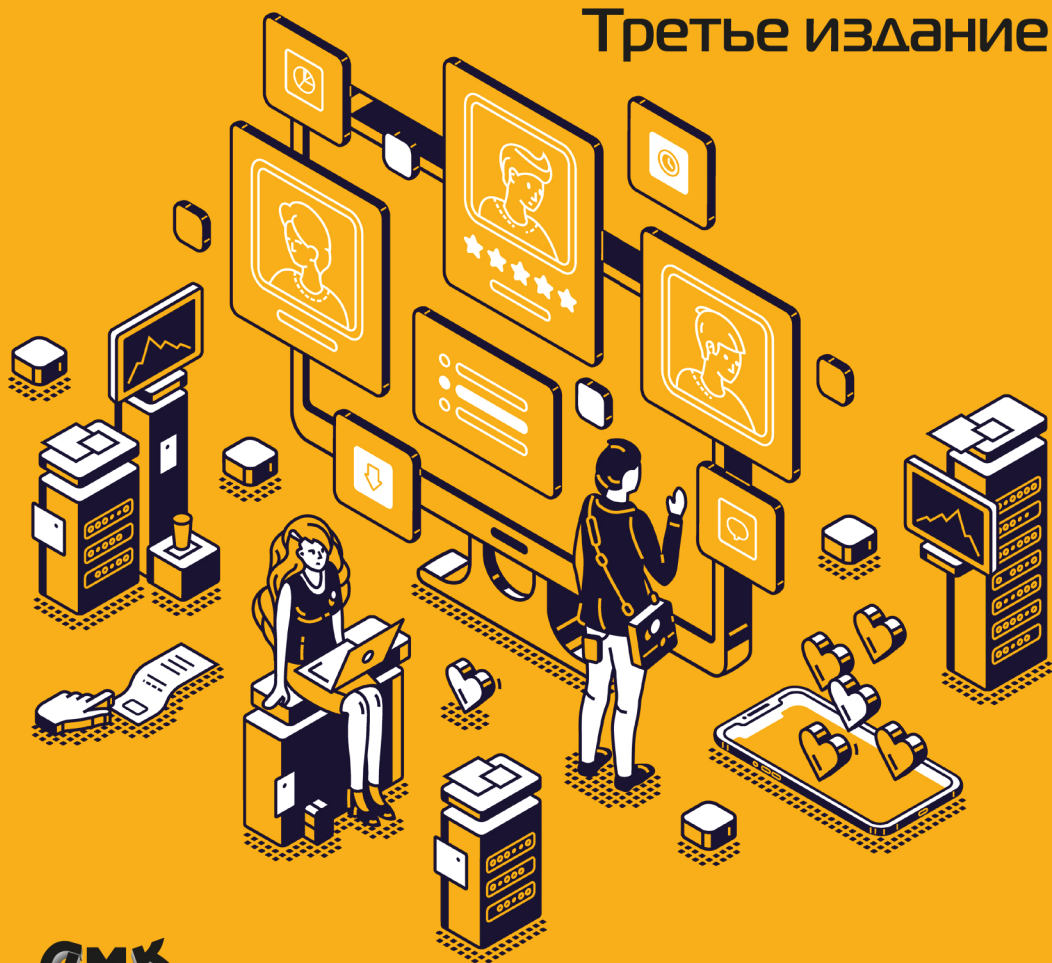


Рэндал Э. Брайант  
Дэвид Р. О'Хамарон

# Компьютерные системы

Архитектура и программирование

Третье издание



Рэндал Э. Брайант  
Дэвид Р. О'Халларон

# Компьютерные системы: архитектура и программирование

3-е издание

# Computer Systems

## A Programmer's Perspective

Third edition

Randal E. Bryant  
Carnegie Mellon University

David R. O'Hallaron  
Carnegie Mellon University

PEARSON

# Компьютерные системы: архитектура и программирование

3-е издание

Рэндал Э. Брайант  
университет Карнеги–Меллона

Дэвид Р. О'Халларон  
университет Карнеги–Меллона



УДК 004.2  
ББК 32.972  
Б87

Б87 Рэндал Э. Брайант, Дэвид Р. О'Халларон

Компьютерные системы: архитектура и программирование. 3-е изд. / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2022. – 994 с.: ил.

**ISBN 978-5-97060-492-2**

В книге описываются стандартные элементы архитектуры, такие как центральный процессор, память, порты ввода-вывода, а также операционная система, компилятор, компоновщик и сетевое окружение. Демонстрируются способы представления данных и программ на машинном уровне, приемы оптимизации программ, особенности управления потоками выполнения и виртуальной памятью, а также методы сетевого и параллельного программирования. Приведенные в книге примеры для процессоров, совместимых с Intel (x86\_64), написаны на языке С и выполняются в операционной системе Linux.

Издание адресовано студентам и преподавателям по IT-специальностям, а также будет полезно разработчикам, желающим повысить свой профессиональный уровень и писать программы, эффективно использующие возможности компьютерной архитектуры.

Authorized translation from the English language edition, entitled *Computer Systems: A Programmer's Perspective*, 3rd Edition, by Randal E. Bryant and David R. O'Hallaron, published by Pearson Education, Inc, publishing as Pearson, Copyright © 2016.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0-13-409266-9 (англ.)

ISBN 978-5-97060-492-2 (рус.)

Copyright © 2016, 2011, and 2003 by  
Randal E. Bryant and David R. O'Hallaron, 2021  
© Оформление, перевод на русский язык,  
издание, ДМК Пресс, 2022

---

*Студентам и преподавателям курса 15-213  
университета Карнеги–Меллона, вдохновившим  
нас на переработку и уточнение этого издания*

# Оглавление

<b>Предисловие от издательства .....</b>	<b>17</b>
<b>Вступление.....</b>	<b>18</b>
<b>Об авторах.....</b>	<b>34</b>
<b>Глава 1. Экскурс в компьютерные системы.....</b>	<b>36</b>
1.1. Информация – это биты + контекст.....	38
1.2. Программы, которые переводятся другими программами в различные формы .....	39
1.3. Как происходит компиляция .....	41
1.4. Процессоры читают и интерпретируют инструкции, хранящиеся в памяти .....	42
1.4.1. Аппаратная организация системы.....	42
1.4.2. Выполнение программы hello .....	44
1.5. Различные виды кеш-памяти .....	46
1.6. Устройства памяти образуют иерархию.....	47
1.7. Операционная система управляет работой аппаратных средств.....	48
1.7.1. Процессы .....	49
1.7.2. Потоки.....	50
1.7.3. Виртуальная память.....	51
1.7.4. Файлы.....	53
1.8. Обмен данными в сетях.....	53
1.9. Важные темы .....	55
1.9.1. Закон Амдала .....	56
1.9.2. Конкуренция и параллелизм .....	57
1.9.3. Важность абстракций в компьютерных системах.....	60
1.10. Итоги.....	61
Библиографические заметки.....	61
Решения упражнений .....	61
<b>Часть I</b>	
<b>Структура программы и ее выполнение .....</b>	<b>63</b>
<b>Глава 2. Представление информации и работа с ней.....</b>	<b>64</b>
2.1. Хранение информации.....	68
2.1.1. Шестнадцатеричная система счисления.....	68
2.1.2. Размеры данных .....	71
2.1.3. Адресация и порядок следования байтов .....	74
2.1.4. Представление строк .....	80
2.1.5. Представление программного кода .....	81
2.1.6. Введение в булеву алгебру .....	82
2.1.7. Битовые операции в С .....	85
2.1.8. Логические операции в С.....	87

2.1.9. Операции сдвига в С.....	88
<b>2.2. Целочисленные представления .....</b>	<b>90</b>
2.2.1. Целочисленные типы .....	91
2.2.2. Представление целых без знака .....	92
2.2.3. Представление в дополнительном коде.....	94
2.2.4. Преобразования между числами со знаком и без знака.....	99
2.2.5. Числа со знаком и без знака в С.....	104
2.2.6. Расширение битового представления числа .....	106
2.2.7. Усечение чисел .....	109
2.2.8. Советы по приемам работы с числами со знаком и без знака .....	111
<b>2.3. Целочисленная арифметика .....</b>	<b>113</b>
2.3.1. Сложение целых без знака .....	113
2.3.2. Сложение целых в дополнительном коде .....	118
2.3.3. Отрицание целых в дополнительном коде .....	123
2.3.4. Умножение целых без знака .....	124
2.3.5. Умножение целых в дополнительном коде .....	124
2.3.6. Умножение на константу.....	128
2.3.7. Деление на степень двойки .....	130
2.3.8. Заключительные размышления о целочисленной арифметике .....	134
<b>2.4. Числа с плавающей точкой.....</b>	<b>135</b>
2.4.1. Дробные двоичные числа.....	136
2.4.2. Представление значений с плавающей точкой в стандарте IEEE .....	139
2.4.3. Примеры чисел .....	141
2.4.4. Округление .....	146
2.4.5. Операции с плавающей точкой .....	148
2.4.6. Значения с плавающей точкой в С .....	150
<b>2.5. Итоги.....</b>	<b>151</b>
Библиографические заметки.....	152
Домашние задания.....	153
Правила представления целых чисел на битовом уровне .....	154
Правила представления чисел с плавающей точкой на битовом уровне .....	165
Решения упражнений .....	167
<b>Глава 3. Представление программ на машинном уровне .....</b>	<b>184</b>
3.1. Историческая перспектива .....	187
<b>3.2. Программный код.....</b>	<b>190</b>
3.2.1. Машинный код.....	190
3.2.2. Примеры кода .....	192
3.2.3. Замечание по форматированию.....	195
<b>3.3. Форматы данных.....</b>	<b>197</b>
<b>3.4. Доступ к информации .....</b>	<b>198</b>
3.4.1. Спецификаторы операндов .....	200
3.4.2. Инструкции перемещения данных .....	201
3.4.3. Примеры перемещения данных .....	205
3.4.4. Вталкивание данных в стек и выталкивание из стека .....	208
<b>3.5. Арифметические и логические операции.....</b>	<b>209</b>

3.5.1. Загрузка эффективного адреса .....	210
3.5.2. Унарные и бинарные операции .....	212
3.5.3. Операции сдвига .....	212
3.5.4. Обсуждение .....	213
3.5.5. Специальные арифметические операции .....	215
3.6. Управление .....	218
3.6.1. Флаги условий .....	218
3.6.2. Доступ к флагам .....	219
3.6.3. Инструкции перехода .....	222
3.6.4. Кодирование инструкций перехода .....	223
3.6.5. Реализация условного ветвления потока управления .....	225
3.6.6. Реализация условного ветвления потока данных .....	229
3.6.7. Циклы .....	235
3.6.8. Оператор switch .....	245
3.7. Процедуры .....	250
3.7.1. Стек времени выполнения .....	251
3.7.2. Передача управления .....	252
3.7.3. Передача данных .....	256
3.7.4. Локальные переменные на стеке .....	258
3.7.5. Локальные переменные в регистрах .....	260
3.7.6. Рекурсивные процедуры .....	262
3.8. Распределение памяти под массивы и доступ к массивам .....	264
3.8.1. Базовые принципы .....	264
3.8.2. Арифметика указателей .....	266
3.8.3. Вложенные массивы .....	267
3.8.4. Массивы фиксированных размеров .....	268
3.8.5. Массивы переменных размеров .....	271
3.9. Структуры разнородных данных .....	273
3.9.1. Структуры .....	273
3.9.2. Объединения .....	276
3.9.3. Выравнивание .....	279
3.10. Комбинирование инструкций управления потоком выполнения и передачи данных в машинном коде .....	282
3.10.1. Указатели .....	283
3.10.2. Жизнь в реальном мире: использование отладчика GDB .....	284
3.10.3. Ссылки на ячейки за границами выделенной памяти и переполнение буфера .....	286
3.10.4. Предотвращение атак методом переполнения буфера .....	290
3.10.5. Поддержка кадров стека переменного размера .....	295
3.11. Вычисления с плавающей точкой .....	298
3.11.1. Операции перемещения и преобразования данных .....	300
3.11.2. Операции с плавающей точкой в процедурах .....	305
3.11.3. Арифметические операции с плавающей точкой .....	305
3.11.4. Определение и использование констант с плавающей точкой .....	307
3.11.5. Поразрядные логические операции с числами с плавающей точкой .....	308
3.11.6. Операции сравнения значений с плавающей точкой .....	309

3.11.7. Заключительные замечания об операциях с плавающей точкой.....	312
3.12. Итоги .....	312
Библиографические заметки.....	313
Домашние задания.....	314
Решения упражнений .....	325
<b>Глава 4. Архитектура процессора .....</b>	<b>349</b>
4.1. Архитектура системы команд Y86-64.....	352
4.1.1. Состояние, видимое программисту .....	352
4.1.2. Инструкции Y86-64.....	353
4.1.3. Кодирование инструкций .....	355
4.1.4. Исключения в архитектуре Y86-64.....	360
4.1.5. Программы из инструкций Y86-64.....	361
4.1.6. Дополнительные сведения об инструкциях Y86-64.....	366
4.2. Логическое проектирование и язык HCL .....	368
4.2.1. Логические вентили .....	368
4.2.2. Комбинационные цепи и булевы выражения в HCL.....	369
4.2.3. Комбинационные цепи для слов и целочисленные выражения в HCL...	371
4.2.4. Принадлежность множеству .....	375
4.2.5. Память и синхронизация .....	375
4.3. Последовательные реализации Y86-64 (SEQ) .....	378
4.3.1. Организация обработки в несколько этапов .....	378
4.3.2. Аппаратная реализация последовательной архитектуры SEQ .....	387
4.3.3. Синхронизация в последовательной реализации SEQ .....	391
4.3.4. Реализация этапов в последовательной версии SEQ .....	394
4.4. Общие принципы конвейерной обработки .....	402
4.4.1. Вычислительные конвейеры.....	402
4.4.2. Подробное описание работы конвейера.....	404
4.4.3. Ограничения конвейерной обработки.....	406
4.4.4. Конвейерная обработка с обратной связью.....	408
4.5. Конвейерные реализации Y86-64 .....	409
4.5.1. SEQ+: переупорядочение этапов обработки.....	409
4.5.2. Добавление конвейерных регистров.....	411
4.5.3. Переупорядочение сигналов и изменение их маркировки.....	415
4.5.4. Прогнозирование следующего значения PC .....	416
4.5.5. Риски конвейерной обработки .....	418
4.5.6. Обработка исключений .....	431
4.5.7. Реализация этапов в PIPE .....	434
4.5.8. Управляющая логика конвейера.....	441
4.5.9. Анализ производительности .....	451
4.5.10. Незаконченная работа.....	454
4.6. Итоги.....	457
4.6.1. Имитаторы Y86-64.....	458
Библиографические заметки.....	458
Домашние задания.....	459
Решения упражнений .....	465

<b>Глава 5. Оптимизация производительности программ</b>	<b>478</b>
5.1. Возможности и ограничения оптимизирующих компиляторов	481
5.2. Выражение производительности программы	484
5.3. Пример программы	486
5.4. Устранение неэффективностей в циклах	490
5.5. Сокращение вызовов процедур	493
5.6. Устранение избыточных ссылок на память	495
5.7. Общее описание современных процессоров	498
5.7.1. Общие принципы функционирования	498
5.7.2. Производительность функционального блока	502
5.7.3. Абстрактная модель работы процессора	504
5.8. Развертывание циклов	510
5.9. Увеличение степени параллелизма	514
5.9.1. Несколько аккумуляторов	515
5.9.2. Переупорядочение операций	520
5.10. Обобщение результатов оптимизации комбинирующего кода	524
5.11. Некоторые ограничивающие факторы	525
5.11.1. Вытеснение регистров	525
5.11.2. Прогнозирование ветвлений и штрафы за ошибки предсказания	526
5.12. Понятие производительности памяти	530
5.12.1. Производительность операций загрузки	530
5.12.2. Производительность операций сохранения	531
5.13. Жизнь в реальном мире: методы повышения производительности	537
5.14. Выявление и устранение узких мест производительности	538
5.14.1. Профилирование программ	538
5.14.2. Использование профилировщика при выборе кода для оптимизации	540
5.15. Итоги	544
Библиографические заметки	545
Домашние задания	545
Решения упражнений	548
<b>Глава 6. Иерархия памяти</b>	<b>553</b>
6.1. Технологии хранения информации	554
6.1.1. Память с произвольным доступом	554
6.1.2. Диски	562
6.1.3. Твердотельные диски	572
6.1.4. Тенденции развития технологий хранения	574
6.2. Локальность	577
6.2.1. Локальность обращений к данным программы	577
6.2.2. Локальность выборки инструкций	579
6.2.3. В заключение о локальности	579
6.3. Иерархия памяти	581
6.3.1. Кеширование в иерархии памяти	582
6.3.2. В заключение об иерархии памяти	585
6.4. Кеш-память	586

6.4.1. Обобщенная организация кеш-памяти .....	586
6.4.2. Кеш с прямым отображением.....	588
6.4.3. Ассоциативные кеши.....	595
6.4.4. Полностью ассоциативные кеши.....	597
6.4.5. Проблемы с операциями записи .....	600
6.4.6. Устройство реальной иерархии кешей.....	601
6.4.7. Влияние параметров кеша на производительность .....	602
6.5. Разработка программ, эффективно использующих кеш .....	603
6.6. Все вместе: влияние кеша на производительность программ .....	608
6.6.1. Гора памяти.....	608
6.6.2. Переупорядочение циклов для улучшения пространственной локальности .....	612
6.6.3. Использование локальности в программах.....	615
6.7. Итоги .....	616
Библиографические заметки.....	616
Домашние задания.....	617
Решения упражнений .....	627
<b>Часть II</b>	
<b>Выполнение программ в системе .....</b>	<b>633</b>
<b>Глава 7. Связывание .....</b>	<b>634</b>
7.1. Драйверы компиляторов .....	636
7.2. Статическое связывание.....	637
7.3. Объектные файлы .....	638
7.4. Перемещаемые объектные файлы .....	638
7.5. Идентификаторы и таблицы имен.....	640
7.6. Разрешение ссылок .....	643
7.6.1. Как компоновщик разрешает ссылки на повторяющиеся имена .....	644
7.6.2. Связывание со статическими библиотеками.....	648
7.6.3. Как компоновщики разрешают ссылки на статические библиотеки.....	651
7.7. Перемещение .....	652
7.7.1. Записи перемещения .....	653
7.7.2. Перемещение ссылок.....	654
7.8. Выполняемые объектные файлы .....	657
7.9. Загрузка выполняемых объектных файлов.....	659
7.10. Динамическое связывание с разделяемыми библиотеками.....	660
7.11. Загрузка и связывание с разделяемыми библиотеками из приложений .....	662
7.12. Перемещаемый программный код .....	665
7.13. Подмена библиотечных функций .....	668
7.13.1. Подмена во время компиляции .....	669
7.13.2. Подмена во время компоновки.....	670
7.13.3. Подмена во время выполнения.....	671
7.14. Инструменты управления объектными файлами.....	673
7.15. Итоги .....	673



Библиографические заметки.....	674
Домашние задания.....	674
Решения упражнений .....	677
<b>Глава 8. Управление исключениями .....</b>	<b>680</b>
8.1. Исключения.....	682
8.1.1. Обработка исключений .....	683
8.1.2. Классы исключений.....	685
8.1.3. Исключения в системах Linux/x86-64 .....	687
8.2. Процессы .....	690
8.2.1. Логический поток управления .....	691
8.2.2. Конкурентные потоки управления.....	692
8.2.3. Изолированное адресное пространство .....	693
8.2.4. Пользовательский и привилегированный режимы .....	693
8.2.5. Переключение контекста .....	694
8.3. Системные вызовы и обработка ошибок .....	695
8.4. Управление процессами .....	696
8.4.1. Получение идентификатора процесса .....	697
8.4.2. Создание и завершение процессов .....	697
8.4.3. Утилизация дочерних процессов.....	701
8.4.4. Приостановка процессов.....	706
8.4.5. Загрузка и запуск программ .....	707
8.4.6. Запуск программ с помощью функций fork и execve .....	709
8.5. Сигналы .....	712
8.5.1. Терминология сигналов .....	714
8.5.2. Посылка сигналов .....	715
8.5.3. Получение сигналов .....	717
8.5.4. Блокировка и разблокировка сигналов.....	720
8.5.5. Обработка сигналов.....	721
8.5.6. Синхронизация потоков во избежание неприятных ошибок конкурентного выполнения .....	730
8.5.7. Явное ожидание сигналов .....	732
8.6. Нелокальные переходы .....	735
8.7. Инструменты управления процессами.....	739
8.8. Итоги.....	739
Библиографические заметки.....	740
Домашние задания.....	740
Решения упражнений .....	747
<b>Глава 9. Виртуальная память .....</b>	<b>750</b>
9.1. Физическая и виртуальная адресация .....	752
9.2. Пространства адресов.....	753
9.3. Виртуальная память как средство кеширования.....	754
9.3.1. Организация кеша DRAM.....	754
9.3.2. Таблицы страниц .....	755
9.3.3. Попадание в кеш DRAM .....	756
9.3.4. Промах кеша DRAM .....	757

9.3.5. Размещение страниц.....	758
9.3.6. И снова о локальности.....	759
9.4. Виртуальная память как средство управления памятью.....	759
9.5. Виртуальная память как средство защиты памяти.....	761
9.6. Преобразование адресов.....	762
9.6.1. Интегрирование кешей и виртуальной памяти.....	765
9.6.2. Ускорение трансляции адресов с помощью TLB.....	766
9.6.3. Многоуровневые таблицы страниц.....	767
9.6.4. Все вместе: сквозное преобразование адресов.....	769
9.7. Практический пример: система памяти Intel Core i7/Linux.....	773
9.7.1. Преобразование адресов в Core i7.....	774
9.7.2. Система виртуальной памяти Linux.....	776
9.8. Отображение в память.....	780
9.8.1. И снова о разделяемых объектах.....	781
9.8.2. И снова о функции fork.....	783
9.8.3. И снова о функции execve.....	783
9.8.4. Отображение в память на уровне пользователя с помощью функции mmap.....	785
9.9. Динамическое распределение памяти.....	786
9.9.1. Функции malloc и free.....	787
9.9.2. Что дает динамическое распределение памяти.....	790
9.9.3. Цели механизмов распределения памяти и требования к ним.....	791
9.9.4. Фрагментация.....	792
9.9.5. Вопросы реализации.....	793
9.9.6. Неявные списки свободных блоков.....	794
9.9.7. Размещение распределенных блоков.....	796
9.9.8. Разбиение свободных блоков.....	796
9.9.9. Увеличение объема динамической памяти.....	797
9.9.10. Объединение свободных блоков.....	797
9.9.11. Объединение с использованием граничных тегов.....	798
9.9.12. Все вместе: реализация простого механизма распределения памяти.....	800
9.9.13. Явные списки свободных блоков.....	807
9.9.14. Раздельные списки свободных блоков.....	808
9.10. Сборка мусора.....	811
9.10.1. Основы сборки мусора.....	811
9.10.2. Алгоритм сборки мусора Mark&Sweep.....	813
9.10.3. Консервативный алгоритм Mark&Sweep для программ на C.....	814
9.11. Часто встречающиеся ошибки.....	815
9.11.1. Разыменование недопустимых указателей.....	815
9.11.2. Чтение неинициализированной области памяти.....	816
9.11.3. Переполнение буфера на стеке.....	816
9.11.4. Предположение о равенстве размеров указателей и объектов, на которые они указывают.....	816
9.11.5. Ошибки занижения или завышения на единицу.....	817
9.11.6. Ссылка на указатель вместо объекта.....	817

9.11.7. Неправильное понимание арифметики указателей .....	818
9.11.8. Ссылки на несуществующие переменные .....	818
9.11.9. Ссылка на данные в свободных блоках .....	818
9.11.10. Утечки памяти .....	819
9.12. Итоги .....	819
Библиографические заметки .....	820
Домашние задания .....	821
Решения упражнений .....	824
<b>Часть III</b>	
<b>Взаимодействие программ .....</b>	<b>829</b>
<b>Глава 10. Системный уровень ввода/вывода .....</b>	<b>830</b>
10.1. Ввод/вывод в Unix .....	831
10.2. Файлы .....	832
10.3. Открытие и закрытие файлов .....	833
10.4. Чтение и запись файлов .....	835
10.5. Надежные чтение и запись с помощью пакета RIO .....	836
10.5.1. Функции RIO небуферизованного ввода/вывода .....	837
10.5.2. Функции RIO буферизованного ввода .....	838
10.6. Чтение метаданных файла .....	842
10.7. Чтение содержимого каталога .....	843
10.8. Совместное использование файлов .....	845
10.9. Переадресация ввода/вывода .....	848
10.10. Стандартный ввод/вывод .....	849
10.11. Все вместе: какие функции ввода/вывода использовать? .....	849
10.12. Итоги .....	851
Библиографические заметки .....	852
Домашние задания .....	852
Решения упражнений .....	853
<b>Глава 11. Сетевое программирование .....</b>	<b>854</b>
11.1. Программная модель клиент-сервер .....	854
11.2. Компьютерные сети .....	855
11.3. Всемирная сеть интернет .....	860
11.3.1. IP-адреса .....	861
11.3.2. Доменные имена интернета .....	863
11.3.3. Интернет-соединения .....	866
11.4. Интерфейс сокетов .....	867
11.4.1. Структуры адресов сокетов .....	868
11.4.2. Функция socket .....	869
11.4.3. Функция connect .....	869
11.4.4. Функция bind .....	870
11.4.5. Функция listen .....	870
11.4.6. Функция accept .....	870
11.4.7. Преобразование имен хостов и служб .....	872
11.4.8. Вспомогательные функции для интерфейса сокетов .....	876

11.4.9. Примеры эхо-клиента и эхо-сервера .....	879
11.5. Веб-серверы .....	881
11.5.1. Основные сведения о вебе .....	881
11.5.2. Веб-контент .....	882
11.5.3. Транзакции HTTP .....	884
11.5.4. Обслуживание динамического контента .....	886
11.6. Все вместе: разработка небольшого веб-сервера TINY .....	889
11.7. Итоги .....	896
Библиографические заметки .....	896
Домашние задания .....	897
Решения упражнений .....	898
<b>Глава 12. Конкурентное программирование .....</b>	<b>901</b>
12.1. Конкурентное программирование с процессами .....	903
12.1.1. Конкурентный сервер, основанный на процессах .....	904
12.1.2. Достоинства и недостатки подхода на основе процессов .....	905
12.2. Конкурентное программирование с мультиплексированием ввода/вывода .....	906
12.2.1. Конкурентный на основе мультиплексирования ввода/вывода, управляемый событиями .....	909
12.2.2. Достоинства и недостатки мультиплексирования ввода/вывода .....	913
12.3. Конкурентное программирование с потоками выполнения .....	914
12.3.1. Модель выполнения многопоточных программ .....	914
12.3.2. Потоки Posix .....	915
12.3.3. Создание потоков .....	916
12.3.4. Завершение потоков .....	916
12.3.5. Утилизация завершившихся потоков .....	917
12.3.6. Обособление потоков .....	917
12.3.7. Инициализация потоков .....	918
12.3.8. Конкурентный многопоточный сервер .....	918
12.4. Совместное использование переменных несколькими потоками выполнения .....	920
12.4.1. Модель памяти потоков .....	921
12.4.2. Особенности хранения переменных в памяти .....	921
12.4.3. Совместно используемые переменные .....	922
12.5. Синхронизация потоков выполнения с помощью семафоров .....	922
12.5.1. Граф выполнения .....	925
12.5.2. Семафоры .....	928
12.5.3. Использование семафоров для исключительного доступа к ресурсам .....	929
12.5.4. Использование семафоров для организации совместного доступа к ресурсам .....	930
12.5.5. Все вместе: конкурентный сервер на базе предварительно созданных потоков .....	935
12.6. Использование потоков выполнения для организации параллельной обработки .....	938
12.7. Другие вопросы конкурентного выполнения .....	944

12.7.1. Безопасность в многопоточном окружении .....	944
12.7.2. Реентерабельность .....	946
12.7.3. Использование библиотечных функций в многопоточных программах.....	947
12.7.4. Состояние гонки.....	948
12.7.5. Взаимоблокировка (тупиковые ситуации).....	950
12.8. Итоги.....	953
Библиографические заметки.....	953
Домашние задания.....	954
Решения упражнений .....	958
<b>Приложение А. Обработка ошибок .....</b>	<b>963</b>
А.1. Обработка ошибок в системе Unix .....	963
А.2. Функции-обертки обработки ошибок.....	965
<b>Библиография.....</b>	<b>968</b>
<b>Предметный указатель .....</b>	<b>975</b>

# Предисловие от издательства

## Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте [www.dmkpress.com](http://www.dmkpress.com), зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com); при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу [http://dmkpress.com/authors/publish\\_book/](http://dmkpress.com/authors/publish_book/) или напишите в издательство по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

## Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com), и мы исправим это в следующих тиражах.

## Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Pearson очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

# Вступление

Данная книга предназначена для программистов, желающих повысить свой профессиональный уровень изучением того, что происходит «под кожаным системным блоком» компьютерной системы.

Целью авторов является попытка разъяснить устойчивые концепции, лежащие в основе всех компьютерных систем, а также демонстрация конкретных видов влияния этих идей на корректность, производительность и полезные свойства прикладных программ. Многие книги по компьютерным системам написаны *для создателей* таких систем и описывают, как сконструировать оборудование или реализовать системное программное обеспечение, включая операционную систему, компилятор и сетевой интерфейс. Эта книга, напротив, написана *для программиста* и рассказывает, как прикладные программисты могут использовать свои знания о системах для создания более качественных программ. Конечно, знакомство с требованиями к системам является хорошим первым шагом в обучении их созданию, поэтому эта книга послужит также ценным введением для тех, кто продолжает заниматься созданием аппаратного и программного обеспечения систем. Большинство книг по компьютерным системам также имеют тенденцию сосредотачиваться только на одном аспекте системы, например на аппаратной архитектуре, операционной системе, компиляторе или сети. Эта книга охватывает все эти аспекты и рассматривает их с позиции программиста.

Доскональное изучение и освоение изложенных в книге концепций позволит читателю со временем превратиться в редкий тип профессионального программиста, понимающего саму суть происходящего и способного решить любую задачу. Вы сможете писать программы, которые эффективнее используют возможности операционной системы и системного программного обеспечения, действуют правильно в широком диапазоне рабочих условий и параметров, работают быстрее и не содержат уязвимостей для кибератак. При этом будет заложена основа для изучения таких специфических тем, как компиляторы, архитектура компьютерных систем, операционные системы, сети и кибербезопасность.

## Что нужно знать перед прочтением

Эта книга посвящена системам с аппаратной архитектурой x86-64, являющиеся последним этапом на пути развития, который прошли Intel и ее конкуренты, начинавшие с микропроцессора 8086 в 1978 году. В соответствии с соглашениями об именовании, принятыми в Intel в отношении их линейки микропроцессоров, этот класс микропроцессоров в просторечии называется «x86». По мере развития полупроводниковых технологий, позволяющих размещать на одном кристалле все больше и больше транзисторов, производительность и объем внутренней памяти процессоров значительно увеличились. В ходе этого прогресса они перешли от 16-разрядных слов к 32-разрядным и выпустили процессор IA32, а совсем недавно произошел переход к 64-разрядным словам и появилась архитектура x86-64.

Мы рассмотрим, как машины с этой архитектурой выполняют программы на языке C в Linux. Linux – одна из операционных систем, ведущих свою родословную от операционной системы Unix, первоначально разработанной в Bell Laboratories. К другим членам этого класса операционных систем относятся Solaris, FreeBSD и MacOS X. В последние годы эти операционные системы сохраняли высокий уровень совместимости благодаря усилиям по стандартизации POSIX и Standard Unix Specification. То есть сведения, что приводятся в этой книге, почти напрямую применимы ко всем этим «Unix-подобным» операционным системам.

**Еще незнакомы с С?** Совет по языку программирования С

В помощь читателям, плохо знакомым или незнакомым с языком С, авторами предлагаются примечания, подобные данному, для подчеркивания функций, особенно важных для С. Предполагается, что читатели знакомы с С++ или Java.

В тексте содержится множество примеров программного кода, которые мы компилировали и опробовали в системах Linux. Мы предполагаем, что у вас есть доступ к такой системе, что вы можете входить в нее и умеете выполнять простые действия, такие как получение списка файлов или переход в другой каталог. Если ваш компьютер работает под управлением Microsoft Windows, то мы рекомендуем установить одну из множества виртуальных машин (например, VirtualBox или VMWare), которые позволяют программам, написанным для одной операционной системы (гостевой ОС), запускаться в другой (несущей ОС, или хост-ОС).

Также предполагается, что читатель знаком с С или С++. Если весь опыт программиста ограничивается работой с Java, переход потребует от него больше усилий, но авторы окажут всю необходимую помощь. Java и С имеют общий синтаксис и управляющие операторы. Однако в С есть свои особенности (в частности, указатели, явное распределение динамической памяти и форматируемый ввод/вывод), которых нет в Java. К счастью, С – не очень сложный язык, он прекрасно описан в классической книге Брайана Кернигана и Денниса Ритчи [61]. Вне зависимости от «подкованности» читателя в области программирования, эта книга послужит ценным дополнением к его библиотеке. Если прежде вы использовали только интерпретируемые языки, такие как Python, Ruby или Perl, то вам определенно стоит посвятить некоторое время изучению С, прежде чем продолжить читать эту книгу.

В начальных главах книги рассматривается взаимодействие между программами на С и их аналогами на машинном языке. Все примеры на машинном языке были созданы с помощью компилятора GNU GCC на процессоре x86-64. Наличия какого бы то ни было опыта работы с аппаратными средствами, машинными языками или программирования в ассемблере не предполагается.

**Как читать книгу**

Изучение принципов работы компьютерных систем с точки зрения программиста – занятие очень увлекательное, потому что проходит в интерактивном режиме. Изучив что-то новое, вы тут же можете это проверить и получить результат, что называется, из первых рук. На самом деле авторы полагают, что единственным способом познания систем является их *практическое исследование*: либо путем решения конкретных упражнений, либо написанием и выполнением программ в реально существующих системах.

Система является предметом изучения всей книги. При представлении какой-либо новой концепции в тексте она будет сопровождаться иллюстрацией в форме одной или нескольких *практических задач*, которые нужно сразу же постараться решить, чтобы проверить правильность понимания изложенного. Решения упражнений приводятся в конце каждой главы. Попробуйте сначала самостоятельно решать эти практические задачи и только потом проверяйте правильность выбранного пути. В конце каждой главы также представлены *домашние задания* различной степени сложности. Каждому домашнему заданию присвоен определенный уровень сложности:

- ◆ для решения достаточно нескольких минут; требуется минимальный объем программирования (или не требуется вообще);
- ◆◆ для решения потребуется до 20 минут. Часто нужно написать и опробовать программный код. Многие такие задания созданы на основе задач, приведенных в примерах;



- ◆◆◆ для решения потребуется приложить значительные усилия; по времени на решение может уйти до 2 часов. Как правило, для выполнения этих заданий требуется написать и опробовать значительный объем кода;
- ◆◆◆ лабораторная работа, на выполнение которой может уйти до 10 часов.

Все примеры кода в тексте книги отформатированы автоматически (без всякого ручного вмешательства) и получены из программ на C, скомпилированных с помощью GCC и протестированных в Linux. Конечно, в вашей системе может быть установлена другая версия gcc или вообще другой компилятор, генерирующий другой машинный код, но общее поведение примеров должно быть таким же. Весь исходный код доступен на веб-странице книги [csapp.cs.cmu.edu](http://csapp.cs.cmu.edu). Имена файлов с исходным кодом документируются с использованием горизонтальных полос, окружающих отформатированный код. Например, программу из листинга 1 можно найти в файле `hello.c` в каталоге `code/intro/`. Мы советуем обязательно опробовать примеры программ у себя по мере их появления.

#### Листинг 1. Типичный пример кода

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6     return 0;
7 }
```

`code/intro/hello.c`

`code/intro/hello.c`

Чтобы не раздувать объем и без того большой книги, мы создали несколько приложений, размещенных в интернете, которые содержат сведения, дополняющие основную книгу. Они отмечены в книге заголовками в форме «Приложение в интернете ТЕМА:ПОДТЕМА», где ТЕМА кратко описывает основную тему, а ПОДТЕМА – раздел темы. Например, «Приложение в интернете DATA:BOOL» содержит сведения по булевой алгебре, дополняющие описание представлений данных в главе 2, а «Приложение в интернете ARCH:VLOG» содержит описание приемов проектирования процессоров с использованием языка описания оборудования Verilog, дополняющее обсуждение конструкции процессора в главе 4. Все эти приложения доступны на веб-странице книги (<https://csapp.cs.cmu.edu/3e/waside.html>).

## Обзор книги

Книга состоит из 12 глав, охватывающих основные принципы компьютерных систем:

### Глава 1. Экскурс в компьютерные системы.

В этой главе описываются основные идеи и темы, относящиеся к компьютерным системам, на примере исследования жизненного цикла простой программы «hello, world».

### Глава 2. Представление информации и работа с ней.

Здесь описывается компьютерная арифметика с упором на свойства представлений числа без знака и числа в дополнительном двоичном коде, которые имеют значение для программистов. В данной главе рассматривается представление чисел и, следовательно, диапазон значений, которые можно запрограммировать для отдельно взятого размера слова. Авторы обсуждают влияние

преобразований типов чисел со знаком и без знака и математические свойства арифметических операций. Для начинающих программистов часто оказывается откровением, что сложение (в дополнительном коде) или умножение двух положительных чисел может дать отрицательный результат. С другой стороны, арифметика дополнительного кода удовлетворяет многим алгебраическим свойствам целочисленной арифметики, благодаря чему компилятор может преобразовать операцию умножения на константу в последовательность сдвигов и сложений. Для иллюстрации принципов и применения булевой алгебры авторы используют поразрядные операции С. Формат IEEE с плавающей точкой описывается в терминах представления значений и математических свойств операций с плавающей точкой.

Абсолютное понимание компьютерной арифметики принципиально для создания надежных программ. Например, программисты и компиляторы не могут заменить выражение  $(x < y)$  на  $(x - y < 0)$  из-за возможности переполнения. Они не могут даже заменить его выражением  $(-y < -x)$  из-за асимметричности диапазона отрицательных и положительных чисел в дополнительном коде. Арифметическое переполнение является обычным источником ошибок программирования и уязвимостей в системе безопасности, однако мало в какой книге можно найти описание свойств компьютерной арифметики, сделанное с точки зрения самого программиста.

#### **О чем рассказывается во врезках?**

На протяжении всей книги вам будут встречаться подобные примечания. В них приводятся некоторые дополнительные сведения к обсуждаемой теме. Примечания преследуют несколько целей. Одни представляют собой небольшие исторические экскурсы. Например, как появились С, Linux и Internet? Другие разъясняют какие-либо понятия. Например, чем отличаются кеш, множество и блок? Третьи примечания описывают примеры «из жизни». Например, как ошибка в вычислениях с плавающей точкой уничтожила французскую ракету или какие геометрические и функциональные параметры имеют коммерческие жесткие диски. И наконец, некоторые примечания – это всего лишь забавные комментарии.

### *Глава 3. Представление программ на машинном уровне.*

Авторы научат читать машинный код x86-64, созданный компилятором С. Здесь представлены основные шаблоны инструкций для различных управляющих структур, таких как условные операторы, циклы и операторы выбора. Также рассматривается реализация процедур, включая выделение места на стеке, условные обозначения использования реестров и передачу параметров. В главе рассматриваются различные структуры данных, например структуры, объединения и массивы, их размещение в памяти и доступ к ним. Здесь еще будет показано, как выглядят программы с точки зрения машины, что поможет понять распространенные уязвимости, такие как переполнение буфера, и шаги, которые программист, компилятор и операционная система могут предпринять для уменьшения этих угроз. Изучение данной главы поможет повысить профессиональный уровень, потому что при этом появится понимание, как компьютер воспринимает программы.

### *Глава 4. Архитектура процессора.*

В этой главе описываются комбинаторные и последовательные логические элементы, после чего демонстрируется, как эти элементы можно объединить в

информационный канал, выполняющий упрощенный набор инструкций x86-64 с названием «Y86-64». Сначала будет рассматриваться однопоточный тракт данных. Его архитектура проста, но не отличается высоким быстродействием. Затем будет представлено понятие *конвейерной обработки*, в которой различные шаги, необходимые для обработки инструкции, реализуются как отдельные этапы. В каждый конкретный момент этапы конвейера могут обрабатывать разные инструкции. Получившийся в результате пятиступенчатый процессорный конвейер намного более реалистичен. Управляющая логика процессора описывается с использованием простого языка описания аппаратных средств – HCL. Проекты аппаратного обеспечения, написанные на HCL, можно компилировать и объединять в симуляторы, а затем использовать для создания описания Verilog, пригодного для производства реального оборудования.

### *Глава 5. Оптимизация производительности программ.*

В этой главе представлен ряд методов повышения производительности кода, при этом идея состоит в том, что программисты учатся писать код на С так, чтобы компилятор мог затем сгенерировать эффективный машинный код. Сначала рассматриваются преобразования, сокращающие объем работы, которую предстоит выполнить программе, и, следовательно, которые должны стать стандартной практикой при написании любых программ для любых машин. Затем мы перейдем к преобразованиям, повышающим степень параллелизма на уровне команд в сгенерированном машинном коде, чтобы повысить их производительность на современных «суперскалярных» процессорах. Для обоснования этих преобразований будет представлена простая модель работы современных процессоров и показано, как измерить потенциальную производительность программы с точки зрения критических путей с использованием графического представления программы. Вы будете удивлены, насколько можно ускорить программу, применив простые преобразования к коду на С.

### *Глава 6. Иерархия памяти.*

Память является для программистов одной из самых «заметных» частей компьютерной системы. До этой главы читатели полагались на концептуальную модель памяти в форме одномерного массива с постоянным временем доступа. На практике память представляет собой иерархию запоминающих устройств разной емкости, стоимости и быстродействия. В главе рассматриваются разные типы памяти, такие как ОЗУ и ПЗУ, а также геометрические параметры и устройство существующих современных дисковых накопителей, организация этих запоминающих устройств в иерархию. Авторы показывают возможность иерархической организации посредством локальности ссылок. Данные идеи конкретизируются представлением уникального взгляда на систему памяти как на «гору памяти» со «скалами» временной локальности и «склонами» пространственной локальности. В заключение рассказывается, как повысить производительность программных приложений путем усовершенствования их временной и пространственной локальности.

### *Глава 7. Связывание.*

В данной главе описывается статическое и динамическое связывание, включая такие понятия, как: перемещаемые и выполняемые объектные файлы, разрешение символов, перемещение, статические библиотеки, разделяемые библиотеки, перемещаемый код и подмена библиотечных функций (library interpositioning). Тема связывания редко рассматривается в книгах по компьютерным системам, но авторы решили включить ее в эту книгу по двум причи-

нам. Во-первых, некоторые типичные ошибки, с которыми сталкиваются программисты, как раз возникают на этапе связывания и особенно характерны для крупных программных пакетов. Во-вторых, объектные файлы, создаваемые компоновщиками, связаны с такими понятиями, как загрузка, виртуальная память и отображение памяти.

#### *Глава 8. Управление исключениями.*

В этой главе авторы отступают от однопрограммной модели, вводя общую концепцию потока управления исключениями (не совпадающего с обычным потоком управления путем ветвления в условных операторах и в точках вызова процедур). Мы рассмотрим примеры потоков управления исключениями, существующих на всех уровнях системы, от аппаратных исключений и прерываний низкого уровня до переключения контекста между параллельными процессами, внезапных изменений в потоке управления, вызванных передачей сигналов Linux, и нелокальных переходов в C, разрывающих стройную структуру стека.

В этой части книги будет представлено фундаментальное понятие *процесса* как абстракции выполняющейся программы. Здесь авторы расскажут, как работают процессы, как их создавать и как ими можно управлять из прикладных программ, и покажут, как прикладные программисты могут запустить несколько процессов с помощью системных вызовов Linux. По окончании этой главы вы сможете написать простую командную оболочку для Linux с поддержкой управления заданиями. Эта глава также станет первым знакомством с недетерминированным поведением, возникающим при параллельных вычислениях.

#### *Глава 9. Виртуальная память.*

Описывает представление системы виртуальной памяти с целью дать некоторое понимание ее особенностей и принципов работы. Здесь вы узнаете, как разные процессы, действующие одновременно, могут использовать один и тот же диапазон адресов, совместно использовать одни страницы памяти и иметь индивидуальные копии других. В этой главе также описываются вопросы, связанные с управлением виртуальной памятью и манипуляциями с ней. В частности, мы уделим большое внимание инструментам распределения памяти из стандартной библиотеки, таким как `malloc` и `free`. Обсуждение данной темы преследует несколько целей. Прежде всего оно подкрепляет концепцию о том, что пространство виртуальной памяти является всего лишь массивом байтов, который программа может разделить на блоки разного размера для хранения данных. Помогает понять последствия ошибок обращения к памяти в программах, такие как утечки и недействительные ссылки в указателях. Наконец, многие программисты реализуют свои инструменты распределения памяти, оптимизированные под требования и характеристики конкретного приложения. Эта глава в большей степени, чем любая другая, демонстрирует преимущества неразрывного освещения аппаратных и программных аспектов компьютерных систем. Книжки о традиционных компьютерных архитектурах и операционных системах обычно представляют виртуальную память только с одной стороны.

#### *Глава 10. Системный уровень ввода/вывода.*

В этой главе рассматриваются основные концепции ввода/вывода в системе Unix, такие как файлы и дескрипторы. Авторы описывают совместное использование файлов, принципы работы переадресации ввода/вывода и доступ к метаданным файлов. Здесь также представлен пример разработки надежного пакета буферизованного ввода/вывода, прекрасно справляющегося с любопыт-

ным поведением подсистемы ввода/вывода, известным как *недостача*, когда библиотечные функции возвращают только часть ввода. В главе описывается стандартная библиотека ввода/вывода языка C и ее связь с подсистемой ввода/вывода в Linux с упором на ограничения стандартного ввода/вывода, делающие его непригодным для сетевого программирования. Вообще говоря, темы, охваченные в этой главе, служат основой для двух следующих глав, посвященных сетевому и параллельному программированию.

#### *Глава 11. Сетевое программирование.*

Сети являются своеобразными устройствами ввода/вывода для программ, объединяющими многие из понятий, описанных ранее: процессы, сигналы, порядок следования байтов, отображение памяти и динамическое распределение пространства запоминающих устройств. Сетевые программы также являются одними из первых кандидатов на применение приемов параллельного программирования, о котором рассказывается в следующей главе. Данная глава – лишь тонкий срез глобального предмета сетевого программирования, необходимый для создания простенького веб-сервера. Здесь будет представлена модель клиент–сервер, лежащая в основе всех сетевых приложений. Авторы представят взгляд программиста на сеть Интернет и покажут, как писать сетевые клиенты и серверы, используя интерфейс сокетов. И наконец, в главе будет представлен протокол HTTP и разработан простой веб-сервер.

#### *Глава 12. Конкурентное программирование.*

Эта глава описывает принципы конкурентного (параллельного) программирования на примере сетевого сервера. Авторы сравнивают и противопоставляют три основных механизма, используемых для создания конкурентных программ: процессы, мультиплексирование ввода/вывода и потоки выполнения – и показывают возможность их использования при создании серверов, способных обслуживать множество одновременных соединений. Здесь же описаны основные принципы синхронизации с использованием семафорных операций *P* и *V*, безопасность потоков выполнения и реентерабельность, а также состояние взаимоблокировки. Конкурентное программирование играет важную роль в большинстве сетевых приложений. Также в этой главе описывается конкурентное программирование на уровне потоков выполнения, что позволяет ускорить решение задач на многоядерных процессорах. Чтобы все ядра правильно и эффективно работали над одной вычислительной задачей, требуется тщательная координация потоков, выполняющихся конкурентно.

### **Что нового в этом издании**

Первое издание этой книги было опубликовано в 2003 году, а второе – в 2011. Учитывая быстрое развитие компьютерных технологий, содержимое книги на удивление хорошо сохранило свою актуальность. Принципиальное устройство компьютеров Intel x86, на которых выполняются программы на C под управлением Linux (и других похожих операционных систем), мало изменилось за эти годы. Однако изменения в аппаратных технологиях, компиляторах, интерфейсах программных библиотек и накопленный опыт преподавания потребовали существенного пересмотра материала книги.

Самым большим изменением по сравнению со вторым изданием является переход с представления, основанного на сочетании IA32 и x86-64, к представлению, основанному исключительно на x86-64. Это смещение акцента повлияло на содержимое многих глав. Вот краткое перечисление основных значительных изменений.

### *Глава 1. Экскурс в компьютерные системы.*

Мы переместили обсуждение закона Амдала из главы 5 в эту главу.

### *Глава 2. Представление информации и работа с ней.*

В многочисленных отзывах читатели и рецензенты сообщают, что некоторые сведения в этой главе сложны для понимания. Поэтому мы постарались упростить форму подачи материала, поясняя моменты, которые обсуждаются в строгом математическом стиле. Это позволит читателям сначала просмотреть математические детали, чтобы получить общее представление, а затем вернуться к подробному описанию.

### *Глава 3. Представление программ на машинном уровне.*

Мы перешли от представления, основанного на комбинации IA32 и x86-64, к представлению только на основе x86-64. Мы также обновили примеры кода, генерируемые более свежими версиями gcc. Как результат глава претерпела существенные изменения, включая изменение порядка, в котором представлены некоторые концепции. Мы также добавили описание аппаратной поддержки вычислений с плавающей точкой. А для сохранения совместимости добавили приложение в интернете, описывающее машинный код для IA32.

### *Глава 4. Архитектура процессора.*

Мы обновили описание архитектуры процессора, выполнив переход с 32-разрядной архитектуры на архитектуру с поддержкой 64-разрядных слов и операций.

### *Глава 5. Оптимизация производительности программ.*

Мы обновили эту главу, отразив возможности последних поколений процессоров x86-64 в плане производительности. С введением большего количества функциональных модулей и более сложной логики управления разработанная нами модель производительности программ, основанная на представлении программ в форме потока данных, стала предсказывать производительность еще надежнее, чем раньше.

### *Глава 6. Иерархия памяти.*

Мы обновили эту главу с учетом последних технологий.

### *Глава 7. Связывание.*

Мы переписали эту главу, перейдя на архитектуру x86-64, расширили обсуждение использования глобальной таблицы смещений GOT и таблицы компоновки процедур PLT для создания перемещаемого кода и добавили новый раздел о мощном методе связывания, известном как *library interpositioning* (подмена библиотечных функций).

### *Глава 8. Управление исключениями.*

Мы добавили более строгое описание обработчиков сигналов, включив функции, которые можно безопасно вызывать при обработке асинхронных сигналов, специальные рекомендации по написанию обработчиков сигналов и использование `sigsuspend` для приостановки обработчиков.

### *Глава 9. Виртуальная память.*

Эта глава изменилась незначительно.

*Глава 10. Системный уровень ввода/вывода.*

Мы добавили новый раздел о файлах и иерархии файлов, но в остальном эта глава изменилась незначительно.

*Глава 11. Сетевое программирование.*

Мы представили новые методы протоколонеависимого и потокобезопасного сетевого программирования с использованием современных функций `getaddrinfo` и `getnameinfo`, пришедших на смену устаревшим и нереентерабельным функциям `gethostbyname` и `gethostbyaddr`.

*Глава 12. Параллельное программирование.*

Мы расширили обсуждение параллельного программирования, добавив потоки выполнения, использование которых позволяет программам быстрее решать свои задачи на многоядерных машинах.

Также мы добавили и пересмотрели множество практических и домашних заданий по всей книге.

## Происхождение книги

Книга родилась из вводного курса, разработанного в университете Карнеги–Меллона (УКМ) осенью 1998 года и получившего название «15-213: Введение в компьютерные системы». С тех пор курс читается в каждом семестре, и в каждом семестре его слушают более 400 студентов, от второкурсников до аспирантов, самых разных специальностей. Данный курс стал основой для большинства других курсов по компьютерным системам в университете Карнеги–Меллона.

Идея этого курса заключалась в том, чтобы познакомить студентов с компьютерами, взглянув на них с другой стороны. Мало кто из студентов смог бы самостоятельно построить компьютерную систему. С другой стороны, от большинства обучающихся и даже инженеров по вычислительной технике требуется повседневное использование компьютеров и умение программировать. Поэтому авторы данной книги решили начать знакомство с системами с точки зрения программиста и при выборе тем использовали следующий своеобразный фильтр: тема будет освещаться только в том случае, если она связана с производительностью, корректностью или с полезными свойствами пользовательских программ на C.

К примеру, исключены темы, связанные с аппаратными сумматорами и конструкцией шин. В курсе также имелись темы, посвященные машинному языку, но вместо подробного рассмотрения языка ассемблера мы предпочли сконцентрироваться на том, как компилятор транслирует конструкции языка C в машинный код, включая операции с указателями, циклы, вызовы процедур и возврат из них. Кроме того, мы решили более широко взглянуть на систему как на комплекс аппаратных и программных средств и включили в книгу такие темы, как связывание, загрузка, процессы, сигналы, оптимизация производительности, виртуальная память, ввод/вывод, а также сетевое и параллельное программирование.

Данный подход позволил сделать курс практичным, конкретным, наглядным и на редкость интересным для студентов. Ответная реакция с их стороны и со стороны коллег по факультету была незамедлительной и положительной, и авторы книги поняли, что преподаватели из других учебных заведений тоже смогут воспользоваться их работами. Это и стало предпосылкой появления данной книги, написанной лекционным конспектом курса и которую мы теперь обновили, чтобы отразить изменения в технологиях и подходах к реализации систем.

Благодаря выходу новых изданий и переводам этой книги на разные языки она и многие ее варианты стали частью учебных программ по информатике и компьютерной инженерии в сотнях колледжей и университетов по всему миру.



## Преподавателям: курсы на основе этой книги

Преподаватели могут использовать эту книгу для проведения нескольких видов курсов по компьютерным системам. В табл. 1 перечислены пять категорий таких курсов. Конкретный курс зависит от требований учебной программы, личного вкуса, а также опыта и способностей студентов. Курсы в табл. 1 перечислены слева направо в порядке близости ко взгляду программиста на систему. Вот их краткое описание.

- УК.** Курс «Устройство компьютеров» с традиционными темами, раскрытыми в нетрадиционном стиле. Охватываются такие традиционные темы, как логическая модель, архитектура процессора, язык ассемблера и системы памяти. При этом больше внимания должно уделяться программной стороне. Например, обсуждение данных должно быть связано с типами данных и операциями в программах на С, а обсуждение ассемблерного кода основываться не на рукописном машинном коде, а на сгенерированном компилятором С.
- УК+.** Курс УК с дополнительным упором на аппаратную сторону и производительность прикладных программ. По сравнению с УК, студенты, слушающие курс УК+, узнают больше об оптимизации кода и об улучшении производительности памяти своих программ на языке С.
- ВКС.** Базовый курс «Введение в компьютерные системы», разработанный для подготовленных программистов, которые понимают влияние оборудования, операционной системы и системы компиляции на производительность и правильность прикладных программ. Существенное отличие от УК+ – отсутствие охвата низкоуровневой архитектуры процессора. Вместо этого программисты учатся работать с высокоуровневой моделью современного процессора. Курс ВКС хорошо вписывается в 10-недельную четверть, но при необходимости может быть продлен до 15-недельного семестра, если проходить его в неторопливом темпе.
- ВКС+.** Базовый курс «Введение в компьютерные системы» с дополнительным охватом таких тем системного программирования, как ввод/вывод системного уровня, сетевое и параллельное программирование. В университете Карнеги–Меллона это семестровый курс, охватывающий все главы данной книги, кроме низкоуровневой архитектуры процессора.
- СП.** Курс «Системное программирование». Этот курс похож на ВКС+, но в нем не преподается оптимизация операций с плавающей запятой и производительности, а также уделяется больше внимания системному программированию, включая управление процессами, динамическое связывание, ввод/вывод на уровне системы, сетевое программирование и параллельное программирование. Преподаватели могут добавить информацию из других источников для обсуждения дополнительных тем, таких как демоны, управление терминалом и межпроцессные взаимодействия в Unix.

Как показывает табл. 1, эта книга дает студентам и преподавателям массу возможностей. Если вы хотите, чтобы ваши ученики познакомились с низкоуровневой архитектурой процессоров, то этот вариант доступен в курсах УК и УК+. С другой стороны, если вы хотите переключиться с текущего курса «Устройство компьютеров» на курс ВКС или ВКС+, но опасаетесь вносить радикальные изменения сразу, то можете организовать постепенный переход к ВКС. Вы можете начать с курса УК, который преподносит традиционные темы нетрадиционным способом, а освоившись с этим материалом – переходить к УК+ и в конечном итоге к ВКС. Если студенты не имеют опыта программирования на С (например, они программировали только на Java), то



вы можете потратить несколько недель на чтение лекций о С, а затем переходить к чтению курса УК или ВКС.

**Таблица 1.** Пять категорий курсов о компьютерных системах, основанных на книге «Компьютерные системы: архитектура и программирование». Курс ВКС+ – это курс 15-213 в университете Карнеги–Меллона.

*Примечание:* символ ⊕ означает частичный охват главы, как то: (1) только аппаратная часть; (2) без динамического распределения памяти; (3) без динамического связывания; (4) без представления данных с плавающей точкой

Глава	Тема	Курс				
		УК	УК+	ВКС	ВКС+	СП
1	Экскурс в компьютерные системы	•	•	•	•	•
2	Представление данных	•	•	•	•	⊕ <sup>(4)</sup>
3	Машинный язык	•	•	•	•	•
4	Архитектура процессора	•	•			
5	Оптимизация кода		•	•	•	
6	Иерархия памяти	⊕ <sup>(1)</sup>	•	•	•	⊕ <sup>(1)</sup>
7	Связывание			⊕ <sup>(3)</sup>	⊕ <sup>(3)</sup>	•
8	Управление исключениями			•	•	•
9	Виртуальная память	⊕ <sup>(2)</sup>	•	•	•	•
10	Ввод/вывод на уровне системы				•	•
11	Сетевое программирование				•	•
12	Параллельное программирование				•	•

Наконец, отметим, что курсы УК+ и СП могут образовать хорошую последовательность из двух семестров (или четверти и семестра). Или же можно подумать о чтении курса ВКС+ как состоящего из ВКС и СП.

### Преподавателям: примеры лабораторных работ в классе

Курс ВКС+ в университете Карнеги–Меллона получил очень высокие оценки от студентов. Медианный балл 5,0/5,0 и средний балл 4,6/5,0 являются типичными оценками студентов курса. Основными достоинствами студенты называют забавные, увлекательные и актуальные лабораторные работы, которые доступны на веб-странице книги. Вот примеры лабораторных работ, которые поставляются с книгой.

#### *Представление данных.*

Эта лабораторная работа требует от студентов реализовать простые логические и арифметические функции с использованием строго ограниченного подмножества языка С. Например, они должны вычислить абсолютное значение числа, используя только битовые операции. Эта лабораторная работа помогает студентам понять представление типов данных в языке С на двоичном уровне и особенности побитовых операций.

#### *Двоичные бомбы.*

*Двоичная бомба* – это программа, которая передается студентам в виде скомпилированного файла. При запуске она предлагает пользователю ввести шесть разных строк. Если при вводе будет допущена ошибка, то бомба «взрывается» – выводит сообщение об ошибке и регистрирует событие на сервере оценки. Студенты должны «обезвредить» свои уникальные бомбы, дизассемблировав

программы и определив, как должны выглядеть эти шесть строк. Лабораторная работа учит студентов понимать язык ассемблера, а также заставляет их научиться пользоваться отладчиком.

#### *Переполнение буфера.*

Студенты должны изменить поведение двоичного выполняемого файла, используя уязвимость переполнения буфера. Эта лабораторная работа учит студентов осторожности обращения со стеком и показывает опасности кода, уязвимого для атак переполнения буфера.

#### *Архитектура.*

Некоторые домашние задания из главы 4 можно объединить в лабораторную работу, где студенты изменяют HCL-описание процессора, добавляя новые инструкции, изменяя политику прогнозирования ветвлений или добавляя и удаляя обходные пути и регистрируя порты. Полученные процессоры можно моделировать и запускать с помощью автоматических тестов, которые обнаруживают большинство возможных ошибок. Эта лабораторная работа позволяет студентам познакомиться с захватывающими аспектами проектирования процессоров, не требуя полного знания языков логического проектирования и описания оборудования.

#### *Производительность.*

Студенты должны оптимизировать производительность основных функций приложения, таких как свертка или транспонирование матриц. Эта лабораторная работа наглядно демонстрирует свойства кеш-памяти и дает студентам опыт низкоуровневой оптимизации программ.

#### *Кеш.*

Эта лабораторная работа является альтернативой лабораторной работе «Производительность». В ней студенты должны написать симулятор кеша общего назначения, а затем оптимизировать базовые функции программы транспонирования матрицы так, чтобы минимизировать количество промахов кеша. При этом мы используем инструмент Valgrind для трассировки реальных адресов.

#### *Командная оболочка.*

Студенты создают свою программу командной оболочки Unix с поддержкой управления заданиями, включая комбинации клавиш **Ctrl+C** и **Ctrl+Z**, а также команды `fg`, `bg` и `jobs`. В этой лабораторной работе учащиеся впервые знакомятся с параллельным программированием и получают четкое представление об управлении процессами в Unix, сигналах и их обработке.

#### *Распределение памяти.*

Студенты реализуют свои версии `malloc`, `free` и (необязательно) `realloc`. Эта лабораторная работа помогает студентам получить четкое представление о структуре и организации данных и требует от них оценки различных компромиссов между эффективностью потребления памяти и времени выполнения.

#### *Прокси.*

Студенты реализуют параллельный веб-прокси, находящийся между браузером и остальной частью Всемирной паутины. Эта лабораторная работа знакомит студентов с такими темами, как веб-клиенты и серверы, и связывает воедино многие концепции курса, такие как порядок следования байтов, файловый ввод/вывод, управление процессами, сигналы, обработка сигналов, отображение памяти, сокеты и параллельное выполнение. Студентам нравится видеть, как работают их программы, обслуживающие реальные веб-браузеры и веб-серверы.

В руководстве для преподавателя курса «Компьютерные системы: архитектура и программирование» подробно описаны все лабораторные работы, а также даются инструкции по загрузке вспомогательного программного обеспечения.

### Благодарности к третьему изданию

Нам приятно поблагодарить всех, кто помог в создании этого третьего издания книги «Компьютерные системы: архитектура и программирование».

Мы хотели бы сказать спасибо нашим коллегам из университета Карнеги–Меллона, которые преподавали курс ВКС на протяжении многих лет и представили так много ценных отзывов: Гая Блеллока (Guy Blelloch), Роджера Данненберга (Roger Dannenberg), Дэвида Экхардта (David Eckhardt), Франца Франчетти (Franz Franchetti), Грегга Гангера (Greg Ganger), Сета Гольдштейна (Seth Goldstein), Халеда Харраса (Khaled Harras), Грегга Кесдена (Greg Kesden), Брюса Мэггса (Bruce Maggs), Тодда Моури (Todd Mowry), Андреаса Новацика (Andreas Nowatzky), Фрэнка Пфеннинга (Frank Pfenning), Маркуса Пуэшеля (Markus Pueschel) и Энтони Роу (Anthony Rowe). Дэвид Винтерс (David Winters) очень помог в установке и настройке эталонного Linux-сервера.

Джейсон Фриттс (Jason Fritts; университет Сент-Луиса) и Синди Норрис (Cindy Norris; штат Аппалачи) предоставили нам подробные и вдумчивые рецензии ко второму изданию. Или Гонг (Yili Gong; Уханьский университет) перевел книгу на китайский язык, обеспечил поддержку страницы книги с исправлениями для китайской версии и предоставил множество замечаний об ошибках. Годмар Бэк (Godmar Back; Технологический институт Вирджинии) помог значительно улучшить книгу, познакомив нас с понятиями безопасности при обработке асинхронных сигналов и протоколонеависимого сетевого программирования.

Большое спасибо нашим внимательным читателям, сообщившим об ошибках во втором издании: Рами Аммари (Rami Ammari), Полу Анагностопулосу (Paul Anagnostopoulos), Лукасу Бааренфангеру (Lucas Bärenfänger), Годмару Бэку (Godmar Back), Джи Бину (Ji Bin), Шарбелу Боусеману (Sharbel Bousemaan), Ричарду Каллахану (Richard Callahan), Сету Чайкену (Seth Chaiken), Ченгу Чену (Cheng Chen), Либо Чену (Libo Chen), Тао Ду (Tao Du), Паскалю Гарсия (Pascal Garcia), Или Гонгу (Yili Gong), Рональду Гринбергу (Ronald Greenberg), Дорухану Гулозу (Dorukhan Gülöz), Донгу Хану (Dong Han), Доминику Хельму (Dominik Helm), Рональду Джонсу (Ronald Jones), Мустафе Каздагли (Mustafa Kazdagli), Гордону Киндлманну (Gordon Kindlmann), Санкару Кришнану (Sankar Krishnan), Канаку Кшетри (Kanak Kshetri), Джунлину Лу (Junlin Lu), Цянцян Луо (Qiangqiang Luo), Себастьяну Луи (Sebastian Luy), Лей Ма (Lei Ma), Эшвину Нанджаппа (Ashwin Nanjappa), Грегуару Пароди (Gregoire Paradis), Йонасу Пфеннингеру (Jonas Pfenninger), Карлу Пичотта (Karl Pichotta), Дэвиду Рэмси (David Ramsey), Каустабху Рою (Kaustabh Roy), Дэвиду Селвараджу (David Selvaraj), Санкару Шанмугаму (Sankar Shanmugam), Доминику Смулковска (Dominique Smulkowska), Дару Сорбо (Dag Sørbø), Майклу Спиру (Michael Spear), Ю Танака (Yu Tanaka), Стивену Трикановичу (Steven Tricanowicz), Скотту Райту (Scott Wright), Вайки Райту (Waiki Wright), Чженшень Яну (Zhengshan Yan), Хану Сю (Han Xu), Фиро Янь (Firo Yang), Шуанг Янь (Shuang Yang), Джону Е (John Ye), Такето Ёсида (Taketo Yoshida), Яну Чжу (Yan Zhu) и Майклу Зинку (Michael Zink).

Также благодарим наших читателей, которые внесли свой вклад в создание лабораторных работ, в том числе Годмара Бэка (Godmar Back; Технологический институт Вирджинии), Таймона Била (Taumon Beal; Вустерский политехнический институт), Арана Клаусона (Aran Clauson; университет Западного Вашингтона), Кэри Грея (Cary Gray; колледж Уитона), Пола Хайдака (Paul Haiduk; аграрно-технический университет Западного Техаса), Лена Хейми (Len Hamey; университет Маккуори), Эдди Келера (Eddie Kohler; Гарвард), Хью Лауэра (Hugh Lauer; Вустерский политехнический институт), Роберта Марморштейна (Robert Marmorstein; университет Лонгвуда) и Джеймса Рили (James Riely; университет Де Поля).

И снова Пол Анагностопулос (Paul Anagnostopoulos) из Windfall Software мастерски справился с набором книги и руководил производственным процессом. Большое спасибо Полу и его звездной команде: Ричарду Кэмпбу (Richard Camp; корректура), Дженифер МакКлейн (Jennifer McClain, корректура), Лорел Мюллер (Laurel Muller; художественное оформление) и Теду Ло (Ted Laux; составление предметного указателя). Пол даже заметил ошибку в нашем описании происхождения аббревиатуры BSS, которая оставалась незамеченной с момента выхода первого издания!

Наконец, мы хотим поблагодарить наших друзей из Prentice Hall. Марсию Хортон (Marcia Horton) и нашего редактора Мэтта Гольдштейна (Matt Goldstein), которые неутомимо поддерживали и ободряли нас, и мы глубоко благодарны им за это.

## Благодарности ко второму изданию

Мы глубоко признательны всем, кто так или иначе помог нам писать это второе издание книги «Компьютерные системы: архитектура и программирование».

В первую очередь мы благодарим наших коллег, преподававших курс ВКС в университете Карнеги–Меллона, за их бесценные отзывы и поддержку: Гая Блеллока (Guy Blelloch), Роджера Данненберга (Roger Dannenberg), Дэвида Экхардта (David Eckhardt), Грегга Гангера (Greg Ganger), Сета Гольдштейна (Seth Goldstein), Грегга Кесдена (Greg Kesden), Брюса Мэггса (Bruce Maggs), Тодда Моури (Todd Mowry), Андреаса Новацика (Andreas Nowatzky), Фрэнка Пфеннинга (Frank Pfenning) и Маркуса Пуэшеля (Markus Pueschel).

Также благодарим наших остроглазых читателей, которые сообщили об обнаруженных ими ошибках и опечатках в первом издании: Дэниела Амеланга (Daniel Amelang), Руи Баптиста (Rui Baptista), Куарупа Баррейринхаса (Quarup Barreirinhas), Майкла Бомбика (Michael Bombyk), Йорга Брауэра (Jörg Brauer), Джордана Бро (Jordan Brough), Исин Цяо (Yixin Cao), Джеймса Кэролла (James Carroll), Руи Карвальо (Rui Carvalho), Хонг-Ки Чоя (Hyoung-Kee Choi), Эла Дэвиса (Al Davis), Гранта Дэвиса (Grant Davis), Кристиана Дюфур (Christian Dufour), Мао Фан (Mao Fan), Тима Фримана (Tim Freeman), Инге Фрика (Inge Frick), Макса Гебхардта (Max Gebhardt), Джеффа Голдблата (Jeff Goldblat), Томаса Гросса (Thomas Gross), Аните Гупта (Anita Gupta), Джона Хемптона (John Hampton), Хип Хонга (Hiep Hong), Грегга Израэльсена (Greg Israelsen), Рональда Джонса (Ronald Jones), Хауди Каземи (Haudy Kazemi), Брайана Келла (Brian Kell), Константина Кусулиса (Constantine Kousoulis), Сашу Краковяк (Sacha Krakowiak), Аруна Кришнасвами (Arun Krishnaswamy), Мартина Куласа (Martin Kulas), Майкла Ли (Michael Li), Зеянь Ли (Zeyang Li), Рики Лю (Ricky Liu), Марио Ло Конте (Mario Lo Conte), Дирка Мааса (Dirk Maas), Девона Мейси (Devon Macey), Карла Марциника (Carl Marciniak), Уилла Марреро (Will Marrero), Симона Мартинса (Simone Martins), Тао Мен (Tao Men), Марка Моррисси (Mark Morrissey), Венката Наиду (Venkata Naidu), Бхаса Налаботула (Bhas Nalabothula), Томаса Ниманна (Thomas Niemann), Эрика Пескина (Eric Peskin), Дэвида По (David Po), Энн Роджерс (Anne Rogers), Джона Росса (John Ross), Майкла Скотта (Michael Scott), Сейки (Seiki), Рэй Ши (Ray Shih), Даррена Шульца (Darren Shultz), Эрика Силкенсена (Erik Silkensen), Сурьянто (Suryanto), Эмиля Тарази (Emil Tarazi), Наванана Тера-Ампорнпунта (Nawanan Theera-Ampornpunt), Джо Трдинича (Joe Trdinich), Майкла Тригобоффа (Michael Trigoboff), Джеймса Труппа (James Troup), Мартина Вонпатека (Martin Vopatek), Алана Уэста (Alan West), Бетси Вольф (Betsy Wolff), Тима Вонга (Tim Wong), Джеймса Вудраффа (James Woodruff), Скотта Райта (Scott Wright), Джеки Сяо (Jackie Xiao), Гуаньпэн Сюй (Guanpeng Xu), Цин Сюй (Qing Xu), Карен Янь (Caren Yang), Инь Юншен (Yin Yongsheng), Ван Юаньсюань (Wang Yuanxuan), Стивена Чжану (Steven Zhang) и Дай Чжун (Day Zhong). Особая благодарность Инге Фрик (Inge Frick), которая обнаружила малоизвестную ошибку в нашем примере с блокировкой и копированием, и Рики Лю (Ricky Liu) за его потрясающие навыки корректуры.

Наши коллеги из Intel Labs – Эндрю Чен (Andrew Chien) и Лимор Фикс (Limor Fix) – оказывали нам невероятную поддержку на протяжении всей работы над книгой. Стив

Шлоссер (Steve Schlosser) любезно предоставил некоторые характеристики дисководов. Кейси Хелфрич (Casey Helfrich) и Майкл Райан (Michael Ryan) установили и обслуживали наш новый сервер с процессором Core i7 на борту. Майкл Козуч (Michael Kozuch), Бабу Пиллаи (Babu Pillai) и Джейсон Кэмпбелл (Jason Campbell) предоставили ценную информацию о производительности системы памяти, многоядерных системах и энергосбережении. Фил Гиббонс (Phil Gibbons) и Шимин Чен (Shimin Chen) поделились своим значительным опытом в разработке твердотельных дисков.

Мы смогли привлечь многих талантливых специалистов, в том числе Вен-Мей Хву (Wen-Mei Hwu), Маркуса Пуэшеля (Markus Pueschel) и Иржи Симса (Jiri Simsa), и они не только поделились с нами подробными отзывами, но и дали множество ценных советов. Джеймс Хой (James Hoe) помог создать Verilog-версию процессора Y86 и выполнил всю работу, необходимую для синтеза действующего оборудования.

Большое спасибо нашим коллегам, представившим свои отзывы к рукописи, среди них: Джеймс Арчибалд (James Archibald; университет Бригама Янга), Ричард Карвер (Richard Carver; университет Джорджа Мейсона), Мирела Дамиан (Mirela Damian; университет Вилланова), Питер Динда (Peter Dinda; Северо-Западный университет), Джон Фиор (John Fiore; университет Темпл), Джейсон Фриттс (Jason Fritts; университет Сент-Луиса), Джон Грейнер (John Greiner; университет Райса), Брайан Харви (Brian Harvey; университет Калифорнии, Беркли), Дон Хеллер (Don Heller; университет штата Пенсильвания), Вей Чунг Сю (Wei Chung Hsu; университет Миннесоты), Мишель Хью (Michelle Hugue; университет Мэриленда), Джереми Джонсон (Jeremy Johnson; университет Дрекселя), Джефф Кеннинг (Geoff Kuenning; колледж Харви Мадда), Рики Лю (Ricky Liu), Сэм Мэдден (Sam Madden; Массачусетский технологический институт), Фред Мартин (Fred Martin; Массачусетский университет, Лоуэлл), Абрахам Матта (Abraham Matta; Бостонский университет), Маркус Пуэшель (Markus Pueschel; Университет Карнеги-Меллона), Норман Рэмси (Norman Ramsey; Университет Тафтса), Гленн Рейнманн (Glenn Reinmann; Калифорнийский университет в Лос-Анджелесе), Микела Тауфер (Michela Taufer; университет Делавэра) и Крейг Зиллес (Craig Zilles; Иллинойсский университет в Урбане-Шампейне).

Пол Анагностопулос (Paul Anagnostopoulos) из Windfall Software проделал большую работу по верстке книги и руководил производственной группой. Большое спасибо Полу и его превосходной команде: Рик Кэмп (Rick Camp; редактор), Джо Сноуден (Joe Snowden; верстальщик), Мэри Эллен Н. Оливер (MaryEllen N. Oliver; корректор), Лорел Мюллер (Laurel Muller; художник) и Теду Лауксу (Ted Laux; составление предметного указателя).

Наконец, мы хотим поблагодарить наших друзей из Prentice Hall. Марсию Хортон (Marcia Horton), которая всегда была рядом с нами, и нашего редактора Мэтта Гольдштейна (Matt Goldstein), от начала и до конца обеспечивавшего безупречное руководство процессом. Мы глубоко благодарны им за их помощь, поддержку и идеи.

## Благодарности к первому изданию

Мы безмерно благодарны всем друзьям и коллегам за их вдумчивую критику и сердечную поддержку. Отдельное спасибо студентам курса 15-213, чья энергия и энтузиазм «не давали нам засохнуть». Спасибо Нику Картеру (Nick Carter) и Винни Фьюриа (Vinnie Furia) за то, что любезно предоставили нам пакет mallos.

Гай Блеллок (Guy Blelloch), Грег Кесден (Greg Kesden), Брюс Мэггс (Bruce Maggs) и Тодд Маори (Todd Mowry) являлись преподавателями курса на протяжении нескольких семестров, и их неоценимая поддержка помогала нам постоянно совершенствовать представляемый материал. Духовным руководством и постоянным содействием во время работы мы обязаны Хербу Дерби (Herb Derby). Алан Фишер (Allan Fisher), Гарт Гибсон (Garth Gibson), Томас Гросс (Thomas Gross), Сатья (Satya), Питер Стинкесте (Peter Steenkiste) и Хью Чанг (Hui Zhang) дали нам толчок для начала работы над книгой.

Предложение Гарта «запустило маховик», было поддержано и тщательно проработано командой Алана Фишера. Марк Стелик (Mark Stehlik) и Питер Ли (Peter Lee) оказали неоценимую помощь в организации материала этой книги и привели его в соответствие с учебным планом. Грег Кесден (Greg Kesden) дал ценные отзывы, описав влияние ВКС на курс ОС (операционные системы). Грег Гэнгер (Greg Ganger) и Джирри Шиндлер (Jiri Schindler) любезно предоставили некоторые характеристики дисководов и ответили на наши вопросы о существующих в настоящее время дисках. Том Стрикер (Tom Stricker) показал нам «гору памяти». Джеймс Хоу (James Hoe) поделился интересными идеями о том, как следует подавать материал об архитектуре процессора.

Группа студентов – Халил Амири (Khalil Amiri), Анжела Демке Браун (Angela Demke Brown), Крис Колохан (Chris Colohan), Джейсон Кроуфорд (Jason Crawford), Питер Динда (Peter Dinda), Хулио Лопес (Julio Lopez), Брюс Лоукамп (Bruce Lowekamp), Джефф Пирс (Jeff Pierce), Санджей РАО (Sanjay Rao), Баладжи Сарпешкар (Balaji Sarpeshkar), Блейк Шолль (Blake Scholl), Санжи Сешиа (Sanjit Seshia), Грег Стефан (Greg Steffan), Тианкай Ту (Tiankai Tu), Кип Уокер (Kip Walker) и Йинглайн Цзе (Yinglian Xie) – помогали в разработке содержимого курса. В частности, Крис Колохан придумал увлекательный (и забавный) стиль представления материала, используемый до сих пор, а также разработал легендарную «двоичную бомбу», оказавшуюся превосходным инструментом обучения работе с машинным кодом и концепциям отладки.

Крис Бауэр (Chris Bauer), Алан Кокс (Alan Cox), Питер Динда (Peter Dinda), Сандия Дуаркадис (Sandhya Dwarkadas), Джон Грейнер (John Greiner), Брюс Джейкоб (Bruce Jacob), Барри Джонсон (Barry Johnson), Дон Хеллер (Don Heller), Брюс Лоукамп (Bruce Lowekamp), Грег Моррисетт (Greg Morrisett), Брайан Ноубл (Brian Noble), Бобби Отмер (Bobbie Othmer), Билл Пью (Bill Pugh), Майкл Скотт (Michael Scott), Марк Сматермен (Mark Smotherman), Грег Стефан (Greg Steffan) и Боб Уайер (Bob Wier) потратили кучу времени на ознакомление с рукописью книги. Отдельное спасибо Элу Дэвису (Al Davis; университет Юты), Питеру Динда (Peter Dinda; Северо-Западный университет), Джону Грейнеру (John Greiner; университет Райе), Вей Су (Wei Hsu; университет Миннесоты), Брюсу Лоукампу (Bruce Lowekamp; колледж Уильяма и Мэри), Бобби Отмеру (Bobbie Othmer; Университет Миннесоты), Майклу Скотту (Michael Scott; университет Рочестера) и Бобу Уайеру (Bob Wier; колледж Роки Маунтин) за тестирование бета-версий лабораторных работ. Также огромное спасибо всем их студентам!

Нам очень хочется поблагодарить наших коллег из Prentice Hall. Марсия Хортон (Marcia Horton), Эрик Фрэнк (Eric Frank) и Гарольд Стоун (Harold Stone) оказывали постоянную неослабевающую поддержку в течение всего процесса работы. Гарольд при этом помогал в точном описании исторических фактов по созданию процессорных архитектур RISC и CISC. Джерри Ралия (Jerry Ralya) глубоко изучала материал и многому научила нас в плане литературного изложения своих мыслей.

И наконец, хочется выразить свою признательность техническим писателям Брайану Кернигану (Brian Kernighan) и покойному В. Ричарду Стивенсу (W. Richard Stevens) за то, что они доказали нам, что и техническую литературу можно писать красиво.

Огромное спасибо всем.

Рэнди Брайант  
Дэйв О'Халларон  
*Питсбург, Пенсильвания*



# Об авторах



**Рэндал Э. Брайант (Randal E. Bryant)** в 1973 г. получил степень бакалавра в Мичиганском университете, после чего поступил в аспирантуру Технологического института в Массачусетсе. В 1981 г. получил степень доктора наук по теории вычислительных машин и систем. В течение трех лет работал ассистентом профессора в Калифорнийском технологическом институте; на факультет в Карнеги–Меллон пришел в 1984 г. Пять лет возглавлял факультет информатики и затем десять лет занимал пост декана этого же факультета. В настоящее время является профессором информатики в университете. Он также проводит встречи с представителями Департамента электротехники и вычислительной техники.

Вот уже 40 лет профессор Брайант преподает курсы по компьютерным системам как на уровне бакалавриата, так и на уровне магистратуры. За многие годы преподавания курсов компьютерной архитектуры он начал смещать акцент с проектирования компьютеров к тому, как программисты могли бы писать более эффективные и надежные программы, более полно понимая системы. Вместе с профессором О’Холлароном разработал в университете Карнеги–Меллон учебный курс 15-213 «Введение в компьютерные системы», легший в основу данной книги. Также преподавал курсы по алгоритмам, программированию, компьютерным сетям, распределенным системам и проектированию СБИС (сверхбольших интегральных схем).

Исследования профессора Брайанта имеют отношение к проектированию инструментальных программных средств в помощь разработчикам аппаратных средств при верификации корректности создаваемых ими систем. Сюда входят несколько видов моделирующих программ, а также инструменты формальной верификации, доказывающие корректность проектирования посредством математических методов. Им опубликовано свыше 150 технических работ. Результаты исследований профессора Брайанта используются ведущими производителями компьютерной техники, включая Intel, IBM, Fujitsu и Microsoft. Является лауреатом многих наград за исследования, в числе которых две награды за изобретения, а также награда за технические достижения от Semiconductor Research Corporation (SRC), премия Канеллакиса за теоретические и практические исследования от Association for Computer Machinery (ACM), премия Бейкера (W. R. G. Baker Award), премия Эммануэля Пиора (Emmanuel Piore Award), премия Фила Кауфмана (Phil Kaufman Award) и премия Ричарда Ньютона (A. Richard Newton Award) от Institute of Electrical and Electronics Engineers (IEEE). Является сотрудником как ACM, так и IEEE и членом Национальной академии США и Американской академии искусств и наук.



**Дэвид Р. О'Холларон (David R. O'Hallaron)** – профессор информатики, электротехники и вычислительной техники в университете Карнеги–Меллона. Получил докторскую степень в университете Вирджинии. С 2007 по 2010 год занимал должность директора Intel Labs в Питтсбурге.

В течение 20 лет преподавал курсы по компьютерным системам на уровне бакалавриата и магистратуры по таким темам, как компьютерная архитектура, введение в компьютерные системы, проектирование параллельных процессоров и сетевые службы. Вместе с профессором Брайантом разработал курс в университете Карнеги–Меллона, который привел к созданию этой книги. В 2004 году был награжден премией

Герберта Саймона (Herbert Simon Award) за выдающиеся успехи в преподавании от школы компьютерных наук CMUSchool of Computer Science, лауреаты которой выбираются на основе опросов студентов.

Профессор О'Халларон занимается исследованиями в области компьютерных систем, уделяя особое внимание программным системам для научных вычислений, вычислений с интенсивным использованием данных и виртуализации. Одной из самых известных примеров его работ является проект Quake, в котором участвовала группа специалистов по информатике, инженеров-строителей и сейсмологов. Все вместе они реализовали возможность предсказывать движение земной коры во время сильных землетрясений. В 2003 году профессор О'Халларон и другие члены команды Quake получили премию Гордона Белла (Gordon Bell Prize) – высшую международную награду в области высокопроизводительных вычислений. В настоящее время он работает над проблемой автогрейдинга, то есть над программами, способными оценивать качество других программ.



# Глава 1

## Экскурс в компьютерные системы

- 1.1. Информация – это биты + контекст.
- 1.2. Программы, которые переводятся другими программами в различные формы.
- 1.3. Как происходит компиляция.
- 1.4. Процессоры читают и интерпретируют инструкции, хранящиеся в памяти.
- 1.5. Различные виды кеш-памяти.
- 1.6. Устройства памяти образуют иерархию.
- 1.7. Операционная система управляет работой аппаратных средств.
- 1.8. Обмен данными в сетях.
- 1.9. Важные темы.
- 1.10. Итоги.

Библиографические заметки.

Домашние задания.

Решения упражнений.

**К**омпьютерная система состоит из аппаратных средств и программного обеспечения, которые взаимодействуют, обеспечивая выполнение прикладных программ. Конкретные реализации систем со временем претерпевают изменения, однако идеи, лежащие в их основе, остаются неизменными. Все аппаратные и программные компоненты, из которых состоят вычислительные системы и которые выполняют одни и те же функции, похожи друг на друга. Эта книга предназначена для программистов, желающих повысить свою квалификацию за счет лучшего понимания того, как эти компоненты работают и какое влияние они оказывают на правильное функционирование их программ.

Вам предстоит увлекательное путешествие. Если вы не пожалеете времени для изучения идей, изложенных в этой книге, то вы вступите на путь, который в конечном итоге позволит вам стать представителем немногочисленной категории профессиональных программистов, обладающих четким пониманием принципов работы вычислительной системы и ее влияния на ваши прикладные программы.

Вы приобретете специальные навыки, например будете знать, как избежать странных числовых ошибок, вызванных особенностями представления чисел конкретным

компьютером. Узнаете, как оптимизировать программный код на языке С путем применения специальных приемов, которые используют особенности современных процессоров и систем памяти. Получите представление о том, как компилятор реализует вызовы процедур и как использовать эти знания, чтобы избежать прорех в системе защиты, вызванных сбоями, возникающими в результате переполнения буферов, которые мешают работе сетевого программного обеспечения. Научитесь распознавать и избегать неприятных ошибок во время связывания, которые приводят в замешательство программистов средней руки. Узнаете, как написать свою командную оболочку, свой пакет процедур динамического распределения памяти и даже свой собственный веб-сервер. Познакомитесь с перспективами и подводными камнями параллельного выполнения – с темой, которая приобретает все большее значение с распространением процессоров, имеющих несколько ядер.

### Происхождение языка программирования С

Язык С разрабатывался с 1969 года по 1973 год Деннисом Ритчи (Dennis Ritchie), сотрудником Bell Laboratories. Национальный институт стандартизации США (American National Standards Institute, ANSI) утвердил стандарт ANSI C в 1989 году. Этот стандарт дает определение языка программирования С и набора библиотечных функций, известных как стандартная библиотека С. Керниган и Ритчи описали язык в своей классической книге, которую в кругах программистов любовно называют не иначе как «K&R» [61]. По словам Ритчи [92], язык С – это «причудливый, порочный и в то же время безусловный успех». Так все-таки почему успех?

- *Язык С был тесно связан с операционной системой Unix.* Он разрабатывался с самого начала как язык системного программирования для Unix. Большая часть ядра, а также все вспомогательные инструменты и библиотеки были написаны на С. По мере роста популярности Unix во второй половине семидесятых и в начале восьмидесятых годов прошлого столетия многим пришлось столкнуться с языком С, и многим он понравился. Поскольку система Unix была практически полностью написана на С, она легко переносилась на новые машины, а это, в свою очередь, увеличивало круг пользователей и самого языка С, и операционной системы Unix.
- *С – простой, компактный язык.* Его разработкой занимался один человек, а не многочисленный комитет, и результатом явился четкий непротиворечивый язык с небольшим бременем прошлого. В книге K&R дается полное описание языка и стандартной библиотеки, приводятся многочисленные примеры и упражнения, и на это потребовалось всего 261 страница. Простота языка С облегчает его изучение и перенос на разные компьютеры.
- *С разрабатывался для решения практических задач.* Первоначально его целью была реализация операционной системы Unix. Потом обнаружилось, что на нем можно писать любые программы, потому что сам язык давал такую возможность.

С – это язык системного программирования, и в то же время в нем имеются все средства, обеспечивающие возможность написания прикладных программ. Он полностью удовлетворяет потребности некоторой части программистов, но все же подходит не для всех ситуаций. Указатели языка С часто оказываются причиной различных недоразумений и программных ошибок. Языку не хватает также прямой поддержки таких полезных абстракций, как классы, объекты и исключения. Более новые версии этого языка, такие как C++ и Java, позволяют решать подобного рода проблемы и для программ прикладного уровня.

В своих, ставших классическими книгах по программированию на языке С [61] Керниган (Kernighan) и Ритчи (Ritchie) начинают знакомить читателя с языком программирования на примере простой программы `hello` (которая всего лишь выводит текст приветствия), представленной в листинге 1.1. И хотя это очень простая программа, все

основные части системы должны работать согласованно, чтобы довести ее до успешного завершения. В каком-то смысле цель этой книги заключается в том, чтобы помочь вам понять, что происходит и как, когда вы запускаете программу `hello` в своей системе.

**Листинг 1.1.** Программа `hello` (источник: [60])

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6     return 0;
7 }
```

`code/intro/hello.c`

`code/intro/hello.c`

Мы начнем изучение систем с того, что исследуем жизненный цикл программы `hello` от момента ее написания программистом до момента, когда она выполняется системой, выводит свое незатейливое послание и завершается. Двигаясь вперед по жизненному циклу этой программы, мы будем вводить основные понятия, терминологию и компоненты, вступающие в игру. В последующих главах мы подробнее остановимся на этих понятиях и идеях.

## 1.1. Информация – это биты + контекст

Наша программа `hello` начинает свой жизненный цикл как *исходная программа* (или *файл с исходным кодом*), которую программист создает с помощью текстового редактора и сохраняет в файле с именем `hello.c`. Исходный код программы представляет собой последовательность битов, каждый из которых принимает значение 0 или 1, организованных в 8-битные блоки, получившие название *байты*. Каждый байт в исходном коде представляет некоторый символ.

Большинство компьютерных систем представляют текстовые символы в стандарте ASCII (American Standard Code for Information Interchange – американский стандартный код обмена информацией), согласно которому каждый символ представляется уникальным однобайтным целым числом<sup>1</sup>. Например, в листинге 1.2 программа `hello.c` показана в кодах ASCII.

**Листинг 1.2.** Представление текстового файла `hello` в кодах ASCII

#	i	n	c	l	u	d	e	SP	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	SP	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	SP	SP	SP	SP	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	SP	w	o	r	l	d	\	n	"	)	;	\n	SP
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	SP	SP	r	e	t	u	r	n	SP	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

<sup>1</sup> Для представления текстов на языках, отличных от английского, используются другие методы кодирования. См. врезку «Стандарт Юникода для представления текста» в главе 2.

Программа `hello.c` хранится в файле как последовательность байтов. Каждый байт принимает целочисленное значение, соответствующее некоторому символу. Например, первый байт имеет значение 35, которому соответствует символ «#». Второй байт имеет значение 105, которому соответствует символ «i», и т. д. Обратите внимание, что текстовая строка заканчивается невидимым символом *перевода строки* «\n», который представлен целым значением 10. Такие файлы, как `hello.c`, содержащие исключительно символы, называются *текстовыми файлами*, а все другие – *двоичными файлами*.

Представление файла `hello.c` иллюстрирует одну из фундаментальных идей. Вся информация в системе, включая файлы на дисках, программы и данные пользователей, хранящиеся в памяти, а также данные, передаваемые по сети, представляется в виде битовых блоков. Единственное, что отличает разные виды данных друг от друга, – контекст, в котором мы их рассматриваем. Например, в разных контекстах одна и та же последовательность байтов может представлять целое число, число с плавающей точкой, строку символов или машинную инструкцию.

Как программисты мы должны понимать машинное представление чисел, поскольку они не тождественны целым или вещественным числам. Они суть конечные приближения, которые могут вести себя непредсказуемым образом. Эта фундаментальная идея широко используется в главе 2.

## 1.2. Программы, которые переводятся другими программами в различные формы

Программа `hello` начинает свою жизнь как программа на языке высокого уровня, поскольку в этой форме она может быть прочитана и понята человеком. Однако, чтобы запустить программу `hello.c` в системе, операторы на языке C должны быть преобразованы другими программами в некоторую последовательность инструкций на *машинном языке* низкого уровня. Эти инструкции затем упаковываются в *выполняемую объектную программу* и сохраняются в двоичном файле на диске. Объектные программы также называются *выполняемыми объектными файлами*.

В операционной системе Unix преобразование исходного файла в объектный выполняется *драйвером компилятора*:

```
unix> gcc -o hello hello.c
```

Здесь драйвер компилятора GCC читает исходный файл `hello.c` и транслирует его в выполняемый объектный файл `hello`. Трансляция выполняется в четыре этапа, как показано на рис. 1.1. Совокупность программ, выполняющих эти четыре этапа (*препроцессор, компилятор, ассемблер и компоновщик*), называется *системой компиляции*.

- *Этап препроцессора* (или этап предварительной обработки). Препроцессор (`cpp`) изменяет исходную программу в соответствии с директивами, которые начинаются с символа «#». Например, директива `#include <stdio.h>` в строке 1 программы `hello.c` заставляет препроцессор прочитать содержимое системного заголовочного файла `stdio.h` и вставить его непосредственно в текст программы. В результате получается другая программа на языке C, обычно с расширением `.i`.
- *Этап компиляции*. Компилятор (`ccl`) транслирует текстовый файл `hello.i` в текстовый файл `hello.s`, который содержит *программу на языке ассемблера*. Эта программа включает следующее определение функции `main`:

```
1 main:
2     subq $8, %rsp
3     movl $.LC0, %edi
4     call puts
```

```

5    movl $0, %eax
6    addq $8, %rsp
7    ret

```

Каждый оператор в строках 2–7 этого определения точно описывает одну из инструкций низкоуровневого машинного языка в текстовой форме. Польза языка ассемблера прежде всего в том, что он представляет общий выходной язык для компиляторов разных языков высокого уровня. Например, компиляторы языка C и языка Fortran генерируют выходные файлы на одном и том же языке ассемблера.

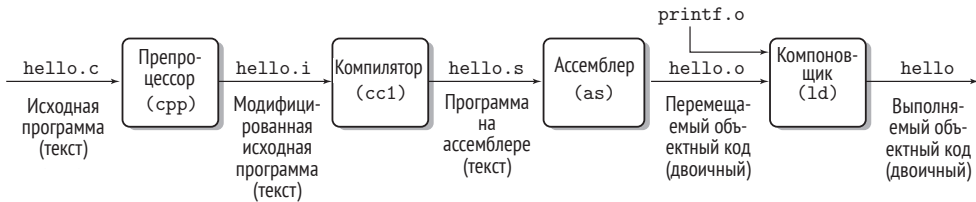


Рис. 1.1. Система компиляции

- *Этап ассемблирования.* Ассемблер (as) транслирует файл hello.s в машинные инструкции, упаковывает их в форму, известную как *перемещаемый объектный код*, и запоминает результат в объектном файле hello.o. Файл hello.o – это двоичный файл, содержащий 17 байт, которые кодируют машинные инструкции, составляющие функцию main. Если открыть hello.o в текстовом редакторе, то вы увидите совершенно непонятную абракадабру.
- *Этап компоновки.* Обратите внимание, что наша программа hello вызывает функцию printf из *стандартной библиотеки C*, которая поставляется в комплекте с любым компилятором языка C. Функция printf находится в отдельном предварительно скомпилированном объектном файле с именем printf.o, который тем или иным способом должен быть объединен с нашей программой hello.o. Это объединение осуществляет компоновщик (ld). В результате получается выполняемый объектный файл (или просто *выполняемый файл*), готовый к загрузке и выполнению системой.

### О проекте GNU

GCC – один из множества полезных инструментов, созданных в рамках проекта GNU (сокращенно от «GNU's Not Unix» – «GNU – не Unix»). Проект GNU – это освобожденная от налогов благотворительная акция, основанная Ричардом Столлменом (Richard Stallman) в 1984 году с амбициозной целью разработать законченную Unix-подобную систему, исходный код которой не обременен ограничениями на его изменение и распространение. В рамках проекта GNU была разработана среда со всеми основными компонентами операционной системы Unix, за исключением ядра, которое разрабатывалось отдельно, а именно в рамках проекта Linux. Среда GNU включает редактор emacs, компилятор GCC, отладчик GDB, ассемблер, компоновщик, утилиты для манипуляции двоичными файлами и другие компоненты. Компилятор GCC развился и ныне поддерживает множество разных языков и способен генерировать код для множества разных машин. В число поддерживаемых языков входят: C, C++, Fortran, Java, Pascal, Objective-C и Ada.

Проект GNU – замечательное достижение, однако сплошь и рядом ему не уделяют должного внимания. Современная мода на программные продукты с открытым исходным кодом (обычно ассоциируется с Linux) обязана своим интеллектуальным происхождением понятию свободное программное обеспечение, возникшему в рамках проекта GNU («свободное» – в смысле «свобода слова», но не в смысле «бесплатное пиво»). Более того, операционная система Linux во многом обязана своей популярности инструментальным средствам GNU, которые позволяют развернуть среду для ядра системы Linux.

## 1.3. Как происходит компиляция

В случае простых программ, таких как `hello.c`, мы можем рассчитывать, что система компиляции произведет правильный и эффективный машинный код. Однако есть несколько важных причин, почему программисты должны понимать, как работает система компиляции.

- *Оптимизация производительности программы.* Современные компиляторы – это сложные инструменты, которые обычно производят эффективный программный код. Однако мы, программисты, должны хотя бы немного понимать машинный код и знать, как компилятор транслирует разные инструкции на С, чтобы принимать осознанные решения при разработке своих программ. Например, всегда ли оператор `switch` является более эффективным, чем некоторая последовательность операторов `if-then-else`? Какие накладные расходы несет вызов функции? Является ли оператор `while` более эффективным, чем оператор `for`? Какой способ обращения к элементам массива эффективнее – по указателю или по индексам? Почему наш цикл выполняется намного быстрее, если накапливать сумму в локальной переменной вместо аргумента, который передается по ссылке? Можно ли ускорить функцию, если просто расставить круглые скобки в арифметическом выражении?

В главе 3 мы представим машинный язык x86-64 последних поколений компьютеров с Linux, Macintosh и Windows. Там мы расскажем, как компиляторы транслируют различные конструкции языка С в этот язык. В главе 5 вы узнаете, как оптимизировать производительность своих программ, выполняя простые преобразования в коде на С, которые помогают компилятору лучше справляться со своей задачей. А в главе 6 вы будете изучать иерархическую природу системы памяти, узнаете, как компиляторы языка С хранят массивы данных и как можно использовать эти знания, чтобы ускорить работу программ на С.

- *Понимание ошибок времени компоновки.* Как показывает наш опыт, некоторые из наиболее запутанных программных ошибок порождаются компоновщиком, особенно при создании больших программных систем. Например, что означает сообщение компоновщика о том, что он не может разрешить ссылку? Чем отличаются статические и глобальные переменные? Что случится, если в разных файлах на С объявить две глобальные переменные с одинаковыми именами? Чем отличаются статические и динамические библиотеки? Почему важен порядок перечисления библиотек в командной строке? И самое неприятное: почему ошибки, источником которых является компоновщик, остаются незаметными до выполнения программы? Ответы на все эти вопросы вы получите в главе 7.
- *Как избежать прорех в системе защиты.* В течение многих лет уязвимость, порождаемая переполнением буфера, была причиной большей части проблем в сетевых серверах. Такие уязвимости существуют в силу того обстоятельства, что слишком мало программистов понимают необходимость тщательно ограничивать объемы и формы данных, которые поступают из ненадежных источников. Первым шагом в освоении приемов безопасного программирования является изучение особенностей хранения в программном стеке данных и управляющей информации. Мы расскажем, как правильно использовать стек и избежать уязвимостей, порождаемых переполнением буферов, в главе 3, когда приступим к изучению языка ассемблера. Там же мы познакомимся с методами, которые могут использовать программисты, компиляторы и операционные системы для снижения угрозы атаки.

## 1.4. Процессоры читают и интерпретируют инструкции, хранящиеся в памяти

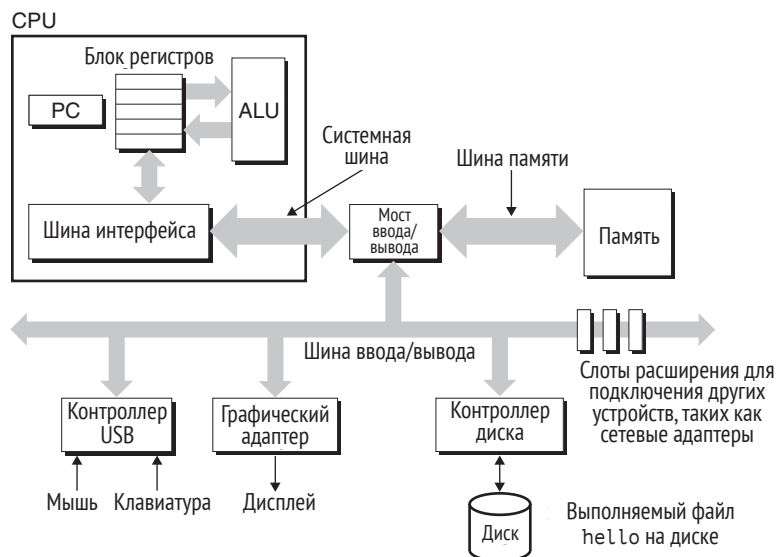
Итак, наша исходная программа `hello.c` прошла через систему компиляции, которая преобразовала ее в выполняемый объектный файл с именем `hello` и сохранила на диске. Чтобы запустить выполняемый файл в системе Unix, нужно ввести его имя с клавиатуры в другой программе, известной как *командная оболочка*:

```
linux>./hello
hello, world
linux>
```

Командная оболочка – это интерпретатор командной строки, который выводит на экран приглашение к вводу, ждет, когда вы введете с клавиатуры команду, а затем выполняет ее. Если первое слово в команде не является именем какой-либо встроенной команды, то оболочка предполагает, что это слово является именем выполняемого файла, который она должна загрузить и выполнить. Поэтому в данном случае оболочка загружает и запускает программу `hello`, после чего ждет, когда та завершится. Программа `hello` выводит свое сообщение на экран и завершается, после чего оболочка выводит приглашение к вводу следующей команды и ждет, когда будет введена новая команда.

### 1.4.1. Аппаратная организация системы

Чтобы понять, что происходит с программой `hello` во время выполнения, мы должны иметь представление о том, как устроена аппаратная часть типичной вычислительной системы, блок-схема которой показана на рис. 1.2. Эта конкретная блок-схема отражает устройство современных систем Intel, однако примерно такое же устройство имеют все вычислительные системы. Пусть вас сейчас не беспокоит сложность этой блок-схемы – мы будем рассматривать различные ее детали на протяжении всей книги.



**Рис. 1.2.** Аппаратная организация типичной вычислительной системы.

CPU: центральное процессорное устройство или просто процессор,  
 ALU: арифметико-логическое устройство, PC (program counter): счетчик команд,  
 USB (Universal Serial Bus): универсальная последовательная шина



## Шины

Вычислительную систему пронизывает совокупность электрических проводников, так называемых *шин*, по которым байты информации циркулируют между компонентами системы. Обычно шины конструируются таким образом, чтобы по ним можно было передавать байты порциями фиксированного размера, которые называют *словами*. Число байтов в слове (*размер слова*) является одним из фундаментальных параметров, которые изменяются от системы к системе. В большинстве современных систем используются слова с размером 4 байта (32 бита) или 8 байт (64 бита). В этой книге мы будем использовать понятие «слово» без определения конкретного размера и уточнять его в случаях, когда это необходимо.

## Устройства ввода/вывода

*Устройства ввода/вывода* – это средства связи с внешним миром. В нашем примере системы имеется четыре устройства ввода/вывода: клавиатура и мышь для ввода данных пользователем, устройство для отображения данных пользователю и дисковое устройство (или просто диск) для долгосрочного хранения данных и программ. В начальный момент выполняемый файл хранится на диске.

Каждое устройство ввода/вывода подключено к шине ввода/вывода посредством *контроллера* или *адаптера*. Различие между ними заключается в их конструктивных особенностях. Контроллеры – это платы с наборами микросхем, установленные в самом устройстве или на главной печатной плате (ее еще часто называют *материнской платой*). Адаптер – это плата, которая подключается через контактное гнездо на материнской плате. Независимо от конструкции таких устройств, их назначение заключается в том, чтобы передавать информацию между шиной и устройством ввода/вывода в обоих направлениях.

В главе 6 более подробно описана работа таких устройств ввода/вывода, как диск. В главе 10 вы узнаете, как пользоваться интерфейсом ввода/вывода системы Unix для доступа к устройствам из вашей прикладной программы. Мы же сосредоточим основное внимание на особо интересном классе устройств – сетевых адаптерах, принцип работы с которыми, впрочем, легко обобщить на любые другие устройства.

## Основная память

*Основная память* – временное хранилище, в котором находятся сама программа и данные, которыми она манипулирует во время выполнения. Физически основная память состоит из совокупности микросхем *динамической памяти с произвольным доступом* (Dynamic Random Access Memory, DRAM). Логически основная память организована в виде линейного массива байтов, каждый из которых имеет свой уникальный адрес (индекс элемента массива); отсчет адресов начинается с нуля. Машинные инструкции, составляющие программу, могут состоять из разного числа байтов. Размер элементов данных, соответствующих переменным в программах на C, зависит от их типов. Например, на машине x86\_64, работающей под Linux, тип данных *short* требует двух байт, типы *int* и *float* – четырех байт, а типы *long* и *double* – восьми байт.

В главе 6 мы более подробно рассмотрим, как работают технологии памяти, такие как DRAM, и как из них конструируется основная память.

## Процессор

*Центральный процессор* (ЦП; Central Processing Unit, CPU), или просто процессор, – это механизм, который интерпретирует (или *выполняет*) инструкции, хранящиеся в основной памяти. Его ядро составляет устройство памяти с емкостью в одно слово (или *регистр*) – *счетчик команд* (Program Counter, PC). В любой конкретный момент времени он хранит адрес некоторой машинной инструкции в основной памяти<sup>2</sup>.

<sup>2</sup> Аббревиатура PC также часто расшифровывается как Personal Computer (персональный компьютер). Различие между понятиями должно быть очевидно из контекста.



С момента включения системы и до ее выключения процессор снова и снова выполняет инструкцию, на которую указывает счетчик команд, и затем обновляет значение счетчика, выполняя переход к следующей инструкции. Кажется, что процессор работает в соответствии с очень простой моделью выполнения инструкций, определяемой его *архитектурным набором команд*. Согласно этой модели инструкции выполняются в строгой последовательности, а выполнение одной инструкции происходит в несколько этапов. Сначала процессор читает инструкцию из памяти по адресу в счетчике команд (PC), интерпретирует биты инструкции, выполняет простые операции, определяемые инструкцией, а затем обновляет значение счетчика, записывая в него адрес следующей инструкции, которая может размещаться за текущей или где-то в другом месте в памяти.

Существует всего несколько таких простых операций, и все они связаны с обслуживанием основной памяти, блока *регистров* и *арифметико-логического устройства* (Arithmetic/Logic Unit, ALU). Блок регистров – небольшое запоминающее устройство из совокупности регистров, каждый из которых имеет свое уникальное имя и может хранить одно слово. Устройство ALU вычисляет новые значения данных и адресов. Назовем лишь несколько примеров простых операций, которые процессор может выполнить по требованию той или иной инструкции:

- *загрузка*: копирует байт или слово из основной памяти в регистр, затирая при этом предыдущее содержимое этого регистра;
- *сохранение*: копирует байт или слово из регистра в некоторую ячейку основной памяти, затирая при этом предыдущее содержимое этой ячейки;
- *выполнение арифметико-логической операции*: копирует содержимое двух регистров в ALU, выполняет соответствующую операцию с этими словами и запоминает результат в одном из регистров, затирая при этом предыдущее содержимое этого регистра;
- *переход*: извлекает слово из самой инструкции и копирует его в счетчик команд (PC), затирая предыдущее его содержимое.

Мы говорим «кажется, что процессор работает в соответствии с очень простой моделью, определяемой его архитектурным набором», но на самом деле механика работы современных процессоров намного сложнее, чем было описано выше. Поэтому мы будем различать архитектурный набор команд процессора, определяющий эффект каждой машинной инструкции, и его микроархитектуру, определяющую фактическую реализацию процессора. В процессе знакомства с машинным кодом в главе 3 мы рассмотрим абстракцию архитектурного набора. В главе 4 вы узнаете больше о том, как в действительности устроены процессоры. В главе 5 мы опишем модель работы процессора с поддержкой предсказаний и оптимизации производительности программ на машинном языке.

### 1.4.2. Выполнение программы hello

Ознакомившись с простым описанием аппаратной организации и операций, мы начинаем понимать, что происходит, когда мы запускаем наш пример программы. Здесь мы должны опустить множество деталей, которые учтем позже, а пока нас вполне удовлетворяет общая картина.

Сначала свои инструкции выполнит командная оболочка, ожидающая, когда мы введем команду с клавиатуры. Как только мы введем символы `./hello`, командная оболочка прочитает каждый из них в регистр и сохранит в основной памяти, как показано на рис. 1.3.

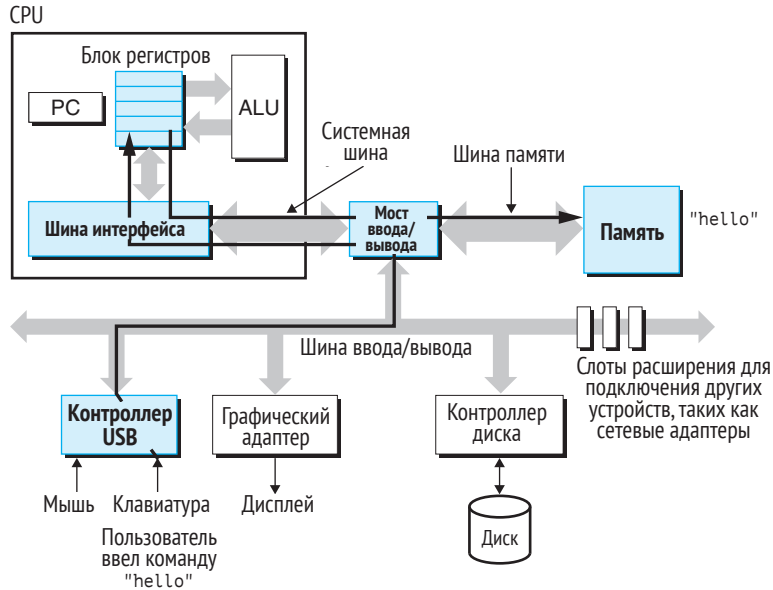


Рис. 1.3. Чтение команды hello с клавиатуры

Когда мы нажмем клавишу **Enter**, оболочка воспримет это как сигнал окончания ввода команды. После этого она загрузит выполняемый файл hello, осуществив последовательность инструкций, которая скопирует в основную память программные коды и данные, содержащиеся в объектном файле hello на диске. Данные включают строку символов "hello, world\n", которая в конечном итоге будет выведена на экран.

Используя метод, известный как *прямой доступ к памяти* (Direct Memory Access, DMA; см. главу 6), данные перемещаются с диска непосредственно в оперативную память, минуя процессор. Этот шаг показан на рис. 1.4.

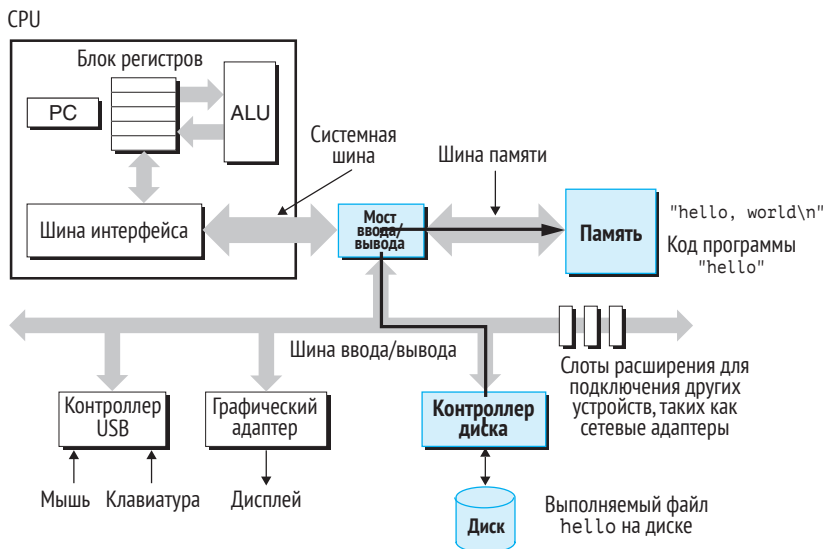


Рис. 1.4. Загрузка выполняемого файла в основную память

Как только код и данные из объектного файла `hello` будут загружены в память, процессор начинает выполнять машинные инструкции подпрограммы `main` в программе `hello`. Эти инструкции копируют байты строки `"hello, world\n"` из основной памяти в регистры, а оттуда – на дисплей, на экране которого они затем отображаются. Эти этапы показаны на рис. 1.5.

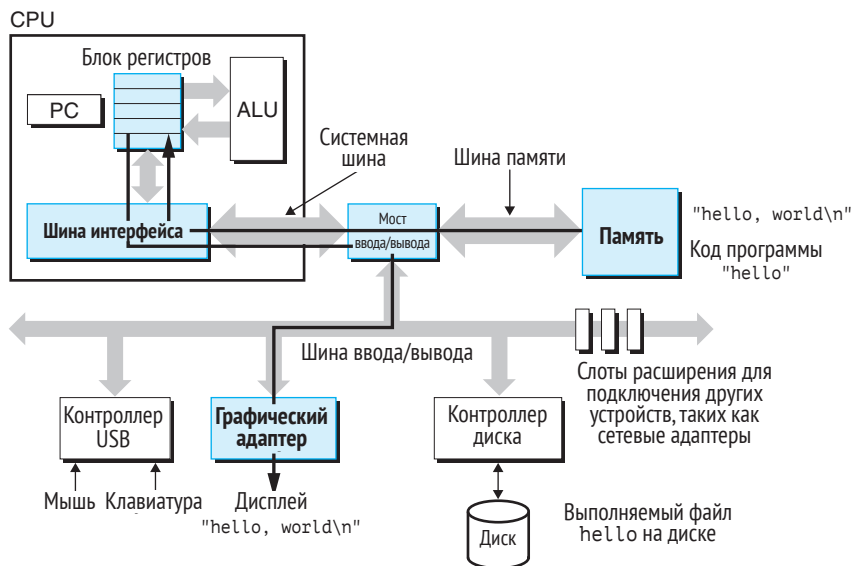


Рис. 1.5. Вывод строки из основной памяти на экран дисплея

## 1.5. Различные виды кеш-памяти

Важный урок из этого простого примера заключается в том, что система затрачивает уйму времени на перемещение информации из одного места в другое. Машинные инструкции в программе `hello` в начальный момент хранятся на диске. Когда производится загрузка программы, они копируются в основную память. По мере выполнения программы ее инструкции копируются из основной памяти в процессор. Аналогично строка данных `"hello, world\n"`, которая первоначально хранилась на диске, копируется в основную память, а затем из основной памяти на устройство отображения. С точки зрения программиста такой большой объем копирования ложится тяжким бременем на систему и существенно снижает «истинную производительность» программы. Следовательно, главная цель системных проектировщиков заключается в том, чтобы максимально ускорить операции копирования данных.

В силу физических законов чем больше запоминающее устройство, тем медленнее оно работает. И в то же время создание быстродействующих запоминающих устройств обходится дороже, чем более медленных. Например, емкость диска может оказаться в 1000 раз больше объема оперативной памяти, но, чтобы прочесть слово с диска, требуется в 10 млн раз больше времени, чем из основной памяти.

Аналогично типичный блок регистров может хранить лишь несколько сотен байтов информации, в то время как основная память – миллиарды. Однако процессор способен читать данные из регистров примерно в 100 раз быстрее, чем из основной памяти. Более того, по мере развития полупроводниковых технологий *расхождение в скорости доступа к регистрам и к основной памяти* продолжают углубляться. Увеличить быстродействие процессора намного проще, чем заставить основную память работать быстрее.

Чтобы уменьшить разрыв между процессором и основной памятью, создатели процессоров включили в них небольшие быстродействующие устройства хранения, получившие название *кеш-память* (или просто кеш) и служащие для временного хранения информации, которая, возможно, потребуется процессору в ближайшем будущем. На рис. 1.6 показана типичная система с кеш-памятью. *Кеш L1* (его еще называют кешем первого уровня) внутри процессора может хранить десятки тысяч байт, и доступ к ним осуществляется почти так же быстро, как к регистрам. Еще больший объем имеет кеш L2 (кеш второго уровня) – он может вместить от сотен тысяч до миллионов байт. Этот кеш связан с процессором специальной шиной. Скорость доступа к кешу L2 в 5 раз ниже скорости доступа к кешу L1, и все равно она в 5–10 раз выше скорости доступа к основной памяти. Кешы L1 и L2 построены по технологии, известной как *статическая память с произвольным доступом* (Static Random Access Memory, SRAM). Более новые и более мощные системы имеют три уровня кешей: L1, L2 и L3. Идея кеширования состоит в том, чтобы дать системе возможность получить положительный эффект от наличия большого объема памяти и очень короткого времени доступа за счет улучшения *локальности* – тенденции программ использовать данные и код, находящиеся в ограниченных областях памяти. Использование кешей для хранения данных, к которым программа, вероятно, будет часто обращаться, позволяет выполнять большинство операций с памятью, используя быстрые кешы.

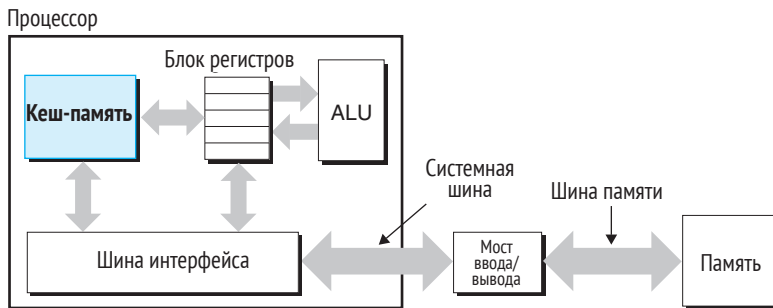


Рис. 1.6. Различные виды кеш-памяти

Один из самых важных уроков этой книги заключается в том, что прикладные программисты, которые знают о наличии кеш-памяти, могут воспользоваться ею и увеличить производительность своих программ на порядок. Мы будем изучать эти важные устройства и узнаем, как ими пользоваться, в главе 6.

## 1.6. Устройства памяти образуют иерархию

Идея поместить небольшое, зато более быстрое запоминающее устройство (кеш-память) между процессором и более емким, но с худшим быстродействием запоминающим устройством (основной памятью) оказалась весьма плодотворной. Фактически запоминающие устройства в любой вычислительной системе образуют *иерархию*, подобную изображенной на рис. 1.7. По мере движения по этой иерархии сверху вниз устройства становятся все медленнее, объемнее, а стоимость хранения одного байта уменьшается. Регистры находятся на вершине иерархии, которая обозначается как уровень 0 (L0). Кеш-память занимает уровни с 1 по 3 (L1–L3). Основная память находится на уровне 4 (L4) и т. д.

Основная идея иерархии памяти заключается в том, что память одного уровня служит кешем для следующего нижнего уровня. То есть блок регистров – это кеш для кеш-памяти L1. Кешы L1 и L2 служат кешами для кеш-памяти L2 и L3 соответственно.

Кеш L3 служит кешем для основной памяти, а та, в свою очередь, – кешем для диска. В некоторых сетевых системах с распределенными файловыми системами локальный диск служит кешем для данных, хранящихся на дисках других систем.



Рис. 1.7. Пример иерархии памяти

Подобно тому, как программисты используют знание структуры кешей L1 и L2 для повышения производительности своих программ, можно использовать структуру всей иерархии памяти. Более подробно эти вопросы будут рассмотрены в главе 6.

## 1.7. Операционная система управляет работой аппаратных средств

Вернемся к нашей программе `hello`. Когда оболочка загружала и запускала программу `hello` и когда программа `hello` отображала свое сообщение, ни та ни другая программа не обращалась напрямую к клавиатуре, диску или основной памяти. Для этого они пользовались услугами, предоставляемыми операционной системой. Операционную систему можно представить как некоторый слой программного обеспечения между прикладной программой и аппаратными средствами, как показано на рис. 1.8. Любые операции с аппаратными средствами прикладные программы должны выполнять через операционную систему.

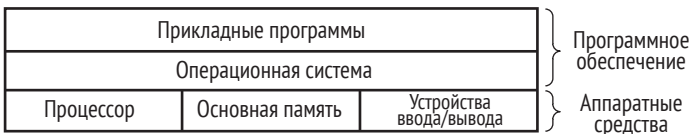


Рис. 1.8. Многослойная организация компьютерной системы

Операционная система прежде всего должна отвечать двум основным требованиям: (1) защищать аппаратные средства от катастрофических действий вышедшей из-под контроля программы и (2) предоставлять приложениям простые и единообраз-

ные механизмы манипулирования сложными и часто сильно отличающимися низкоразрядными аппаратными средствами. Для этого операционная система предлагает набор фундаментальных абстракций, показанных на рис. 1.8: *процессы*, *виртуальную память* и *файлы*. Как видно по рис. 1.9, файлы являются абстракцией устройств ввода/вывода, виртуальная память является абстракцией как основной памяти, так и дисковых устройств ввода/вывода, а процессы – абстракцией процессора, основной памяти и устройств ввода/вывода.

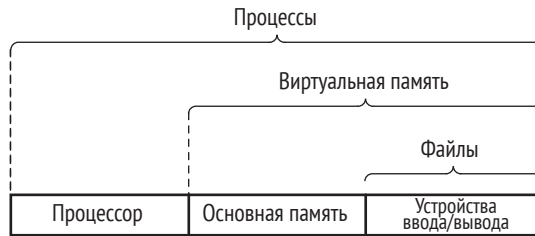


Рис. 1.9. Абстракции, предоставляемые операционной системой

### 1.7.1. Процессы

Когда программа, такая как `hello`, запускается в современной системе, операционная система создает иллюзию, что она – единственная программа, выполняющаяся в системе. С точки зрения программы создается впечатление, что только она распоряжается процессором, основной памятью и устройствами ввода/вывода. Процессор как бы выполняет инструкции программы подряд, одну за другой, без прерываний, и только код программы и ее данные являются единственными объектами, пребывающими в памяти системы. Источником таких иллюзий является понятие процесса, одной из наиболее важных и успешных идей в теории вычислительных машин и систем.

*Процесс* – это абстракция операционной системы, представляющая выполняемую программу. В одной и той же системе одновременно может выполняться множество процессов, и в то же время каждому процессу кажется, что только он пользуется аппаратными средствами. Под *одновременным* (или *параллельным*) выполнением мы понимаем поочередное выполнение инструкций то одного, то другого процесса. В большинстве систем существует больше процессов, выполняющихся одновременно, чем процессоров, на которых они выполняются.

Традиционные системы могут выполнять только одну программу в каждый конкретный момент времени, тогда как новые *многоядерные* процессоры способны одновременно выполнять программный код нескольких программ. В любом случае может показаться, что один процессор реализует несколько процессов одновременно, если будет переключаться между ними очень быстро. Операционная система проделывает такое чередование с помощью механизма *переключения контекста*. Чтобы упростить остальную часть этого обсуждения, мы будем рассматривать только *однопроцессорную систему* с единственным процессором. К обсуждению *многопроцессорных* систем мы вернемся в разделе 1.9.2.

Операционная система хранит всю информацию о состоянии, необходимую для правильного выполнения процесса. Состояние, известное как *контекст*, содержит такие сведения, как текущее значение счетчика команд РС, состояние блока регистров и содержимое основной памяти. В любой конкретный момент времени в системе выполняется только один процесс. Когда операционная система принимает решение передать управление некоторому другому процессу, она производит *переключение контекста*, запоминая контекст текущего процесса и восстанавливая контекст нового, после чего передает управление новому процессу. Новый процесс возобновляет выполнение точ-

но с того места, в котором он был прерван. Эту идею иллюстрирует рис. 1.10 на примере нашей программы `hello`.

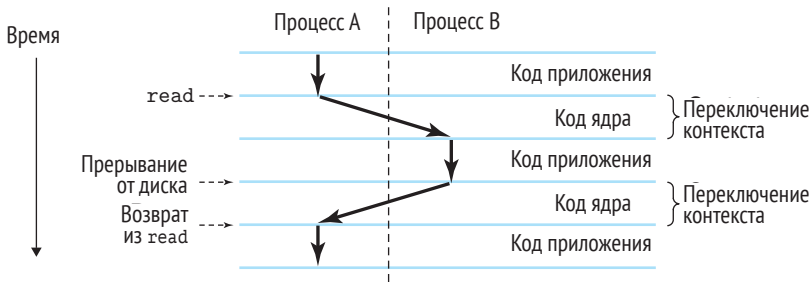


Рис. 1.10. Переключение контекстов процессов

В рассматриваемом нами примере существует два процесса: процесс командной оболочки и процесс `hello`. Первоначально система выполняет только один процесс, а именно процесс командной оболочки, которая ожидает ввода команды. Когда мы обращаемся к нему с требованием запустить программу `hello`, оболочка выполняет наше требование, вызывая специальную функцию, так называемый *системный вызов*, который передает управление операционной системе. Операционная система сохраняет контекст процесса оболочки, создает новый процесс `hello` с его контекстом, а затем передает управление новому процессу `hello`. После завершения `hello` операционная система восстанавливает контекст процесса оболочки и возвращает ему управление, а он затем ждет ввода следующей команды.

Как показано на рис. 1.10, переключение с одного процесса на другой производится *ядром* операционной системы. Ядро – это часть кода операционной системы, которая всегда находится в памяти. Когда прикладная программа требует от операционной системы какого-либо действия, например чтения из файла или записи в файл, она выполняет специальную инструкцию *системного вызова*, передавая управление ядру. Затем ядро выполняет запрошенную операцию и возвращает управление прикладной программе. Обратите внимание, что ядро – это не отдельный процесс, а блок кода и структур данных, которые система использует для управления всеми процессами.

Реализация абстракции процесса требует тесного взаимодействия аппаратных и программных средств операционной системы. В главе 8 мы выясним, как это делается, а также как прикладные программы могут создавать свои собственные процессы и управлять ими.

## 1.7.2. Потoki

Обычно мы представляем себе процесс как единственный поток управления, однако в современных системах процесс может состоять из множества единиц выполнения, которые называют *потоками выполнения* (или нитями), каждый поток выполняется в контексте процесса, использует тот же программный код и глобальные данные, что и сам процесс. Роль потоков как программных моделей непрерывно возрастает в связи с требованиями к поддержке параллельных вычислений, предъявляемыми сетевыми серверами, поскольку организовать совместное использование данных несколькими потоками намного проще, чем несколькими процессами, а также в силу того, что потоки обычно значительно эффективнее процессов. Кроме того, многопоточность является одним из способов ускорить работу программ, когда в системе имеется несколько процессоров, о чем будет рассказываться в разделе 1.9.2. С базовыми понятиями параллельных вычислений, включая создание многопоточных программ, вы познакомитесь в главе 12.



### Unix, Posix и Standard Unix Specification

В шестидесятые годы прошлого столетия господствовали большие и сложные операционные системы, такие как OS/360, разработанная компанией IBM, и Multics, разработанная компанией Honeywell. И если OS/360 была одной из наиболее успешных операционных систем того периода, то Multics вличила жалкое существование в течение многих лет и не смогла добиться широкого признания. Компания Bell Laboratories первоначально была одним из партнеров, разрабатывавших проект Multics, но в 1969 году отказалась от участия в этом проекте по причине его чрезмерной сложности и ввиду отсутствия положительных результатов. Полученный при разработке системы отрицательный опыт сподвиг группу исследователей компании – Кена Томпсона (Ken Thompson), Денниса Ритчи (Dennis Ritchie), Дуга Макилроя (Doug McIlroy) и Джо Оссанну (Joe Ossanna) – приступить в 1969 году к работе над более простой операционной системой для компьютера PDP-7 компании DEC, написанной исключительно на машинном языке. Многие из идей новой системы, такие как иерархическая файловая система и понятие командной оболочки как процесса пользовательского уровня, были заимствованы из системы Multics, но реализованы в виде более простого и компактного пакета программ. В 1970 году Брайан Керниган (Brian Kernighan) выбрал для новой системы название «Unix» как противовес названию «Multics», тем самым подчеркнув неповоротливость и тяжеловесность системы Multics (в некотором приближении Multics можно перевести как «многогранный», в том же контексте Unix можно перевести как «одногранный». – *Прим. перев.*). Ядро Unix было переписано на языке C в 1973 году, а сама операционная система была представлена широкой публике в 1974 году [93].

Поскольку компания Bell Labs предоставила высшим учебным заведениям исходные коды на очень выгодных условиях, у операционной системы Unix появилось множество сторонников среди студентов и преподавателей различных университетов. Работа, оказавшая большое влияние на дальнейшее развитие, была выполнена в Калифорнийском университете Беркли в конце семидесятых и в начале восьмидесятых годов, когда исследователи из Беркли добавили виртуальную память и сетевые протоколы в последовательность выпусков, получивших название Unix 4.xBSD (Berkeley Software Distribution). Одновременно компания Bell Labs наладила выпуск своих собственных версий Unix, которые стали известны как System V Unix. Версии других поставщиков программного обеспечения, таких как система Sun Microsystems Solaris, были построены на базе оригинальных версий BSD и System V.

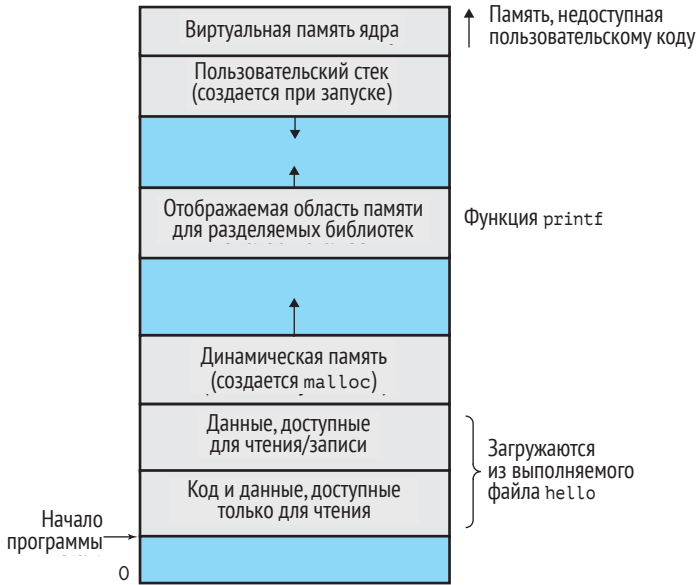
Осложнения возникли в середине восьмидесятых годов, когда производители операционной системы предприняли попытку выбрать собственные направления в развитии, добавляя новые свойства, часто нарушающие совместимость с прежними версиями. Чтобы пресечь эти сепаратистские тенденции, институт стандартизации IEEE (Institute for Electrical and Electronics Engineers – Институт инженеров по электротехнике и электронике) возглавил усилия по стандартизации системы Unix. Позже Ричард Столлман (Richard Stallman) окрестил продукт этих усилий как «Posix». В результате было получено семейство стандартов, известное как стандарты Posix, которые решали такие проблемы, как интерфейс языка C для системных вызовов в Unix, командные оболочки и утилиты, потоки и сетевое программирование. Запущенный вслед за этим отдельный проект по стандартизации, известный как стандартная спецификация Unix (Standard Unix Specification, SUS), объединил свои усилия с Posix для создания единого унифицированного стандарта систем Unix. В результате этого сотрудничества различия между разными версиями Unix почти исчезли.

### 1.7.3. Виртуальная память

*Виртуальная память* – это абстракция, которая порождает в каждом процессе иллюзию, что лишь он один использует основную память. Каждый процесс имеет одно и то же представление о памяти, которое известно как его *виртуальное адресное прост-*



ранство. Виртуальное адресное пространство для процессов операционной системы Linux показано на рис. 1.11. (Другие Unix-системы используют ту же топологию.) В системе Linux верхняя часть адресного пространства резервируется для программного кода и данных операционной системы, которые являются общими для всех процессов. В нижней части адресного пространства находятся программный код и данные, принадлежащие процессам пользователей. Обратите внимание на тот факт, что адреса на схеме увеличиваются снизу вверх.



**Рис. 1.11.** Виртуальное адресное пространство процесса (области нарисованы без соблюдения масштаба)

Виртуальное адресное пространство, с точки зрения каждого процесса, состоит из некоторого числа четко определенных областей, каждая из которых выполняет свою функцию. Эти области будут подробно описаны далее в книге, тем не менее давайте кратко рассмотрим каждую из них прямо сейчас, начав с меньших адресов и продвигаясь в сторону их увеличения.

- *Программный код и данные.* Программный код каждого процесса всегда начинается с одного и того же адреса, за ним следуют ячейки памяти, соответствующие глобальным переменным. Область с кодом и данными инициализируется непосредственно содержимым выполняемого объектного файла, в нашем случае `hello`. Более подробно эта часть адресного пространства будет описана в главе 7, где мы исследуем связывание и загрузку.
- *Динамическая память (куча).* Непосредственно за кодом и данными следует область динамической памяти (кучи) программы. В отличие от областей с программным кодом и глобальными данными, размеры которых фиксируются, как только процесс начнет выполняться, динамическая память может расширяться и сокращаться в размерах во время выполнения программы, при вызове некоторых функций из стандартной библиотеки C, таких как `malloc` и `free`. Мы продолжим подробное изучение динамической памяти после того, как в главе 9 рассмотрим вопросы управления виртуальной памятью.

- *Совместно используемые (разделяемые) библиотеки.* Ближе к середине адресного пространства находится область с программным кодом и данными *совместно используемых библиотек*, таких как стандартная библиотека C или библиотека математических функций. Понятие совместно используемой библиотеки является мощным, но в то же время несколько трудным для понимания. Вы узнаете, как с ними работать, когда мы приступим к изучению динамического связывания в главе 7.
- *Стек.* В верхней части виртуального адресного пространства находится *стек пользователя*, который применяется компилятором для реализации вызовов функций. Как и динамическая память, стек пользователя может динамически расширяться и сокращаться в размерах во время выполнения программы. В частности, каждый раз, когда программа вызывает какую-либо функцию, размер стека увеличивается. Каждый раз, когда функция возвращает управление, стек сокращается. В главе 3 вы узнаете, как компилятор использует стек.
- *Виртуальная память ядра.* Верхняя часть адресного пространства зарезервирована для ядра. Прикладным программам запрещено читать содержимое этой области, записывать в нее данные или напрямую вызывать функции ядра.

Для правильной работы виртуальной памяти требуется сложное взаимодействие аппаратных и программных средств операционной системы, включая аппаратное преобразование каждого адреса, генерируемого процессором. Основная идея заключается в том, чтобы сохранить содержимое виртуальной памяти процесса на диске, а затем использовать основную память как кеш диска. В главе 9 мы покажем вам, как работает этот механизм и почему это так важно для функционирования современных систем.

### 1.7.4. Файлы

*Файл* – это всего лишь последовательность байтов, не более и не менее. Каждое устройство ввода/вывода, в том числе диски, клавиатуры, устройства отображения и даже сети, моделируется соответствующим файлом. Все операции ввода/вывода в системе выполняются путем чтения и записи в файлы посредством нескольких системных вызовов, образующих *подсистему ввода/вывода Unix*.

Это простое и элегантное понятие файла, тем не менее, обладает глубоким смыслом, поскольку обеспечивает унифицированное представление всего разнообразия файлов, которые могут входить в состав системы. Например, прикладные программисты, манипулирующие содержимым дискового файла, могут быть абсолютно не знакомы с дисковыми технологиями и при этом чувствовать себя вполне комфортно. Более того, одна и та же программа будет прекрасно выполняться в разных системах, использующих разные дисковые технологии. Подсистема ввода/вывода Unix будет рассматриваться в главе 10.

## 1.8. Обмен данными в сетях

До этого момента в нашем экскурсе мы рассматривали систему как изолированную совокупность аппаратных и программных средств. На практике современные системы часто соединены с другими системами посредством компьютерных сетей. С точки зрения отдельной системы, сеть можно рассматривать как еще одно устройство ввода/вывода, как показано на рис. 1.12. Когда система копирует некоторую последовательность байтов из основной памяти в сетевой адаптер, поток данных устремляется через сеть в другую машину, а не, скажем, в локальный дисковый накопитель. Аналогично система может читать данные, отправленные другими машинами, и сохранять в своей основной памяти.

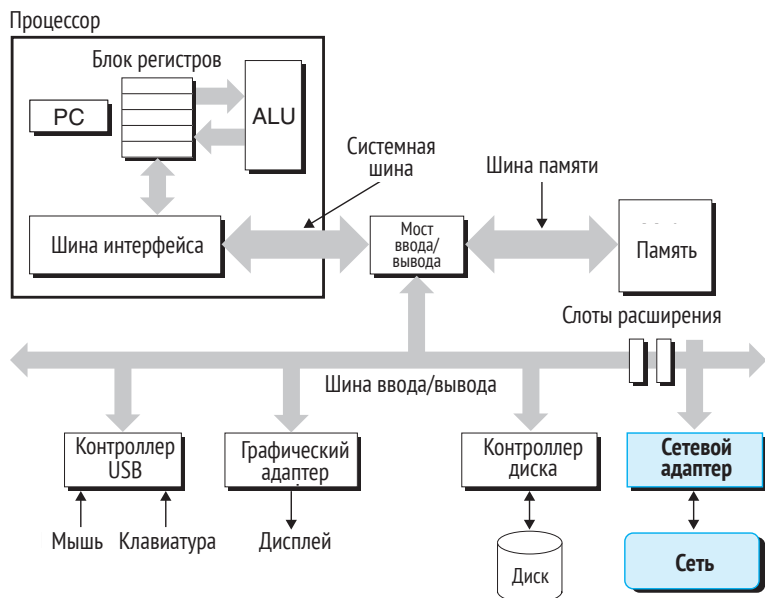


Рис. 1.12. Сеть – это еще одно устройство ввода/вывода

С пришествием глобальных сетей, таких как интернет, копирование информации между машинами стало одним из наиболее важных применений компьютерных систем. Например, такие приложения, как электронная почта, обмен мгновенными сообщениями, Всемирная паутина, FTP (File Transfer Protocol – протокол передачи файлов) и telnet, основаны на копировании информации по сети.

Возвращаясь к нашему примеру hello, мы можем воспользоваться знакомым приложением telnet, чтобы запустить программу hello на удаленной машине. Предположим, что мы решили воспользоваться *клиентом* на нашей машине для подключения к *telnet-серверу* на удаленной машине. После регистрации на удаленной машине запустится удаленная командная оболочка, которая будет ждать от нас ввода команд. С этого момента процесс дистанционного запуска программы hello требует выполнения пяти простых действий, представленных на рис. 1.13.

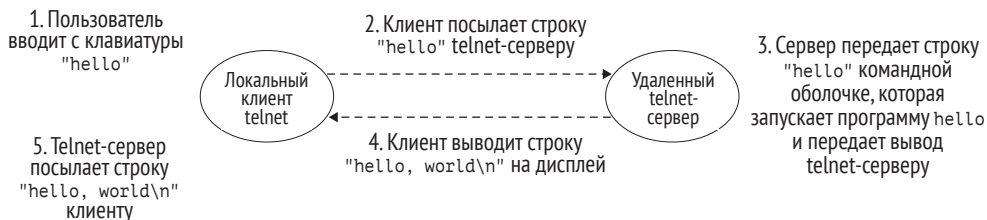


Рис. 1.13. Использование telnet для запуска программы hello на удаленной машине

После ввода строки hello на стороне клиента и нажатия клавиши **Enter** клиент отправит эту строку telnet-серверу. Когда сервер получит эту строку из сети, он передаст ее своей командной оболочке. Затем удаленная командная оболочка запустит программу hello и передаст выходную строку telnet-серверу. Наконец, telnet-сервер перешлет полученную строку клиенту по сети, а тот отобразит ее на локальном дисплее.

Такой тип обмена между клиентами и серверами характерен для всех сетевых приложений. В главе 11 вы узнаете, как создавать сетевые приложения, и примените полученные знания для создания простого веб-сервера.

### Проект Linux

В августе 1991 года финский аспирант по имени Линус Торвальдс (Linus Torvalds) скромно объявил о завершении разработки ядра новой Unix-подобной операционной системы:

От: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)

Сетевая телеконференция: comp.os.minix

Тема: Что бы вы хотели прежде всего видеть в minix?

Резюме: ограниченный опрос, касающийся моей новой операционной системы

Дата: 25 августа 91 20:57:08 по Гринвичу

Вниманию всех, кто пользуется системой minix:

Я разрабатываю (бесплатную) операционную систему (это всего лишь хобби, система небольшая и спроектирована непрофессионально, в отличие от GNU) для персональных компьютеров AT 386/486. Проект вызревал с апреля, сейчас он приобретает законченный вид. Я хотел бы узнать мнение людей, работающих с minix (одобряют или не одобряют мою систему), поскольку моя операционная система в какой-то степени напоминает minix (то же физическое размещение файловой системы, в силу практических причин, наряду с другими общими чертами).

В настоящий момент я перенес программы bash(1.08) и gcc(1.40), и, как ни странно, они работают. Это означает, что через несколько месяцев мне удастся получить кое-что полезное, и мне хотелось бы знать, что бы хотело видеть большинство в моем программном продукте. Благодарен за любые предложения, в то же время я не обещаю, что все выполню.

Linus (torvalds@kruuna.helsinki.fi)

Как указывает Торвальдс, отправной точкой для создания Linux стала операционная система Minix, разработанная Эндрю С. Таненбаумом (Andrew S. Tanenbaum) в образовательных целях [113].

Все остальное, как говорится, уже история. Операционная система Linux превратилась в техническое и культурное явление. Объединившись с проектом GNU, проект Linux позволил получить полную, совместимую со стандартами Posix версию операционной системы Unix, включая ядро и всю поддерживающую его инфраструктуру. Система Linux успешно работает на самых разных компьютерах, от карманных до мэйнфреймов. Группа разработчиков компании IBM умудрилась впихнуть ее даже в наручные часы!

## 1.9. Важные темы

На этом мы завершаем наш начальный экскурс в компьютерные системы. Важная причина отказаться от дальнейшего обсуждения заключается в том, что система – это нечто большее, чем только аппаратные средства. Это скорее переплетение аппаратных средств и системного программного обеспечения, которые должны действовать сообща для достижения окончательной цели – выполнения прикладных программ. Остальная часть книги представит некоторые дополнительные подробности об аппаратном и программном обеспечении и покажет, как, зная эти подробности, можно писать более быстрые, надежные и безопасные программы.

В заключение этой главы выделим несколько важных концепций, которые затрагивают все аспекты компьютерных систем. Мы будем отмечать их важность на протяжении всей книги.

### 1.9.1. Закон Амдала

Джин Амдал (Gene Amdahl), один из пионеров компьютерных вычислений, сделал простое, но далеко идущее наблюдение об эффективности повышения производительности одной части системы, которое стало известно как *закон Амдала*. Согласно этому закону, когда мы ускоряем одну часть системы, прирост общей производительности системы зависит не только от величины ускорения этой части системы, но и насколько значимой она является. Рассмотрим систему, в которой для выполнения некоторого приложения требуется время  $T_{\text{old}}$ . Предположим, что некоторая часть системы потребляет долю  $\alpha$  этого времени, и мы повысили ее производительность в  $k$  раз. То есть компоненту изначально требовалось время  $\alpha T_{\text{old}}$ , а теперь  $(\alpha T_{\text{old}})/k$ . Таким образом, общее время выполнения будет

$$\begin{aligned} T_{\text{new}} &= (1 - \alpha)T_{\text{old}} + (\alpha T_{\text{old}})/k \\ &= T_{\text{old}}[(1 - \alpha) + \alpha/k]. \end{aligned}$$

Отсюда прирост скорости  $S = T_{\text{old}}/T_{\text{new}}$  можно вычислить как

$$S = \frac{1}{(1 - \alpha) + \alpha/k}. \quad (1.1)$$

Рассмотрим пример, когда часть системы, которая изначально потребляла 60 % времени ( $\alpha = 0,6$ ), ускоряется в 3 раза ( $k = 3$ ). В этом случае общее ускорение составит  $1/[0,4 + 0,6/3] = 1,67\times$ . Несмотря на значительное ускорение основной части системы, общее ускорение оказалось значительно меньше ускорения одной части. Это основная идея закона Амдала – чтобы значительно ускорить работу всей системы, нужно повысить скорость очень большой части всей системы.

#### Упражнение 1.1 (решение в конце главы)

Представьте, что вы работаете водителем грузовика и вас наняли для перевозки груза картофеля из города Бойсе, штат Айдахо, в город Миннеаполис, штат Миннесота, на расстояние 2500 километров. По вашим оценкам, вы ездите со средней скоростью 100 км/ч, то есть на выполнение рейса потребуется в общей сложности 25 часов.

1. Вы слышали в новостях, что в штате Монтана, на который приходится 1500 км пути, только что отменили ограничение скорости. Ваш грузовик может двигаться со скоростью 150 км/ч. Каким могло бы быть общее ускорение рейса?
2. Вы можете купить новый турбокомпрессор для своего грузовика на сайте [www.fasttrucks.com](http://www.fasttrucks.com). У них есть множество разных моделей, но чем эффективнее турбокомпрессор, тем дороже он стоит. Насколько быстро вы должны проехать штат Монтана, чтобы получить общее ускорение в 1,67 раза?

#### Упражнение 1.2 (решение в конце главы)

Отдел маркетинга вашей компании пообещал вашим клиентам, что производительность следующей версии программного обеспечения улучшится в 2 раза. Вам поручили выполнить это обещание. Вы определили, что можно ускорить только 80 % системы. Насколько (то есть какое значение  $k$ ) вам нужно ускорить эту часть, чтобы достичь общего увеличения производительности в 2 раза?

### Выражение относительной производительности

Лучший способ выразить увеличение производительности – это отношение вида  $T_{old}/T_{new}$ , где  $T_{old}$  – время, необходимое для выполнения в исходной версии системы, а  $T_{new}$  – время, необходимое для выполнения в измененной версии. Если улучшение действительно имеет место быть, то это число будет больше 1,0. Для обозначения таких отношений мы используем окончание «х», то есть запись «2,2х» читается как «в 2,2 раза».

Также часто используется более традиционный способ выражения относительного изменения в процентах, особенно в случаях, когда изменение небольшое, но определение этого способа неоднозначно. Как должен вычисляться процент? Как  $100 \cdot (T_{old} - T_{new})/T_{old}$ ? Или, может быть, как  $100 \cdot (T_{old} - T_{new})/T_{new}$ ? Или как-то иначе? К тому же этот способ менее наглядный для больших изменений. Понять фразу «производительность улучшилась на 120 %» сложнее, чем «производительность улучшилась в 2,2 раза».

Один интересный частный случай закона Амдала – рассмотреть эффект выбора  $k$  равным  $\infty$ . То есть можно взять некоторую часть системы и ускорить ее до такой степени, что на ее работу будет тратиться ничтожно мало времени. В результате получаем

$$S_{\infty} = \frac{1}{(1 - \alpha)}. \quad (1.2)$$

Так, например, если мы сможем ускорить 60 % системы до такой степени, что она практически не будет потреблять времени, то чистое ускорение все равно составит только  $1/0,4 = 2,5\times$ .

Закон Амдала описывает общий принцип ускорения любого процесса. Однако он может применяться не только к ускорению компьютерных систем, но также может помочь компании, пытающейся снизить стоимость производства бритвенных лезвий, или студенту, желающему улучшить свой средний балл. И все же он наиболее актуален в мире компьютеров, где производительность нередко улучшается в 2 или более раз. Таких высоких значений можно достичь только путем оптимизации больших частей системы.

## 1.9.2. Конкуренция и параллелизм

На протяжении всей истории развития цифровой вычислительной техники постоянными движущими силами, толкающими к совершенствованию, были два требования: компьютеры должны делать больше и работать быстрее. Оба этих параметра улучшаются, когда процессор одновременно может выполнять большее количество задач. Для обозначения общей идеи одновременного выполнения множества действий мы используем термин *конкуренция*, а для обозначения использования конкуренции для ускорения работы системы – термин *параллелизм*. Параллелизм может использоваться в компьютерной системе на нескольких уровнях абстракции. Мы выделяем здесь три уровня, от самого верхнего до самого нижнего в системной иерархии.

### Конкуренция на уровне потоков

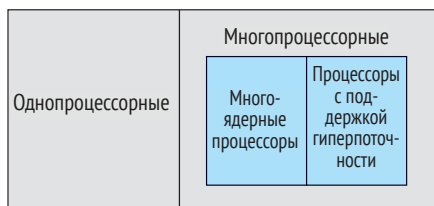
Основываясь на абстракции процессов, можно разрабатывать системы, в которых несколько программ выполняются одновременно, что приводит к *конкуренции*. Используя механизм потоков, можно даже запустить несколько потоков управления в рамках одного процесса. Поддержка конкурентного выполнения появилась в компьютерных системах с момента появления механизма разделения времени в начале 1960-х годов. В ту пору конкурентное выполнение только моделировалось – компьютер просто быстро переключался между выполняемыми процессами, подобно тому, как жонглер держит в воздухе сразу несколько шаров. Эта форма конкуренции позволяет нескольким пользователям одновременно взаимодействовать с системой, например получать

страницы с одного веб-сервера. Она также дает возможность одному пользователю одновременно выполнять несколько задач, например открыть веб-браузер в одном окне, текстовый процессор в другом и одновременно воспроизводить потоковую музыку. До недавнего времени в большинстве случаев все вычисления выполнялись одним процессором, даже если ему приходилось переключаться между несколькими задачами. Эта конфигурация известна как *однопроцессорная система*.

Когда в системе имеется несколько процессоров, все они управляются одним ядром операционной системы, и мы получаем *многопроцессорную систему*. Такие системы были доступны для крупномасштабных вычислений начиная с 1980-х годов, но с появлением *многоядерных* процессоров и технологии *гиперпоточности* они стали обычным явлением. На рис. 1.14 показана классификация этих различных типов процессоров.

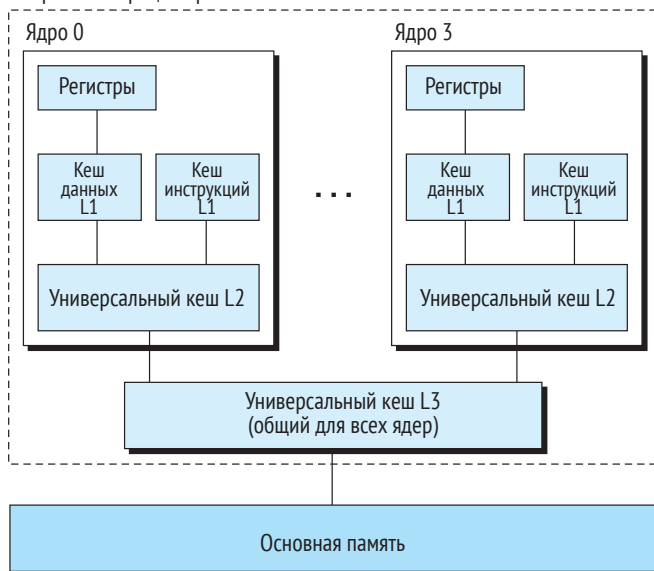
Многоядерные процессоры состоят из нескольких процессоров (называемых «ядрами»), интегрированных на один кристалл. На рис. 1.15 показана организация типичного многоядерного процессора, имеющего в одной микросхеме четыре ядра, каждое со своими кешами L1 и L2, причем каждый кеш L1 разделен на две части: одна предназначена для хранения недавно выбиравшихся инструкций, а другая – данных. Ядра совместно используют кеш-память более высокого уровня (L3), а также интерфейс с основной памятью. Эксперты прогнозируют, что вскоре на одном кристалле будут размещаться десятки, а то и сотни ядер.

Все системы



**Рис. 1.14.** Классификация систем в зависимости от количества процессоров. Многопроцессорные системы становятся все более распространенными с появлением многоядерных процессоров и технологии гиперпоточности

Микросхема процессора



**Рис. 1.15.** Организация многоядерного процессора. Четыре процессорных ядра размещены на одном кристалле



Гиперпоточность, которую иногда называют *одновременной многопоточностью*, – это технология, позволяющая одному процессору выполнять сразу несколько потоков управления. Это предполагает наличие нескольких копий определенных аппаратных средств процессора, таких как счетчики инструкций и блоки регистров, при этом другие аппаратные компоненты, такие как блок арифметических операций с плавающей точкой, наличествуют в единственном числе. В отличие от обычного процессора, которому требуется около 20 000 тактов для переключения между потоками, гиперпотоковый процессор выбирает потоки для выполнения на циклической основе. Это позволяет процессору полнее использовать свои ресурсы. Например, если один поток должен дожидаться загрузки некоторых данных в кеш, то процессор может продолжить выполнение другого потока. Например, каждое ядро в процессоре Intel Core i7 может выполнять два потока, поэтому четырехъядерная система фактически способна параллельно выполнять восемь потоков.

Использование технологий многопроцессорной обработки позволяет повысить производительность системы, предлагая два преимущества. Во-первых, уменьшает необходимость моделирования конкуренции при выполнении нескольких задач. Как уже упоминалось, даже персональный компьютер, используемый одним человеком, может выполнять множество действий одновременно. Во-вторых, дает возможность выполнять каждую отдельную прикладную программу быстрее, правда при условии, что в ней имеется несколько потоков управления, способных эффективно выполняться параллельно. То есть даже притом что принципы параллелизма формировались и изучались на протяжении более 50 лет, только появление многоядерных и гиперпоточных систем способствовало появлению желания использовать приемы разработки прикладных программ, которые могут применять параллелизм на уровне потоков, реализованный на аппаратном уровне. Более подробно поддержка конкурентного выполнения и ее использование рассматриваются в главе 12.

### Параллелизм на уровне инструкций

На гораздо более низком уровне абстракции современные процессоры могут выполнять несколько инструкций одновременно. Это свойство известно как *параллелизм на уровне инструкций*. Например, ранним микропроцессорам, таким как Intel 8086 1978 года выпуска, для выполнения одной инструкции требовалось несколько тактов (обычно 3–10). Более современные процессоры могут выполнять по 2–4 инструкции за такт. Для выполнения любой конкретной инструкции требуется намного больше времени, например 20 тактов или больше, но процессор использует ряд хитрых приемов для одновременной обработки до 100 инструкций. В главе 4 мы рассмотрим использование *конвейерной обработки*, в которой действия, необходимые для выполнения инструкции, разделены на этапы, а аппаратное обеспечение процессора организовано в виде последовательности стадий, каждая из которых выполняет один из этапов. Стадии могут работать параллельно, выполняя разные части разных инструкций. Мы увидим, что для поддержания скорости выполнения, близкой к 1 инструкции на такт, не требуется ничего особенно сложного.

Процессоры, которые могут выполнять более одной инструкции за такт, называют *суперскалярными*. Большинство современных процессоров поддерживают суперскалярные операции. В главе 5 мы опишем высокоуровневую модель таких процессоров и покажем, как прикладные программисты могут использовать эту модель для прогнозирования производительности своих программ, а затем писать программы так, чтобы сгенерированный код достигал более высокой степени параллелизма на уровне инструкций и, следовательно, работал быстрее.

### Одиночный поток команд, множественный поток данных

На самом низком уровне многие современные процессоры имеют специальное оборудование, позволяющее одной инструкции параллельно выполнять несколько опера-



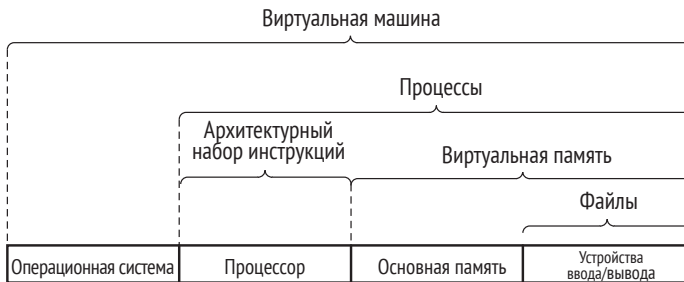
ций. Этот режим известен как *одиночный поток команд, множественный поток данных* (Single-Instruction, Multiple-Data, SIMD). Например, в последних поколениях процессоров Intel и AMD есть инструкции, которые могут параллельно складывать 8 пар чисел с плавающей точкой одинарной точности (тип данных float в языке C).

Эти инструкции SIMD предоставляются в основном для ускорения приложений, обрабатывающих изображения, звук и видео. Некоторые компиляторы пытаются автоматически использовать параллелизм этого вида в программах на C, но все же более надежным методом является разработка программ с использованием специальных типов *векторных* данных, поддерживаемых компиляторами, такими как GCC. Мы кратко опишем этот стиль программирования в приложении в интернете OPT:SIMD в рамках общего описания способов оптимизации программ в главе 5.

### 1.9.3. Важность абстракций в компьютерных системах

*Абстракции* – одна из важнейших концепций информатики. Например, одной из рекомендуемых практик программирования является создание простого прикладного программного интерфейса (Application Program Interface, API) – набора функций, позволяющих программистам использовать код, не вникая в особенности его работы. Разные языки программирования предоставляют разные формы и уровни поддержки абстракции, такие как объявления классов в Java и прототипов функций в C.

Мы уже познакомились с некоторыми абстракциями, изображенными на рис. 1.16, которые встречаются в компьютерных системах. *Архитектурный набор команд* обеспечивает абстракцию физического устройства процессора. Согласно этой абстракции, программа в машинном коде ведет себя так, будто она выполняется на процессоре, который обрабатывает инструкции по одной. Физический процессор устроен намного сложнее и может выполнять несколько инструкций параллельно, но всегда в соответствии с этой простой последовательной моделью. Придерживаясь одной и той же модели выполнения, разные реализации процессоров могут обрабатывать один и тот же машинный код, предлагая различные затраты и производительность.



**Рис. 1.16.** Некоторые абстракции, предоставляемые компьютерной системой.

Основная задача компьютерных систем – обеспечить абстрактные представления на разных уровнях, чтобы скрыть сложность фактических реализаций

На уровне операционной системы мы ввели три абстракции: *файлы* как абстракцию устройств ввода/вывода, *виртуальную память* как абстракцию памяти программ и *процессы* как абстракцию выполняющихся программ. К этим абстракциям мы можем добавить еще одну: *виртуальную машину*, предоставляющую абстракцию всего компьютера, включая операционную систему, процессор и программы. Идея виртуальной машины была представлена компанией IBM еще в 1960-х годах, но лишь не так давно она стала привлекать особое внимание как способ управления компьютерами, которые должны запускать программы, разработанные для разных операционных систем (таких как

Microsoft Windows, Mac OS X и Linux) или разных версий одной и той же операционной системы.

Мы вернемся к этим абстракциям в следующих разделах книги.

## 1.10. Итоги

Компьютерные системы состоят из аппаратных и программных средств, которые взаимодействуют с целью выполнения прикладных программ. Информация внутри компьютера представлена в виде групп битов, которые интерпретируются в зависимости от контекста. Программы транслируются другими программами в различные формы. Сначала они представлены в виде исходного текста ASCII, затем преобразуются компиляторами и компоновщиками в выполняемые файлы.

Процессоры читают и интерпретируют двоичные инструкции, находящиеся в основной памяти. Поскольку большую часть времени компьютеры тратят на копирование данных между основной памятью, устройствами ввода/вывода и регистрами процессора, память системы образует некоторую иерархию, на вершине которой находятся регистры процессора, далее следуют несколько уровней аппаратной кеш-памяти, затем основная DRAM-память и дисковая память. Чем выше находится устройство памяти в иерархии, тем выше его быстродействие и стоимость в пересчете на один бит. Кроме того, устройства памяти, находящиеся выше в иерархии, служат кешем для устройств памяти, находящихся ниже. Программисты могут оптимизировать производительность своих программ на языке C, изучив и воспользовавшись особенностями иерархии памяти.

Ядро операционной системы играет роль посредника между прикладными программами и аппаратными средствами. Оно реализует три фундаментальные абстракции:

- 1) файлы, абстрагирующие устройства ввода/вывода;
- 2) виртуальную память, абстрагирующую как основную память, так и дисковую;
- 3) процессы, абстрагирующие процессоры, основную память и устройства ввода/вывода.

Наконец, сети дают компьютерным системам возможность обмениваться данными между собой. С точки зрения конкретной системы, сеть есть не что иное, как устройство ввода/вывода.

## Библиографические заметки

Ритчи (Ritchie) написал интересные заметки о первых шагах языка C и системы Unix [91, 92]. Ритчи и Томпсон (Ritchie and Thompson) впервые опубликовали отчет о системе [93]. Зильбершатц, Галвин и Ганье (Silberschatz, Galvin and Gagne) представили исчерпывающую историю появления разных версий Unix [102]. Веб-страницы проектов GNU ([www.gnu.org](http://www.gnu.org)) и Linux ([www.linux.org](http://www.linux.org)) содержат текущую и историческую информацию разного характера. Информация о стандартах Posix тоже доступна онлайн ([www.unix.org](http://www.unix.org)).

## Решения упражнений

### Решение упражнения 1.1

Эта задача показывает, что закон Амдала применим не только к компьютерным системам.

1. В терминах уравнения 1.1 мы имеем  $\alpha = 0,6$  и  $k = 1,5$ . Если говорить более конкретно, преодоление 1500 километров через штат Монтана займет 10 часов.

Остальная часть пути также займет 10 часов. Несложные вычисления дают нам ускорение  $25/(10 + 10) = 1,25\times$ .

2. В терминах уравнения 1.1 мы имеем  $\alpha = 0,6$ , и требуется определить  $k$ , чтобы на выходе получить  $S = 1,67$ . Если говорить более конкретно, то чтобы ускорить поездку в 1,67 раза, мы должны уменьшить общее время до 15 часов. На преодоление части пути за пределами штата Монтана по-прежнему потребуются 10 часов, поэтому мы должны пересечь Монтану за 5 часов. Для этого нужно ехать со скоростью 300 км/ч, что довольно много для грузовика!

### Решение упражнения 1.2

Лучше всего действие закона Амдала исследовать на наглядных примерах. Эта задача требует взглянуть на уравнение 1.1 с необычной точки зрения.

Для решения данной задачи нужно просто применить уравнение. Итак, дано:  $S = 2$  и  $\alpha = 0,8$ , мы должны найти  $k$ :

$$2 = \frac{1}{(1 - 0,8) + 0,8/k}$$
$$0,4 + 1,6/k = 1,0$$
$$k = 2,67$$



# Часть I

## Структура программы и ее выполнение

Наше исследование компьютерных систем мы начнем с изучения самого компьютера, состоящего из процессора и подсистемы памяти. Для начала мы выясним способы представления основных типов данных, таких как целые и вещественные числа. После этого мы сможем перейти к рассмотрению того, как машинные инструкции обрабатывают эти данные и как компилятор преобразует программы на языке C в эти инструкции. Далее мы изучим некоторые подходы к реализации процессоров, чтобы лучше понимать, как используются аппаратные ресурсы при выполнении инструкций. Понимая суть работы компиляторов и машинного кода, можно повысить производительность программ, создавая эффективно компилируемый исходный код. В заключение мы исследуем особенности организации подсистемы памяти – одного из самых сложных компонентов современных компьютерных систем.

Данная часть книги обеспечит читателей глубоким пониманием особенностей представления и выполнения прикладных программ. Приобретенные навыки помогут вам писать надежные и безопасные программы, максимально использующие вычислительные ресурсы.

## Представление информации и работа с ней

- 2.1. Хранение информации.
- 2.2. Целочисленное представление.
- 2.3. Целочисленная арифметика.
- 2.4. Числа с плавающей точкой.
- 2.5. Итоги.

Библиографические заметки.

Домашние задания.

Решения упражнений.

Современные компьютеры хранят и обрабатывают информацию, представленную в виде двоичных сигналов. Эти скромные двоичные знаки, или *биты*, формируют основу цифровой революции. Всем знакомая десятичная система счисления используется уже 1000 лет; она изобретена в Индии. В XII веке арабские математики усовершенствовали ее, а на Западе она появилась в XIII веке «с помощью» итальянского математика Леонардо Пизано (1170–1250), более известного под именем Фибоначчи. Использование десятичного исчисления естественно для людей, у которых на руках по десять пальцев, однако для машин, способных хранить и обрабатывать информацию, более приемлемы двоичные величины. Двоичные сигналы легко представлять, хранить и передавать, к примеру как наличие или отсутствие отверстия в перфоленте, как высокое или низкое напряжение в электрической цепи или как магнитная стрелка, указывающая одним концом на север, а другим на юг. Электронные схемы для хранения информации и осуществления расчетов по двоичным сигналам очень просты и надежны; они позволяют производителям объединять миллионы таких схем на одном кремниевом кристалле.

Сам по себе отдельный бит не представляет особой ценности. Однако при объединении битов в группы и применении особой *интерпретации*, придающей определенное значение разным комбинациям битов, можно представить элементы любого конечного множества. Например, применяя систему двоичных чисел, можно использовать группы битов для кодирования неотрицательных чисел. Используя стандартную систему кодировки символов, можно кодировать буквы и символы в документе. В данной главе рассматриваются оба этих вида кодировок, а также кодировки для представления отрицательных чисел и аппроксимации вещественных чисел.

Далее мы рассмотрим три важнейших способа представления чисел. Представление без знака основано на традиционном двоичном представлении чисел больше или равных нулю. Представление в дополнительном коде является наиболее распространенным способом кодирования целых чисел со знаком, которые могут быть как положительными, так и отрицательными. Представление чисел с плавающей точкой – это двоичная версия привычного нам обозначения вещественных чисел. Используя различные представления, компьютеры выполняют арифметические операции, например сложение и умножение, аналогичные соответствующим операциям с целыми и вещественными числами.

Для чисел компьютеры используют ограниченное количество битов, поэтому некоторые операции могут вызывать *переполнение*, когда числа оказываются слишком большими, чтобы их можно было представить в двоичном виде ограниченным количеством битов. Переполнение может приводить к поразительным результатам. Например, на большинстве современных компьютеров (использующих 32-разрядное представление типа `int`) расчет выражения

$$200 * 300 * 400 * 500$$

дает результат –884 901 888. Это противоречит правилам целочисленной арифметики – произведение положительных чисел дало отрицательный результат.

С другой стороны, компьютерная целочисленная арифметика удовлетворяет многим известным правилам целочисленной арифметики. Например, ассоциативность и коммутативность умножения; так, вычисление любого из следующих выражений на языке C даст в результате –884 901 888:

$$\begin{aligned} &(500 * 400) * (300 * 200) \\ &((500 * 400) * 300) * 200 \\ &((200 * 500) * 300) * 400 \\ &400 * (200 * (300 * 500)) \end{aligned}$$

Компьютер может и не дать ожидаемого результата, но он, по крайней мере, последователен!

Арифметика с плавающей точкой обладает совершенно другими математическими свойствами. Произведение множества положительных чисел всегда будет положительным, хотя в случае переполнения вы получите в результате особую величину:  $+\infty$ . С другой стороны, арифметика с плавающей точкой не ассоциативна из-за конечной точности представления. Например, выражение  $(3.14+1e20)-1e20$  на большинстве машин вернет 0.0, а  $3.14+(1e20-1e20)$  вернет 3.14. Различия свойств целочисленной арифметики и арифметики с плавающей точкой проистекают из разницы в подходах к обработке конечного количества их представлений – целочисленные представления охватывают сравнительно небольшой диапазон значений, но точно отражают каждое число в этом диапазоне, в то время как представления с плавающей точкой могут охватывать весьма широкий диапазон значений, но конкретные значения отражают только приблизительно.

Путем изучения фактических числовых представлений можно определить, какие диапазоны величин могут быть представлены, а также свойства различных арифметических операций. Понимание этого аспекта совершенно необходимо для создания программ, работающих корректно во всем диапазоне числовых величин и переносимых на разные типы машин, операционных систем и компиляторов. Как вы узнаете далее, из-за некоторых тонкостей компьютерной арифметики возникло множество уязвимостей. На заре компьютерных вычислений программные ошибки причиняли людям неудобства, только когда они возникали случайно, теперь есть легионы хакеров, которые пытаются использовать любую найденную ошибку, чтобы получить несанкционированный доступ к чужим системам. Это возлагает на программистов особую обя-



занность понимать, как работают их программы и как их можно заставить действовать нежелательным образом.

### Как читать эту главу

В этой главе мы собираемся исследовать фундаментальные свойства представлений чисел и других форм данных в компьютере, а также свойства операций, которые компьютеры выполняют с этими данными. Это требует от нас углубиться в язык математики, писать формулы и уравнения и подробно описывать, как выводятся некоторые важные свойства.

Чтобы вам проще было ориентироваться, мы выбрали следующую организацию главы: каждое свойство сначала будет описано с использованием строгой математической нотации, а затем проиллюстрировано и подкреплено примерами и неформальным обсуждением. Мы рекомендуем внимательно изучить каждую математическую формулировку снова и снова, а также примеры и обсуждение, пока у вас не появится полное понимание того, о чем говорится и что является важным. Для более сложных свойств мы также представим их вывод, структурированный как математическое доказательство. При первом чтении вы можете пропустить эти выводы, но потом обязательно вернитесь к ним и постарайтесь понять их до конца.

Мы также советуем не пропускать практические задачи, которые будут встречаться в ходе обсуждения. Практические задачи вовлекают в активное обучение, помогая превращать мысли в жизнь. Получив опыт решения таких задач, вам будет намного проще проследить ход наших рассуждений в выводах свойств. Также имейте в виду, что математические навыки, необходимые для понимания этого материала, доступны любому, освоившему курс алгебры средней школы.

Для кодирования числовых величин в компьютерах используется несколько различных двоичных представлений. Вы познакомитесь с такими представлениями в главе 3, когда мы приступим к изучению темы программирования на машинном уровне. А в этой мы опишем приемы кодирования и дадим некоторое практическое обоснование используемых представлений чисел.

Для выполнения математических операций непосредственно на уровне битов разработаны несколько способов. Знание этих способов чрезвычайно важно для понимания машинного кода, сгенерированного компилятором, стремящегося оптимизировать вычисление арифметических выражений.

Наша трактовка основана на базовом наборе математических принципов. Сначала мы рассмотрим основные способы представления чисел, после чего выведем такие свойства, как диапазон представимых чисел, их представление на двоичном уровне, а также свойства арифметических операций. Мы полагаем, что данный материал полезнее рассматривать с такой абстрактной точки зрения, потому что программистам необходимо обладать четким пониманием, как компьютерная арифметика соотносится с более знакомой арифметикой целых и вещественных чисел.

Язык программирования C++ основан на C и использует точно такие же представления данных и операции. Все сказанное в данной главе о C в равной степени относится и к C++. Определение языка Java, напротив, создало новый набор стандартов для представления данных и операций. Если стандарт C предназначался для широкого применения, то стандарт Java довольно специфичен в отношении форматов представления данных. В этой главе в нескольких местах мы особо выделим представления и операции, поддерживаемые в Java.

### История развития языка C

Как рассказывалось во врезке «Происхождение языка программирования C» в главе 1, язык C был разработан Деннисом Ритчи из Bell Laboratories для использования с операционной системой Unix (также разработанной в Bell Labs). В то время большинство системного программного обеспечения, такого как операционные системы, приходилось писать в основном на ассемблере, чтобы иметь доступ к низкоуровневым представлениям различных типов данных. Например, на других языках высокого уровня того времени было невозможно написать распределитель памяти, такой как библиотечная функция `malloc`.

Оригинальная версия языка C, созданная в Bell Labs, была задокументирована в первом издании книги Брайана Кернигана и Денниса Ритчи [60]. Со временем язык C продолжал эволюционировать благодаря усилиям нескольких групп стандартизации. Первая серьезная ревизия оригинальной версии C из Bell Labs привела к появлению стандарта ANSI C в 1989 году, разработанному группой под эгидой Американского национального института стандартов (American National Standards Institute, ANSI). ANSI C сильно отличался от Bell Labs C, особенно в способе объявления функций. ANSI C описан во втором издании книги Кернигана и Ритчи [61], которая до сих пор считается одной из лучших книг о C.

Международная организация по стандартизации (International Standards Organization, ISO) взяла на себя ответственность за продолжение стандартизации языка C, приняв в 1990 году версию, которая была практически такой же, как ANSI C, и получила название «ISO C90».

Эта же организация спонсировала обновление языка в 1999 году, в результате чего появился «ISO C99». Среди прочего, эта версия включала некоторые новые типы данных и обеспечивала поддержку текстовых строк с символами, которых нет в английском языке. В 2011 году был утвержден более свежий стандарт, получивший название «ISO C11». И снова в язык были добавлены новые типы данных и функции. Большинство из этих новшеств сохранили *обратную совместимость*, то есть программы, написанные в соответствии с более ранним стандартом (по крайней мере, ISO C90), будут показывать такое же поведение после компиляции с использованием компилятора, поддерживающего новые стандарты.

Коллекция компиляторов GNU (GNU Compiler Collection, GCC) может компилировать программы в соответствии с соглашениями, принятыми в нескольких различных версиях языка C, опираясь на различные параметры командной строки, перечисленные в табл. 2.1. Например, чтобы скомпилировать программу `prog.c` согласно стандарту ISO C11, можно ввести такую команду:

```
linux> gcc -std=c11 prog.c
```

**Таблица 2.1.** Выбор разных версий C в командной строке GCC

Версия C	Параметр командной строки GCC
GNU 89	<i>без параметров</i> , <code>-std=gnu89</code>
ANSI, ISO C90	<code>-ansi</code> , <code>-std=c89</code>
ISO C99	<code>-std=c99</code>
ISO C11	<code>-std=c11</code>

Похожий эффект дают параметры `-ansi` и `-std=c89` – код компилируется в соответствии со стандартом ANSI или ISO C90. (C90 иногда называют «C89», потому что разработка этого стандарта началась в 1989 году.) Параметр `-std=c99` заставляет компилятор следовать соглашениям ISO C99.

На момент написания этой книги, когда в командной строке не указывался ни один из перечисленных параметров, программа компилировалась в соответствии с версией ISO C90, при этом допускалось использовать некоторые особенности из стандартов C99 и C11, а также из C++ и некоторые другие, поддерживаемые компилятором GCC. В рамках проекта GNU была разработана версия, сочетающая в себе ISO C11 и другие возможности, которые можно включить с помощью параметра командной строки `-std=gnu11`. (В настоящее время эта реализация не завершена.) Со временем она станет версией по умолчанию.



### Роль указателей в С

Указатели являются основной особенностью С. Они обеспечивают механизм ссылок на элементы структур данных, включая массивы. Подобно переменной, указатель имеет два аспекта: значение и тип. Значением является адрес местоположения определенного объекта, а типом – тип объекта (например, целое число или число с плавающей точкой), хранящегося в этом местоположении.

Истинное понимание указателей требует изучения их представления и реализации на машинном уровне. Этому будет уделено основное внимание в главе 3, а кульминацией станет более подробное описание в разделе 3.10.1.

## 2.1. Хранение информации

Вместо отдельных битов в большинстве компьютеров наименьшим элементом хранения в памяти являются блоки по 8 бит – *байты*. На машинном уровне программа видит память как очень большой массив байтов, называемый *виртуальной памятью*. Каждый байт имеет уникальный номер, называемый *адресом*, а множество всех возможных адресов называется *виртуальным адресным пространством*. Как подсказывает название, виртуальное адресное пространство – это всего лишь концептуальное представление памяти в программе. Фактическая реализация, описываемая в главе 9, основана на использовании оперативной (RAM) и дисковой памяти, а также специальных аппаратных и программных средств операционной системы, обеспечивающих программы таким массивом байтов, кажущимся непрерывным.

В следующих главах мы покажем, как компилятор и система времени выполнения делят пространство памяти на более управляемые единицы, предназначенные для хранения различных *программных объектов*, то есть данных, инструкций и управляющей информации. Распределение и управление памятью для разных частей программы используют различные механизмы. Все управление осуществляется в рамках виртуального адресного пространства. Например, значение указателя в С, на что бы он не указывал – на целое число, структуру или на какой-то другой объект, – является виртуальным адресом первого байта этого объекта в памяти. Компилятор С также имеет в своем распоряжении информацию о типе каждого указателя, благодаря чему может генерировать разный машинный код для доступа к значению, на которое ссылается указатель, в зависимости от типа этого значения. Однако, несмотря на то что компилятор С обладает информацией о типе, сгенерированная им программа в машинном коде не несет никакой информации о типах данных. На машинном уровне каждый объект программы – это просто блок байтов, а сама программа – последовательность байтов.

### 2.1.1. Шестнадцатеричная система счисления

Один байт состоит из восьми бит. В двоичной системе интервал его значений от  $00000000_2$  до  $11111111_2$ . В десятичном целочисленном представлении байт охватывает диапазон от  $0_{10}$  до  $255_{10}$ . Двоичное представление слишком громоздкое, а для десятичного очень утомительно выполнять преобразования в битовые комбинации и обратно. Вместо всего этого битовые комбинации записываются в шестнадцатеричной нотации. В шестнадцатеричной системе счисления используются цифры от 0 до 9 и буквы от А до F. В табл. 2.2 показаны десятичные и двоичные значения шестнадцатеричных цифр. При записи в шестнадцатеричной форме байт охватывает диапазон от  $00_{16}$  до  $FF_{16}$ .

**Таблица 2.2.** Шестнадцатеричная форма записи чисел.

Каждая шестнадцатеричная цифра представляет одно из 16 возможных значений

<b>Шестнадцатеричное число</b>	0	1	2	3	4	5	6	7
<b>Десятичная величина</b>	0	1	2	3	4	5	6	7
<b>Двоичная величина</b>	0000	0001	0010	0011	0100	0101	0110	0111
<b>Шестнадцатеричное число</b>	8	9	A	B	C	D	E	F
<b>Десятичная величина</b>	8	9	10	11	12	13	14	15
<b>Двоичная величина</b>	1000	1001	1010	1011	1100	1101	1110	1111

В С числовые константы, начинающиеся с 0x или 0X, интерпретируются как шестнадцатеричные. Буквы от A до F можно записывать как в верхнем, так и в нижнем регистре. Например, число FA1D37B<sub>16</sub> можно записать как 0xFA1D37B, как 0xfa1d37b или даже использовать верхний и нижний регистры, например 0xFa1D37b. В этой книге для представления шестнадцатеричных величин мы будем использовать обозначения, принятые в С.

Обычной задачей при работе с машинным кодом является преобразование вручную между десятичным, двоичным и шестнадцатеричным представлениями битовых комбинаций. Преобразование между двоичным и шестнадцатеричным представлениями выполняется просто, по одному шестнадцатеричному знаку за раз. Для этого можно использовать таблицу соответствий, представленную в табл. 2.2. Один из простых способов выполнения преобразований в уме – запоминание десятичных эквивалентов шестнадцатеричных чисел A, C и F. Шестнадцатеричные величины B, D и E можно перевести в десятичные вычислением их величин относительно первых трех.

Например, предположим, что дано число 0x17A4C. Его можно преобразовать в двоичный формат путем развертывания каждой шестнадцатеричной цифры следующим образом:

Шестнадцатеричное	1	7	3	A	4	C
Двоичное	0001	0111	0011	1010	0100	1100

Это дает двоичное значение 000101110011101001001100.

И наоборот, имея двоичное число 1111001010110110110011, преобразовать его в шестнадцатеричное можно, предварительно разбив его на группы по четыре бита. Но имейте в виду, что если общее число битов не кратно четырем, то двоичное представление нужно дополнить ведущими нулями слева. Затем каждая группа из четырех бит преобразуется в соответствующее шестнадцатеричное число:

Двоичное	11	1100	1010	1101	1011	0011
Шестнадцатеричное	3	C	A	D	B	3

### Упражнение 2.1 (решение в конце главы)

Выполните следующие преобразования:

1. 0x39A7F8 – в двоичное.
2. Двоичное 1100100101111011 – в шестнадцатеричное.
3. 0xD5E4C – в двоичное.
4. Двоичное 1001101110011110110101 – в шестнадцатеричное.

Когда величина  $x$  является степенью двойки, т. е.  $x = 2^n$  для некоторого  $n$ , тогда  $x$  можно легко записать в шестнадцатеричной форме, помня, что двоичное представление  $x$  –

это просто 1, за которой следует  $n$  нулей. Шестнадцатеричная цифра 0 – это четыре двоичных нуля. Поэтому для  $n$ , записанного в форме  $i + 4j$ , где  $0 \leq i \leq 3$ ,  $x$  можно записать как шестнадцатеричную цифру 1 ( $i = 0$ ), 2 ( $i = 1$ ), 4 ( $i = 2$ ) или 8 ( $i = 3$ ), за которой следует  $j$  шестнадцатеричных нулей. В качестве примера рассмотрим число  $x = 2048 = 2^{11}$ . В данном случае мы имеем  $n = 11 = 3 + 4 \cdot 2$ , что дает шестнадцатеричное число  $0x800$ .

### Упражнение 2.2 (решение в конце главы)

Заполните пустые клетки в следующей таблице, подставив десятичные и шестнадцатеричные представления различных степеней двойки:

Степень	$2^n$ (десятичное)	$2^n$ (шестнадцатеричное)
9	512	0x200
19	_____	_____
_____	16 384	_____
_____	_____	0x10000
17	_____	_____
_____	32	_____
_____	_____	0x80

В общем случае преобразование между десятичным и шестнадцатеричным представлениями требует использования операций умножения или деления. Чтобы преобразовать десятичное число  $x$  в шестнадцатеричное, можно многократно делить  $x$  на 16, получая частное  $q$  и остаток  $r$ , так что  $x = q \cdot 16 + r$ . Затем  $r$  преобразуется в шестнадцатеричную цифру, которая занимает младшую позицию, и процесс повторяется для шестнадцатеричного числа  $q$ , доводится повторением процесса с  $q$  до наименьшего значимого. Вот пример преобразования десятичного числа 314 156:

$$\begin{aligned}
 314,156 &= 19,634 \cdot 16 + 12 & (C) \\
 19,634 &= 1,227 \cdot 16 + 2 & (2) \\
 1,227 &= 76 \cdot 16 + 11 & (B) \\
 76 &= 4 \cdot 16 + 12 & (C) \\
 4 &= 0 \cdot 16 + 4 & (4)
 \end{aligned}$$

Из получившегося результата можно рассчитать шестнадцатеричное представление  $0x4CB2C$ .

И наоборот, для преобразования шестнадцатеричного числа в десятичное нужно умножить каждую из шестнадцатеричных цифр на соответствующую степень числа 16. Например, десятичный эквивалент числа  $0x7AF$  рассчитывается следующим образом:

$$7 \cdot 16^2 + 10 \cdot 16 + 15 = 7 \cdot 256 + 10 \cdot 16 + 15 = 1792 + 160 + 15 = 1967.$$

### Упражнение 2.3 (решение в конце главы)

Один байт можно представить двумя шестнадцатеричными цифрами. Заполните пустые клетки таблицы, подставляя десятичные, двоичные и шестнадцатеричные величины разным значениям байтов:

Десятичное	Двоичное	Шестнадцатеричное
0	0000 0000	0x00
167	_____	_____
62	_____	_____
188	_____	_____
_____	0011 0111	_____
_____	1000 1000	_____
_____	1111 0011	_____
_____	_____	0x52
_____	_____	0xAC
_____	_____	0xE7

**О преобразовании между десятичной и шестнадцатеричной системами счисления**

Преобразование больших чисел из десятичного в шестнадцатеричное представление и обратно лучше всего поручить калькулятору или компьютеру. Также для этого можно использовать множество готовых инструментов. Самый простой способ – воспользоваться поисковой системой и выполнить запрос, такой как:

или

Преобразовать 0xabcd в десятичную форму

Преобразовать 123 в шестнадцатеричную форму.

**Упражнение 2.4 (решение в конце главы)**

Без преобразования чисел из десятичной системы счисления в двоичную попытайтесь решить следующие арифметические задачи, преобразуя ответы в шестнадцатеричную форму. *Подсказка:* просто модифицируйте методы, используемые для выполнения десятичного сложения и вычитания для использования шестнадцатеричной системы счисления.

- 1.  $0x503c + 0x8 = \underline{\hspace{2cm}}$
- 2.  $0x503c - 0x40 = \underline{\hspace{2cm}}$
- 3.  $0x503c + 64 = \underline{\hspace{2cm}}$
- 4.  $0x50ea - 0x503c = \underline{\hspace{2cm}}$

**2.1.2. Размеры данных**

Каждый компьютер имеет *размер машинного слова*, указывающий номинальный размер указателя. Поскольку виртуальные адреса кодируются такими словами, то наиболее важным системным параметром, определяемым размером слова, является максимальный размер виртуального адресного пространства. То есть для машины с размером слова в  $w$  бит диапазон виртуальных адресов охватывает от 0 до  $2^w - 1$ , обеспечивая программе доступ максимум к  $2^w$  байт.

В настоящее время наблюдается масштабный переход от машин с размером слова в 32 бита к машинам с размером слова в 64 бита. В первую очередь такой переход был выполнен в сфере масштабных научных вычислений и баз данных, далее последовали настольные компьютеры и ноутбуки, а совсем недавно 64-разрядные процессоры появились на смартфонах. Размер слова в 32 бита ограничивает виртуальное адресное пространство 4 гигабайтами (записывается 4 Гбайт), т. е. немногим более  $4 \times 10^9$  байт. Переход на 64-разрядный размер слова увеличил размер виртуального адресного пространства до 16 *эксабайт*, т. е. около  $1,84 \times 10^{19}$  байт.

Большинство 64-разрядных машин способны также выполнять программы, скомпилированные для 32-разрядной архитектуры, обеспечивая тем самым обратную совместимость. Так, например, если программу `prog.c` скомпилировать с директивой

```
linux> gcc -m32 prog.c
```

то эта программа будет корректно выполняться и на 32-разрядной машине, и на 64-разрядной. Программа, скомпилированная с директивой

```
linux> gcc -m64 prog.c
```

напротив, будет выполняться только на 64-разрядной машине. Далее мы будем называть программы «32-разрядными» или «64-разрядными», только чтобы подчеркнуть, как они скомпилированы, но не для того, чтобы указать, на машинах какого типа они могут выполняться.

Компьютеры и компиляторы поддерживают множество форматов данных, используя различные способы кодирования данных, например целых чисел и чисел с плавающей точкой, а также различные размеры. Например, многие машины имеют инструкции для манипуляций с отдельными байтами, а также целыми числами длиной 2, 4 и 8 байт. Они поддерживают числа с плавающей точкой длиной 4 и 8 байт.

Язык C поддерживает множество форматов данных, целых и с плавающей точкой. В табл. 2.3 перечислены разные типы данных и их размеры в байтах. (В разделе 2.2 мы обсудим связь между типичными размерами и определяемыми стандартом языка C.) Точный размер некоторых типов данных зависит от способа компиляции программы. Мы покажем размеры для типичных 32- и 64-разрядных программ. Целые числа могут быть *со знаком*, т. е. способны представлять отрицательные и положительные значения, или *без знака*, т. е. способны представлять только неотрицательные значения. Тип данных `char` представляет единственный байт. Несмотря на то что название `char` связано с запоминанием одного символа (`character`) в текстовой строке, его также можно использовать для хранения целых величин. Типы данных `short`, `int` и `long` представляют значения разных размеров. Но даже при компиляции в 64-разрядной системе тип данных `int` обычно имеет размер 4 байта. Тип данных `long` в 32-разрядных программах обычно имеет размер 4 байта, а в 64-разрядных – 8 байт.

#### Новичок в C? Объявление указателей

Для любого типа данных *T* объявление

```
T *p;
```

указывает, что *p* – это переменная указателя, указывающего на объект типа *T*. Например:

```
char *p;
```

– это объявление указателя на объект типа `char`.

Чтобы избежать неоднозначности, связанной с «типичными» размерами и настройками компилятора, в ISO C99 был определен класс типов данных с фиксированными

размерами, не зависящими от настроек компилятора и архитектуры компьютера. Среди них типы `int32_t` и `int64_t`, имеющие размеры 4 и 8 байт соответственно. Использование целочисленных типов фиксированного размера – лучший способ для программистов сохранить полный контроль над представлениями данных.

Большинство типов данных кодируют числовые значения как значения со знаком, за исключением случаев, когда им предшествует ключевое слово `unsigned` или используется специальное объявление типа без знака фиксированного размера. Исключением является тип данных `char`. Большинство компиляторов и машин обрабатывают этот тип как целочисленный тип со знаком, однако стандарт C не гарантирует этого. Вместо этого, как показано в квадратных скобках, программист должен использовать объявление `signed char`, чтобы гарантировать 1-байтное представление со знаком. Однако во многих контекстах поведение программы не зависит от того, является тип данных `char` со знаком или без знака.

Язык C допускает разный порядок следования ключевых слов и позволяет включать или опускать необязательные ключевые слова. Например, все следующие объявления определяют один и тот же тип:

```
unsigned long
unsigned long int
long unsigned
long unsigned int
```

Мы же в этой книге будем следовать форме объявления, показанной в табл. 2.3.

**Таблица 2.3.** Типичные размеры (в байтах) основных типов данных в языке C. Размер зависит от способа компиляции программы. Здесь показаны типичные размеры для 32- и 64-разрядных программ

Объявление в языке C		Байты	
Со знаком	Без знака	В 32-разрядных программах	В 64-разрядных программах
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

В табл. 2.3 также показано, что указатель (например, переменная, объявленная с типом `char *`) использует полноразмерное слово программы. Большинство машин также поддерживают два разных формата с плавающей точкой: одинарной точности, в C объявляется как `float`, и двойной точности, объявляется как `double`. Для переменных этих типов отводится 4 и 8 байт соответственно.

Программисты должны стремиться делать свои программы переносимыми между разными машинами и компиляторами. Один из аспектов переносимости – нечувствительность к точным размерам различных типов данных. Стандарты C устанавливают нижние границы числовых диапазонов для разных типов данных, как будет показано ниже, но не устанавливают верхних границ (кроме типов фиксированного размера). На 32-разрядных машинах и в 32-разрядных программах, преобладавших в период примерно с 1980-е по 2010-е годы, многие программы писались с учетом размеров, ука-

занных в табл. 2.3 для 32-разрядных программ. С переходом на 64-разрядные машины многие скрытые зависимости от размера слова стали приводить к ошибкам при переносе старых программ на новые машины. Например, многие программисты традиционно полагали, что объект, объявленный с типом `int`, можно использовать для хранения указателя. Это предположение было справедливо для большинства 32-разрядных программ, но привело к проблемам в 64-разрядных программах.

### 2.1.3. Адресация и порядок следования байтов

Для программных объектов, занимающих несколько байтов, необходимо установить два правила: каков будет адрес объекта и как должны располагаться байты в памяти. Практически во всех машинах такие объекты хранятся в виде непрерывных последовательностей байтов, а адресом многобайтного объекта служит наименьший адрес ячейки памяти. Например, предположим, что переменная `x` типа `int` имеет адрес `0x100`, т. е. выражение взятия адреса `&x` вернет `0x100`. Тогда четыре байта, составляющих значение переменной `x`, будут храниться в ячейках памяти `0x100`, `0x101`, `0x102` и `0x103`.

Существует два общепринятых правила, определяющих порядок следования байтов в таких объектах. Рассмотрим целое число длиной  $w$  бит, имеющее битовое представление  $[x_{w-1}, x_{w-2}, \dots, x_1, x_0]$ , где  $x_{w-1}$  – наибольший значащий бит, а  $x_0$  – наименьший значимый. Если предположить, что  $w$  кратно восьми, то эти биты можно сгруппировать в байты, где наибольший значащий байт будет включать биты  $[x_{w-1}, x_{w-2}, \dots, x_{w-8}]$ , наименьший значащий – биты  $[x_7, x_6, \dots, x_0]$ , а другие байты будут включать биты из середины. В одних машинах объект будет храниться в памяти в порядке от наименьшего значащего байта к наибольшему значащему, а в других – наоборот. Первое правило: порядок, когда первым следует наименьший значащий байт, называется *обратным*, или *остроконечным* (*little endian*). Второе правило: порядок, когда первым следует наибольший значащий байт, называется *прямым*, или *тупоконечным* (*big endian*).

Теперь вернемся к нашему примеру с переменной `x` типа `int` с адресом `0x100`, хранящей шестнадцатеричное значение `0x01234567`. Порядок расположения байтов в ячейках с адресами от `0x100` до `0x103` зависит от типа машины:

**Прямой порядок** (*big endian* – тупоконечный):

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

**Обратный порядок** (*little-endian* – остроконечный):

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

Обратите внимание, что в слове `0x01234567` старший байт имеет шестнадцатеричное значение `0x01`, а младший – `0x67`.

В большинстве Intel-совместимых машин используется исключительно обратный (*little-endian*) порядок следования байтов. В большинстве машин, выпускаемых IBM и Oracle (в 2010 году компания Oracle приобрела компанию Sun Microsystems и вошла в число производителей вычислительных машин), используется прямой (*big-endian*) порядок следования байтов. Обратите внимание, что мы сказали «в большинстве». Соглашения не разделяются строго по корпоративным границам. Например, обе компании, IBM и Oracle, производят машины, использующие Intel-совместимые процессоры, поддерживающий обратный (*little-endian*) порядок следования байтов. Многие современные микропроцессоры поддерживают *прямой* (*big-endian*) порядок байтов, и их

можно настроить для работы как с прямым (big-endian), так и с обратным (little-endian) порядком следования байтов. Однако на практике порядок байтов фиксируется с выбором конкретной операционной системы. Например, микропроцессоры ARM, используемые во многих сотовых телефонах, поддерживают оба порядка следования байтов, но две наиболее распространенные операционные системы – Android (от Google) и IOS (от Apple) – используют только обратный (little-endian) порядок байтов.

#### **О происхождении терминов «little endian» (остроконечный) и «big endian» (тупоконечный)**

Вот как в 1726 году Джонатан Свифт описывал историю последователей разбивания яйца с тупого и острого конца:

...Лиллипутия и Блефуску... Эти две могущественные державы ведут между собой ожесточенную войну на протяжении тридцати шести лун. Поводом к войне послужили следующие обстоятельства. Всеми разделяется убеждение, что варенные яйца при употреблении в пищу испокон веков разбивались с тупого конца; но дед нынешнего императора, будучи ребенком, порезал себе палец за завтраком, разбивая яйцо означенным древним способом. Тогда император, отец ребенка, обнародовал указ, предписывающий всем его подданным под страхом строгого наказания разбивать яйца с острого конца. Этот закон до такой степени озлобил население, что, по словам наших летописей, стал причиной шести восстаний, во время которых один император потерял жизнь, а другой – корону. Мятежи эти постоянно разжигались монархами Блефуску, а после их подавления изгнанники всегда находили приют в этой империи. Насчитывают до одиннадцати тысяч фанатиков, которые в течение этого времени пошли на казнь, лишь бы не разбивать яйца с острого конца. Были напечатаны сотни томов, посвященных этой полемике, но книги Тупоконечников давно запрещены, и вся партия лишена законом права занимать государственные должности.

В свое время Свифт зло иронизировал по поводу непрекращающихся стычек между Англией (Лиллипутия) и Францией (Блефуску). Дэнни Коэн (Danny Cohen), родоначальник сетевых протоколов, впервые применил эти термины для описания упорядочения байтов [24], после чего они получили широкое распространение.

Люди оказываются удивительно эмоциональными, вступая в спор о том, какой порядок байтов правильный. На самом деле термины «little endian» (остроконечный) и «big endian» (тупоконечный) заимствованы из книги Джонатана Свифта «Путешествия Гулливера», где описывается вражда двух группировок, спорящих о том, с какого конца следует разбивать сваренное всмятку яйцо: с тупого или острого. Точно так же, как и в случае с пресловутым яйцом, не существует технологического смысла ставить один способ упорядочения байтов выше другого, и фактически вся полемика сводится к банальному пикированию на общественно-политические темы. Пока будет существовать выбор из двух способов и его приверженцы будут последовательны, до тех пор выбор будет произвольным.

Для большинства прикладных программистов порядок следования байтов, используемый их машинами, полностью невидим; программы, скомпилированные для любого класса машин, дают идентичные результаты. Однако иногда порядок байтов становится проблемой. Первый случай – когда двоичные данные передаются по сети между разными машинами. Проблема возникает, когда данные, созданные машиной с обратным (little-endian) порядком байтов, отправляются на машину с прямым (big-endian) порядком байтов или наоборот, в результате чего для принимающей программы байты в словах следуют не по порядку. Чтобы избежать таких проблем, сетевые приложения должны следовать установленным соглашениям о порядке следования



байтов: отправляющая машина должна преобразовывать данные из внутреннего представления гарантированно в сетевой стандарт, а принимающая – из сетевого стандарта в свое внутреннее представление. Мы увидим примеры этих преобразований в главе 11.

Второй случай, когда порядок байтов приобретает важность, – интерпретация последовательностей байтов, представляющих целочисленные данные. Это часто имеет место при исследовании машинного кода программ. Например, в файле с машинным кодом для процессора Intel x86\_64 имеется следующая строка:

```
4004d3: 01 05 43 0b 20 00 add %eax,0x200b43(%rip)
```

Эта строка создана *дисассемблером*, инструментом, извлекающим последовательно инструкции из выполняемого файла программы. Более подробно такие инструменты и способы интерпретации подобных строк мы рассмотрим в главе 3. А пока просто заметим, что эта строка содержит последовательность шестнадцатеричных байтов 01 05 43 0b 20 0 – представление инструкции, складывающей слово данных со значением, хранящимся по адресу, который сам вычисляется сложением числа 0x200b43 с содержимым *счетчика инструкций* – адреса следующей инструкции. Если взять последние четыре байта данной последовательности 43 0b 20 00 и записать их в обратном порядке, то получится 00 20 0b 43. Если отбросить начальный ноль, то получится значение 0x200b43, записанное в правой части. То, что байты следуют в обратном порядке, – обычное дело для машин с обратным (little-endian) порядком байтов. Естественный способ записи последовательности байтов следующий: младший байт записывается слева, а старший – справа, хоть это и противоречит обычному способу записи чисел, когда старший значащий разряд записывается слева, а младший – справа.

Третий случай, когда порядок байтов важен, – при разработке программ, которые, что называется, действуют «в обход» обычной системы типов. В языке C это можно сделать с использованием *приведения* (cast) или *объединения* (union), чтобы получить возможность ссылаться на объект, который может интерпретироваться по-разному. В общем и целом подобные «трюки», мягко говоря, не поощряются большинством программистов, однако они могут быть полезными и даже необходимыми для программирования на системном уровне.

В листинге 2.1 показан код на C, в котором используется приведение типа для доступа и вывода различных программных объектов в виде последовательностей байтов. Для определения типа данных `byte_pointer` как указателя на объект типа `unsigned char` используется `typedef`. Такой указатель на байт ссылается на последовательность байтов, в которой каждый байт интерпретируется как неотрицательное целое число. Первая подпрограмма `show_bytes` принимает адрес последовательности байтов через параметр типа `byte_pointer` и счетчик байтов. Счетчик байтов передается через параметр типа `size_t` – предпочтительный тип для представления размеров разных структур данных. Она выводит отдельные байты в шестнадцатеричном виде. Директива форматирования `% 2x` указывает, что целое число должно быть преобразовано в шестнадцатеричный вид, содержащий не меньше двух цифр.

**Листинг 2.1.** Функции вывода программных объектов в виде последовательностей байтов. Они используют приведение типа, чтобы выполнить свою работу в обход системы типов. Подобные функции легко определить для других типов данных

```
1 #include <stdio.h>
2
3 typedef unsigned char *byte_pointer;
4
5 void show_bytes(byte_pointer start, size_t len) {
```

```

6     int i;
7     for (i = 0; i < len; i++)
8         printf("%.2x", start[i]);
9     printf("\n");
10 }
11
12 void show_int(int x) {
13     show_bytes((byte_pointer) &x, sizeof(int));
14 }
15
16 void show_float(float x) {
17     show_bytes((byte_pointer) &x, sizeof(float));
18 }
19
20 void show_pointer(void *x) {
21     show_bytes((byte_pointer) &x, sizeof(void *));
22 }

```

Процедуры `show_int`, `show_float` и `show_pointer` демонстрируют, как можно использовать процедуру `show_bytes` для вывода в виде последовательностей байтов программных объектов с типами `int`, `float` и `void *` соответственно. Обратите внимание, что они просто передают указатель `&x` на свой аргумент `x` в вызов `show_bytes`, приводя указатель к типу `unsigned char *`. Эта операция приведения типа указывает компилятору, что программа будет интерпретировать указатель как указатель на последовательность байтов, а не на объект исходного типа данных. Этот указатель будет затем указывать на самый младший адрес, занимаемый объектом.

Эти процедуры используют оператор `sizeof` для определения количества байтов, занимаемых объектом. Как правило, выражение `sizeof(T)` возвращает количество байтов, необходимых для хранения объекта типа `T`. Использование `sizeof` вместо фиксированного значения – это одно из условий написания кода, переносимого на разные типы машин.

Мы запустили код из листинга 2.2 на нескольких разных машинах и получили результаты, показанные в табл. 2.4. В эксперименте использовались следующие машины:

Linux 32	с процессором Intel IA32 и работающая под управлением Linux;
Windows	с процессором Intel IA32 и работающая под управлением Windows;
Sun	с процессором Sun Microsystems SPARC и работающая под управлением Solaris (в настоящее время эти машины выпускаются Oracle);
Linux 64	с процессором Intel x86-64 и работающая под управлением Linux.

**Листинг 2.2.** Примеры представления в виде последовательностей байтов.  
Этот код выводит объекты данных в виде последовательностей байтов

```

1 void test_show_bytes(int val) {
2     int ival = val;
3     float fval = (float) ival;
4     int *pval = &ival;
5     show_int(ival);
6     show_float(fval);
7     show_pointer(pval);
8 }

```

*code/data/show-bytes.c*

*code/data/show-bytes.c*

**Таблица 2.4.** Представления различных значений данных в виде последовательностей байтов. Результаты для `int` и `float` идентичны, за исключением порядка байтов. Значения указателя зависят от типа машины

Машина	Значение	Тип	Байты (в шестнадцатеричном виде)
Linux 32	12345	<code>int</code>	39 30 00 00
Windows	12345	<code>int</code>	39 30 00 00
Sun	12345	<code>int</code>	00 00 30 39
Linux 64	12345	<code>int</code>	39 30 00 00
Linux 32	12345.0	<code>float</code>	00 e4 40 46
Windows	12345.0	<code>float</code>	00 e4 40 46
Sun	12345.0	<code>float</code>	46 40 e4 00
Linux 64	12345.0	<code>float</code>	00 e4 40 46
Linux 32	<code>&amp;ival</code>	<code>int *</code>	e4 f9 ff bf
Windows	<code>&amp;ival</code>	<code>int *</code>	b4 cc 22 00
Sun	<code>&amp;ival</code>	<code>int *</code>	ef ff fa 0c
Linux 64	<code>&amp;ival</code>	<code>int *</code>	b8 11 e5 ff ff 7f 00 00

Аргумент 12345 имеет шестнадцатеричное представление `0x00003039`. Для типа `int` получаем идентичные результаты для всех машин, кроме порядка байтов. Можно заметить, что наименьший значащий байт `0x39` выводится первым в Linux 32, Windows и Linux 64, что указывает на использование обратного (*little-endian*) порядка байтов в этих машинах, а в Sun – последним, что указывает на использование прямого (*big-endian*) порядка байтов. Подобным же образом идентичны байтовые представления данных типа `float`, кроме порядка байтов. Однако значения указателей абсолютно различны. В разных конфигурациях машин и операционных систем используются разные правила распределения памяти. Стоит отметить одну особенность: машины с Linux 32, Windows и Sun используют 4-байтные адреса, а Linux 64 – 8-байтные.

#### Новичок в C? Об определении своих имен для типов данных

Объявление `typedef` в языке C дает возможность определять свои имена для типов данных. Это может заметно улучшить читаемость кода, потому что глубоко вложенные объявления типов порой очень сложно расшифровывать.

Синтаксис `typedef` подобен синтаксису объявления переменной, за исключением того, что в данном случае используется имя типа, а не переменной. То есть объявление переменной с типом `byte_pointer` в листинге 2.1 равноценно объявлению переменной с типом `unsigned char`. Например, строки

```
typedef int *int_pointer;
int_pointer ip;
```

определяют тип `int_pointer` как указатель на значение типа `int` и объявляют переменную `ip` с этим типом. Как вариант эту переменную можно было объявить как

```
int *ip;
```



**Создание таблицы ASCII**

Командой `man ascii` можно вывести таблицу с кодами символов ASCII.

**Упражнение 2.5 (решение в конце главы)**

Рассмотрим три следующих вызова `show_bytes`:

```
int val = 0x87654321;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```

Укажите, какие значения будут выведены каждым вызовом на машинах с обратным (little-endian) и прямым (big-endian) порядками следования байтов.

- |                            |                       |
|----------------------------|-----------------------|
| A. Обратный порядок: _____ | Прямой порядок: _____ |
| B. Обратный порядок: _____ | Прямой порядок: _____ |
| C. Обратный порядок: _____ | Прямой порядок: _____ |

**Упражнение 2.6 (решение в конце главы)**

Используя `show_int` и `show_float`, мы определили, что целое число 3510593 имеет шестнадцатеричное представление `0x00359141`, тогда как число с плавающей точкой 3510593.0 имеет шестнадцатеричное представление `0x4A564504`.

1. Запишите эти два шестнадцатеричных значения в двоичной форме.
2. Сдвиньте две строки относительно друг друга до совпадения битов. Сколько битов совпадает?
3. Какие части строк не совпадают?

**2.1.4. Представление строк**

Строка в языке C кодируется массивом символов и завершается нулем (нулевым символом). Значения символов определяются кодировкой, самой распространенной из которых является ASCII. Следовательно, если запустить процедуру `show_bytes` с аргументами "12345" и 6 (чтобы включить завершающий нулевой символ), тогда в результате получится последовательность байтов 31 32 33 34 35 00. Обратите внимание, что в кодировке ASCII десятичная цифра *x* имеет код `0x3x`, а завершающий байт имеет шестнадцатеричное представление `0x00`. Тот же результат будет получен в любой системе, использующей ASCII, независимо от порядка следования байтов и размера слова. Вследствие этого текстовые данные менее зависимы от платформы, нежели двоичные.

**Упражнение 2.7 (решение в конце главы)**

Что выведет следующий вызов `show_bytes`?

```
const char *s = "abcdef";
show_bytes((byte_pointer) s, strlen(s));
```

Обратите внимание, что буквы от «a» до «z» имеют коды ASCII от `0x61` до `0x7a`.

### Стандарт Юникода для представления текста

Набор символов ASCII подходит для кодировки документов на английском языке, однако в нем отсутствуют некоторые буквы, например французского языка, такие как «ç». Данный набор символов абсолютно непригоден для представления документов на таких языках, как греческий, русский и китайский.

За прошедшие годы было разработано и предложено множество методов кодирования текста на разных языках. Консорциум Unicode Consortium разработал наиболее полный стандарт кодирования текста. Текущий стандарт Unicode (версия 7.0) включает более 100 000 символов для широкого спектра языков, в том числе языки Древнего Египта и Вавилона. К их чести, технический комитет Unicode Technical Committee отклонил предложение включить стандартное письмо Клингонов – вымышленной цивилизации из телесериала *Star Trek* (Звездный путь).

Базовая кодировка, известная как «универсальный набор символов» Юникод, использует 32-разрядное представление символов. Казалось бы, это требует, чтобы каждый символ в текстовой строке занимал 4 байта. Однако возможны альтернативные варианты кодирования, когда для наиболее распространенных символов требуется всего 1 или 2 байта, а для менее распространенных – больше. В частности, в кодировке UTF-8 каждый символ кодируется как последовательность байтов так, что для стандартных символов ASCII используются те же однобайтовые коды, что и в ASCII, в том смысле, что все последовательности символов ASCII выглядят точно так же и в кодировке UTF-8.

Язык программирования Java использует Юникод для представления своих строк. Программные библиотеки поддержки Юникода доступны также для C.

## 2.1.5. Представление программного кода

Рассмотрим следующую функцию на C:

```
1 int sum(int x, int y) {
2     return x + y;
3 }
```

При компилировании на машинах, которые мы использовали в экспериментах выше, будет создан машинный код, имеющий следующее представление в виде байтов:

<b>Linux 32</b>	55 89 e5 8b 45 0c 03 45 08 c9 c3
<b>Windows</b>	55 89 e5 8b 45 0c 03 45 08 5d c3
<b>Sun</b>	81 c3 e0 08 90 02 00 09
<b>Linux 64</b>	55 48 89 e5 89 7d fc 89 75 f8 03 45 fc c9 c3

Как видите, кодирование инструкций отличается. В разных типах машин используются разные и несовместимые инструкции и кодировки. Даже идентичные процессоры, действующие под управлением разных операционных систем, имеют различия в соглашениях о кодировании и, следовательно, не совместимы на двоичном уровне. Двоичный код редко можно переносить между разными комбинациями машин и операционных систем.

Фундаментальная идея компьютерных систем заключается в том, что с точки зрения машины программа представляет собой обычную последовательность байтов. Машина не имеет информации об исходном коде программы, за исключением, возможно, некоторых вспомогательных таблиц, создаваемых в помощь при отладке. Более подробно эту тему мы рассмотрим в главе 3, когда будем изучать программирование на машинном языке.

### 2.1.6. Введение в булеву алгебру

В основе кодирования, хранения и обработки информации компьютерами лежат двоичные значения, поэтому вокруг значений 0 и 1 развился обширный массив математических дисциплин. Все началось с работы Джорджа Буля (George Boole; 1815–1864), вышедшей в 1850 году и получившей название *булева алгебра*. Джордж Буль заметил, что, кодируя логические значения «истина» и «ложь» в виде двоичных значений 1 и 0, можно сформулировать алгебру, отражающую основные принципы логических рассуждений.

Простейшая булева алгебра определяется на множестве двух элементов  $\{0,1\}$ . В табл. 2.5 показано несколько операций в этой алгебре. Символы представления операций подобраны так, что совпадают с битовыми операторами в языке C, которые мы рассмотрим ниже. Булева операция  $\sim$  соответствует логической операции НЕ, обозначаемой в логике высказываний символом  $\neg$ . То есть говорится, что  $\neg P$  истинно, когда  $P$  не истинно, и наоборот. Соответственно,  $\sim p$  равно 1, когда  $p$  равно нулю, и наоборот. Булева операция  $\&$  соответствует логической операции И, обозначаемой в логике высказываний символом  $\wedge$ . Мы говорим, что  $P \wedge Q$  выполняется, когда и  $P$  и  $Q$  истинны. Соответственно,  $p \& q$  равно единице только тогда, когда  $p = 1$  и  $q = 1$ . Булева операция  $|$  соответствует логической операции ИЛИ, обозначаемой в логике высказываний символом  $\vee$ . Мы говорим, что  $P \vee Q$  выполняется, когда истинно  $P$  или  $Q$ . Соответственно,  $p | q$  равно единице только тогда, когда  $p = 1$  или  $q = 1$ . Булева операция  $\wedge$  соответствует логической операции ИСКЛЮЧАЮЩЕЕ ИЛИ, обозначаемой в логике высказываний символом  $\oplus$ . Мы говорим, что  $P \oplus Q$  выполняется, когда истинно  $P$  или  $Q$ , но не оба. Соответственно,  $p \wedge q$  равно единице, когда либо  $p = 1$  и  $q = 0$ , либо  $p = 0$  и  $q = 1$ .

**Таблица 2.5. Операции булевой алгебры.** Двоичные значения 1 и 0 кодируют логические значения «истина» и «ложь», а операции  $\sim$ ,  $\&$ ,  $|$ , и  $\wedge$  кодируют логические операции НЕ, И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ соответственно

$\sim$		$\&$	0	1	$ $	0	1	$\wedge$	0	1
0	1	0	0	0	0	0	1	0	0	1
1	0	1	0	1	1	1	1	1	1	0

Клод Шэннон (Claude Shannon; 1916–2001), ставший впоследствии основоположником теории информации, впервые обратил внимание на связь булевой алгебры и дискретной логики. В своей магистерской диссертации (в 1937 году) он доказал, что булеву алгебру можно применять к проектированию и анализу сетей электромеханических реле. Несмотря на то что с тех пор компьютерные технологии продвинулись очень далеко, булева алгебра по-прежнему играет центральную роль в проектировании и анализе цифровых систем.

Четыре логические операции можно распространить на *битовые векторы*, строки нулей и единиц некоторой фиксированной длины  $w$ . Мы определяем операции над битовыми векторами как применение к соответствующим элементам аргументов. Пусть  $a$  и  $b$  обозначают битовые векторы  $[a_{w-1}, a_{w-2}, \dots, a_0]$  и  $[b_{w-1}, b_{w-2}, \dots, b_0]$  соответственно. Мы определяем  $a \& b$  как битовый вектор с длиной  $w$ , где  $i$ -й элемент равен значению выражения  $a_i \& b_i$  для  $0 \leq i < w$ . Аналогичным образом на битовые векторы распространяются операции  $|$ ,  $\wedge$  и  $\sim$ .

В качестве примера рассмотрим случай, когда  $w = 4$  и аргументами являются векторы  $a = [0110]$  и  $b = [1100]$ . Четыре операции  $a \& b$ ,  $a | b$ ,  $a \wedge b$  и  $a \sim b$  дают в результате:

$\&$	$\frac{0110}{1100}$	$ $	$\frac{0110}{1100}$	$\wedge$	$\frac{0110}{1100}$	$\sim$	$\frac{0110}{1100}$
	0100		1110		1010		0011



Одним из полезных применений битовых векторов является представление конечных множеств. Например, любое подмножество  $A \subseteq \{0, 1, \dots, w-1\}$  можно представить в виде битового вектора  $[a_{w-1}, \dots, a_1, a_0]$ , где  $a_i = 1$  в том и только том случае, если  $i \in A$ . Например, помня, что  $a_{w-1}$  записывается слева, а  $a_0$  – справа, вектор  $a = [01101001]$  представляет множество  $A = \{0, 3, 5, 6\}$ , а вектор  $b = [01010101]$  – множество  $B = \{0, 2, 4, 6\}$ . При такой интерпретации булевы операции  $|$  и  $\&$  соответствуют объединению и пересечению множеств, операция  $\sim$  соответствует дополнению множества. Например, операция  $a \& b$  дает битовый вектор  $[01000001]$ , тогда как  $A \cap B = \{0, 6\}$ .

Мы еще увидим примеры практического применения представления множеств в виде битовых векторов. Например, в главе 8 будет показано, что существует ряд различных *сигналов*, которые могут прервать выполнение программы. Мы сможем выборочно включать или отключать различные сигналы, задав маску в виде битового вектора, где 1 в битовой позиции  $i$  указывает, что сигнал  $i$  включен, а 0 – отключен. То есть маска представляет собой множество разрешенных сигналов.

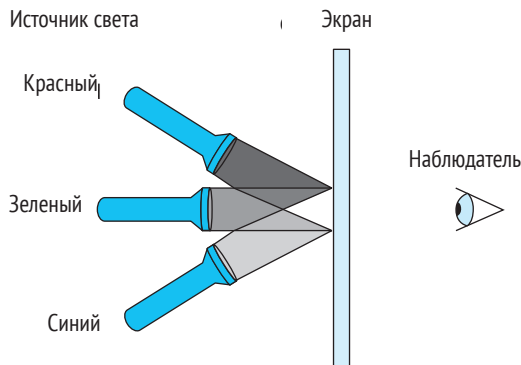
### Упражнение 2.8 (решение в конце главы)

Заполните следующую таблицу, подставив результаты булевых операций с битовыми векторами.

Операция	Результат
$a$	$[01101001]$
$b$	$[01010101]$
$\sim a$	_____
$\sim b$	_____
$a \& b$	_____
$a   b$	_____
$a \wedge b$	_____

### Упражнение 2.9 (решение в конце главы)

Компьютеры создают цветные изображения на мониторе или жидкокристаллическом дисплее путем смешения трех разных цветов светового спектра: красного, зеленого и синего. Представьте простую схему с тремя разными цветами, каждый из которых можно спроецировать на стеклянный экран:



Мы можем создать восемь разных цветов, включая (1) или выключая (0) тот или иной источник света:

Красный (R)	Зеленый (G)	Синий (B)	Цвет
0	0	0	Черный
0	0	1	Синий
0	1	0	Зеленый
0	1	1	Голубой
1	0	0	Красный
1	0	1	Алый
1	1	0	Желтый
1	1	1	Белый

Каждый из этих цветов можно представить в виде битового вектора с длиной 3 и применить к ним булевы операции.

1. Дополнение цвета формируется выключением включенного цвета и включением выключенного. Какие цвета будут считаться дополнениями к восьми перечисленным?
2. Опишите эффект применения булевых операций к следующим цветам:

Синий | Зеленый = \_\_\_\_\_

Желтый & Голубой = \_\_\_\_\_

Красный ^ Алый = \_\_\_\_\_

#### Приложение в интернете DATA:BOOL. Еще о булевой алгебре и булевых кольцах

Логические операции  $|$ ,  $\&$  и  $\sim$ , применяемые к битовым векторам с длиной  $w$ , образуют *булеву алгебру* для любого целого числа  $w > 0$ . Самым простым является случай, когда  $w = 1$  и имеется только два элемента, но в более общем случае имеется  $2^w$  битовых векторов с длиной  $w$ . Булева алгебра обладает многими свойствами арифметики целых чисел. Например, подобно тому, как умножение распределяется по сложению, записывается как  $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ , логическая операция  $\&$  распределяет по  $|$ , записывается как  $a \& (b | c) = (a \& b) | (a \& c)$ . Вдобавок, однако, логическая операция  $|$  распределяется по  $\&$ , поэтому можно написать  $a | (b \& c) = (a | b) \& (a | c)$ , тогда как следующее тождество  $a + (b \cdot c) = (a + b) \cdot (a + c)$  неверно для любых целых чисел.

Рассматривая операции  $\wedge$ ,  $\&$  и  $\sim$ , применяемые к битовым векторам с длиной  $w$ , мы получаем другую математическую форму, известную как *булево кольцо*. Булевы кольца имеют много общих свойств с целочисленной арифметикой. Например, одно из свойств целочисленной арифметики состоит в том, что каждое значение  $x$  имеет аддитивную инверсию  $-x$ , такую что  $x + -x = 0$ . Аналогичное свойство имеет место для булевых колец, где  $\wedge$  рассматривается как операция «сложения», но в данном случае каждый элемент является своей собственной аддитивной инверсией. То есть  $a \wedge a = 0$  для любого значения  $a$ . (Здесь мы используем 0 для представления битового вектора из одних нулей.) Все это справедливо и для одиночных битов, поскольку  $0 \wedge 0 = 1 \wedge 1 = 0$ , и для битовых векторов. Это свойство сохраняется даже после переупорядочения членов выражения, поэтому  $(a \wedge b) \wedge a = b$ . Это свойство приводит к некоторым интересным результатам и хитростям, которые мы исследуем в упражнении 2.10.

2.1.7. Битовые операции в C

Одной из полезных особенностей C является поддержка булевых операций с битами. На самом деле символы, использованные для представления булевых операций, применяются и в C: | – ИЛИ (OR), & – И (AND); ~ – НЕ (NOT) и ^ – ИСКЛЮЧАЮЩЕЕ ИЛИ (EXCLUSIVE-OR). Они применяются к данным любых целочисленных типов, включая перечисленные в табл. 2.3. Вот некоторые примеры вычисления выражений с данными типа char:

Выражение C	Двоичное выражение	Двоичный результат	Шестнадцатеричный результат
~0x41	~[0100 0001]	[1011 1110]	0xBE
~0x00	~[0000 0000]	[1111 1111]	0xFF
0x69 & 0x55	[0110 1001] & [0101 0101]	[0100 0001]	0x41
0x69   0x55	[0110 1001]   [0101 0101]	[0111 1101]	0x7D

Как показывают примеры, лучшим способом определения результата выражения с битовыми операциями является преобразование шестнадцатеричных аргументов в двоичные представления, выполнение операции с этими двоичными представлениями и обратное преобразование в шестнадцатеричную форму.

Упражнение 2.10 (решение в конце главы)

В качестве примера применимости свойства  $a \wedge a = 0$  к любому битовому вектору  $a$  рассмотрим следующую программу:

```
1 void inplace_swap(int *x, int *y) {
2     *y = *x ^ *y; /* Шаг 1 */
3     *x = *x ^ *y; /* Шаг 2 */
4     *y = *x ^ *y; /* Шаг 3 */
5 }
```

Уже по названию можно утверждать, что суть этой процедуры заключается в перестановке значений, хранящихся в ячейках, на которые ссылаются переменные-указатели  $x$  и  $y$ . Обратите внимание, что в отличие от обычной методики перестановки двух значений, здесь нет необходимости в третьей ячейке для временного хранения одного из значений на время перемещения другого. Такой способ не дает выигрыша в производительности; выигрыш имеет место лишь в форме интеллектуального развлечения.

Начиная с исходных значений  $a$  и  $b$  в ячейках, на которые ссылаются  $x$  и  $y$ , заполните следующую таблицу значениями, получающимися сохраненными в ячейках после каждого шага. Для демонстрации эффекта используйте свойства  $\wedge$ . Помните о том, что каждый элемент является собственной аддитивной инверсией ( $a \wedge a = 0$ ).

Шаг	*x	*y
Начальное состояние	$a$	$b$
Шаг 1	_____	_____
Шаг 2	_____	_____
Шаг 3	_____	_____

**Упражнение 2.11 (решение в конце главы)**

Вооружившись функцией `inplace_swap` из упражнения 2.10, вы решили написать код, который будет менять местами элементы массива, находящиеся на противоположных концах, двигаясь к середине.

В результате вы пришли к следующей функции:

```

1 void reverse_array(int a[], int cnt) {
2     int first, last;
3     for (first = 0, last = cnt-1;
4         first <= last;
5         first++, last--)
6         inplace_swap(&a[first], &a[last]);
7 }
```

Применив свою функцию к массиву, содержащему элементы 1, 2, 3 и 4, вы обнаружили, что новый массив, как и ожидалось, содержит элементы 4, 3, 2 и 1. Но, попробовав применить функцию к массиву с элементами 1, 2, 3, 4 и 5, вы с удивлением обнаружили, что в массиве теперь хранятся элементы 5, 4, 0, 2 и 1. Фактически код, правильно обрабатывающий массивы четной длины, при применении к массиву с нечетной длиной обнуляет средний элемент.

1. Для массива с нечетной длиной  $\text{cnt} = 2k + 1$  какие значения получат переменные `first` и `last` в последней итерации функции `reverse_array`?
2. Почему этот вызов функции `inplace_swap` обнуляет элемент массива?
3. Какое простое изменение кода `reverse_array` устранил эту проблему?

Одним из распространенных применений битовых операций является реализация операций *маскирования*, где маской называется комбинация битов, определяющая выбранное множество битов в слове. Например, маска `0xFF` (единицы в младших восьми битах) выделяет младший байт в слове. Битовая операция `x & 0xFF` вернет значение, состоящее из младшего значимого байта `x`, и обнулит все остальные байты. Например, при `x = 0x89ABCDEF` упомянутое выражение вернет `0x000000EF`. Выражение `~0` вернет маску, состоящую из одних единиц, независимо от размера представления данных. Эту маску можно также записать как `0xFFFFFFFF` для 32-разрядного типа данных `int`, но такой код не является переносимым.

**Упражнение 2.12 (решение в конце главы)**

Напишите выражения на C, использующие переменную `x` со значениями, перечисленными ниже. Ваш код должен правильно обрабатывать слова любого размера  $w \geq 8$ . Для справки мы приведем результат вычисления выражений для `x = 0x87654321` с  $w = 32$ .

1. Младший байт `x`, все остальные биты сброшены в 0. [`0x00000021`]
2. Все байты в `x`, кроме младшего, являются дополнениями исходных значений, а младший байт остается без изменений. [`0x789ABC21`]
3. Все биты в младшем байте получают единичные значения, а все остальные байты в `x` остаются без изменений. [`0x876543FF`]

**Упражнение 2.13 (решение в конце главы)**

С конца 1970-х до конца 1980-х годов очень популярным был компьютер Digital Equipment VAX. Вместо инструкций булевых операций «И» и «ИЛИ» он имел инструкции `bis` (bit set –

установить бит) и `bic` (bit clear – сбросить бит). Обе инструкции принимают слово данных `x` и слово-маску `m` и возвращают результат `z`, состоящий из битов в `x`, измененных в соответствии с битами в `m`. Инструкция `bis` устанавливает в 1 биты в `z`, которым соответствуют единичные биты в `m`. Инструкция `bic` сбрасывает в 0 биты в `z`, которым соответствуют единичные биты в `m`.

Чтобы увидеть, как эти операции связаны с битовыми операциями в языке C, предположим, что у нас есть функции `bis` и `bic`, реализующие операции установки и сброса битов, и мы должны использовать их для реализации функций, выполняющих битовые операции `|` и `^`, без использования каких-либо других операторов C. Добавьте недостающий код в листинг ниже. *Подсказка:* напишите выражения на языке C для операций `bis` и `bic`.

```
/* Объявления функций, реализующих операции bis и bic */
int bis(int x, int m);
int bic(int x, int m);

/* Вычислить x|y, используя только функции bis и bic */
int bool_or(int x, int y) {
    int result = _____;
    return result;
}

/* Вычислить x^y, используя только функции bis и bic */
int bool_xor(int x, int y) {
    int result = _____;
    return result;
}
```

### 2.1.8. Логические операции в C

Язык C также имеет набор *логических* операторов `||`, `&&` и `!`, соответствующих операциям «ИЛИ», «И» и «НЕ» логики высказываний. Они очень похожи на битовые операции, но действуют совершенно иначе. Логические операции рассматривают любой ненулевой аргумент как истинный, а аргумент 0 – как ложный. Они возвращают либо 1, либо 0, сообщая об истинности или ложности результата соответственно. Вот несколько примеров оценки логических выражений:

Выражение	Результат
<code>!0x41</code>	<code>0x00</code>
<code>!0x00</code>	<code>0x01</code>
<code>!!0x41</code>	<code>0x01</code>
<code>0x69 &amp;&amp; 0x55</code>	<code>0x01</code>
<code>0x69    0x55</code>	<code>0x01</code>

Обратите внимание, что поведение битовых операций совпадает с поведением их логических двойников только в одном случае – когда аргументы ограничены значениями 0 или 1.

Второе важное отличие логических операторов `&&` и `||` от их битовых двойников `&` и `|` состоит в том, что логические операторы не вычисляют второй аргумент, если результат выражения однозначно определяется первым аргументом. Например, выражение `a && 5/a` никогда не спровоцирует деление на ноль, а выражение `p && *p++` никогда не вызовет разыменование пустого указателя.

Упражнение 2.14 (решение в конце главы)

Предположим, что  $x$  и  $y$  имеют байтовые значения  $0x66$  и  $0x39$  соответственно. Заполните следующую таблицу, подставляя результаты различных выражений на C:

Выражение	Значение	Выражение	Значение
$x \& y$	_____	$x \&\& y$	_____
$x   y$	_____	$x    y$	_____
$\sim x   \sim y$	_____	$!x    !y$	_____
$x \& !y$	_____	$x \&\& \sim y$	_____

Упражнение 2.15 (решение в конце главы)

Используя только битовые и логические операции, запишите выражение на языке C, эквивалентное выражению  $x == y$ . Другими словами, результатом должна быть 1, когда  $x$  и  $y$  равны, и 0 в противном случае.

2.1.9. Операции сдвига в C

В C также имеется набор операций *сдвига* комбинаций битов влево и вправо. Для операнда  $x$ , имеющего битовое представление  $[x_{w-1}, x_{w-2}, \dots, x_0]$ , выражение  $x \ll k$  дает величину с битовым представлением  $[x_{w-k-1}, x_{w-k-2}, \dots, x_0, 0, \dots, 0]$ . То есть биты в  $x$  сдвигаются на  $k$  бит влево с удалением  $k$  старших бит и заполнением нулями  $k$  бит справа. Величина сдвига должна находиться в диапазоне от 0 до  $w - 1$ . Операции сдвига ассоциативны слева направо, то есть выражение  $x \ll j \ll k$  эквивалентно выражению  $(x \ll j) \ll k$ .

Существует аналогичная операция сдвига вправо:  $x \gg k$ , однако ее поведение имеет отличия. Вообще говоря, все машины поддерживают две формы сдвига вправо:

- *логическая*. Логический сдвиг вправо заполняет  $k$  бит слева нулями, возвращая результат  $[0, \dots, 0, x_{w-1}, x_{w-2}, \dots, x_k]$ ;
- *арифметическая*. Арифметический сдвиг вправо заполняет  $k$  бит слева исходным значением самого старшего бита, возвращая результат  $[x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_k]$ . Это правило может показаться немного странным, но, как мы увидим далее, оно полезно при работе с целочисленными данными со знаком.

Для примера в следующей таблице показан результат применения различных операций сдвига к двум разным значениям 8-разрядного аргумента  $x$ :

Операция	Значение 1	Значение 2
Аргумент $x$	[01100011]	[10010101]
$x \ll 4$	[00110000]	[01010000]
$x \gg 4$ (логический сдвиг)	[00000110]	[00001001]
$x \gg 4$ (арифметический сдвиг)	[00000110]	[11111001]

Цифры, выделенные курсивом, обозначают значения, заполняющие биты справа (сдвиг влево) или слева (сдвиг вправо). Обратите внимание, что все результаты, кроме одного, получены заполнением битов нулями. Исключение составляет случай арифметического сдвига вправо значения [10010101]. Поскольку старший бит в исходном значении равен 1, то он используется как значение заполнения.

Стандарт C не определяет точно, какой сдвиг вправо будет выполнен для целочисленных значений со знаком. К сожалению, это означает, что любой код, принимающий ту или иную форму, может впоследствии столкнуться с проблемами переносимости. Впрочем, как показывает практика, почти все комбинации компиляторов и машин используют для данных со знаком арифметический сдвиг вправо, и большинство программистов принимают это как должное. Для данных без знака, напротив, должен выполняться логический сдвиг вправо.

В отличие от C, язык Java точно определяет, как должны выполняться сдвиги вправо. Выражение  $x \gg k$  выполняет арифметический сдвиг вправо значения  $x$  на  $k$  позиций, а  $x \ggg k$  выполняет логический сдвиг.

#### Сдвиг на $k$ для больших значений $k$

Каким должен быть результат сдвига для типа данных, состоящего из  $w$  бит, на некоторое число  $k \geq w$ ? Например, каков должен быть результат вычисления следующих выражений, если предположить, что тип данных `int` имеет  $w = 32$ :

```
int      lval = 0xFEDCBA98 << 32;
int      aval = 0xFEDCBA98 >> 36;
unsigned uval = 0xFEDCBA98u >> 40;
```

Стандарты C старательно избегают указывать, что делать в таком случае. На многих машинах инструкции сдвига учитывают только младшие  $\log_2 w$  бит величины сдвига при сдвиге  $w$ -разрядного значения, поэтому величина сдвига вычисляется как  $k \bmod w$ . Например, при  $w = 32$  указанные выше три операции сдвига дадут те же результаты, что и операции сдвига на 0, 4 и 8 бит соответственно:

```
lval      0xFEDCBA98
aval      0xFFEDCBA9
uval      0x00FEDCBA
```

Однако такое поведение не гарантируется для программ на C, поэтому количество сдвигов должно быть меньше размера слова. Java, напротив, явно требует, чтобы количество сдвигов вычислялось с применением формулы, показанной выше.

#### Проблемы с приоритетом операторов в операциях сдвига

Иногда может возникнуть соблазн написать выражение  $1 \ll 2 + 3 \ll 4$ , подразумевая  $(1 \ll 2) + (3 \ll 4)$ . Однако в C это выражение эквивалентно выражению  $1 \ll (2 + 3) \ll 4$ , потому что сложение (и вычитание) имеют более высокий приоритет, чем операторы сдвига. А в результате действия правила ассоциативности слева направо, согласно которому можно расставить скобки как  $(1 \ll (2 + 3)) \ll 4$ , получается значение 512 вместо ожидаемого 52.

Неправильное определение приоритета в выражениях C – частый источник ошибок, и их очень трудно обнаружить при проверке. В случае сомнений всегда расставляйте круглые скобки!



Упражнение 2.16 (решение в конце главы)

Заполните таблицу, подставив результаты применения различных операций сдвига к од- нобайтной величине. Для этой цели лучше всего использовать двоичные представления. Преобразуйте исходные величины в двоичные значения, выполните сдвиги, после чего пре- образуйте результаты обратно в шестнадцатеричную форму. Каждый ответ должен состоять из 8 двоичных или двух шестнадцатеричных цифр.

x		x << 3		Логический x >> 2		Арифметический x >> 2	
Шестнадца- теричное	Двоичное	Двоичное	Шестнадца- теричное	Двоичное	Шестнадца- теричное	Двоичное	Шестнадца- теричное
0xC3	_____	_____	_____	_____	_____	_____	_____
0x75	_____	_____	_____	_____	_____	_____	_____
0x87	_____	_____	_____	_____	_____	_____	_____
0x66	_____	_____	_____	_____	_____	_____	_____

2.2. Целочисленные представления

В этом разделе описываются два разных способа использования битов для представле- ния целых чисел: один из этих способов позволяет представлять только неотрицатель- ные числа, а другой – отрицательные числа, положительные и ноль. Позже мы увидим, что они сильно взаимосвязаны как математическими свойствами, так и реализациями на машинном уровне. Мы также рассмотрим влияние расширения или сжатия целых чисел, чтобы их можно было уместить в представления с разной длиной.

В табл. 2.6 представлена математическая терминология, которую мы будем исполь- зовать для точного определения и описания характеристик кодирования целочислен- ных данных и выполнения операций с ними в компьютерах. Эта терминология будет поясняться в ходе обсуждения. Таблица 2.6 включена сюда для справки.

**Таблица 2.6.** Терминология для описания целочисленных данных и арифметических операций. Нижний индекс *w* обозначает количество битов в представлении данных. Столбец «Раздел» содержит название раздела, в котором определяется термин

Символ	Тип	Значение	Раздел
$B2T_w$	Функция	Двоичное в дополнительный код	2.2.3
$B2U_w$	Функция	Двоичное в целое без знака	2.2.2
$U2B_w$	Функция	Целое без знака в двоичное	2.2.2
$U2T_w$	Функция	Целое без знака в дополнительный код	2.2.4
$T2B_w$	Функция	Дополнительный код в двоичное	2.2.3
$T2U_w$	Функция	Дополнительный код в целое без знака	2.2.4
$TMin_w$	Константа	Минимальное значение дополнительного кода	2.2.3
$TMax_w$	Константа	Максимальное значение дополнительного кода	2.2.3
$UMax_w$	Константа	Максимальное значение целого без знака	2.2.2
$+^t_w$	Операция	Сложение целых в дополнительном коде	2.3.2
$+^u_w$	Операция	Сложение целых без знака	2.3.1

Символ	Тип	Значение	Раздел
$*_w^t$	Операция	Умножение целых в дополнительном коде	2.3.5
$*_w^u$	Операция	Умножение целых без знака	2.3.4
$-_w^t$	Операция	Отрицание целых в дополнительном коде	2.3.3
$-_w^u$	Операция	Отрицание целых без знака	Упражнение 2.27

### 2.2.1. Целочисленные типы

Язык C поддерживает разные *целочисленные* типы данных, представляющие ограниченные диапазоны целых чисел. Они показаны в табл. 2.7 и 2.8 с диапазонами их значений, «характерными» для 32- и 64-разрядных программ. Размер каждого типа определяется ключевым словом – его названием: `char`, `short`, `long`, а также признаком отсутствия знака (`unsigned`). По умолчанию целое может быть отрицательным. Как мы видели в табл. 2.3, размеры некоторых типов зависят от режима компиляции программы – 32- или 64-разрядного. Разные размеры позволяют представлять разные диапазоны значений. Единственный тип, размер которого зависит от машины, – это `long`. Большинство 64-разрядных программ используют 8-байтное представление для этого типа, назначая ему более широкий диапазон значений, чем 4-байтное представление в 32-разрядных программах.

Рассматривая табл. 2.7 и 2.8, обратите внимание на одну важную особенность: диапазоны не симметричны – диапазон отрицательных чисел простирается на единицу больше диапазона положительных чисел. Вы узнаете, почему это происходит, когда мы будем рассматривать представления отрицательных чисел.

**Таблица 2.7.** Типичные диапазоны целочисленных типов данных в 32-разрядных программах на C

Тип данных в C	Минимальное значение	Максимальное значение
[signed] char	-128	127
unsigned char	0	255
short	-32 768	32 767
unsigned short	0	65 535
int	-2 147 483 648	2 147 483 647
unsigned	0	4 294 967 295
long	-2 147 483 648	2 147 483 647
unsigned long	0	4 294 967 295
int32_t	-2 147 483 648	2 147 483 647
uint32_t	0	4 294 967 295
int64_t	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
uint64_t	0	18 446 744 073 709 551 615

**Таблица 2.8.** Типичные диапазоны целочисленных типов данных в 64-разрядных программах на C

Тип данных в C	Минимальное значение	Максимальное значение
[signed] char	-128	127
unsigned char	0	255
short	-32 768	32 767
unsigned short	0	65 535

Тип данных в C	Минимальное значение	Максимальное значение
int	-2 147 483 648	2 147 483 647
unsigned	0	4 294 967 295
long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long	0	18 446 744 073 709 551 615
int32_t	-2 147 483 648	2 147 483 647
uint32_t	0	4 294 967 295
int64_t	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
uint64_t	0	18 446 744 073 709 551 615

Стандарты C определяют минимальные диапазоны значений, которые должен представлять каждый тип данных. Как показано в табл. 2.9, эти диапазоны такие же или меньше, чем в типичных реализациях, представленных в табл. 2.7 и 2.8. В частности, кроме типов данных фиксированного размера, они требуют симметричности диапазонов положительных и отрицательных чисел. Также можно видеть, что тип данных `int` может быть реализован с использованием 2-байтного представления, хотя это попахивает возвратом во времена 16-разрядных машин. Также можно видеть, что тип `long` может быть реализован с использованием 4-байтного представления, что характерно для 32-разрядных программ. Типы данных фиксированного размера гарантируют неизменность диапазонов значений и что они будут в точности такими, как показано в табл. 2.7, включая асимметрию между отрицательным и положительным диапазонами.

**Таблица 2.9.** Гарантированные диапазоны целочисленных типов данных в C. Стандарты C требуют, чтобы типы данных имели диапазоны значений не уже указанных здесь

Тип данных в C	Минимальное значение	Максимальное значение
[signed] char	-127	127
unsigned char	0	255
short	-32 767	32 767
unsigned short	0	65 535
int	-32 767	32 767
unsigned	0	65 535
long	-2 147 483 647	2 147 483 647
unsigned long	0	4 294 967 295
int32_t	-2 147 483 648	2 147 483 647
uint32_t	0	4 294 967 295
int64_t	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
uint64_t	0	18 446 744 073 709 551 615

**Новичок в C?** Числа со знаком и без знака в C, C++ и Java

C и C++ поддерживают числа со знаком (по умолчанию) и без знака. Java поддерживает только числа со знаком.

**2.2.2. Представление целых без знака**

Предположим, что имеется целочисленный тип, представляющий значения из  $w$  бит. Битовый вектор записывается либо как  $\vec{x}$ , что обозначает вектор как единое целое, либо

как  $[x_{w-1}, x_{w-2}, \dots, x_0]$  с перечислением отдельных битов в векторе. Рассматривая  $\vec{x}$  как число, записанное в двоичном виде, мы получаем интерпретацию  $\vec{x}$  как числа без знака. В этом представлении каждый бит  $x_i$  имеет значение 0 или 1, причем в последнем случае он соответствует величине  $2^i$ , которая является составной частью числа. Эта интерпретация выражается как функция  $B2U_w$  («двоичное с длиной  $w$  в целое без знака»).

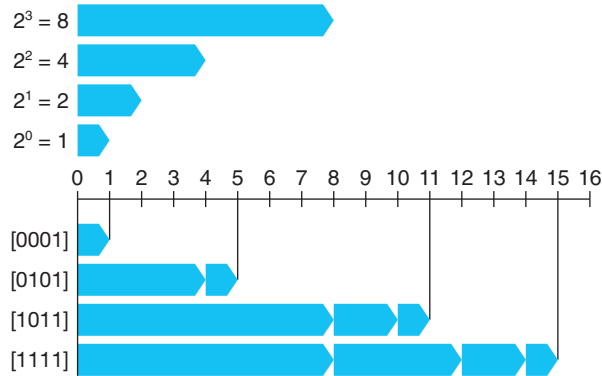
**ПРИНЦИП:** определение представления целого без знака.

Для вектора  $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$ :

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i. \quad (2.1)$$

Нотация  $\doteq$  в этом уравнении означает, что левая часть определяется как равная правой части. Функция  $B2U_w$  преобразует строки нулей и единиц длиной  $w$  в неотрицательные целые числа. Например, на рис. 2.1 показано, как  $B2U$  отображает битовые векторы в целые числа в следующих случаях:

$$\begin{aligned} B2U_4([0001]) &= 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1 \\ B2U_4([0101]) &= 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5 \\ B2U_4([1011]) &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11 \\ B2U_4([1111]) &= 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15 \end{aligned} \quad (2.2)$$



**Рис. 2.1.** Примеры целых чисел без знака для  $w = 4$ . Когда  $i$ -й бит в двоичном представлении имеет значение 1, он добавляет  $2^i$  к значению

Каждая  $i$ -я позиция в двоичном представлении на рис. 2.1 представлена синей полосой с длиной  $2^i$ , направленной вправо. Числовое значение битового вектора равно сумме длин полос, соответствующих битам со значениями 1.

Рассмотрим диапазон целых значений без знака, который можно представить с использованием  $w$  бит. Наименьшее значение (0) дает битовый вектор  $[00 \dots 0]$ , а наибольшее – битовый вектор  $[11 \dots 1]$ , которому соответствует целочисленное значение  $UMax_w \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$ . Используя в качестве примера 4-разрядное представление, мы получаем  $UMax_4 = B2U_4([1111]) = 2^4 - 1 = 15$ . Таким образом, функцию  $B2U_w$  можно определить как отображение  $B2U_w: \{0, 1\}^w \rightarrow \{0, \dots, UMax_w\}$ .

Двоичное представление без знака имеет важное свойство, заключающееся в том, что каждое число от 0 до  $2^{w-1}$  имеет уникальное представление в виде  $w$ -разрядного значения. Например, существует только одно представление десятичного значения 11 как 4-разрядного целого числа без знака, а именно [1011]. Мы выделили это как математический принцип. Эти принципы мы сначала будем формулировать, а потом объяснять.

**ПРИНЦИП:** уникальность представления чисел без знака.

Функция  $B2U_w$  является биекцией (взаимно однозначным соответствием).

Математическим термином *биекция* обозначается функция  $f$ , которая действует в двух направлениях: она отображает значение  $x$  в значение  $y$ , где  $y = f(x)$ , но также действует в обратную сторону, потому что для каждого  $y$  существует уникальное значение  $x$  такое, что  $f(x) = y$ . Это записывается с использованием нотации, обратной функции  $f^{-1}$ , например  $x = f^{-1}(y)$ . Функция  $B2U_w$  отображает каждый битовый вектор с длиной  $w$  в уникальное число от 0 до  $2^w - 1$  и имеет обратную функцию, которую мы называем  $U2B_w$  («unsigned to binary» – из целого без знака в двоичное), отображающую каждое число в диапазоне от 0 до  $2^w - 1$  в уникальную комбинацию из  $w$  бит.

#### Упражнение 2.17 (решение в конце главы)

Предположив, что  $w = 4$ , каждой возможной шестнадцатеричной цифре можно присвоить числовую величину, интерпретируя эту цифру как число без знака или как число в дополнительном коде. Заполните следующую таблицу в соответствии с этими интерпретациями, суммируя ненулевые степени 2, как в уравнениях (2.1) и (2.2):

$\vec{x}$			
Шестнадцатеричное	Двоичное	$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
0xE	[1110]	$2^3 + 2^2 + 2^1 = 14$	$-2^3 + 2^2 + 2^1 = -2$
0x0	_____	_____	_____
0x5	_____	_____	_____
0x8	_____	_____	_____
0xD	_____	_____	_____
0xF	_____	_____	_____

### 2.2.3. Представление в дополнительном коде

Во многих приложениях необходимы также отрицательные значения. Наиболее распространенное представление чисел со знаком в компьютерах – *дополнительный код* (т. е. форма представления с дополнением до двух). Дополнительный код отличается способом интерпретации самого старшего бита как имеющего отрицательный вес. Мы выразим эту интерпретацию как функцию  $B2T_w$  («binary to two's complement» – двоичное в дополнительный код):

**ПРИНЦИП:** определение представления в дополнительном коде.

Для вектора  $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$ :

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i. \quad (2.3)$$

### Еще о целочисленных типах фиксированного размера

Для некоторых программ важно, чтобы типы данных кодировались с использованием представлений определенных размеров. Например, программам, обменивающимся данными через интернет с использованием стандартного протокола, важно использовать типы данных, совместимые с указанными в протоколе. Мы видели, что некоторые типы данных в C, особенно `long`, имеют разные диапазоны на разных машинах, и на самом деле стандарты C определяют только минимальные, но не точные диапазоны для типов данных. Мы можем выбрать типы данных, которые на большинстве машин совместимы с представлениями, определяемыми стандартами, но это не дает гарантий переносимости.

Мы уже встречались с 32- и 64-разрядными версиями целочисленных типов фиксированного размера (табл. 2.3); они являются частью более крупного класса типов данных. Стандарт ISO C99 определяет этот класс целочисленных типов в файле `stdint.h`. В этом файле объявляется набор типов данных в форме `intN_t` и `uintN_t` для представления  $N$ -разрядных целых чисел со знаком и без знака для разных значений  $N$ . Точные значения  $N$  зависят от реализации, но большинство компиляторов поддерживают значения 8, 16, 32 и 64. Благодаря этому можно однозначно объявить 16-разрядную переменную для хранения целого без знака, используя тип `uint16_t`, а 32-разрядную переменную для хранения целого со знаком, используя тип `int32_t`.

Помимо этих типов данных имеется также набор макросов, определяющих минимальные и максимальные значения чисел для каждого  $N$ . Они имеют имена в форме `INTN_MIN`, `INTN_MAX` и `UINTN_MAX`.

Процедура форматированного вывода при применении к типам фиксированного размера требует использования макросов, которые разворачиваются в строки формата в зависимости от системы. Так, например, значения переменных  $x$  и  $y$  типа `int32_t` и `uint64_t` можно вывести следующим вызовом `printf`:

```
printf ("x = %" PRIid32 ", y = %" PRIu64 "\n", x, y);
```

При компиляции 64-разрядной программы макрос `PRIid32` разворачивается в строку `"d"`, а `PRIu64` – в пару строк `"l"` `"u"`. Когда препроцессор C встречает последовательность строковых констант, разделенных только пробелами (или другими пробельными символами), то объединяет их вместе. В результате приведенный выше вызов `printf` превращается в

```
printf ("x = %d, y = %lu\n", x, y);
```

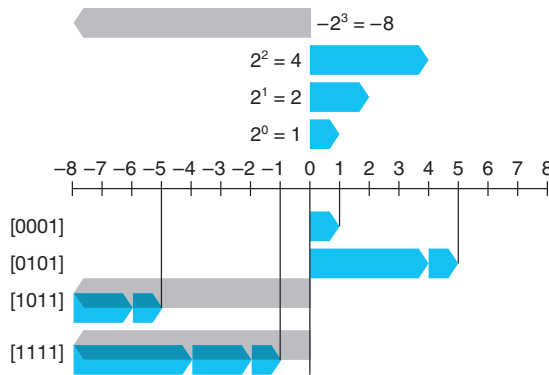
Использование макросов гарантирует создание правильной строки формата независимо от флагов компиляции.

Самый старший бит  $x_{w-1}$  также называется *битом знака*. Его «вес» равен  $-2^{w-1}$ , т. е. задает отрицательный вес следующего за ним представления без знака. Когда бит знака установлен в 1, представленное значение отрицательно, а когда установлен в 0, значение неотрицательно. Для примера на рис. 2.2 показано отображение битовых векторов, производимое функцией  $B2T$  для следующих случаев:

$$\begin{aligned}
 B2T_4([0001]) &= -0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1 \\
 B2T_4([0101]) &= -0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5 \\
 B2T_4([1011]) &= -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 0 + 2 + 1 = -5 \\
 B2T_4([1111]) &= -1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 4 + 2 + 1 = -1
 \end{aligned} \tag{2.4}$$

На рис. 2.2 мы показываем, что знаковый бит имеет отрицательный вес, изобразив его как серую полосу, направленную влево. Числовое значение битового вектора опре-

деляется как сумма возможной серой полосы, направленной влево, и синих полосок, направленных вправо.



**Рис. 2.2.** Примеры чисел в дополнительном коде для  $w = 4$ .

Бит 3 играет роль знакового бита; когда он имеет значение 1, он вносит в общую сумму величину  $-2^3 = -8$ . Это взвешивание изображено как серая полоса, направленная влево

Как видите, комбинации битов идентичны на рис. 2.1 и 2.2 (а также в уравнениях 2.2 и 2.4), но числовые значения различаются, когда старший бит равен 1, потому что в одном случае он имеет вес +8, а в другом – вес -8.

Рассмотрим диапазон значений чисел в дополнительном коде, который можно представить с использованием  $w$  бит. Наименьшее значение дает битовый вектор  $[10 \dots 0]$  (установлен бит с отрицательным весом и сброшены все остальные), которое выражается как  $TMin_w \doteq -2^{w-1}$ . Наибольшее значение дает битовый вектор  $[01 \dots 1]$  (сброшен бит с отрицательным весом и установлены все остальные), которое выражается как  $TMax_w \doteq \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$ . Используя в качестве примера 4-разрядное представление, мы получаем  $TMin_4 = B2T_4([1000]) = -2^3 = -8$  и  $TMax_4 = B2T_4([0111]) = 2^2 + 2^1 + 2^0 = 4 + 2 + 1 = 7$ .

Как видите,  $B2T_w$  отображает комбинацию битов с длиной  $w$  в число между  $TMin_w$  и  $TMax_w$  и определяется как отображение  $B2T_w: \{0, 1\}^w \rightarrow \{TMin_w, \dots, TMax_w\}$ . Как было показано в обсуждении представления целых чисел без знака, каждое число в пределах заданного диапазона имеет уникальное представление в виде  $w$ -разрядного вектора. Это приводит нас к принципу уникальности чисел в дополнительном коде, аналогичному принципу уникальности чисел без знака.

**ПРИНЦИП:** уникальность представления чисел в дополнительном коде.

Функция  $B2T_w$  является биекцией (взаимно однозначным соответствием).

Мы определяем функцию  $T2B_w$  («two's complement to binary» – дополнительный код в двоичное) как обратную функции  $B2T_w$ . То есть для числа  $x$ , такого что  $TMin_w \leq x \leq TMax_w$ , функция  $T2B_w(x)$  дает уникальную  $w$ -разрядную комбинацию, представляющую  $x$ .

В табл. 2.10 показаны некоторые важные комбинации битов и их численные значения для слов разной длины. Первые три определяют диапазоны представимых целых чисел в терминах значений  $UMax_w$ ,  $TMin_w$  и  $TMax_w$ . Мы будем часто обращаться к этим трем особым значениям в последующем обсуждении. Мы опустим нижний индекс  $w$  и будем использовать обозначения  $Umax$ ,  $Tmin$  и  $Tmax$ , когда  $w$  можно вывести из контекста или он не является важным для обсуждения.



Таблица 2.10. Особые числа в шестнадцатеричном представлении

Значение	Размер слова <i>w</i>			
	8	16	32	64
$UMax_w$	0xFF 255	0xFFFF 65 535	0xFFFFFFFF 4 294 967 295	0xFFFFFFFFFFFFFFFF 18 446 744 073 709 551 615
$TMin_w$	0x80 -128	0x8000 -32 768	0x80000000 -2 147 483 648	0x8000000000000000 -9 223 372 036 854 775 808
$TMax_w$	0x7F 127	0x7FFF 32 767	0x7FFFFFFF 2 147 483 647	0x7FFFFFFFFFFFFFFF 9 223 372 036 854 775 807
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

Здесь стоит особо выделить несколько моментов. Во-первых, как показано в табл. 2.6 и 2.7, диапазоны значений в дополнительном коде асимметричны:  $|TMin| = |TMax| + 1$ , т. е. для  $TMin$  нет положительного эквивалента. Далее мы увидим, что это приводит к проявлению некоторых особенных свойств арифметики чисел в дополнительном коде и может оказаться источником небольших программных ошибок. Эта асимметрия возникает из-за того, что половина комбинаций битов (в которых бит знака установлен в 1) представляют отрицательные числа, а половина (в которых бит знака сброшен в 0) представляют неотрицательные числа. Поскольку 0 – неотрицательное число, это означает, что положительных чисел в дополнительном коде на одно меньше, чем отрицательных.

Во-вторых, максимальное значение без знака несколько превышает удвоенное максимальное значение в дополнительном коде:  $UMax = 2TMax + 1$ . Все комбинации битов, обозначающие отрицательные числа в дополнительном коде, становятся положительными значениями в представлении без знака.

### Упражнение 2.18 (решение в конце главы)

В главе 3 будут рассматриваться листинги, сгенерированные *дисассемблером* – программой, преобразующей выполняемый программный файл в удобочитаемую форму ASCII. Такие файлы содержат множество шестнадцатеричных чисел, обычно представляющих величины в форме дополнительного кода. Умение распознавать эти числа и понимать их значение (например, отрицательные они или положительные) является важным профессиональным навыком для любого программиста.

Для строк, отмеченных в следующем листинге буквами от A до I (справа), преобразуйте шестнадцатеричные значения, показанные (в 32-разрядном дополнительном коде) справа от названий инструкций (sub, mov и add), в их десятичные эквиваленты:

```

4004d0: 48 81 ec e0 02 00 00    sub    $0x2e0,%rsp      A.
4004d7: 48 8b 44 24 a8          mov    -0x58(%rsp),%rax  B.
4004dc: 48 03 47 28             add    0x28(%rdi),%rax   C.
4004e0: 48 89 44 24 d0          mov    %rax,-0x30(%rsp)  D.
4004e5: 48 8b 44 24 78          mov    0x78(%rsp),%rax   E.
4004ea: 48 89 87 88 00 00 00    mov    %rax,0x88(%rdi)   F.
4004f1: 48 8b 84 24 f8 01 00    mov    0x1f8(%rsp),%rax  G.
4004f8: 00
4004f9: 48 03 44 24 08          add    0x8(%rsp),%rax
4004fe: 48 89 84 24 c0 00 00    mov    %rax,0xc0(%rsp)   H.
400505: 00
400506: 48 8b 44 d4 b8          mov    -0x48(%rsp,%rdx,8),%rax  I.

```

В табл. 2.10 также показано представление констант  $-1$  и  $0$ . Обратите внимание, что  $-1$  имеет то же битовое представление, что и  $UMax$ , – в виде строки из одних единиц. Числовое значение  $0$  в обоих представлениях имеет вид строки из одних нулей.

Стандарт C не требует представления целых чисел со знаком в форме дополнительного кода, однако такая реализация используется практически на всех машинах. Для поддержания переносимости кода не следует делать никаких конкретных предположений о диапазоне представимых величин, кроме тех, что перечислены в табл. 2.9, либо о представлении чисел со знаком. С другой стороны, многие программы написаны с учетом представления чисел со знаком в виде дополнительного кода и «типичных» диапазонов, перечисленных в табл. 2.7 и 2.8, и эти программы переносимы на широкий спектр машин и компиляторов. Файл `<limits.h>` в библиотеке C определяет множество констант, разграничивающих диапазоны разных типов целочисленных данных для каждой конкретной машины, на которой выполняется компилятор. Например, он определяет константы `INT_MAX`, `INT_MIN` и `UINT_MAX`, описывающие диапазоны целых чисел со знаком и без знака. Для машин, использующих дополнительный код, где тип данных `int` имеет  $n$  бит, эти константы соответствуют величинам  $TMax_w$ ,  $TMin_w$  и  $UMax_w$ .

В Java используется довольно специфический стандарт, определяющий диапазоны и представления целочисленных типов. Он требует использовать представление в дополнительном коде и задает точные диапазоны для 64-разрядного случая (табл. 2.8). В Java однобайтный тип данных называется `byte` вместо `char`. Такая детальность требований предназначена для того, чтобы программы на Java могли вести себя одинаково, независимо от аппаратной архитектуры или операционной системы.

Чтобы лучше понять представление в дополнительном коде, рассмотрим следующий пример:

```
1 short x = 12345;
2 short mx = -x;
3
4 show_bytes((byte_pointer) &x, sizeof(short));
5 show_bytes((byte_pointer) &mx, sizeof(short));
```

При выполнении на машине с прямым (big endian) порядком следования байтов этот код выведет `30 39` и `cf c7`, сообщая, что `x` имеет шестнадцатеричное представление `0x039`, а `mx` – шестнадцатеричное представление `0xcfc7`. Если развернуть их в двоичное представление, то получатся комбинации битов `[0011000000111001]` для `x` и `[1100111111000111]` для `mx`. Как показано в табл. 2.11, для этих двух комбинаций уравнение (2.3) дает значения  $12\,345$  и  $-12\,345$ .

**Таблица 2.11.** Представление в дополнительном коде чисел  $12\,345$  и  $-12\,345$  и числа без знака  $53\,191$ . Обратите внимание, что последние два имеют идентичные битовые представления

12 345			-12 345		53 191	
Вес	Бит	Значение	Бит	Значение	Бит	Значение
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64

12 345			-12 345		53 191	
Вес	Бит	Значение	Бит	Значение	Бит	Значение
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1024	0	0	1	1024	1	1024
2048	0	0	1	2048	1	2048
4096	1	4096	0	0	0	0
8192	1	8192	0	0	0	0
16 384	0	0	1	16 384	1	16 384
±32 768	0	0	1	-32 768	1	32 768
Итого		12 345		-12 345		53 191

### Альтернативные представления чисел со знаком

Существует два других стандартных представления чисел со знаком.

Обратный код (дополнение до единиц). То же самое, что и дополнительный код, за исключением того, что самый старший бит имеет вес  $-(2^{w-1} - 1)$ , а не  $-2^{w-1}$ :

$$B2O_w(\vec{x}) \doteq -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i.$$

Прямой код (величина знака). Самый старший бит в прямом коде определяет вес остальных битов – положительный или отрицательный:

$$B2S_w(\vec{x}) \doteq (-1)^{x_{w-1}} \cdot \left( \sum_{i=0}^{w-2} x_i 2^i \right).$$

Оба этих представления обладают интересным свойством: они определяют разные способы представления числа 0. В обоих представлениях [00 ... 0] интерпретируется как +0. Величину -0 можно представить в прямом коде как [10 ... 0], а в обратном коде – как [11 ... 1]. Несмотря на то что машины, основанные на представлении в прямом коде, созданы достаточно давно, почти на всех современных машинах используется представление в дополнительном коде. Далее вы увидите, что прямой код используется для представления чисел с плавающей точкой.

Обратите внимание на различие в терминах «дополнение до двух» (дополнительный код) и «дополнение до единиц» (обратный код), где слово «два» используется в единственном числе, а «единиц» – во множественном. Термин «дополнение до двух» возник из того факта, что для неотрицательного  $x$   $w$ -разрядное представление  $-x$  вычисляется как  $2^w - x$  (одна двойка). Термин «дополнение до единиц» проистекает из свойства вычисления  $-x$  в этой нотации как [111...1] –  $x$  (множество единиц).

## 2.2.4. Преобразования между числами со знаком и без знака

Язык C позволяет выполнять преобразования между разными числовыми типами данных. Например, предположим, что переменная  $x$  объявлена как `int`, а  $u$  – как `unsigned`. Выражение `(unsigned) x` преобразует значение  $x$  в значение без знака, а `(int) u` – значе-

ние  $u$  в целое число со знаком. Какими должны быть результаты таких преобразований? С математической точки зрения можно представить несколько различных соглашений. Очевидно, что любое значение, которое можно представить в обеих формах, должно сохраниться. С другой стороны, преобразование отрицательного значения в значение без знака может дать ноль. Преобразование значения без знака, которое слишком велико для представления в дополнительном коде, может дать  $TMax$ . Однако в большинстве реализаций C ответ на этот вопрос основан на уровне битового представления, а не числового.

Например, рассмотрим следующий код:

```
1 short   int    v = -12345;
2 unsigned short uv = (unsigned short) v;
3 printf("v = %d, uv = %u\n", v, uv);
```

Если запустить его на машине, где используется дополнительный код, то он сгенерирует следующий вывод:

```
v = -12345, uv = 53191
```

Здесь мы видим, что преобразование (приведение типа) заключается в сохранении идентичных значений битов и в изменении способа интерпретации. В табл. 2.11 мы видели, что 16-разрядное представление числа  $-12\,345$  в дополнительном коде идентично 16-разрядному беззнаковому представлению числа  $53\,191$ . Преобразование из типа `short` в тип `unsigned short` изменяет значение числа, но не изменяет битового представления.

Теперь рассмотрим следующий код:

```
1 unsigned u = 4294967295u;    /* UMax */
2 int      tu = (int) u;
3 printf("u = %u, tu = %d\n", u, tu);
```

Если запустить его на машине, где используется дополнительный код, то он сгенерирует следующий вывод:

```
u = 4294967295, tu = -1
```

Как показано в табл. 2.10 (для 32-разрядного слова), комбинации битов, представляющие число  $4\,294\,967\,295$  ( $UMax_{32}$ ) в форме без знака и  $-1$  в дополнительном коде, идентичны. При преобразовании типа `unsigned` в `int` базовое битовое представление остается неизменным.

Это общее правило преобразования между числами со знаком и без знака с одинаковым размером слова в большинстве реализаций C – числовые значения могут изменяться, но комбинации битов – никогда. Давайте сформулируем эту идею на языке математики. Мы определили функции  $U2B_w$  и  $T2B_w$ , которые отображают числа в их битовые представления либо в форме без знака, либо в форме дополнительного кода. То есть для целого числа  $x$  в диапазоне  $0 \leq x < UMax_w$  функция  $U2B_w(x)$  дает уникальное  $w$ -разрядное представление  $x$  без знака. Точно так же для  $x$  в диапазоне  $TMin_w \leq x \leq TMax_w$  функция  $T2B_w(x)$  дает уникальное  $w$ -разрядное представление  $x$  в дополнительном коде.

Теперь определим функцию  $T2U_w$  как  $T2U_w(x) \doteq B2U_w(T2B_w(x))$ . Эта функция принимает число в диапазоне от  $TMin_w$  до  $TMax_w$  и возвращает число в диапазоне от  $0$  до  $UMax_w$ , где два числа имеют идентичные битовые представления, за исключением того, что аргумент представлен в дополнительном коде, а результат – в виде числа без знака. Точно так же для  $x$  между  $0$  и  $UMax_w$  функция  $U2T_w$ , определенная как  $U2T_w(x) \doteq B2T_w(U2B_w(x))$ , дает число, имеющее такое же представление в дополнительном коде, что и в представлении  $x$  в форме без знака.

Продолжая предыдущие примеры, мы видим из табл. 2.11, что  $T2U_{16}(-12\,345) = 53\,191$ , а  $U2T_{16}(53\,191) = -12\,345$ . То есть 16-разрядная комбинация, которая в шестнадцатеричном виде выглядит как  $0 \times \text{CFC7}$ , является представлением числа  $-12\,345$  в дополнительном коде и представлением числа без знака  $53\,191$ . Также обратите внимание, что  $12\,345 + 53\,191 = 65\,536 = 2^{16}$ . Это свойство обобщается на взаимосвязь между двумя числовыми значениями, представленными (в дополнительном коде и в форме без знака) данной комбинацией битов. Точно так же из табл. 2.10 мы видим, что  $T2U_{32}(-1) = 4\,294\,967\,295$  и  $U2T_{32}(4\,294\,967\,295) = -1$ . То есть  $UMax$  имеет такое же битовое представление в форме без знака, что и  $-1$  в форме дополнительного кода. Здесь также можно заметить взаимосвязь между этими двумя числами:  $1 + UMax_w = 2^w$ .

Таким образом, мы видим, что функция  $T2U$  описывает преобразование числа в форме дополнительного кода в его аналог без знака, а  $U2T$  – преобразование в обратную сторону. Они описывают результат преобразования между этими типами данных в большинстве реализаций C.

Связь между значениями в дополнительном коде и в представлении без знака для данной комбинации битов, которую мы видели на нескольких примерах, можно выразить как свойство функции  $T2U$ :

**ПРИНЦИП:** преобразование из представления в дополнительном коде в представление без знака.

Для  $x$  такого, что  $TMin_w \leq x \leq TMax_w$ :

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases}. \quad (2.5)$$

Например, мы видели, что  $T2U_{16}(-12\,345) = -12\,345 + 2^{16} = 53\,191$ , а также что  $T2U_w(-1) = -1 + 2^w = UMax_w$ .

Это свойство можно вывести, сравнивая уравнения 2.1 и 2.3.

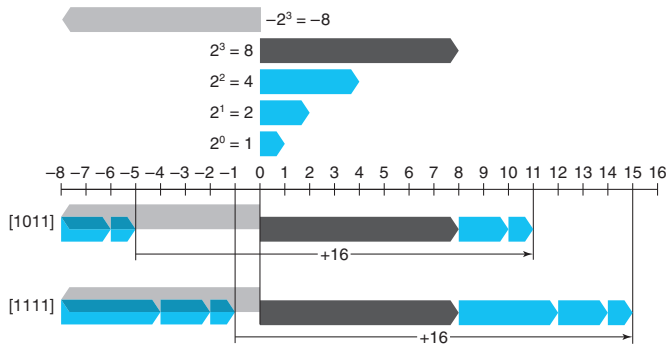
**ВЫВОД:** преобразование из дополнительного кода в представление без знака.

Сравнивая уравнения 2.1 и 2.3, можно заметить, что для комбинации битов  $\vec{x}$ , если вычислить разность  $B2U_w(\vec{x}) - B2T_w(\vec{x})$ , взвешенные суммы битов от 0 до  $w-2$  будут компенсировать друг друга, оставляя значение  $B2U_w(\vec{x}) - B2T_w(\vec{x}) = x_{w-1}(2^{w-1} - -2^{w-1}) = x_{w-1}2^w$ . Это дает соотношение  $B2U_w(\vec{x}) - B2T_w(\vec{x}) = 2^w$ . Таким образом, мы получаем

$$B2U_w(T2B_w(x)) = T2U_w(x) = x + x_{w-1}2^w. \quad (2.6)$$

В представлении  $x$  в дополнительном коде бит  $x_{w-1}$  определяет знак числа, давая нам два случая из уравнения 2.5.

Для примера на рис. 2.3 показано, как функции  $B2U$  и  $B2T$  присваивают значения комбинациям битов для  $w = 4$ . В случае использования дополнительного кода старший бит служит битом знака, который изображен как серая полоса, направленная влево. Для случая представления без знака этот бит имеет положительный вес, который показан как черная полоса, направленная вправо. При переходе от представления в дополнительном коде к представлению без знака старший бит меняет свой вес с  $-8$  на  $+8$ . Как следствие отрицательные значения в дополнительном коде увеличиваются на  $2^4 = 16$  в представлении без знака. Таким образом,  $-5$  превращается в  $+11$ , а  $-1$  – в  $+15$ .



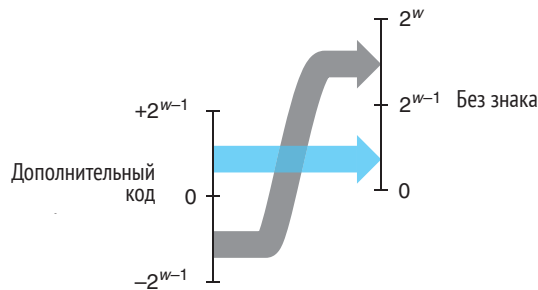
**Рис. 2.3.** Сравнение представлений без знака и в дополнительном коде для  $w = 4$ . В дополнительном коде вес самого старшего бита равен  $-8$ , а в представлении без знака  $+8$ , что дает чистую разницу в 16

**Упражнение 2.19 (решение в конце главы)**

Используя таблицу, заполненную при решении упражнения 2.17, заполните следующую таблицу, описывающую функцию  $T2U_4$ :

$x$	$T2U_4(x)$
-8	_____
-3	_____
-2	_____
-1	_____
0	_____
5	_____

На рис. 2.4 проиллюстрировано поведение функции  $T2U$ : при проецировании числа со знаком на его эквивалент без знака отрицательные числа преобразуются в большие положительные числа, тогда как неотрицательные числа остаются без изменений.



**Рис. 2.4.** Преобразование числа из представления в дополнительном коде в представление без знака. Функция  $T2U$  преобразует отрицательные числа в большие положительные числа

**Упражнение 2.20 (решение в конце главы)**

Объясните, как уравнение (2.5) применяется к записям в таблице, созданной в упражнении 2.19.

Двигаясь в обратном направлении, можно вывести отношение между числом  $u$  без знака и его эквивалентом со знаком  $U2T_w(u)$ .

**ПРИНЦИП:** преобразование из представления без знака в представление в дополнительном коде.

Для  $u$  такого, что  $0 \leq u \leq UMax_w$ :

$$U2T_w(u) = \begin{cases} u, & u \leq TMax_w \\ u + 2^w, & u > TMax_w \end{cases} \quad (2.7)$$

Этот принцип можно обосновать следующим образом:

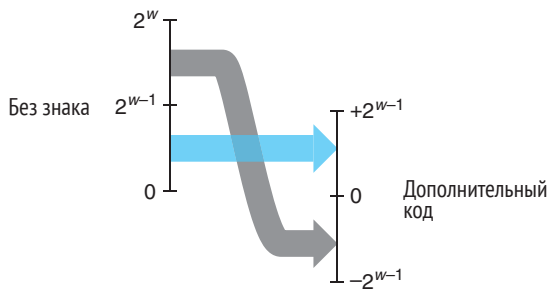
**ВЫВОД:** преобразование представления без знака в представление в дополнительном коде.

Пусть  $\tilde{u} = U2Bw(u)$ . Этот битовый вектор также является представлением  $U2T_w(u)$  в дополнительном коде. Уравнения 2.1 и 2.3 можно объединить и получить:

$$U2T_w(u) = -u_{w-1}2^w + u. \quad (2.8)$$

В представлении  $u$  без знака бит  $u_{w-1}$  определяет, больше ли  $u$ , чем  $TMax_w = 2^{w-1} - 1$ , что дает два случая из уравнения 2.7.

Поведение функции  $U2T$  проиллюстрировано на рис. 2.5. Для маленьких ( $\leq TMax_w$ ) чисел преобразование из представления без знака в представление со знаком сохраняет числовую величину. Для больших величин ( $> TMax_w$ ) число преобразуется в отрицательную величину.



**Рис. 2.5.** Преобразование числа из представления без знака в представление в дополнительном коде. Функция  $U2T$  преобразует числа больше  $2^{w-1}$  в отрицательные числа

Итак, мы рассмотрели результаты преобразования в обоих направлениях между представлениями чисел без знака и в дополнительном коде. Для значений в диапазоне  $0 < x < TMax_w$  имеем  $T2U_w(x) = x$  и  $U2T_w(x) = x$ . То есть числа в этом диапазоне имеют идентичные представления без знака и в дополнительном коде. Для величин вне это-



го диапазона преобразования либо добавляют, либо вычитают  $2^w$ . Например,  $T2U_w(-1) = -1 + 2^w = UMax_w$  – ближайшее к нулю отрицательное число, отображается в самое большое число без знака. В другом крайнем случае мы видим, что  $T2U_w(TMin_w) = -2^{w-1} + 2^w = 2^{w-1} = TMax_w + 1$  – наибольшее по величине отрицательное число отображается в число без знака как раз за пределами диапазона положительных чисел в дополнительном коде. Исследуя пример в табл. 2.11, можно заметить, что  $T2U_{16}(-12\ 345) = 65\ 536 + -12\ 345 = 53\ 191$ .

### 2.2.5. Числа со знаком и без знака в C

Как показано в табл. 2.7 и 2.8, C поддерживает арифметику обоих чисел, со знаком и без знака, для всех целочисленных типов данных. Хотя стандарт C не определяет конкретного представления чисел со знаком, почти на всех машинах используется дополнительный код. Вообще говоря, большинство чисел имеют знак по умолчанию. Например, при объявлении такой константы, как 12345 или `0x1A2B`, считается, что число имеет знак. Для создания константы без знака необходимо добавить в конец символ U или u (например, 12345U или `0x1A2Bu`).

В C возможно преобразование из величин без знака в величины со знаком. Хотя C не определяет точно, как должно выполняться такое преобразование, большинство систем следуют такому правилу: битовое представление числа не меняется. Таким образом, для преобразования числа без знака в число со знаком используется функция  $U2T_w$ , а для преобразования числа со знаком в число без знака – функция  $T2U_w$ , где  $w$  – число битов в типе данных.

Преобразования могут выполняться явно, как в следующем примере:

```
1 int tx, ty;
2 unsigned ux, uy;
3
4 tx = (int) ux;
5 uy = (unsigned) ty;
```

Или неявно, когда выражение одного типа присваивается переменной другого типа, как в следующем коде:

```
1 int tx, ty;
2 unsigned ux, uy;
3
4 tx = ux; /* Преобразуется в значение со знаком */
5 uy = ty; /* Преобразуется в значение без знака */
```

При выводе числовых величин с помощью `printf` необходимо использовать директивы `%d`, `%u` и `%x`, чтобы вывести число как десятичное со знаком, десятичное без знака и шестнадцатеричное соответственно. Обратите внимание, что `printf` не использует никакой информации о типе, и поэтому можно вывести значение типа `int` с директивой `%u` или значение типа `unsigned` с директивой `%d`. Например, рассмотрим следующий код:

```
1 int x = -1;
2 unsigned u = 2147483648; /* 2 в 31-й степени */
3
4 printf("x = %u = %d\n", x, x);
5 printf("u = %u = %d\n", u, u);
```

Если эту программу скомпилировать в 32-разрядном формате, то она выведет:

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

В обоих случаях `printf` сначала выведет слово как число без знака, а затем – как со знаком. Здесь можно видеть, как действуют процедуры преобразования  $T2U_{32}(-1) = UMax_{32} = 2^{32} - 1$  и  $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$ .

Такое, возможно, малопонятное поведение возникает вследствие особенностей обработки в С выражений, содержащих комбинацию величин со знаком и без знака. Когда операция применяется к операндам, один из которых является числом со знаком, а другой – числом без знака, то С неявно преобразует аргумент со знаком в аргумент без знака и выполняет операцию, предполагая, что числа неотрицательные. Как будет показано далее, это правило мало отличается от поведения стандартных арифметических операций, но приводит к малопонятным результатам для операторов сравнения, таких как `<` и `>`. В табл. 2.12 показаны некоторые примеры выражений сравнения и их результаты, когда тип `int` имеет 32-разрядное представление в дополнительном коде. Рассмотрим сравнение `-1 < 0U`. Поскольку второй операнд – число без знака, первый неявно преобразуется в число без знака, поэтому данное выражение эквивалентно сравнению `4294967295U < 0U` (напомним, что  $T2U_w(-1) = UMax_w$ ), которое, разумеется, возвращает `false`. Другие случаи тоже можно проанализировать подобным способом.

**Таблица 2.12.** Результат действия правил преобразования типов в С. Неочевидные случаи отмечены знаком «\*». Когда один из сравниваемых операндов – число без знака, то другой операнд неявно преобразуется в значение без знака. Смотрите приложение в интернете DATA:TMIN, где поясняется, почему мы записываем  $TMin_{32}$  как `-2147483647-1`

Выражение			Тип	Результат
<code>0</code>	<code>==</code>	<code>0U</code>	Без знака	1
<code>-1</code>	<code>&lt;</code>	<code>0</code>	Со знаком	1
<code>-1</code>	<code>&lt;</code>	<code>0U</code>	Без знака	<code>0 *</code>
<code>2147483647</code>	<code>&gt;</code>	<code>-2147483647-1</code>	Со знаком	1
<code>2147483647U</code>	<code>&gt;</code>	<code>-2147483647-1</code>	Без знака	<code>0 *</code>
<code>2147483647</code>	<code>&gt;</code>	<code>(int) 2147483648U</code>	Со знаком	<code>1 *</code>
<code>-1</code>	<code>&gt;</code>	<code>-2</code>	Со знаком	1
<code>(unsigned) -1</code>	<code>&gt;</code>	<code>-2</code>	Без знака	1

### Упражнение 2.21 (решение в конце главы)

Предполагая, что выражения оцениваются на 32-разрядной машине, использующей арифметику в дополнительном коде, заполните следующую таблицу, описывая результаты преобразований и операций сравнения, как это сделано в табл. 2.12.

Выражение	Тип	Результат
<code>-2147483647-1 == 2147483648U</code>	<u>                    </u>	<u>                    </u>
<code>-2147483647-1 &lt; 2147483648</code>	<u>                    </u>	<u>                    </u>
<code>-2147483647-1U &lt; 2147483648</code>	<u>                    </u>	<u>                    </u>
<code>-2147483647-1 &lt; -2147483648</code>	<u>                    </u>	<u>                    </u>
<code>-2147483647-1U &lt; -2147483648</code>	<u>                    </u>	<u>                    </u>

**Приложение в интернете DATA:MIN.** Запись  $TMin$  в C

В табл. 2.12 и в упражнении 2.21 мы предусмотрительно заменили  $TMin_{32}$  на  $-2147483647-1$ . Почему бы просто не написать  $-2147483648$  или  $0x80000000$ ?

Если заглянуть в заголовочный файл `limits.h`, то можно увидеть, что в нем используется тот же метод, что использовали мы для записи  $TMin_{32}$  и  $TMax_{32}$ :

```
/* Минимальное и максимальное значения для типа int со знаком. */
#define INT_MAX    2147483647
#define INT_MIN    (-INT_MAX - 1)
```

Записывать  $TMin_{32}$  таким необычным способом нас заставляет асимметрия представления в дополнительном коде и правила преобразования C. Чтобы понять эту проблему, нужно погрузиться в темные пучины стандартов языка C, однако это поможет оценить некоторые тонкости целочисленных типов данных и их представлений.

## 2.2.6. Расширение битового представления числа

Одной из обычных операций является преобразование между целыми числами с разной длиной слова, сохраняющее числовое значение. Конечно, это невозможно, когда целевой тип данных слишком мал, для представления нужной величины. Однако преобразование типа меньшего размера в тип большего размера всегда должно быть возможно.

Для преобразования числа без знака в больший тип данных достаточно просто добавить в представление ведущие нули. Такая операция называется *дополнением нуля*.

**ПРИНЦИП:** расширение преобразования числа без знака дополнением нулями.

Определим битовые векторы  $\vec{u} = [u_{w-1}, u_{w-2}, \dots, u_0]$  с длиной  $w$  и  $\vec{u}' = [0, \dots, 0, u_{w-1}, u_{w-2}, \dots, u_0]$  с длиной  $w'$ , где  $w' > w$ . Тогда  $B2U_w(\vec{u}) = B2U_{w'}(\vec{u}')$ .

Нетрудно показать, что этот принцип следует непосредственно из определения представления чисел без знака, заданного уравнением 2.1.

Для преобразования числа в дополнительном коде в больший тип данных применяется правило *расширения знакового разряда* с добавлением в представление копий самого старшего бита. Далее мы выделим жирным знаковый бит  $x_{w-1}$ , чтобы показать, как происходит расширение знакового разряда.

**ПРИНЦИП:** расширение знакового разряда в представлении в дополнительном коде.

Определим битовые векторы  $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$  с длиной  $w$  и  $\vec{x}' = [x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]$  с длиной  $w'$ , где  $w' > w$ . Тогда  $B2U_w(\vec{x}) = B2U_{w'}(\vec{x}')$ .

В качестве примера рассмотрим следующий код:

```
1 short sx = -12345;          /* -12345 */
```

```

2 unsigned short usx = sx; /* 53191 */
3 int x = sx;             /* -12345 */
4 unsigned ux = usx;      /* 53191 */
5
6 printf("sx = %d:\t", sx);
7 show_bytes((byte_pointer) &sx, sizeof(short));
8 printf("usx = %u:\t", usx);
9 show_bytes((byte_pointer) &usx, sizeof(unsigned short));
10 printf("x = %d:\t", x);
11 show_bytes((byte_pointer) &x, sizeof(int));
12 printf("ux = %u:\t", ux);
13 show_bytes((byte_pointer) &ux, sizeof(unsigned));

```

Если запустить эту программу, скомпилированную в 32-разрядном режиме, на машине с прямым (big endian) порядком следования байтов, где используется представление в дополнительном коде, то она выведет:

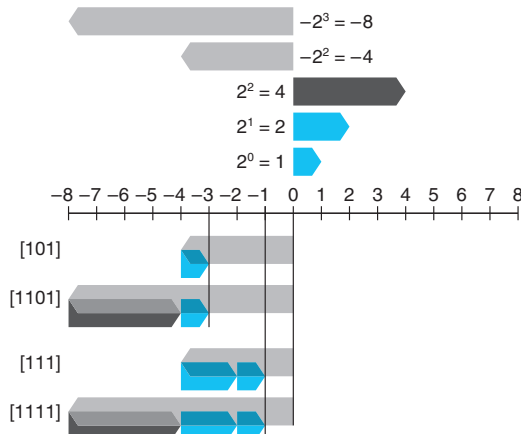
```

sx = -12345:  cf c7
usx = 53191:  cf c7
x  = -12345:  ff ff cf c7
ux  = 53191:  00 00 cf c7

```

Несмотря на то что представление  $-12\,345$  в дополнительном коде и представление  $53\,191$  в форме без знака идентичны в архитектуре с 16-разрядными словами, они различны в архитектуре с 32-разрядными словами. В частности,  $-12\,345$  имеет шестнадцатеричное представление  $0xFFFFCF7$ , тогда как  $53\,191$  – шестнадцатеричное представление  $0x000CFC7$ . Первое расширено дополнительным знаковым разрядом: 16 копий старшего бита 1, в шестнадцатеричном представлении  $0xFFFF$ , добавлены в качестве ведущих битов. Последнее расширено 16 ведущими нулями, в шестнадцатеричном представлении  $0x0000$ .

Для иллюстрации на рис. 2.6 показан результат увеличения размера слова  $w = 3$  до  $w = 4$  за счет расширения знакового разряда. Битовый вектор  $[101]$  представляет значение  $-4 + 1 = -3$ . Расширение знака дает битовый вектор  $[1101]$ , представляющий значение  $-8 + 4 + 1 = -3$ . Как видите, для  $w = 4$  комбинация двух старших битов,  $-8 + 4 = -4$ , совпадает со значением знакового бита для  $w = 3$ . Точно так же битовые векторы  $[111]$  и  $[1111]$  представляют значение  $-1$ .



**Рис. 2.6.** Примеры расширения знака с  $w = 3$  до  $w = 4$ . Для  $w = 4$  общий вес старших двух битов равен  $-8 + 4 = -4$ , что соответствует весу знакового бита для  $w = 3$

Теперь, используя это понимание, мы можем показать, что расширение знака сохраняет значение числа в дополнительном коде.

**ВЫВОД:** расширение числа в дополнительном коде путем расширения знака.

Пусть  $w' = w + k$ . Мы должны доказать, что

$$B2T_{w+k}(\underbrace{[x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]}_{k \text{ раз}}) = B2T_w([x_{w-1}, x_{w-2}, \dots, x_0]).$$

■

Доказательство выполнено методом математической индукции. То есть если можно доказать, что расширение знакового разряда на один бит сохраняет числовую величину, то это свойство будет выполняться при расширении знакового разряда на произвольное число битов. Итак, задача сокращается до доказательства того, что

$$B2T_{w+1}([x_{w-1}, \dots, x_{w-1}, x_{w-2}, \dots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \dots, x_0]).$$

Расширение левой части выражения в соответствии с уравнением (2.3) дает:

$$\begin{aligned} B2T_{w+1}([x_{w-1}, \dots, x_{w-1}, x_{w-2}, \dots, x_0]) &= -x_{w-1} 2^w + \sum_{i=0}^{w-1} x_i 2^i \\ &= -x_{w-1} 2^w + x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1} (2^w - 2^{w-1}) + \sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\ &= B2T_w(x_{w-1}, x_{w-2}, \dots, x_0). \end{aligned}$$

Ключевым свойством, использованным здесь, было равенство  $2^w - 2^{w-1} = 2^{w-1}$ . В соответствии с ним общим эффектом добавления бита с весом  $-2^w$  и преобразования бита, имеющего вес  $-2^{w-1}$ , в бит с весом  $2^{w-1}$  является сохранение первоначального числового значения.

### Упражнение 2.22 (решение в конце главы)

Применив уравнение 2.3, покажите, что каждый из следующих битовых векторов является представлением  $-5$  в дополнительном коде:

1. [1011]
2. [11011]
3. [111011]

Обратите внимание, что второй и третий векторы можно вывести из первого расширением знакового разряда.

Стоит отметить, что относительный порядок преобразования данных из одного разряда в другой и преобразования величин без знака в величины со знаком может повлиять на поведение программы. Рассмотрим следующий код:

```

1 short sx = -12345; /* -12345 */
2 unsigned uy = sx; /* Мистика! */
3
4 printf("uy = %u:\t", uy);
5 show_bytes((byte_pointer) &uy, sizeof(unsigned));

```

Если запустить его на машине с прямым (big endian) порядком следования байтов, то этот код выведет:

```
uy = 4294954951: ff ff cf c7
```

Это показывает, что при преобразовании из short в unsigned программа сначала изменила размер, а затем тип. То есть (unsigned) sx эквивалентно (unsigned) (int) sx, в результате чего было получено значение 4 294 954 951, а не (unsigned) (unsigned short) sx, что дало бы 53 191. Это соглашение действительно определяется стандартами C.

### Упражнение 2.23 (решение в конце главы)

Рассмотрим следующие функции на C:

```

int fun1(unsigned word) {
    return (int) ((word << 24) >> 24);
}

int fun2(unsigned word) {
    return ((int) word << 24) >> 24;
}

```

Предположим, что они выполняются в 32-разрядной программе на машине, использующей арифметику в дополнительном коде. Также предположим, что сдвиг вправо значений со знаком выполняется арифметически, тогда как сдвиг вправо значений без знака – логически.

1. Заполните следующую таблицу, показывающую результаты, возвращаемые этими функциями для нескольких разных аргументов. Вам будет удобнее использовать шестнадцатеричное представление. Просто помните, что старшие биты шестнадцатеричных цифр с 8 по F равны 1.

w	fun1(w)	fun2(w)
0x00000076	_____	_____
0x87654321	_____	_____
0x000000C9	_____	_____
0xEDCBA987	_____	_____

2. Опишите словами, какие вычисления выполняет каждая из этих функций.

## 2.2.7. Усечение чисел

Предположим, что значение вместо расширения требуется сократить. Такое сокращение, например, имеет место в следующем коде:

```

1 int x = 53191;
2 short sx = (short) x; /* -12345 */
3 int y = sx; /* -12345 */

```

Приведение переменной к типу short вызовет усечение 32-разрядного значения типа int до 16-разрядного значения short. Как уже отмечалось выше, такая 16-разрядная комбинация является представлением числа –12 345 в дополнительном коде. При преобразовании обратно в int расширение дополнительным знаковым разрядом заполнит

старшие 16 бит единицами, и в результате получится 32-разрядное представление числа  $-12\ 345$  в дополнительном коде.

При усечении  $w$ -разрядного числа  $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$  до  $k$ -разрядного представления происходит простое отбрасывание старших  $w - k$  бит и в результате получается битовый вектор  $\vec{x}' = [x_{k-1}, x_{k-2}, \dots, x_0]$ . Усечение числа может изменить его значение – это одна из разновидностей переполнения. Для числа без знака мы легко можем определить, какое значение получится.

**ПРИНЦИП:** усечение числа без знака.

Пусть  $\vec{x}$  – битовый вектор  $[x_{w-1}, x_{w-2}, \dots, x_0]$ , а  $\vec{x}'$  – результат усечения до  $k$  бит:  $\vec{x}' = [x_{k-1}, x_{k-2}, \dots, x_0]$ . Пусть  $x = B2U_w(\vec{x})$  и  $x' = B2U_k(\vec{x}')$ . Тогда  $x' = x \bmod 2^k$ .

В основе этого принципа лежит следующая идея: все усеченные биты имеют веса вида  $2^i$ , где  $i \geq k$ , и при выполнении операции деления по модулю каждый из этих весов уменьшается до нуля. Формально эта идея подкрепляется следующим выводом:

**ВЫВОД:** усечение числа без знака.

Применение операции деления по модулю  $k$  уравнению 2.1 дает

$$\begin{aligned} B2U_w(x_{w-1}, x_{w-2}, \dots, x_0) \bmod 2^k &= \left[ \sum_{i=0}^{w-1} x_i 2^i \right] \bmod 2^k \\ &= \left[ \sum_{i=0}^{k-1} x_i 2^i \right] \bmod 2^k \\ &= \sum_{i=0}^{k-1} x_i 2^i \\ &= B2U_k([x_{k-1}, x_{k-2}, \dots, x_0]). \end{aligned}$$

В этом выводе мы используем свойство  $2^i \bmod 2^k = 0$  для любого  $i \geq k$ .

Аналогичное свойство имеет операция усечения числа в дополнительном коде, за исключением того, что потом она преобразует самый старший бит в знаковый бит.

**ПРИНЦИП:** усечение числа в дополнительном коде.

Пусть  $\vec{x}$  – битовый вектор  $[x_{w-1}, x_{w-2}, \dots, x_0]$ , а  $\vec{x}'$  – результат усечения до  $k$  бит:  $\vec{x}' = [x_{k-1}, x_{k-2}, \dots, x_0]$ . Пусть  $x = B2T_w(\vec{x})$  и  $x' = B2T_k(\vec{x}')$ . Тогда  $x' = U2T_k(x \bmod 2^k)$ .

В этой формулировке результатом  $x \bmod 2^k$  будет число от 0 до  $2^k - 1$ . Применение к нему функции  $U2T_k$  приведет к преобразованию веса самого старшего бита  $x_{k-1}$  с  $2^{k-1}$  в  $-2^{k-1}$ . Это можно увидеть на примере преобразования значения  $x = 53\ 191$  из типа `int` в тип `short`. Поскольку  $2^{16} = 65\ 536 \geq x$ , мы получаем  $x \bmod 2^{16} = x$ . Но при преобразовании этого числа в 16-разрядное представление в дополнительном коде мы получим  $x = 53\ 191 - 65\ 536 = -12\ 345$ .

**ВЫВОД:** усечение числа в дополнительном коде.

Используя доказательство, подобное тому, что приводилось для случая усечения числа без знака, мы получаем:

$$B2T_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k = B2U_k([x_{k-1}, x_{k-2}, \dots, x_0]).$$

То есть  $x \bmod 2^k$  можно представить числом без знака, имеющим битовое представление  $[x_{k-1}, x_{k-2}, \dots, x_0]$ . Преобразование его в представление в дополнительном коде дает  $x = U2T_k(x \bmod 2^k)$ .

То есть, учитывая все вышесказанное, эффект усечения чисел без знака можно выразить так:

$$B2U_k([x_{k-1}, x_{k-2}, \dots, x_0]) = B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k, \tag{2.9}$$

а эффект усечения чисел в дополнительном коде – так:

$$B2T_w([x_{k-1}, x_{k-2}, \dots, x_0]) = U2T_k(B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k). \tag{2.10}$$

**Упражнение 2.24 (решение в конце главы)**

Предположим, что 4-разрядное значение (представленное шестнадцатеричными цифрами от 0 до F) усекается до 3-разрядного значения (представленного шестнадцатеричными цифрами от 0 до 7). Заполните следующую таблицу, подставив результат усечения чисел без знака и в дополнительном коде, представленных указанными комбинациями битов:

Шестнадцатеричное число		Без знака		Дополнительный код	
Исходное	Усеченное	Исходное	Усеченное	Исходное	Усеченное
0	0	0	<u>          </u>	0	<u>          </u>
2	2	2	<u>          </u>	2	<u>          </u>
9	1	9	<u>          </u>	-7	<u>          </u>
B	3	11	<u>          </u>	-5	<u>          </u>
F	7	15	<u>          </u>	-1	<u>          </u>

Объясните, как к этим случаям применяются уравнения (2.9) и (2.10).

**2.2.8. Советы по приемам работы с числами со знаком и без знака**

Как было показано выше, неявное приведение знакового типа к беззнаковому влечет за собой некоторое неочевидное поведение. Неочевидные функции часто приводят к программным ошибкам, обусловленным нюансами неявного приведения типов, которые очень трудно обнаружить. Поскольку преобразование выполняется без явного свидетельства в коде, программисты могут не замечать его последствий.

Следующие два упражнения иллюстрируют некоторые трудноуловимые ошибки, которые могут возникнуть из-за неявного приведения типов и использования беззнаковых типов.

**Упражнение 2.25 (решение в конце главы)**

Взгляните на следующий код, реализующий суммирование элементов массива `a`, где количество элементов представлено параметром `length`:

```
1 /* ВНИМАНИЕ: этот код содержит ошибку */
2 float sum_elements(float a[], unsigned length) {
```



```

3   int i;
4   float result = 0;
5
6   for (i = 0; i <= length-1; i++)
7       result += a[i];
8   return result;
9 }

```

Если вызвать эту функцию с аргументом 0, то она должна вернуть 0.0, но вместо этого генерируется ошибка доступа к памяти. Объясните, почему это происходит, и покажите, как можно исправить код.

### Упражнение 2.26 (решение в конце главы)

Вам дано задание написать функцию, которая определяет, длиннее ли одна строка, чем другая. Вы решили использовать библиотечную функцию `strlen` со следующим объявлением:

```

/* Прототип библиотечной функции strlen */
size_t strlen(const char *s);

```

Вот ваша первая попытка использовать эту функцию:

```

/* Определяет, длиннее ли строка s, чем строка t */
/* ВНИМАНИЕ: эта функция содержит ошибку */
int strlonger(char *s, char *t) {
    return strlen(s) - strlen(t) > 0;
}

```

Проверив эту функцию на некоторых примерах данных, вы заметили, что она работает не всегда правильно. Вы провели исследования и выяснили, что при 32-разрядной компиляции тип данных `size_t` определяется (через `typedef`) в заголовочном файле `stdio.h` как беззнаковый.

1. В каких случаях эта функция даст неверный результат?
2. Объясните, как возникает этот неверный результат.
3. Покажите, как исправить код, чтобы он работал надежно.

Мы уже видели несколько примеров, когда тонкие нюансы беззнаковой арифметики, и особенно неявное преобразование типа со знаком в тип без знака, могут привести к ошибкам или уязвимостям. Один из способов избежать подобных ошибок – отказаться от использования чисел без знака. На самом деле мало какие языки, кроме C, поддерживают целые числа без знака. Очевидно, что разработчики этих языков проектировали их более основательно, чем те того заслуживали. Например, Java поддерживает только целые числа со знаком и требует их реализации с арифметикой в дополнительном коде. Обычный оператор сдвига вправо `>>` гарантированно выполняет арифметический сдвиг. Для логического сдвига вправо определен специальный оператор `>>>`.

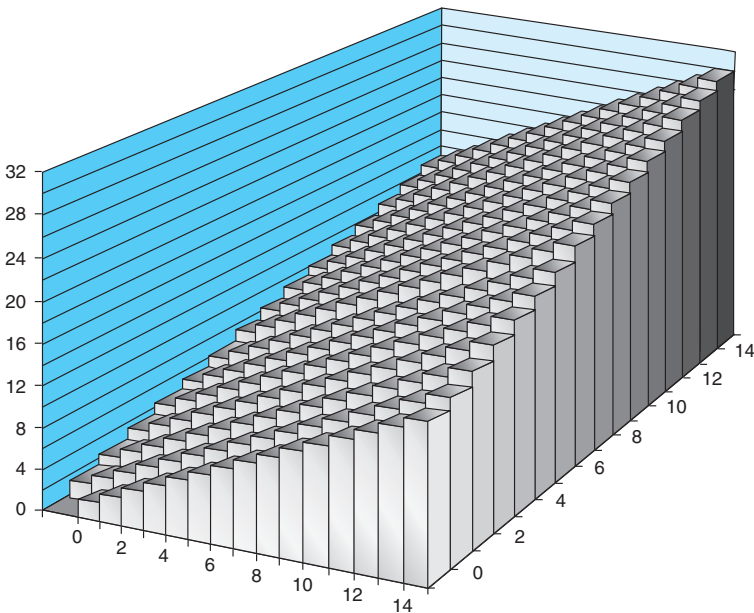
Значения без знака очень полезны, если слова рассматриваются лишь как наборы битов, без какой бы то ни было числовой интерпретации. Это имеет место, например, когда слово используется как набор *флагов*, описывающих булевы условия. Адреса не имеют знака, поэтому системные программисты считают, что типы без знака имеют свои преимущества. Значения без знака также могут пригодиться в математических пакетах для реализации арифметики сравнений по модулю и арифметических операций с многократно увеличенной точностью, где числа представлены в виде массивов слов.

## 2.3. Целочисленная арифметика

Многие начинающие программисты с удивлением обнаруживают, что результатом сложения двух положительных чисел может стать отрицательное число, а сравнение  $x < y$  может дать иной результат, нежели сравнение  $x - y < 0$ . Эти свойства являются порождением конечной природы компьютерной арифметики. Понимание ее нюансов помогает программистам писать более надежный код.

### 2.3.1. Сложение целых без знака

Рассмотрим два неотрицательных целых,  $x$  и  $y$ , таких что  $0 < x, y < 2^w$ . Каждое из этих чисел можно представить в виде  $w$ -разрядного целого без знака. Однако если вычислить их сумму, то результат может оказаться в диапазоне  $0 \leq x + y \leq 2^{w+1} - 2$ . Для представления этой суммы может потребоваться  $w + 1$  бит. Например, на рис. 2.7 показан график функции  $x + y$ , когда  $x$  и  $y$  имеют 4-разрядные представления. Аргументы (показаны по горизонтальным осям) изменяются в диапазоне от 0 до 15, но диапазон суммы шире – от 0 до 30. График данной функции имеет форму наклонной плоскости. Если возникнет необходимость обеспечить поддержку такой суммы с количеством битов  $w + 1$  и возможности ее сложения с другим значением, тогда может потребоваться  $w + 2$  бит и т. д. Такое прогрессирующее увеличение длины слова означает, что для полного представления результатов арифметических операций мы не можем устанавливать границы длины слова. Некоторые языки программирования, например Lisp, на самом деле поддерживают арифметику *бесконечной точности*, чтобы обеспечить поддержку произвольной (разумеется, в пределах машинной памяти) целочисленной арифметики. Чаще всего языки программирования поддерживают арифметику фиксированной точности, и, следовательно, такие операции, как «сложение» и «умножение», отличаются от эквивалентных операций с целыми числами.



**Рис. 2.7.** Целочисленное сложение. Сложение 4-разрядных аргументов может дать в результате 5-разрядную сумму

Давайте определим операцию  $+_w^u$  для аргументов  $x$  и  $y$ , где  $0 \leq x, y < 2^w$ , как результат усеечения целочисленной суммы  $x + y$  до  $w$  бит и последующей интерпретации результата как числа без знака. Эту операцию можно рассматривать как форму арифметики по модулю, вычисление суммы по модулю  $2^w$  путем простого отбрасывания любых битов с весом больше  $2^{w-1}$  в битовом представлении  $x + y$ . Например, рассмотрим 4-разрядные представления чисел  $x = 9$  и  $y = 12 = [1001]$  и  $[1100]$  соответственно. Их сумма равна 21 и имеет 5-разрядное представление  $[10101]$ . Но если отбросить старший бит, мы получим  $[0101]$ , т. е. десятичное значение 5. Это соответствует результату операции  $21 \bmod 16 = 5$ .

### Уязвимость в `getpeername`

В 2002 году программисты, участвовавшие в создании операционной системы с открытым исходным кодом FreeBSD, обнаружили, что их реализация библиотечной функции `getpeername` имеет уязвимость. Упрощенная версия их реализации выглядела примерно так:

```

1 /*
2  * Иллюстрация уязвимости, аналогичной той, что была обнаружена
3  * в реализации getpeername() в ОС FreeBSD
4  */
5
6 /* Объявление библиотечной функции memcpy */
7 void *memcpy(void *dest, void *src, size_t n);
8 /* Область памяти ядра, хранящей данные, которые доступны
9  * пользователю
10 */
11 #define KSIZE 1024
12 char kbuf[KSIZE];
13 /* Копировать до maxlen байт из памяти ядра в буфер пользователя */
14 int copy_from_kernel(void *user_dest, int maxlen) {
15     /* Счетчик байтов len -- наименьшее из размера буфера и maxlen */
16     int len = KSIZE < maxlen ? KSIZE : maxlen;
17     memcpy(user_dest, kbuf, len);
18     return len;
19 }
```

Здесь, в строке 7, показан также прототип библиотечной функции `memcpy`, которая предназначена для копирования указанного количества байтов  $n$  из одной области памяти в другую.

Функция `copy_from_kernel`, начинающаяся в строке 14, предназначена для копирования некоторых данных, поддерживаемых ядром операционной системы, в указанную область памяти, доступную пользователю. Большинство структур данных, поддерживаемых ядром, не должны быть доступны для чтения пользователям, потому что могут содержать конфиденциальную информацию о других пользователях и о других заданиях, выполняемых в системе, но область, объявленная с именем `kbuf`, предназначена для того, чтобы пользователь мог читать из нее. Параметр `maxlen` определяет длину буфера `user_dest`, выделенного пользователем. Вычисление в строке 16 гарантирует, что скопировано будет не больше байтов, чем доступно в исходном или целевом буфере.

А теперь представьте, что какой-то злонамеренный программист пишет код, который вызывает `copy_from_kernel` с отрицательным значением `maxlen`. В этом случае операция взятия минимального из двух чисел в строке 16 присвоит это значение переменной `len`, которое затем будет передано как параметр  $n$  в `memcpy`. Обратите внимание, однако, что параметр  $n$  объявлен с типом `size_t`. Этот тип данных определяется (через `typedef`) в библиотечном файле `stdio.h`.

Обычно в 32-разрядных программах он определяется как `unsigned`, а в 64-разрядных – как `unsigned long`. Поскольку аргумент  $n$  имеет беззнаковый тип, метасру будет рассматривать отрицательное значение как очень большое положительное число и попытается скопировать это количество байтов из области ядра в буфер пользователя. Копирование такого количества байтов (по крайней мере  $2^{31}$ ) на самом деле может не сработать, потому что в процессе копирования могут быть обнаружены недопустимые адреса, но, как бы то ни было, программа может читать области памяти ядра, доступ к которым должен быть для нее закрыт.

Эта проблема обусловлена несоответствием типов данных: в одном месте параметр длины является целым со знаком; в другом месте – целым без знака. Такие несоответствия могут стать источниками ошибок и, как показывает этот пример, даже привести к уязвимостям. К счастью, не было зафиксировано ни одного свидетельства об использовании этой уязвимости в FreeBSD. Разработчики выпустили рекомендацию по безопасности «FreeBSD-SA-02:38.signed-error», в которой системным администраторам сообщалось, как наложить исправление, устраняющее уязвимость. Ошибка может быть исправлена путем объявления параметра `maxlen` в `copy_from_kernel` с типом `size_t`, чтобы он соответствовал параметру  $n$  в метасру. Также следовало объявить локальную переменную `len` и возвращаемое значение с типом `size_t`.

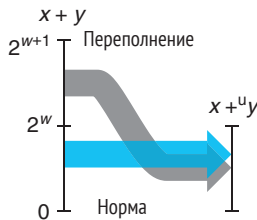
Охарактеризовать операцию  $+_w^u$  можно следующим образом.

**ПРИНЦИП:** сложение целых без знака.

Для  $x$  и  $y$  таких, что  $0 \leq x, y < 2^w$ :

$$x +_w^u y = \begin{cases} x + y, & x + y < 2^w & \text{норма} \\ x + y - 2^w, & 2^w \leq x + y < 2^{w+1} & \text{переполнение} \end{cases} \quad (2.11)$$

Два случая из уравнения 2.11 показаны на рис. 2.8, где сумма  $x + y$  слева отображается в  $w$ -разрядную сумму  $x +_w^u y$  без знака справа. В нормальном случае значение  $x + y$  сохраняется, а в случае переполнения сумма уменьшается на  $2^w$ .



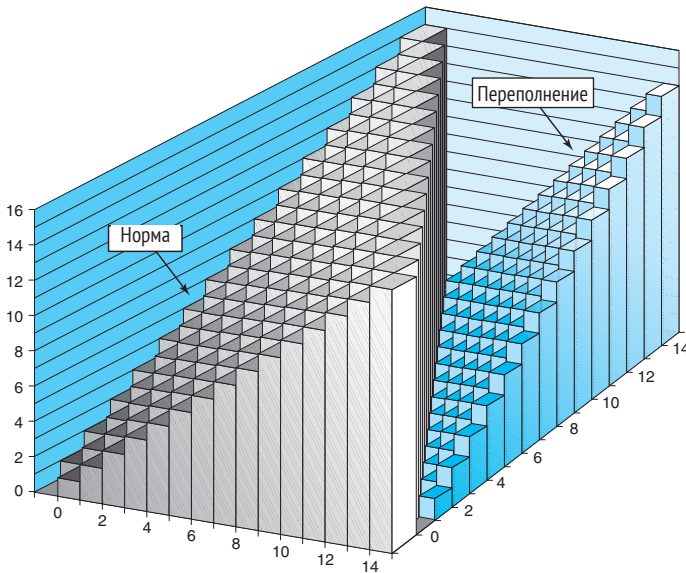
**Рис 2.8.** Связь между сложением целых со знаком и без знака.  
Когда  $x + y$  больше  $2^{w-1}$ , сумма переполняется

**ВЫВОД:** сложение целых без знака.

В общем случае если  $x + y < 2^w$ , то ведущий бит в  $(w + 1)$ -разрядном представлении суммы будет равен 0, и, следовательно, его отбрасывание не изменит числового значения. С другой стороны, если  $2^w \leq x + y < 2^{w+1}$ , то старший бит в  $(w + 1)$ -разрядном пред-

ставлении суммы будет равен 1, и, следовательно, его отбрасывание эквивалентно вычитанию  $2^w$  из суммы.

Говорится, что арифметическая операция дает *переполнение*, когда полный целочисленный результат не умещается в слово, размер которого ограничивается типом данных. Как указано в уравнении 2.11, переполнение возникает тогда, когда сумма двух операндов составляет  $2^w$  или больше. На рис. 2.9 показан график функции сложения целых без знака для длины слова  $w = 4$ . Сумма рассчитывается по модулю  $2^4 = 16$ . Когда  $x + y < 16$ , тогда переполнения не возникает и  $x +_4^u y$  просто равно  $x + y$ . Этому случаю соответствует область наклонной плоскости, обозначенная как «Норма». Когда  $x + y \geq 16$ , тогда происходит переполнение и возникает эффект уменьшения суммы на 16. Этому случаю соответствует область наклонной плоскости, обозначенная как «Переполнение».



**Рис. 2.9.** Сложение целых без знака. При размере слова 4 бита сложение выполняется по модулю 16

В программах на C переполнение не воспринимается как ошибка. Однако иногда бывает желательно проверить факт переполнения.

**ПРИНЦИП:** обнаружение переполнения при сложении целых без знака.

Для  $x$  и  $y$  в диапазоне  $0 \leq x, y \leq UMax_w$ , пусть  $s \doteq x +_w^u y$ . Тогда сказать, что при вычислении  $s$  возникло переполнение, можно, только если  $s < x$  (или, что то же самое,  $s < y$ ).

Так в нашем предыдущем примере мы видели, что  $9 +_4^u 12 = 5$ . В данном случае имело место переполнение, потому что  $5 < 9$ .

**ВЫВОД:** обнаружение переполнения при сложении целых без знака.

Заметим, что  $x + y \geq x$ , и, следовательно, если при вычислении суммы  $s$  не было переполнения, то мы обязательно будем иметь  $s \geq x$ . С другой стороны, если произошло

переполнение, то мы получим  $s = x + y - 2^w$ . Учитывая, что  $y < 2^w$ , получаем  $y - 2^w < 0$ , а значит,  $s = x + (y - 2^w) < x$ .

### Упражнение 2.27 (решение в конце главы)

Напишите функцию со следующим прототипом:

```
/* Определяет, можно ли сложить аргументы без переполнения */
int uadd_ok(unsigned x, unsigned y);
```

Эта функция должна вернуть 1, если аргументы  $x$  и  $y$  можно сложить без переполнения.

### Упражнение 2.28 (решение в конце главы)

Комбинацию битов с длиной  $w = 4$  можно представить одной шестнадцатеричной цифрой. Для интерпретации этих цифр как чисел без знака воспользуйтесь уравнением 2.12 и заполните следующую таблицу, подставляя значения без знака и битовые представления (в шестнадцатеричном виде) аддитивных инверсий указанных цифр.

$x$		$+_w^u x$	
Шестнадцатеричное	Десятичное	Десятичное	Шестнадцатеричное
0	_____	_____	_____
5	_____	_____	_____
8	_____	_____	_____
D	_____	_____	_____
F	_____	_____	_____

Сложение по модулю образует математическую структуру, известную как *абелева группа* и названную в честь норвежского математика Нильса Хенрика Абеля (Niels Henrik Abel; 1802–1829). Эта структура коммутативна (вот тут-то и вступает в силу «абелева» часть) и ассоциативна. Она имеет нейтральный элемент 0, и каждый элемент обладает аддитивной инверсией. Рассмотрим множество чисел без знака, состоящих из  $w$  бит, с операцией сложения  $+_w^u$ . Для каждого значения  $x$  должно иметься некоторое значение  $-_w^u x$  такое, что  $-_w^u x +_w^u x = 0$ . Эту аддитивную обратную операцию можно охарактеризовать следующим образом:

**ПРИНЦИП:** отрицание целого без знака.

Для любого числа  $x$  такого, что  $0 \leq x < 2^w$ , его  $w$ -разрядное отрицание  $-_w^u x$  вычисляется как

$$-_w^u x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \quad (2.12)$$

**ВЫВОД:** отрицание целого без знака.

Когда  $x = 0$ , аддитивная обратная величина явно равна 0. Для случая  $x > 0$  рассмотрим значение  $2^w - x$ . Обратите внимание, что это число находится в диапазоне  $0 < 2^w - x < 2^w$ . Также можно заметить, что  $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$ . Следовательно, это обратная  $x$  величина относительно  $+_w^u$ .

### 2.3.2. Сложение целых в дополнительном коде

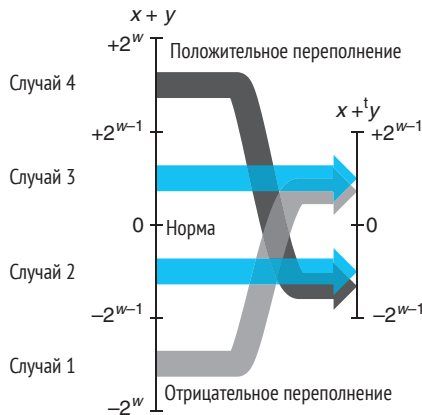
При сложении целых в дополнительном коде мы должны решить, что делать, если результат окажется слишком большим (положительным) или слишком маленьким (отрицательным) для представления. Для целых чисел  $x$  и  $y$  в диапазоне  $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$  их сумма находится в диапазоне  $-2^w \leq x + y \leq 2^w - 2$  и потенциально может потребовать  $w + 1$  бит для точного представления. Как и ранее, будем избегать бесконечного увеличения размера данных путем усечения представления до  $w$  бит. Впрочем, результат не столь прост в математическом отношении, нежели сложение по модулю.

**ПРИНЦИП:** сложение целых в дополнительном коде.

Для целых чисел со знаком  $x$  и  $y$  в диапазоне  $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ :

$$x +_w^u y = \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y & \text{положительное переполнение} \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} & \text{норма} \\ x + y + 2^w, & x + y < -2^{w-1} & \text{отрицательное переполнение} \end{cases} \quad (2.13)$$

Этот принцип иллюстрирует рис. 2.10, где слева показана сумма  $x + y$ , имеющая значение в диапазоне  $-2^w \leq x + y \leq 2^w - 2$ , и результат усечения суммы до  $w$ -разрядного числа в дополнительном коде – справа. (Обозначения «Случай 1» – «Случай 4» на этом рисунке мы используем для анализа в формальном выводе принципа.) Когда сумма  $x + y$  превышает  $TMax_w$  (случай 4), мы говорим, что произошло *положительное переполнение*. В этом случае усечение заключается в вычитании  $2^w$  из суммы. Когда сумма  $x + y$  меньше  $TMin_w$  (случай 1), мы говорим, что произошло *отрицательное переполнение*. В этом случае усечение заключается в добавлении  $2^w$  к сумме.



**Рис. 2.10.** Связь между сложением целых без знака и целых в дополнительном коде.

Когда сумма  $x + y$  меньше, чем  $-2^{w-1}$ , возникает отрицательное переполнение.

Когда больше или равна  $2^{w-1}$ , происходит положительное переполнение

$w$ -разрядная сумма двух целых в дополнительном коде имеет то же битовое представление, что и сумма целых без знака. Фактически большинство компьютеров ис-

пользуют одну и ту же машинную инструкцию для выполнения сложения как целых со знаком, так и без знака.

**ВЫВОД:** сложение целых в дополнительном коде.

Поскольку сложение целых в дополнительном коде дает тот же результат в битовом представлении, что и сложение целых без знака, мы можем охарактеризовать операцию  $+_w^t$  как включающую преобразование ее аргументов в представление без знака, выполнение сложения и последующее преобразование результата в дополнительный код:

$$x +_w^t y = U2T_w(T2U_w(x) +_w^u T2U_w(y)). \quad (2.14)$$

Согласно уравнению 2.6 мы можем записать  $T2U_w(x)$  как  $x_{w-1}2^w + x$  и  $T2U_w(y)$  как  $y_{w-1}2^w + y$ . В соответствии со свойством операции  $+_w^u$ , что она является простым сложением по модулю  $2^w$ , а также свойством сложения по модулю мы имеем:

$$\begin{aligned} x +_w^t y &= U2T_w(T2U_w(x) +_w^u T2U_w(y)) \\ &= U2T_w[(x_{w-1}2^w + x + y_{w-1}2^w + y) \bmod 2^w] \\ &= U2T_w[(x + y) \bmod 2^w]. \end{aligned}$$

Члены  $x_{w-1}2^w$  и  $y_{w-1}2^w$  сокращаются, потому что они равны 0 по модулю  $2^w$ .

Для лучшего понимания идеи определим  $z$  как целочисленную сумму  $z \doteq x + y$ ,  $z'$  как  $z' \doteq z \bmod 2^w$  и  $z''$  как  $z'' \doteq U2T_w(z')$ . Величина  $z''$  равна  $x +_w^t y$ . Анализ можно разделить на четыре случая, как показано на рис. 2.10.

1.  $-2^w \leq z < -2^{w-1}$ . Тогда мы имеем  $z' = z + 2^w$ . Это дает  $0 < z' < -2^{w-1} + 2^w = 2^{w-1}$ . Согласно уравнению (2.7),  $z'$  находится в таком диапазоне, что  $z'' = z'$ . Такой случай называется отрицательным переполнением. Было выполнено сложение двух отрицательных чисел  $x$  и  $y$  (только в этом случае выполняется условие  $z < -2^{w-1}$ ) и получен неотрицательный результат  $z'' = x + y + 2^w$ .
2.  $-2^{w-1} \leq z < 0$ . Тогда мы имеем  $z' = z + 2^w$ , что дает  $-2^{w-1} + 2^w = 2^{w-1} \leq z' < 2$ . Согласно уравнению (2.7),  $z'$  находится в таком диапазоне, что  $z'' = z' - 2^w$  и, следовательно,  $z'' = z' - 2^w = z + 2^w - 2^w = z$ . То есть сумма  $z''$  в дополнительном коде равна целочисленной сумме  $x + y$ .
3.  $0 \leq z < 2^{w-1}$ . Тогда мы имеем  $z' = z$ , что дает  $0 \leq z' < 2^{w-1}$ . И снова сумма  $z''$  в дополнительном коде равна целочисленной сумме  $x + y$ .
4.  $2^{w-1} \leq z < 2^w$ . Тогда мы имеем  $z' = z$ , что дает  $2^{w-1} \leq z' < 2^w$ . Но в этом диапазоне мы имеем  $z'' = z' - 2^w$ , что дает  $z'' = x + y - 2^w$ . Такой случай называется положительным переполнением. Было выполнено сложение двух положительных чисел  $x$  и  $y$  (только в этом случае выполняется условие  $z \geq 2^{w-1}$ ), и получен отрицательный результат  $z'' = x + y - 2^w$ .

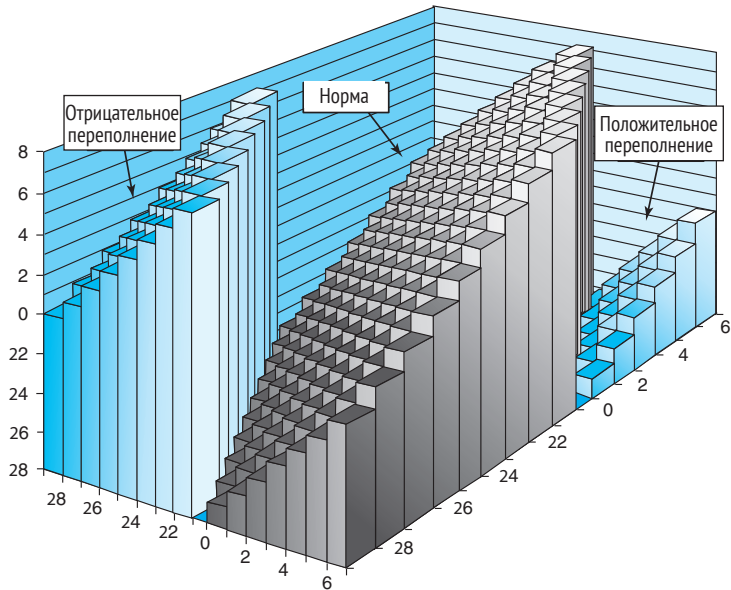
В качестве иллюстрации в табл. 2.13 приводятся несколько примеров сложения целых в дополнительном коде для  $w = 4$ . Для каждого примера указан случай, к которому он относится в выводе уравнения (2.13). Обратите внимание, что  $2^4 = 16$ , и, следовательно, отрицательное переполнение выдает результат на 16 больше целочисленной суммы, а положительное переполнение – на 16 меньше. В таблице также показаны битовые представления операндов и результатов. Обратите внимание, что данный результат можно получить двоичным сложением операндов и усечением его (результата) до 4 бит.



**Таблица 2.13.** Примеры сложения целых в дополнительном коде. Битовое представление 4-разрядной суммы в дополнительном коде можно получить путем двоичного сложения операндов и усечения результата до 4 бит

$x$	$y$	$x + y$	${}_2^1x$	Случай
-8	-5	-13	3	1
[1000]	[1011]	[10011]	[0011]	
-8	-8	-16	0	1
[1000]	[1000]	[10000]	[0000]	
-8	5	-3	-3	2
[1000]	[0101]	[11101]	[1101]	
2	5	7	7	3
[0010]	[0101]	[00111]	[0111]	
5	5	10	-6	4
[0101]	[0101]	[01010]	[1010]	

Рисунок 2.11 иллюстрирует сложение целых в дополнительном коде с длиной слова  $w = 4$ . Диапазон операндов от  $-8$  до  $7$ . Когда  $x + y < -8$ , сложение в дополнительном коде дает отрицательное переполнение, что вызывает увеличение суммы на 16. Когда  $-8 < x + y < 8$ , сложение дает  $x + y$ . Когда  $x + y \geq 8$ , сложение дает положительное переполнение, вызывающее уменьшение суммы на 16. Каждый из этих трех диапазонов образует на схеме наклонную плоскость.



**Рис. 2.11.** Сложение целых в дополнительном коде.

При размере слова 4 бита сложение может дать отрицательное переполнение, когда  $x + y < -8$ , и положительное переполнение, когда  $x + y \geq 8$

Уравнение 2.13 позволяет выявить случаи, когда имеет место переполнение.

**ПРИНЦИП:** определение переполнения при сложении целых в дополнительном коде.

Для  $x$  и  $y$  в диапазоне  $TMin_w \leq x, y \leq TMax_w$  пусть  $s \doteq x +_w^t y$ . Тогда можно сказать, что вычисление  $s$  дало положительное переполнение тогда и только тогда, когда  $x > 0$  и  $y > 0$ , но  $s \leq 0$ . Вычисление дает отрицательное переполнение тогда и только тогда, когда  $x < 0$  и  $y < 0$ , но  $s \geq 0$ .

В табл. 2.13 представлено несколько примеров действия этого принципа для  $w = 4$ . Первая запись показывает случай отрицательного переполнения, когда сложение двух отрицательных чисел дает положительную сумму. Последняя запись показывает случай положительного переполнения, когда сложение двух положительных чисел дает отрицательную сумму.

**ВЫВОД:** определение переполнения при сложении целых в дополнительном коде.

Сначала проанализируем положительное переполнение. Если оба операнда  $x > 0$  и  $y > 0$ , но  $s \leq 0$ , то явно произошло положительное переполнение. И наоборот, положительное переполнение требует выполнения условий (1)  $x > 0$  и  $y > 0$  (в противном случае  $x + y < TMax_w$ ) и (2)  $s \leq 0$  (из уравнения 2.13). Аналогичный набор аргументов справедлив и для отрицательного переполнения.

#### Упражнение 2.29 (решение в конце главы)

Заполните следующую таблицу по аналогии с табл. 2.13. Приведите целые значения 5-разрядных аргументов, их сумм как целочисленных, битовые представления сумм в дополнительном коде и случай из вывода уравнения 2.13.

$x$	$y$	$x + y$	$+_5^t x$	Случай
[10100]	[10001]			
[11000]	[11000]			
[10111]	[01000]			
[00010]	[00101]			
[01100]	[00100]			

#### Упражнение 2.30 (решение в конце главы)

Напишите функцию со следующим прототипом:

```
/* Определяет, можно ли сложить аргументы без переполнения */
int tadd_ok(int x, int y);
```

Эта функция должна возвращать 1, если аргументы  $x$  и  $y$  можно сложить, не вызвав переполнения.

**Упражнение 2.31 (решение в конце главы)**

Вашему коллеге не хватило терпения на анализ условий переполнения при сложении двух целых в дополнительном коде, и он представил следующую реализацию `tadd_ok`:

```
/* Определяет, можно ли сложить аргументы без переполнения */
/* ВНИМАНИЕ: этот код содержит ошибку. */
int tadd_ok(int x, int y) {
    int sum = x+y;
    return (sum-x == y) && (sum-y == x);
}
```

Этот код вызвал у вас смех. Объясните почему.

**Упражнение 2.32 (решение в конце главы)**

Вам поручено написать функцию `tsub_ok` с аргументами  $x$  и  $y$ , которая возвращает 1, если вычисление  $x - y$  не вызовет переполнения. Только что написав функцию для упражнения 2.30, вы пишете следующее:

```
/* Определяет, можно ли вычислить разность аргументов без переполнения */
/* ВНИМАНИЕ: этот код содержит ошибку. */
int tsub_ok(int x, int y) {
    return tadd_ok(x, -y);
}
```

Для каких значений  $x$  и  $y$  эта функция даст неверный результат? Разработка правильной версии этой функции будет предложена в упражнении 2.74 (для самостоятельного решения).

**Упражнение 2.33 (решение в конце главы)**

Комбинацию битов с длиной  $w = 4$  можно представить одной шестнадцатеричной цифрой. Представьте эти цифры в дополнительном коде и заполните следующую таблицу, определив соответствующие аддитивные инверсии.

$x$		$-\frac{1}{4}x$	
Шестнадцатеричное	Десятичное	Десятичное	Шестнадцатеричное
0	_____	_____	_____
5	_____	_____	_____
8	_____	_____	_____
D	_____	_____	_____
F	_____	_____	_____

Какие особенности можно отметить в комбинациях битов, созданных отрицанием целых в дополнительном коде и без знака (см. упражнение 2.28)?

### Приложение в интернете DATA:NEG. Битовое представление отрицания целых в дополнительном коде

Есть несколько интересных способов выполнить отрицание целых в дополнительном коде в битовом представлении. Особенно интересны следующие два метода, которые могут пригодиться, например, когда при отладке программы встречается значение `0xffffffa`, а кроме того, они помогают понять природу представления в виде дополнительного кода.

Один из методов отрицания целых в дополнительном коде в битовом представлении состоит в добавлении битов и последующем увеличении результата. В языке C мы можем заявить, что для любого целого значения  $x$  выражения  $\sim x$  и  $\sim x + 1$  дают идентичный результат.

Вот несколько примеров с 4-разрядным размером слова:

$\vec{x}$		$\sim \vec{x}$		$incr(\sim \vec{x})$	
[0101]	5	[1010]	-6	[1011]	-5
[0111]	7	[1000]	-8	[1001]	-7
[1100]	-4	[0011]	3	[0100]	4
[0000]	0	[1111]	-1	[0000]	0
[1000]	-8	[0111]	7	[1000]	-8

Из нашего предыдущего примера мы знаем, что дополнением для `0xf` является `0x0`, а дополнением для `0xa` является `0x5`, поэтому `0xffffffa` – это представление числа  $-6$  в дополнительном коде.

Второй способ выполнить отрицание числа  $x$  в дополнительном коде основан на разделении битового вектора на две части. Пусть  $k$  – это позиция крайней правой единицы, поэтому битовое представление  $x$  имеет вид  $[x_{w-1}, x_{w-2}, \dots, x_{k+1}, 1, 0, \dots, 0]$ . (Это возможно при условии  $x \neq 0$ .) Отрицание записывается в двоичной форме как  $[\sim x_{w-1}, \sim x_{w-2}, \dots, \sim x_{k+1}, 1, 0, \dots, 0]$ . То есть мы дополняем каждый бит слева от позиции  $k$ .

Проиллюстрируем эту идею на примере некоторых 4-разрядных чисел, где выделим крайний правый шаблон `1, 0, ..., 0` курсивом:

$x$		$\sim x$	
[1100]	-4	[0100]	4
[1000]	-8	[1000]	-8
[0101]	5	[1011]	-5
[0111]	7	[1001]	-7

### 2.3.3. Отрицание целых в дополнительном коде

Как мы знаем, всякое число  $x$  в диапазоне  $TMin_w \leq x \leq TMax_w$  имеет аддитивную инверсию при  $+_w^t$ , которая обозначается как  $-_w^t$ .

**ПРИНЦИП:** отрицание целых в дополнительном коде.

Для  $x$  в диапазоне  $TMin_w \leq x \leq TMax_w$  его отрицание в дополнительном коде  $-_w^t$  вычисляется по формуле:

$$-^t_w x = \begin{cases} TMin_w, & x = TMin_w \\ -x, & x > TMin_w \end{cases}. \quad (2.15)$$

То есть для  $w$ -разрядного сложения целых в дополнительном коде  $TMin_w$  является собственной аддитивной инверсией, в то время как для любого другого значения  $x$  имеется аддитивная инверсия  $-x$ .

**ВЫВОД:** отрицание целых в дополнительном коде.

Обратите внимание, что  $TMin_w + TMin_w = -2^{w-1} + -2^{w-1} = -2^w$ . Эта операция вызовет отрицательное переполнение и, следовательно,  $TMin_w +^t_w TMin_w = -2^w + 2^w = 0$ . Для таких значений  $x$ , что  $x > TMin_w$ , значение  $-x$  также можно представить как  $w$ -разрядное число в дополнительном коде, и их сумма будет  $-x + x = 0$ .

### 2.3.4. Умножение целых без знака

Целые числа  $x$  и  $y$  в диапазоне  $0 \leq x, y \leq 2^w - 1$  можно представить как  $w$ -разрядные числа без знака, однако их произведение  $x \cdot y$  может находиться в диапазоне от 0 до  $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$ . Для представления результата может потребоваться как минимум  $2^w$  бит. Вместо этого беззнаковое умножение в C возвращает младшие  $w$  бит из  $2w$ -разрядного целочисленного произведения. Обозначим это значение как  $x *^u_w y$ .

Усечение числа без знака до  $w$  бит эквивалентно вычислению его значения по модулю  $2^w$ , что дает:

**ПРИНЦИП:** умножение целых без знака.

Для  $x$  и  $y$  таких, что  $0 \leq x, y \leq UMax_w$ :

$$x *^u_w y = (x \cdot y) \bmod 2^w. \quad (2.16)$$

### 2.3.5. Умножение целых в дополнительном коде

Целые числа  $x$  и  $y$  в диапазоне  $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$  можно представить как  $w$ -разрядные числа в дополнительном коде, однако их произведение  $x \cdot y$  может находиться в диапазоне от  $-2^{w-1} \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$  до  $-2^{w-1} \cdot -2^{w-1} = 2^{2w-2}$ . Для представления в дополнительном коде это может потребовать как минимум  $2^w$  бит. Вместо этого умножение со знаком в языке C обычно выполняется усечением  $2w$ -разрядного произведения до  $w$  бит.

Обозначим это значение как  $x *^s_w y$ . Усечение числа в дополнительном коде до  $w$  бит эквивалентно вычислению его значения по модулю  $2^w$  с последующим преобразованием из представления без знака в представление в дополнительном коде, что дает:

**ПРИНЦИП:** умножение целых в дополнительном коде.

Для  $x$  и  $y$  таких, что  $TMin_w \leq x, y \leq TMax_w$ :

$$x *^s_w y = U2T_w((x \cdot y) \bmod 2^w). \quad (2.17)$$

Мы утверждаем, что представление операции умножения на битовом уровне идентично как для целых без знака, так и для целых в дополнительном коде, как отмечено в следующем принципе.

**ПРИНЦИП:** произведения целых без знака и целых в дополнительном коде на битовом уровне эквивалентны.

Пусть  $\vec{x}$  и  $\vec{y}$  – битовые векторы с длиной  $w$ . Определим целые числа  $x$  и  $y$  как значения, представленные этими векторами в форме дополнительного кода:  $x = B2T_w(\vec{x})$  и  $y = B2T_w(\vec{y})$ . Определим неотрицательные целые числа  $x'$  и  $y'$  как значения, представленные этими векторами в форме без знака:  $x' = B2T_w^t(\vec{x})$  и  $y' = B2T_w^u(\vec{y})$ . Тогда

$$T2B_w(x \ast_w^t y) = U2B_w(x' \ast_w^u y').$$

В качестве иллюстрации в табл. 2.14 показаны результаты умножения различных 3-разрядных чисел. Для каждой пары операндов на битовом уровне выполняется умножение без знака и в дополнительном коде, дающее 6-разрядные произведения, а затем усекаем их до 3 бит. Усеченное произведение без знака всегда равно  $x \cdot y \bmod 8$ . Битовые представления обоих усеченных произведений идентичны, даже притом что полные 6-разрядные представления отличаются.

**Таблица 2.14.** Примеры 3-разрядного беззнакового умножения и умножения в дополнительном коде. Несмотря на то что полные представления произведений на битовом уровне могут отличаться, представления усеченных произведений идентичны

Режим	$x$		$y$		$x \cdot y$		Усеченное $x \cdot y$	
Без знака	5	[101]	3	[011]	15	[001111]	7	[111]
В дополнительном коде	-3	[101]	3	[011]	-9	[110111]	-1	[111]
Без знака	4	[100]	7	[111]	28	[011100]	4	[100]
В дополнительном коде	-4	[100]	-1	[111]	4	[000100]	-4	[100]
Без знака	3	[011]	3	[011]	9	[001001]	1	[001]
В дополнительном коде	3	[011]	3	[011]	9	[001001]	1	[001]

**ВЫВОД:** произведения целых без знака и целых в дополнительном коде на битовом уровне эквивалентны.

Из уравнения 2.6 имеем  $x' = x + x_{w-1}2^w$  и  $y' = y + y_{w-1}2^w$ . Вычисление произведения этих значений по модулю  $2^w$  дает:

$$\begin{aligned}
 (x' \cdot y') \bmod 2^w &= [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w \\
 &= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w \\
 &= (x' \cdot y') \bmod 2^w.
 \end{aligned} \tag{2.18}$$

Члены с весом  $2^w$  и  $2^{2w}$  сокращаются из-за оператора модуля. Согласно уравнению 2.17 имеем  $x \ast_w^t y = U2T_w((x \cdot y) \bmod 2^w)$ . Мы можем применить операцию  $T2U_w$  к обеим сторонам, чтобы получить:

$$T2U_w(x \ast_w^t y) = T2U_w(U2T_w((x \cdot y) \bmod 2^w)) = (x \cdot y) \bmod 2^w.$$

Подставив этот результат в уравнения 2.16 и 2.18, получаем:

$$T2U_w(x \ast_w^t y) = (x' \cdot y') \bmod 2^w = x' \ast_w^u y'.$$

Теперь можно применить  $U2B_w$  к обеим сторонам, чтобы получить:

$$U2B_w(T2U_w(x \ast_w^t y)) = T2B_w(x \ast_w^t y) = U2B_w(x' \ast_w^u y').$$

### Уязвимость в библиотеке XDR

В 2002 году обнаружилось, что код, поставляемый компанией Sun Microsystems в составе библиотеки XDR – широко используемого средства обмена структурами данных между программами, – имеет уязвимость, возникающую из-за возможности переполнения при умножении.

Ниже приводится похожий код с такой же уязвимостью:

```

1 /* Иллюстрация уязвимого кода, аналогичного тому, что был обнаружен
2  * в библиотеке XDR компании Sun.
3  */
4 void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
5     /*
6      * Выделяет буфер для ele_cnt объектов с размерами ele_size байт
7      * каждый и копирует в него данные из ele_src
8      */
9     void *result = malloc(ele_cnt * ele_size);
10    if (result == NULL)
11        /* malloc потерпела неудачу */
12        return NULL;
13    void *next = result;
14    int i;
15    for (i = 0; i < ele_cnt; i++) {
16        /* Скопировать i-й объект */
17        memcpy(next, ele_src[i], ele_size);
18        /* Переместить указатель на следующий объект */
19        next += ele_size;
20    }
21    return result;
22 }
```

Функция `copy_elements` предназначена для копирования `ele_cnt` структур данных, каждая из которых имеет размер `ele_size` байт, в буфер, выделенный функцией в строке 9. Требуемый объем буфера вычисляется как `ele_cnt * ele_size`.

Однако представьте, что злонамеренный программист вызывает эту функцию с `ele_cnt`, равным  $1\,048\,577\ (2^{20} + 1)$ , и `ele_size`, равным  $4096\ (2^{12})$ , из 32-разрядной программы. Тогда при умножении в строке 9 произойдет переполнение, в результате чего будет выделено только 4096 байт, а не  $4\,294\,971\,392$ , необходимых для хранения такого количества данных. Цикл, начинающийся со строки 15, будет пытаться скопировать все эти байты, пока не выйдет за границы выделенного буфера и, как следствие, не разрушит другие структуры данных. Это может привести к сбою программы или иным нарушениям в работе системы.

Код, созданный компанией Sun, использовался почти в каждой операционной системе и в таких распространенных программах, как Internet Explorer и система аутентификации Kerberos. Группа реагирования на компьютерные чрезвычайные ситуации (Computer Emergency Response Team, CERT) – организация, управляемая институтом разработки программного обеспечения Карнеги–Меллона, обнаружила эту уязвимость и выпустила рекомендацию «CA-2002-25», получив которую, многие компании поспешили исправить свой код. К счастью, сообщений о проблемах, вызванных этой уязвимостью, не поступало.

Подобная уязвимость существовала во многих реализациях библиотечной функции `calloc`. Впоследствии она была исправлена. К сожалению, многие программисты вызывают функции распределения памяти, такие как `malloc`, используя в качестве аргументов арифметические выражения, не проверяя их на переполнение. Разработка надежной версии `calloc` будет предложена вам в упражнении 2.76.

**Упражнение 2.34 (решение в конце главы)**

Заполните следующую таблицу, подставив результаты умножения различных 3-разрядных чисел, по аналогии с табл. 2.14:

Режим	$x$		$y$		$x \cdot y$		Усеченное $x \cdot y$	
Без знака	_____	[100]	_____	[101]	_____	_____	_____	_____
В дополнительном коде	_____	[100]	_____	[101]	_____	_____	_____	_____
Без знака	_____	[010]	_____	[111]	_____	_____	_____	_____
В дополнительном коде	_____	[010]	_____	[111]	_____	_____	_____	_____
Без знака	_____	[110]	_____	[110]	_____	_____	_____	_____
В дополнительном коде	_____	[110]	_____	[110]	_____	_____	_____	_____

**Упражнение 2.35 (решение в конце главы)**

Вы получили задание разработать функцию `tmult_ok`, которая определяет, можно ли умножить два аргумента, не вызывая переполнения. Вот ваше решение:

```
/* Определяет возможность умножения аргументов без переполнения */
int tmult_ok(int x, int y) {
    int p = x*y;
    /* Либо x == 0, либо разделить p на x и сравнить с y */
    return !x || p/x == y;
}
```

Вы проверили эту функцию с несколькими значениями  $x$  и  $y$ , и, похоже, она работает правильно. Однако ваш коллега говорит вам: «Коль скоро я не могу использовать вычитание для проверки переполнения при сложении (см. упражнение 2.31), то как вы можете использовать деление для проверки переполнения при умножении?»

Придумайте математическое обоснование своего подхода. Для начала докажите, что случай  $x = 0$  обрабатывается правильно. В противном случае рассмотрите  $w$ -разрядные числа  $x$  ( $x \neq 0$ ),  $y$ ,  $p$  и  $q$ , где  $p$  – результат умножения  $x$  и  $y$  в дополнительном коде, а  $q$  – результат деления  $p$  на  $x$ .

1. Покажите, что  $x \cdot y$ , целочисленное произведение  $x$  и  $y$ , можно записать в форме  $x \cdot y = p + t2^w$ , где  $t \neq 0$  тогда и только тогда, когда при вычислении  $p$  имеет место переполнение.
2. Покажите, что  $p$  можно записать в форме  $p = x \cdot q + r$ , где  $|r| < |x|$ .
3. Покажите, что  $q = y$  тогда и только тогда, когда  $r = t = 0$ .

**Упражнение 2.36 (решение в конце главы)**

Для случая, когда тип данных `int` имеет размер 32 бита, разработайте версию `tmult_ok` (упражнение 2.35), которая использует 64-разрядную точность типа данных `int64_t` без использования деления.

**Упражнение 2.37 (решение в конце главы)**

Вам дано задание исправить уязвимость в коде библиотеки XDR, показанном во врезке «Уязвимость в библиотеке XDR», возникающей, когда оба типа данных, `int` и `size_t`, имеют размер 32 бита. Вы решили исключить возможность переполнения при умножении, вы-



числив количество байтов, которые нужно выделить, используя для этого тип `uint64_t`. Вы заменили исходный вызов `malloc` (строка 9), как показано ниже:

```
uint64_t asize =
    ele_cnt * (uint64_t) ele_size;
void *result = malloc(asize);
```

Напомним, что аргумент функции `malloc` имеет тип `size_t`.

1. Устранит ли ваш код уязвимость?
2. Как еще можно изменить код, чтобы устранить уязвимость?

### 2.3.6. Умножение на константу

Традиционно инструкция целочисленного умножения на многих машинах выполнялась довольно медленной, требуя 10 или более тактов, в то время как другие целочисленные операции, такие как сложение, вычитание, операции с битами и сдвиг, требовали только 1 такта. Даже на Intel Core i7 Haswell, который мы используем в качестве эталонной машины, целочисленное умножение занимает 3 такта. Как следствие одна из важных оптимизаций, используемых компиляторами, – это попытка заменить умножение на константу комбинацией операций сдвига и сложения. Сначала мы рассмотрим случай умножения на степень двойки, а затем обобщим его на произвольные константы.

**ПРИНЦИП:** умножение на степень 2.

Пусть  $x$  – целое число без знака, представленное комбинацией битов  $[x_{w-1}, x_{w-2}, \dots, x_0]$ . Тогда для любого  $k \geq 0$  представление  $x2^k$  без знака из  $w + k$  бит задается как  $[x_{w-1}, x_{w-2}, \dots, x_0, 0, \dots, 0]$ , где справа добавлено  $k$  нулей.

Так, например, 11 можно представить в 4-разрядном виде как [1011]. Сдвиг влево на  $k = 2$  дает 6-разрядный вектор [101100], представляющий число без знака  $11 \cdot 4 = 44$ .

**ВЫВОД:** умножение на степень двойки.

Это свойство можно вывести с помощью уравнения 2.1:

$$\begin{aligned} B2U_{w+k}([x_{w-1}, x_{w-2}, \dots, x_0, 0, \dots, 0]) &= \sum_{i=0}^{w-1} x_i 2^{i+k} \\ &= \left[ \sum_{i=0}^{w-1} x_i 2^i \right] \cdot 2^k \\ &= x2^k. \end{aligned}$$

Сдвиг влево на  $k$  бит при фиксированном размере слова отбрасывает старшие  $k$  бит, давая

$$[x_{w-k-1}, x_{w-k-2}, \dots, x_0, 0, \dots, 0],$$

но то же верно для умножения слов фиксированного размера. Таким образом, сдвиг значения влево эквивалентен умножению без знака на степень 2:

**ПРИНЦИП:** умножение целых без знака на степень двойки.

В языке С для беззнаковых переменных  $x$  и  $k$  со значениями  $x$  и  $k$ , такими что  $0 \leq k < w$ , выражение  $x \ll k$  дает значение  $x \ll_w^u 2^k$ .

Поскольку на уровне битов арифметические операции с числами в дополнительном коде фиксированного размера эквивалентны операциям с числами без знака, мы можем утверждать, что между сдвигом влево и умножением на степень 2 в дополнительном коде существует взаимосвязь:

**ПРИНЦИП:** умножение целых в дополнительном коде на степень двойки.

В языке С для переменных  $x$  и  $k$  со значением  $x$  в дополнительном коде и значением  $k$  без знака, таким что  $0 \leq k < w$ , выражение  $x \ll k$  дает значение  $x \ll_w^t 2^k$ .

Обратите внимание, что умножение на степень 2 может вызвать переполнение и в арифметике без знака, и в арифметике в дополнительном коде. Полученный результат показывает, что даже в этом случае можно получить тот же эффект, выполняя сдвиги. Вернемся к нашему предыдущему примеру: мы сдвинули 4-разрядную комбинацию [1011] (числовое значение 11) влево на две позиции и получили [101100] (числовое значение 44). Усечение этого результата до 4 бит дает [1100] (числовое значение  $12 = 44 \bmod 16$ ).

Учитывая, что целочисленное умножение обходится дороже, чем сдвиг и сложение, многие компиляторы С пытаются заменить умножение целого числа на константу комбинацией сдвигов, сложений и вычитаний. Например, предположим, что программа содержит выражение  $x * 14$ . Так как  $14 = 2^3 + 2^2 + 2^1$ , компилятор может переписать это выражение как  $(x \ll 3) + (x \ll 2) + (x \ll 1)$ , заменив одно умножение тремя сдвигами и двумя сложениями. Оба способа дадут один и тот же результат, независимо от того, является ли  $x$  числом без знака или со знаком, даже если умножение вызовет переполнение. Более того, компилятор также может использовать свойство  $14 = 2^4 - 2^1$ , чтобы переписать умножение как  $(x \ll 4) - (x \ll 1)$ , заменив умножение двумя сдвигами и одним вычитанием.

#### Упражнение 2.38 (решение в конце главы)

Как будет показано в главе 3, инструкция LEA может выполнять вычисления в форме  $(a \ll k) + b$ , где  $k$  – либо 0, 1 или 2, а  $b$  – либо 0, либо некоторая программная величина. Компилятор часто использует эту инструкцию для выполнения умножения на константы. Например,  $3 * a$  можно вычислить так:  $(a \ll 1) + a$ .

Рассматривая случаи, когда  $b$  равно 0 или  $a$  и все возможные значения  $k$ , какие числа, кратные  $a$ , можно рассчитать с помощью одной инструкции LEA?

Теперь обобщим наш пример и рассмотрим задачу генерации кода, реализующего выражение  $x * K$  для некоторой константы  $K$ . Компилятор может выразить двоичное представление  $K$  как чередующуюся последовательность нулей и единиц:

$$[(0...0) (1...1) (0...0) \dots (1...1)].$$

Например, 14 можно записать как  $[(0...0) (111) (0)]$ . Рассмотрим серию единиц от позиции  $n$  до позиции  $m$  (где  $n \geq m$ ). (Для случая с числом 14 мы имеем  $n = 3$  и  $m = 1$ .) Влияние этих битов на произведение можно выразить любой из двух форм:

- форма А:  $(x \ll n) + (x \ll (n - 1)) + \dots + (x \ll m)$ ;
- форма В:  $(x \ll (n + 1)) - (x \ll m)$ .

Сложив результаты для каждого прогона, мы сможем вычислить  $x * K$  без инструкции умножения. Конечно, выигрыш от использования комбинаций сдвига, сложения и вычитания по сравнению с одной инструкцией умножения зависит от относительной скорости этих инструкций, а они могут сильно зависеть от машины. Большинство компиляторов выполняют эту оптимизацию, только когда требуется выполнить небольшое количество сдвигов, сложений и вычитаний.

#### Упражнение 2.39 (решение в конце главы)

Как можно изменить выражение в форме В для случая, когда позиция  $n$  соответствует самому старшему биту?

#### Упражнение 2.40 (решение в конце главы)

Для каждого из следующих значений  $K$  найдите способы выразить  $x * K$ , используя только заданное количество операций, при этом можно считать, что сложение и вычитание имеют сопоставимую стоимость. Возможно, вам придется использовать некоторые уловки, помимо простых форм А и В, которые мы рассмотрели выше.

$K$	Сдвигов	Сложений/вычитаний	Выражение
6	2	1	_____
31	1	1	_____
-6	2	1	_____
55	2	2	_____

#### Упражнение 2.41 (решение в конце главы)

Для серии из единиц, начиная с позиции  $n$  и заканчивая позицией  $m$  (где  $n \geq m$ ), мы видели, что можно сгенерировать две формы кода, А и В. Как компилятор должен выбирать, какую форму использовать?

### 2.3.7. Деление на степень двойки

На многих машинах целочисленное деление выполняется еще медленнее, чем целочисленное умножение, требуя до 30 тактов. Деление на степень двойки тоже можно выполнить с помощью операций сдвига, но использоваться будет сдвиг вправо, а не влево. Сдвиг вправо имеет две разновидности – логический сдвиг и арифметический сдвиг, – которые применяются к числам без знака и в дополнительном коде соответственно.

Результат целочисленного деления всегда округляется в сторону нуля.

Чтобы получить точное определение, введем некоторые обозначения. Для любого действительного числа  $a$  определим  $[a]$  как уникальное целое число  $a'$  такое, что  $a' \leq a < a' + 1$ . Например,  $[3,14] = 3$ ,  $[-3,14] = -4$  и  $[3] = 3$ . Аналогичным образом определим  $\lceil a \rceil$  как уникальное целое число  $a'$  такое, что  $a' - 1 < a \leq a'$ . Например,  $\lceil 3,14 \rceil = 4$ ,  $\lceil -3,14 \rceil = -3$  и  $\lceil 3 \rceil = 3$ . Для  $x \geq 0$  и  $y > 0$  целочисленное деление должно дать  $\lfloor x/y \rfloor$ , тогда как

для  $x < 0$  и  $y > 0$  оно должно дать  $\lceil x/y \rceil$ . То есть положительный результат следует округлить в меньшую сторону, а отрицательный – в большую.

Случай использования сдвигов с беззнаковой арифметикой прост отчасти потому, что для значений без знака гарантированно выполняется логический сдвиг вправо.

**ПРИНЦИП:** деление целого без знака на степень 2.

В языке C для переменных  $x$  и  $k$  с беззнаковыми значениями  $x$  и  $k$ , такими что  $0 \leq k < w$ , выражение  $x \gg k$  дает значение  $\lfloor x/2^k \rfloor$ .

В качестве примеров в табл. 2.15 показаны результаты выполнения логического сдвига вправо 16-разрядного числа 12 340 для деления на 1, 2, 16 и 256. Нули, сдвинутые слева, показаны курсивом. Также в последнем столбце показан результат, который получился бы, если бы использовалось деление в действительной арифметике. Эти примеры показывают, что результат сдвига последовательно округляется в сторону нуля, как это принято при целочисленном делении.

**ВЫВОД:** деление целого без знака на степень 2.

Пусть  $x$  – целое число без знака, представленное комбинацией битов  $[x_{w-1}, x_{w-2}, \dots, x_0]$ , и пусть  $k$  находится в диапазоне  $0 \leq k < w$ . Пусть  $x'$  – число без знака с  $w - k$ -разрядным представлением  $[x_{w-1}, x_{w-2}, \dots, x_k]$ , и пусть  $x''$  – число без знака с  $k$ -разрядным представлением  $[x_{k-1}, \dots, x_0]$ . В таком случае мы можем видеть, что  $x = 2^k x' + x''$  и что  $0 \leq x < 2^k$ . Отсюда следует, что  $\lfloor x/2^k \rfloor = x'$ .

Логический сдвиг вправо битового вектора  $[x_{w-1}, x_{w-2}, \dots, x_0]$  на  $k$  даст битовый вектор:

$$[0, \dots, 0, x_{w-1}, x_{w-2}, \dots, x_k].$$

Этот битовый вектор имеет числовое значение  $x'$ , которое, как мы видели, является значением, которое получится при вычислении выражения  $x \gg k$ .

**Таблица 2.15.** Деление чисел без знака на степень 2.

Примеры показывают, что логический сдвиг вправо на  $k$  дает тот же эффект, что и деление на  $2^k$  с последующим округлением в сторону нуля

$k$	$\gg k$ (двоичный)	Десятичное	$12\ 340 / 2^k$
0	0011000000110100	12 340	12 340,0
1	0001100000011010	6170	6170,0
4	0000001100000011	771	771,25
8	000000000110000	48	48,203125

Деление на степень двойки числа в дополнительном коде несколько сложнее. Во-первых, сдвиг должен выполняться *арифметически*, чтобы отрицательные значения оставались отрицательными. Давайте посмотрим, какую ценность принесет такой сдвиг вправо.

**ПРИНЦИП:** деление целого в дополнительном коде на степень 2 с округлением вниз.

Пусть переменная  $x$  имеет значение  $x$  в дополнительном коде и переменная  $k$  имеет значение  $k$  без знака такое, что  $0 \leq k < w$ . Тогда арифметический сдвиг вправо  $x \gg k$  даст значение  $\lfloor x/2^k \rfloor$ .

Для  $x \geq 0$  переменная  $x$  имеет 0 в старшем разряде, поэтому арифметический сдвиг вправо даст такой же эффект, как и логический сдвиг вправо. То есть арифметический сдвиг неотрицательного числа вправо на  $k$  позиций подобен делению на  $2^k$ . В табл. 2.16 показаны примеры применения арифметического сдвига вправо к 16-разрядному представлению отрицательного числа  $-12\ 340$ . Когда округление не требуется ( $k = 1$ ), результат будет равен  $x/2^k$ . Когда требуется округление, сдвиг приводит к округлению результата вниз. Например, сдвиг вправо на четыре позиции приводит к округлению фактического результата  $-771,25$  до  $-772$ . Нам нужно скорректировать стратегию, чтобы правильно обрабатывать деление отрицательных значений  $x$ .

**Таблица 2.16.** Применение арифметического сдвига вправо. Примеры показывают, что арифметический сдвиг вправо подобен делению на степень 2, за исключением того, что округление происходит вниз, а не в сторону нуля

$k$	$\gg k$ (двоичный)	Десятичное	$12\ 340 / 2^k$
0	1100111111001100	$-12\ 340$	$-12\ 340,0$
1	1110011111100110	$-6170$	$-6170,0$
4	1111110011111100	$-772$	$771,25$
8	1111111111001111	$-49$	$-48,203125$

**ВЫВОД:** деление целого в дополнительном коде на степень 2 с округлением вниз.

Пусть  $x$  – целое число в дополнительном коде, представленное комбинацией битов  $[x_{w-1}, x_{w-2}, \dots, x_0]$ , и пусть  $k$  находится в диапазоне  $0 \leq k < w$ .

Пусть  $x'$  – число в дополнительном коде, представленное  $w - k$  битами  $[x_{w-1}, x_{w-2}, \dots, x_k]$ , и пусть  $x''$  – число без знака, представленное младшими битами  $k$   $[x_{k-1}, \dots, x_0]$ . Так же как в случае с целым без знака, мы имеем  $x = 2^k x' + x''$  и  $0 \leq x'' < 2^k$ , что дает  $\lfloor x/2^k \rfloor$ . Обратите внимание, что арифметический сдвиг битового вектора  $[x_{w-1}, x_{w-2}, \dots, x_0]$  вправо на  $k$  позиций дает битовый вектор

$$[x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_k],$$

что является результатом расширения знака от  $w - k$  до  $w$  бит  $[x_{w-1}, x_{w-2}, \dots, x_k]$ . Этот сдвинутый битовый вектор представляет число  $\lfloor x/2^k \rfloor$  в дополнительном коде.

Мы можем скорректировать неправильное округление при сдвиге отрицательного числа вправо, «смещая» значение перед сдвигом.

**ПРИНЦИП:** деление целого в дополнительном коде на степень 2 с округлением вверх.

Пусть переменная  $x$  имеет значение  $x$  в дополнительном коде и переменная  $k$  имеет значение  $k$  без знака такое, что  $0 \leq k < w$ . Тогда выражение  $(x + (1 \ll k) - 1) \gg k$ , когда сдвиг выполняется арифметически, даст значение  $\lceil x/2^k \rceil$ .

В табл. 2.17 показано, как добавление соответствующего смещения перед арифметическим сдвигом вправо дает правильное округление результата. В третьем столбце показан результат добавления смещения к числу  $-12\ 340$ , причем младшие  $k$  бит (которые будут сдвинуты вправо) показаны курсивом. Также видно, что биты слева от них могут увеличиваться или не увеличиваться. В случае, когда округление не требуется ( $k = 1$ ), добавление смещения влияет только на биты, сдвигаемые за пределы числа. В случаях, когда требуется округление, добавление смещения приводит к увеличению старших битов, поэтому результат округляется в сторону нуля.

**Таблица 2.17.** Деление целого в дополнительном коде на степень 2. Добавление смещения перед сдвигом вправо дает в результате округление в сторону нуля

$k$	Смещение	-12 340 + смещение (двоичное представление)	$\gg k$ (двоичное представление)	Десятичное	$-12\,340/2^k$
0	0	1100111111001100	1100111111001100	-12 340	-12 340,0
1	1	1100111111001101	1110011111100110	-6170	-6170,0
4	15	1100111111011011	1111110011111101	-771	-771,25
8	255	1101000011001011	1111111111010000	-48	-48,203125

Прием добавления смещения использует свойство  $\lfloor x/y \rfloor = \lfloor (x + y - 1)/y \rfloor$  для целых чисел  $x$  и  $y$  таких, что  $y > 0$ . Например, когда  $x = -30$  и  $y = 4$ , мы получаем  $x + y - 1 = -27$  и  $\lfloor -30/4 \rfloor = -7 = \lfloor -27/4 \rfloor$ . Когда  $x = -32$  и  $y = 4$ , мы получаем  $x + y - 1 = -29$  и  $\lfloor -32/4 \rfloor = -8 = \lfloor -29/4 \rfloor$ .

**ВЫВОД:** деление целого в дополнительном коде на степень 2 с округлением вверх.

Чтобы увидеть, что  $\lfloor x/y \rfloor = \lfloor (x + y - 1)/y \rfloor$ , предположим, что  $x = qy + r$ , где  $0 \leq r < y$ , что дает  $(x + y - 1)/y = q + (r + y - 1)/y$ , поэтому  $\lfloor (x + y - 1)/y \rfloor = q + \lfloor (r + y - 1)/y \rfloor$ . Последний член будет равен 0, когда  $r = 0$ , и 1, когда  $r > 0$ . То есть, добавляя смещение  $y - 1$  к  $x$  и затем округляя частное в меньшую сторону, мы получим  $q$ , когда  $y$  кратно  $x$ , и  $q + 1$  в противном случае.

В случае, когда  $y = 2^k$ , выражение  $x + (1 \ll k) - 1$  даст значение  $x + 2^k - 1$ , и, соответственно, арифметический сдвиг вправо на  $k$  даст  $\lfloor x/2^k \rfloor$ .

Этот анализ показывает, что на машине, использующей дополнительный код для представления целых и выполняющей сдвиг вправо арифметически, выражение

$$(x < 0 ? x + (1 \ll k) - 1 : x) \gg k$$

даст в результате значение  $x/2^k$ .

#### Упражнение 2.42 (решение в конце главы)

Напишите функцию `div16`, которая возвращает значение  $x/16$  для целочисленного аргумента  $x$ . Функция не должна использовать операцию деления, деления по модулю, умножения и любые условные выражения (`if` или `?:`), любые операторы сравнения (например, `<`, `>` или `==`) или любые циклы. Вы можете предположить, что тип данных `int` имеет длину 32 бита и использует представление в дополнительном коде, а сдвиги вправо выполняются арифметически.

Теперь мы видим, что деление на степень 2 можно реализовать с использованием логических или арифметических сдвигов вправо. Именно по этой причине на большинстве машин доступны два типа сдвига вправо. К сожалению, этот подход не распространяется на деление на произвольные константы. В отличие от умножения, нельзя выразить деление на произвольную константу  $K$  в терминах деления на степени двойки.

#### Упражнение 2.43 (решение в конце главы)

В следующем коде опущены определения констант  $M$  и  $N$ :

```
#define M      /* Мистическое число 1 */
#define N      /* Мистическое число 2 */
int arith(int x, int y) {
```

```

int result = 0;
result = x*M + y/N; /* M и N -- мистические числа. */
return result;
}

```

Этот код скомпилирован для конкретных величин M и N. Компилятор оптимизировал умножение и деление, используя рассмотренные методы. Далее приведен перевод сгенерированного машинного кода обратно на язык C:

```

/* Перевод на язык C ассемблерного кода, выполняющего арифметические
   операции */
int optariph (int x, int y)
{
    int t = x;
    x <= 5;
    x -= t;
    if (y < 0) y += 7;
    y >= 3; /* Арифметический сдвиг */
    return x+y;
}

```

Каковы значения M и N?

### 2.3.8. Заключительные размышления о целочисленной арифметике

Как мы видели, «целочисленная» арифметика, выполняемая компьютерами, на самом деле является формой арифметики с абсолютными значениями чисел. Конечный размер слова, используемый для представления чисел, ограничивает диапазон возможных значений, из-за чего результаты могут переполняться. Мы также увидели, что представление в дополнительном коде обеспечивает возможность представления как отрицательных, так и положительных значений, при этом используются те же реализации на уровне битов, которые используются для выполнения беззнаковой арифметики, – такие операции, как сложение, вычитание, умножение и даже деление, имеют идентичное или очень похожее поведение на уровне битов, независимо от формы представления операндов – без знака или в дополнительном коде.

Мы видели, что некоторые соглашения в языке C могут давать удивительные результаты и быть источниками ошибок, которые трудно выявить или понять. В частности, мы видели, что тип данных без знака, несмотря на концептуальную простоту, может показывать поведение, которого не ожидают даже опытные программисты. Мы также видели, что этот тип данных может возникать совершенно неожиданно, например при записи целочисленных констант и при вызове библиотечных подпрограмм.

#### Упражнение 2.44 (решение в конце главы)

Допустим, что тип `int` имеет размер 32 бита и представляет целые со знаком в дополнительном коде. Сдвиг вправо выполняется арифметически для значений со знаком и логически – для значений без знака. Переменные объявлены и инициализированы следующим образом:

```

int x = foo(); /* Произвольная величина */
int y = bar(); /* Произвольная величина */

unsigned ux = x;
unsigned uy = y;

```

Для каждого из следующих выражений на С доказите их истинность (возвращают значение 1) при всех значениях  $x$  и  $y$  или приведите значения  $x$  и  $y$ , при которых они ложны:

1.  $(x > 0) \ || \ (x-1 < 0)$
2.  $(x \ \& \ 7) \ != \ 7 \ || \ (x < 29 < 0)$
3.  $(x * x) \ >= \ 0$
4.  $x < 0 \ || \ -x \ <= \ 0$
5.  $x > 0 \ || \ -x \ >= \ 0$
6.  $x+y == uy+ux$
7.  $x*-y + uy*ux == -x$

### Об институте IEEE

Институт инженеров по электротехнике и электронике (Institute of Electrical and Electronics Engineers, IEEE) – профессиональное сообщество, охватывающее все электронные и компьютерные технологии. Осуществляет издание научных журналов, спонсирует конференции и организует комитеты по формализации стандартов на темы от передачи электрических сигналов до проектирования программного обеспечения. Еще один пример стандартов IEEE – стандарт 802.11 беспроводных компьютерных сетей.

## 2.4. Числа с плавающей точкой

Числа с плавающей точкой представляют рациональные числа в форме  $V = x \times 2^y$ . Они широко используются для расчетов с очень большими числами ( $|V| \gg 0$ ), числами, близкими к 0 ( $|V| \ll 1$ ), и, в более общем случае, в качестве приближения к реальной арифметике.

До 1980-х годов каждый производитель вычислительной техники изобретал свои собственные правила представления чисел с плавающей точкой и детали выполняемых с ними операций. Кроме того, очень часто они не заботились о точности операций, считая, что скорость и простота реализации имеют куда большее значение, нежели числовая точность.

К 1985 году все изменилось с появлением стандарта IEEE 754 – тщательно проработанной методики представления чисел с плавающей точкой и выполняемых с ними операций. Первые попытки в этом направлении были предприняты еще в 1976 году (их спонсировала корпорация Intel), результатом стало создание сопроцессора 8087, обеспечивающего поддержку арифметики с плавающей точкой для процессора 8086. В помощь проектированию стандарта поддержки арифметики с плавающей точкой для будущих процессоров в качестве консультанта был приглашен Уильям Кахан (William Kahan), профессор университета Беркли в Калифорнии. Ему был дан карт-бланш на объединение усилий комиссии по разработке универсального промышленного стандарта под эгидой IEEE. Комиссия единогласно приняла стандарт, очень похожий на тот, что Кахан создал для Intel. Сейчас практически все компьютеры поддерживают то, что получило название *арифметика с плавающей точкой IEEE*. Этот шаг значительно улучшил переносимость научных программных приложений между разными компьютерами.

В данном разделе мы рассмотрим представление чисел в формате IEEE с плавающей точкой. Также не будут обойдены вниманием вопросы *округления*, когда число нельзя точно представить в определенном формате, по причине чего его приходится округлять вверх или вниз. Затем мы обсудим математические свойства сложения, умножения и операторов отношения. Многие программисты в лучшем случае считают числа с плавающей точкой малоинтересными, а в худшем – загадочными и непонятными. Мы уви-



дим, что в действительности формат IEEE весьма прост и понятен, поскольку основан на небольшом и последовательном наборе принципов.

### 2.4.1. Дробные двоичные числа

Первым шагом к пониманию чисел с плавающей точкой является знакомство с двоичными числами, имеющими дробные значения. Для начала рассмотрим более знакомую систему десятичного счисления. В последней используется представление в форме:

$$d_m d_{m-1} \dots d_1 d_0 . d_{-1} d_{-2} \dots d_{-n},$$

где каждая десятичная цифра  $d$  находится в диапазоне от 0 до 9. Такая запись представляет число  $d$ , определенное как

$$d = \sum_{i=-n}^m 10^i \times d_i.$$

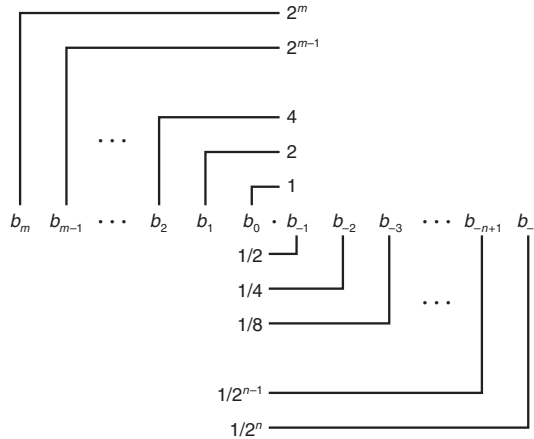
Веса цифр определяются относительно символа десятичной точки, означающей, что цифры слева взвешиваются по неотрицательным степеням 10, давая целочисленные значения, а цифры справа взвешиваются по отрицательным степеням 10, давая дробные значения. Например,  $12.34_{10}$  представляет число

$$1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} = 12 \frac{34}{100}.$$

Рассмотрим, по аналогии, запись в форме

$$b_m b_{m-1} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-n+1} b_{-n},$$

где каждая двоичная цифра (бит)  $b_i$  находится в диапазоне от 0 до 1, как показано на рис. 2.12.



**Рис. 2.12.** Двоичное представление дробных чисел.

Цифры слева от двоичной точки имеют вес  $2^i$ , а цифры справа – вес  $1/2^i$

Такая запись представляет число  $b$ , определенное как

$$d = \sum_{i=-n}^m 2^i \times b_i. \quad (2.19)$$

Теперь символ разделительной точки становится *двоичной точкой*, и цифры слева взвешиваются по неотрицательным степеням 2, а справа – по отрицательным степеням 2. Например,  $101.11_2$  представляет число

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}.$$

Из уравнения (2.19) можно легко понять, что сдвиг двоичной точки на одну позицию влево оказывает эффект деления данного числа на два. Например, если  $101.11_2$  представляет число  $5\frac{3}{4}$ , то  $10.111_2$  представляет число  $2 + 0 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 2\frac{7}{8}$ . Подобным же образом сдвиг двоичной точки на одну позицию вправо оказывает эффект умножения числа на два. Например,  $1011.1_2$  представляет число  $8 + 0 + 2 + 1 + \frac{1}{2} = 11\frac{7}{8}$ .

Обратите внимание, что числа в форме  $0.11\dots1_2$  представляют числа меньше единицы. Например,  $0.111111_2$  представляет  $\frac{63}{64}$ . Для представления таких величин здесь используется сокращение  $1.0 - \varepsilon$ .

При условии что рассматриваются только кодировки конечной длины, в десятичной системе счисления нельзя точно представить такие числа, как  $\frac{1}{3}$  и  $\frac{5}{7}$ . Подобным же образом в дробном двоичном обозначении можно представить только числа, которые можно записать как  $x \times 2^y$ . Другие величины выражаются лишь приближенно. Например, число  $\frac{1}{5}$  можно точно представить в десятичной системе счисления как 0.20. Однако в двоичном дробном виде оно не имеет точного представления и его можно только аппроксимировать с нарастающей точностью путем удлинения двоичного представления:

Представление	Значение	Десятичное
$0.0_2$	$\frac{0}{2}$	$0.0_{10}$
$0.01_2$	$\frac{1}{4}$	$0.25_{10}$
$0.010_2$	$\frac{2}{8}$	$0.25_{10}$
$0.0011_2$	$\frac{3}{16}$	$0.1875_{10}$
$0.00110_2$	$\frac{6}{32}$	$0.1875_{10}$
$0.001101_2$	$\frac{13}{64}$	$0.203125_{10}$
$0.0011010_2$	$\frac{26}{128}$	$0.203125_{10}$
$0.00110011_2$	$\frac{51}{256}$	$0.19921875_{10}$

**Упражнение 2.45 (решение в конце главы)**

Введите недостающую информацию в следующую таблицу:

Дробное значение	Двоичное представление	Десятичное представление
$\frac{1}{8}$	0.01	0.25
$\frac{3}{4}$	_____	_____
$\frac{5}{16}$	_____	_____
_____	10.1101	_____
_____	1.011	_____
_____	_____	5.875
_____	_____	3.1875

**Упражнение 2.46 (решение в конце главы)**

Неточность арифметики с плавающей точкой может иметь катастрофические последствия. 25 февраля 1991 года во время войны в Персидском заливе батарея American Patriot Missile в Дхаране (Саудовская Аравия) не смогла перехватить запущенную иракскую ракету типа «Скад». Ракета взорвалась на территории военной базы США, погубив 28 человек. Генеральное учётное управление США провело подробный анализ сбоя [76] и сделало вывод, что основной причиной стала неточность числовых расчетов. В данном упражнении вам предлагается воспроизвести часть анализа этого ведомства.

Система «Patriot» имеет в своем составе внутренний генератор синхронизирующих импульсов, выполненный в виде счетчика с приращением каждую 0.1 секунды. Для определения времени в секундах программе необходимо умножить показания этого счетчика на 24-рядную величину, являющуюся дробным двоичным приближением  $\frac{1}{10}$ . В частности, двоичное представление  $\frac{1}{10}$  является бесконечной последовательностью 0.000110011[0011]..., где часть, заключенная в скобки, повторяется бесконечно. Компьютер аппроксимировал 0.1, используя только первые 23 бита данной последовательности справа от двоичной точки  $x = 0.00011001100110011001100$ . (См. упражнение 2.51, где обсуждается, как наиболее точно аппроксимировать 0.1.)

1. Каково двоичное представление  $x - 0.1$ ?
2. Каково приближенное десятичное значение  $x - 0.1$ ?
3. При включении системы генератор синхронизирующих сигналов начинает счет с нуля; с этой же отметки начинается отсчет. В данном случае система работала порядка 100 часов. Какова была разница между фактическим временем и вычисленным компьютерной программой?
4. Система прогнозирует время подлета вражеской ракеты, принимая во внимание ее скорость и время последнего показания радара. Если считать, что «Скад» летит со скоростью порядка 2000 м/с, какой величины может достичь ошибка прогноза?

Обычно незначительная ошибка в абсолютном времени, передаваемом тактовым генератором, не влияет на расчет слежения. Оно в большей степени зависит от относительного

времени между двумя последовательными показаниями. Проблема заключалась в том, что программное обеспечение системы «Patriot» было обновлено с целью использования функции чтения показаний с повышенной точностью, однако в новом коде были заменены не все вызовы этой функции. В результате программа слежения использовала точное время для одного показания и неточное – для другого [103].

## 2.4.2. Представление значений с плавающей точкой в стандарте IEEE

Позиционное представление, подобное рассмотренному в предыдущем разделе, неэффективно для очень больших чисел. Например, представление  $5 \times 2^{100}$  будет состоять из комбинации битов 101, за которой следует 100 нулей. Вместо этого предпочтительнее представлять такие числа в форме  $x \times 2^y$  путем задания значений  $x$  и  $y$ .

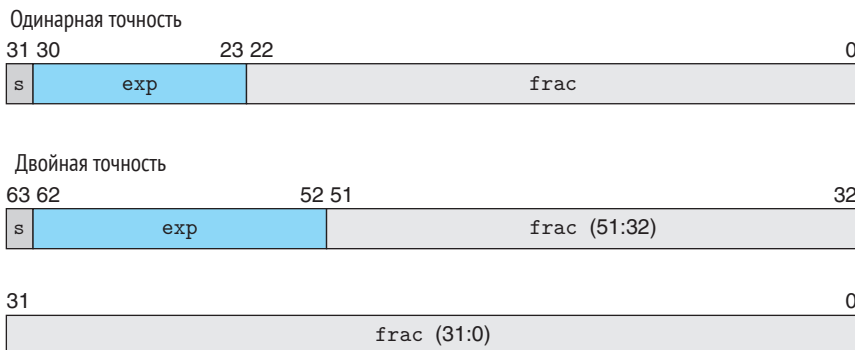
Стандарт IEEE определяет представление чисел с плавающей точкой в виде  $V = (-1)^s \times M \times 2^E$ :

- *символ  $s$*  определяет *знак числа*: отрицательный ( $s = 1$ ) или положительный ( $s = 0$ ), где интерпретация знакового разряда для числового значения 0 рассматривается как особый случай;
- *мантисса  $M$*  – дробное двоичное число в диапазоне от 1 до  $2 - \varepsilon$  или от 0 до  $1 - \varepsilon$ ;
- *показатель  $E$*  взвешивает величину (возможно, отрицательной) степенью 2.

Для кодировки этих величин битовое представление числа с плавающей точкой делится на три поля:

- знаковый бит  $s$  напрямую кодирует символ  $s$ ;
- $k$ -разрядное поле показателя  $\text{exp} = e_{k-1} \dots e_1 e_0$  кодирует показатель  $E$ ;
- $n$ -разрядное поле дробной части  $\text{frac} = f_{n-1} \dots f_1 f_0$  кодирует мантиссу  $M$ , способ кодирования также зависит от того, равно поле показателя 0 или нет.

На рис. 2.13 показано, как эти три поля упакованы в два наиболее распространенных формата. В формате с одинарной точностью (тип `float` в языке C) поля  $s$ ,  $\text{exp}$  и  $\text{frac}$  имеют размеры 1,  $k = 8$  и  $n = 23$  бит, что дает 32-разрядное представление. В формате с двойной точностью (тип `double` в языке C) поля  $s$ ,  $\text{exp}$  и  $\text{frac}$  имеют размеры 1,  $k = 11$ ,  $n = 52$  бит, что дает 64-разрядное представление.



**Рис. 2.13.** Стандартные форматы представления чисел с плавающей запятой.

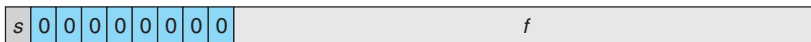
Числа с плавающей запятой представлены тремя полями. Для двух наиболее распространенных форматов они упакованы в 32-разрядные (одинарная точность) или 64-разрядные (двойная точность) слова

Величину, представленную конкретным битовым представлением, можно разделить на три разных варианта (последний имеет два подварианта), в зависимости от значения  $\text{exp}$ . Они показаны на рис. 2.14.

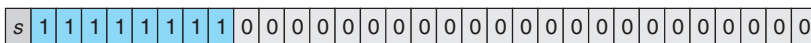
## 1. Нормализованное представление



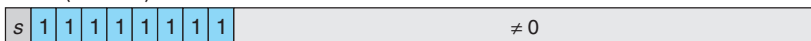
## 2. Ненормализованное представление



### За. Бесконечность



3b. NaN (не число)



**Рис. 2.14.** Категории значений с плавающей запятой одинарной точности. Значение показателя определяет, является число (1) нормализованным, (2) ненормализованным или (3) особым

### Вариант 1: нормализованные значения

Это самый общий случай. Он имеет место, когда комбинация битов  $e_{\text{hr}}$  состоит либо из одних нулей (числовое значение 0), либо из одних единиц (числовое значение 255 для одинарной точности и 2047 для двойной). В этом случае поле показателя интерпретируется как целое со знаком в *смещенной* форме. То есть показатель имеет значение  $E = e - \text{Bias}$  (смещение), где  $e$  – число без знака с битовым представлением  $e_{k-1} \dots e_1 e_0$ , а  $\text{Bias}$  – величина смещения, равная  $2^{k-1} - 1$  (127 для одинарной точности, 1023 – для двойной). Это различие дает диапазон показателей от  $-126$  до  $+127$  для одинарной точности и от  $-1022$  до  $+1023$  – для двойной.

Поле дробной части  $\text{frac}$  интерпретируется как представляющее дробную величину  $f$ , где  $0 \leq f < 1$ , имеющую двоичное представление  $0.f_1 f_2 \dots f_n$ , т. е. с двоичной точкой слева от самого значимого бита. Мантисса определяется как  $M = 1 + f$ . Иногда такое представление называют представлением с неявной ведущей единицей, потому что  $M$  можно рассматривать как число с двоичным представлением  $1.f_1 f_2 \dots f_n$ . Такое представление позволяет получить дополнительный бит точности, поскольку показатель  $E$  всегда можно настроить так, чтобы мантисса  $M$  находилась в диапазоне  $1 \leq M < 2$  (при условии отсутствия переполнения). Поэтому нет необходимости явно представлять ведущий, так как он всегда равен единице.

## Зачем устанавливать смещение для ненормализованных значений

Использование значения показателя 1 – *Bias* вместо простого  $-Bias$  может показаться противоречащим здравому смыслу. Но очень скоро вы увидите, что такое представление упрощает переход от ненормализованных значений к нормализованным.

## Вариант 2: ненормализованные значения

Когда поле показателя состоит только из нулей, то представляемое число находится в ненормализованной форме. В этом случае значение порядка  $E = 1 - Bias$ , а значение мантиссы  $M = f$ , т. е. значение дробной части не имеет неявной ведущей единицы.

Ненормализованные значения служат двум целям. Во-первых, они позволяют представлять числовое значение 0, поскольку для нормализованного представления всегда выполняется условие  $M \geq 1$ , и, следовательно, оно не позволяет представить 0. Фактически представление чисел с плавающей точкой +0.0 имеет комбинацию битов, состоящую из одних нулей: знаковый разряд содержит 0, поле показателя полностью состоит из нулей (что указывает на ненормализованное значение), и поле дробной части тоже состоит из одних нулей, давая  $M = f = 0$ . Интересно, что когда знаковый разряд равен единице, а все другие поля – нулю, то получается значение –0.0. В формате IEEE с плавающей точкой значения –0.0 и +0.0 в одних случаях рассматриваются как разные, а в других – как одинаковые.

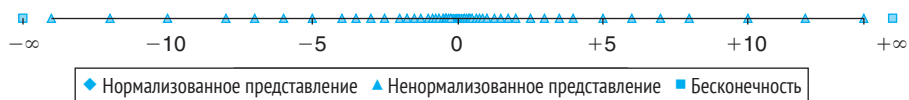
Вторая цель, которую преследуют ненормализованные значения, – представление чисел, близких к 0.0. Они обеспечивают свойство *равномерного приближения к нулю*, при котором возможные числовые значения равномерно располагаются около 0.0.

### Вариант 3: особые значения

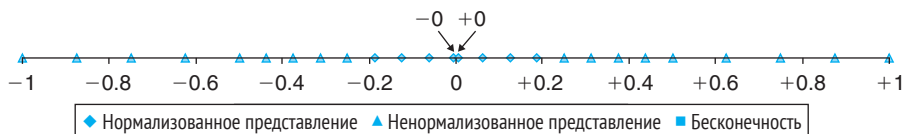
Последняя форма представления значений используется, когда поле показателя состоит из одних единиц. Если при этом поле дробной части состоит из одних нулей, то в результате получается бесконечность: либо  $+\infty$ , когда  $s = 0$ , либо  $-\infty$ , когда  $s = 1$ . Бесконечность может представлять переполнение как при умножении очень больших чисел, так и при делении на ноль. Когда дробная часть не равна нулю, то такое значение называется NaN (сокращенно «Not a Number» – не число). Такие значения возвращаются, когда результат операции нельзя представить вещественным числом или бесконечностью, как, например,  $\sqrt{-1}$  или  $\infty - \infty$ . Они также могут пригодиться в некоторых приложениях для инициализации данных.

### 2.4.3. Примеры чисел

На рис. 2.15 показан набор значений, которые можно представить в гипотетическом 6-разрядном формате, где для порядка отводится  $k = 3$  бита и для дробной части  $n = 2$  бита. Смещение  $Bias = 2^{3-1} - 1 = 3$ . Вверху на рис. 2.15 показаны все представимые значения (кроме NaN). Плюс и минус бесконечность расположены по обоим концам схемы. Нормализованные числа имеют максимальный диапазон  $\pm 14$ . Ненормализованные числа сгруппированы вокруг 0. Они более четко видны во второй части схемы, где показаны только числа в диапазоне от –1.0 до +1.0. Два нуля являются особыми случаями ненормализованных чисел. Обратите внимание, что представляемые числа распределены неравномерно.



(a) Полный диапазон<sup>3</sup>



(b) Значения между –1.0 и +1.0

**Рис. 2.15.** Представляемые значения в 6-разрядном формате с плавающей точкой. Здесь показатель имеет длину  $k = 3$  бита и дробная часть –  $n = 2$  бита. Смещение  $Bias = 3$

В табл. 2.18 приводятся несколько примеров гипотетического 8-разрядного формата представления чисел с плавающей точкой, где для порядка отводится  $k = 4$  бита и для дробной части  $n = 3$  бита. Смещение  $Bias = 2^{4-1} - 1 = 7$ .

**Таблица. 2.18.** Пример неотрицательных значений в 8-разрядном формате с плавающей точкой. Здесь показатель имеет длину  $k = 4$  и дробная часть –  $n = 3$  бита. Смещение  $Bias = 7$

Описание	Битовое представление	Показатель			Дробная часть			Значение	
		$e$	$E$	$2^E$	$f$	$M$	$2^E \times M$	$V$	Десятичное
Ноль	0 0000 000	0	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0,0
Наименьшее положительное	0 0000 001	0	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0,001953
	0 0000 010	0	-6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{2}{512}$	0,003906
	0 0000 011	0	-6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0,005859
...									
Наибольшее ненормализованное	0 0000 111	0	-6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0,013672
Наименьшее нормализованное	0 0001 000	1	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0,015625
	0 0001 001	1	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0,017578
...									
Единица	0 0110 110	6	-1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0,875
	0 0110 111	6	-1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0,9375
	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1,0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1,125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1,25
	...								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224,0
Наибольшее нормализованное	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240,0
Бесконечность	0 1111 000	-	-	-	-	-	-	$\infty$	-

Таблица разделена на три области, представляющие три класса чисел. В разных столбцах показано, как поле показателя кодирует  $E$ , а поле дробной части кодирует ман-

тиссу  $M$ , и вместе они образуют представленное значение  $V = 2^E \times M$ . Ближе к 0 находятся ненормализованные числа, начиная с самого 0. В этом формате они имеют  $E = 1 - 7 = -6$ , что дает вес  $2^E = \frac{1}{64}$ . Дробная часть  $f$  мантииссы  $M$  изменяется в пределах значений  $0, \frac{1}{8}, \dots, \frac{7}{8}$ , давая числа  $V$  в диапазоне от 0 до  $\frac{1}{64} \times \frac{7}{8} = \frac{7}{512}$ .

В табл. 2.18 наименьшие нормализованные числа в этом формате также имеют  $E = 1 - 7 = -6$ , а дробная часть находится в диапазоне  $0, \frac{1}{8}, \dots, \frac{7}{8}$ . Однако мантииссы находятся в диапазоне от  $1 + 0 = 1$  до  $1 + \frac{7}{8} = \frac{15}{8}$ , давая числа  $V$  в диапазоне от  $\frac{8}{512} = \frac{1}{64}$  до  $\frac{15}{512}$ .

Обратите внимание на плавный переход между наибольшим ненормализованным числом  $\frac{7}{512}$  и наименьшим нормализованным числом  $\frac{8}{512}$ . Такая плавность обеспечивается благодаря данному определению  $E$  для ненормализованных значений. Выбором смещения  $1 - Bias$  вместо  $-Bias$  компенсируется отсутствие неявной ведущей единицы в мантииссе ненормализованного числа.

По мере увеличения показателя получаем гораздо большие нормализованные значения, пересекающие 1.0 в направлении наибольшего нормализованного числа. Это число имеет показатель  $E = 7$ , давая вес  $2^E = 128$ . Дробная часть равна  $\frac{7}{8}$ , давая мантииссу  $M = \frac{15}{8}$ . То есть числовое значение  $V = 240$ . Выход за пределы диапазона дает переполнение  $+\infty$ .

Вот одно из интересных свойств данного представления: если рассматривать битовые представления значений в табл. 2.18 как целые числа без знака, то они изменяются по восходящей, как и значения, представляемые числами с плавающей точкой. И это не случайно: формат IEEE проектировался так, чтобы числа с плавающей точкой, можно было сортировать, используя обычную процедуру сортировки целых чисел. Небольшая сложность возникает при обработке отрицательных чисел, поскольку они имеют ведущую единицу и следуют в нисходящем порядке; впрочем, эта трудность преодолима без привлечения арифметики с плавающей точкой для выполнения сравнений (см. упражнение 2.84).

#### Упражнение 2.47 (решение в конце главы)

Рассмотрим 5-разрядное представление числа с плавающей точкой, основанное на формате IEEE, с одним знаковым разрядом, двумя битами показателя ( $k = 2$ ) и двумя битами дробной части ( $n = 2$ ). Смещение показателя составляет  $2^{2-1} - 1 = 1$ .

В следующей таблице перечислен весь диапазон неотрицательных чисел для данного 5-разрядного представления. Заполните таблицу, используя следующие инструкции:

- $e$ : значение показателя как целого без знака;
- $E$ : значение показателя после смещения;
- $2^E$ : числовой вес показателя;
- $f$ : значение дробной части;
- $M$ : значение мантииссы;
- $2^E \times M$ : дробное значение числа (нередуцированное);
- $V$ : дробное значение числа (редуцированное);
- десятичное: десятичное представление числа.

Выразите значения  $2^E, f, M, 2^E \times M$  и  $V$  в виде целых чисел (если возможно) или дробей в форме  $\frac{x}{y}$ . Ячейки с прочерком заполнять не нужно.



Биты	$e$	$E$	$2^E$	$f$	$M$	$2^E \times M$	$V$	Десятичное
0 00 00	_____	_____	_____	_____	_____	_____	_____	_____
0 00 01	_____	_____	_____	_____	_____	_____	_____	_____
0 00 10	_____	_____	_____	_____	_____	_____	_____	_____
0 00 11	_____	_____	_____	_____	_____	_____	_____	_____
0 01 00	_____	_____	_____	_____	_____	_____	_____	_____
0 01 01	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1,25
0 01 10	_____	_____	_____	_____	_____	_____	_____	_____
0 01 11	_____	_____	_____	_____	_____	_____	_____	_____
0 10 00	_____	_____	_____	_____	_____	_____	_____	_____
0 10 01	_____	_____	_____	_____	_____	_____	_____	_____
0 10 10	_____	_____	_____	_____	_____	_____	_____	_____
0 10 11	_____	_____	_____	_____	_____	_____	_____	_____
0 11 00	-	-	-	-	-	-	_____	-
0 11 01	-	-	-	-	-	-	_____	-
0 11 10	-	-	-	-	-	-	_____	-
0 11 11	-	-	-	-	-	-	_____	-

В табл. 2.19 показаны представления и числовые значения некоторых важных чисел с плавающей точкой одинарной и двойной точности. Так же как в случае с 8-разрядным форматом, показанным в табл. 2.18, здесь можно увидеть несколько общих свойств представления чисел с плавающей точкой с  $k$ -разрядным показателем и  $n$ -разрядной дробной частью:

- значение +0.0 всегда имеет битовое представление, состоящее из одних нулей;
- наименьшее положительное ненормализованное значение имеет битовое представление с единицей в младшем бите и нулями во всех остальных. Значение дробной части (и мантиссы)  $M = f = 2^{-n}$ , а значение показателя  $E = -2^{k-1} + 2$ . Отсюда числовое значение  $V = 2^{-n-2^{k-1}+2}$ ;
- наибольшее ненормализованное значение имеет представление, состоящее из поля порядка (все нули) и поля дробной части (все единицы). Значение дробной части (и мантиссы)  $M = f = 1 - 2^{-n}$  (в табл. 2.19 записано как  $1 - \epsilon$ ), и значение порядка  $E = -2^{k-1} + 2$ . Отсюда числовое значение  $V = (1 - 2^{-n}) \times 2^{-n-2^{k-1}+2}$ , что ненамного меньше наименьшего нормализованного значения;
- наименьшее положительное нормализованное значение имеет битовое представление с единицей в младшем бите в поле порядка и нулями во всех остальных. Значение мантиссы  $M = 1$ , а значение порядка  $E = -2^{k-1} + 2$ . Отсюда числовое значение  $V = 2^{-2^{k-1}+2}$ ;
- значение 1.0 имеет битовое представление со всеми единицами в поле порядка, кроме старшего бита, и с нулями во всех остальных битах. Значение мантиссы  $M = 1$ , а значение порядка  $E = 0$ ;
- наибольшее нормализованное значение имеет битовое представление с нулем в знаковом разряде и в младшем бите в поле порядка и с единицами во всех

остальных битах. Дробная часть имеет значение  $f = 1 - 2^{-n}$ , что дает мантиссу  $M = 2 - 2^{-n}$  (в табл. 2.19 записано как  $2 - \epsilon$ ). Значение порядка  $E = 2^{k-1} - 1$ . Отсюда числовое значение

$$V = (2 - 2^{-n}) \times 2^{2^{k-1}-1} = (1 - 2^{-n-1}) \times 2^{2^{k-1}}.$$

**Таблица 2.19.** Примеры неотрицательных чисел с плавающей точкой

Описание	exp	frac	Одинарная точность		Двойная точность	
			Значение	Десятичное	Значение	Десятичное
Ноль	00 ... 00	0 ... 00	0	0.0	0	0.0
Наименьшее ненормализованное	00 ... 00	0 ... 01	$2^{-23} \times 2^{-126}$	$1.4 \times 10^{-45}$	$2^{-52} \times 2^{-1022}$	$4.9 \times 10^{-324}$
Наибольшее ненормализованное	00 ... 00	1 ... 11	$(1 - \epsilon) \times 2^{-126}$	$1.2 \times 10^{-38}$	$(1 - \epsilon) \times 2^{-1022}$	$2.2 \times 10^{-308}$
Наименьшее нормализованное	00 ... 01	0 ... 00	$1 \times 2^{-126}$	$1.2 \times 10^{-38}$	$1 \times 2^{-1022}$	$2.2 \times 10^{-308}$
Единица	01 ... 11	0 ... 00	$1 \times 2^0$	1.0	$1 \times 2^0$	1.0
Наибольшее нормализованное	11 ... 10	1 ... 11	$(2 - \epsilon) \times 2^{127}$	$3.4 \times 10^{38}$	$(2 - \epsilon) \times 2^{1023}$	$1.8 \times 10^{308}$

Одним из полезных упражнений при освоении представления чисел с плавающей точкой является преобразование целых значений в форму с плавающей точкой. Например, в табл. 2.11 было показано число 12345, имеющее двоичное представление [11000000111001]. Создадим его нормализованное представление, сдвинув на 13 позиций вправо за двоичную точку, что даст в результате  $12345 = 1.1000000111001_2 \times 2^{13}$ . Для преобразования в формат одинарной точности IEEE построим поле дробной части, удалив ведущую единицу и добавив 10 нулей в конец, что даст двоичное представление [1000000111001000000000]. Чтобы сконструировать поле порядка, добавим смещение 127 к 13, в результате чего получится число 140, имеющее двоичное представление [10001100]. Объединив все это со знаковым разрядом 0, получим представление числа с плавающей точкой в двоичном виде [01000110010000001110010000000000]. Как рассказывалось в разделе 2.1.3, где мы отметили связь представлений на битовом уровне целого значения 12345 (0x3039) и значения с плавающей точкой одинарной точности 12345.0 (0x4640E400):

```

0 0 0 0 3 0 3 9
0000000000000000000011000000111001
*****
4 6 4 0 E 4 0 0
01000110010000001110010000000000
```

Теперь видно, что совпадение соответствует младшим битам целого числа, прекращаясь как раз перед старшим значимым битом, равным 1 (этот бит образует неявную ведущую единицу), и старшим битом дробной части в представлении с плавающей точкой.

#### Упражнение 2.48 (решение в конце главы)

Как уже упоминалось в упражнении 2.6, целое число 3510593 имеет шестнадцатеричное представление 0x00359141, а число с плавающей точкой одинарной точности 3510593,0 имеет шестнадцатеричное представление 0x4A564504. Выведите данное представление с

плавающей точкой и объясните совпадение между битами целочисленного представления и представления с плавающей точкой.

#### Упражнение 2.49 (решение в конце главы)

1. Для формата плавающей точки с  $n$ -разрядной дробной частью составьте формулу для получения наименьшего положительного целого, которое нельзя представить точно (потому что для полной точности потребуется дробная часть с длиной  $n + 1$  бит). При этом можно допустить, что размер поля порядка  $k$  достаточно велик, чтобы диапазон представимых показателей не являлся ограничением.
2. Какое числовое значение будет иметь это целое в формате с одинарной точностью ( $k = 8$ ,  $n = 23$ )?

### 2.4.4. Округление

Арифметика с плавающей точкой всего лишь аппроксимирует истинную арифметику, потому что данное представление имеет ограниченный диапазон и точность. Следовательно, для значения  $x$ , как правило, необходим систематизированный метод поиска «ближайшего» подходящего значения  $x'$ , которое можно было бы представить в формате с плавающей точкой. В этом заключается задача операции *округления*. Ключевой проблемой здесь является выбор направления округления значения из двух возможных вариантов. Например, пусть у меня в кармане лежат 1,50 доллара, и я хочу округлить их до ближайшего целого числа. Каков должен быть результат: 1 доллар или 2? Существует альтернативный подход: поддерживать верхнюю и нижнюю границы фактического числа. Например, можно определить представляемые значения  $x^-$  и  $x^+$  так, что значение  $x$  гарантированно будет находиться где-то между ними:  $x^- \leq x \leq x^+$ . В формате IEEE представления чисел с плавающей точкой определены четыре разных *режима округления*. По умолчанию используется метод поиска ближайшего соответствия; остальные три можно применять для вычисления верхних и нижних границ.

В табл. 2.20 представлены четыре режима округления, применимых к задаче округления денежной суммы до ближайшего целого доллара. Округление до четного (его еще называют округлением до ближайшего целого) – режим по умолчанию. В этом режиме предпринимается попытка найти ближайшее соответствие. В этом случае 1,40 округлится до 1, а 1,60 – до 2, потому что это ближайшие целые значения. Единственное конструктивное решение, которое придется принять, – это как округлять значения, находящиеся точно посередине между возможными результатами. Режим округления до четного следует правилу, требующему округлять значение вверх или вниз так, чтобы наименее значимая цифра результата была четной. Согласно этому правилу оба значения, 1,50 и 2,50, должны округляться до 2.

**Таблица 2.20.** Иллюстрация режимов округления. Первый режим округляет до ближайшего значения, а остальные три – вверх или вниз

Режим	1,40	1,60	1,50	2,50	–1,50
Округление до четного	1	2	2	2	–2
Округление в сторону нуля	1	1	1	2	–1
Округление вниз	1	1	1	2	–2
Округление вверх	2	2	2	3	–1

Остальные три режима дают в результате гарантированные границы фактического значения. Их можно использовать в некоторых приложениях числовых расчетов. Режим округления в сторону нуля округляет положительные числа вниз, а отрицательные вверх, давая такое значение  $\hat{x}$ , что  $|\hat{x}| \leq |x|$ . Режим округления вниз округляет положительные и отрицательные числа, возвращая результат  $x^-$  такой, что  $x^- \leq x$ . Режим округления вверх округляет положительные и отрицательные числа, возвращая результат  $x^+$  такой, что  $x \leq x^+$ .

Поначалу кажется, что режим округления до четного значения преследует довольно произвольную цель: в чем причина стремления к четным числам? Почему просто не округлять вверх значения, находящиеся посередине между двумя представимыми значениями? Проблема такого правила в том, что легко представить сценарии, в которых округление набора значений данных создаст статистическое смещение при вычислении среднего по выборке. Среднее значение множества чисел, округленных таким способом, будет немного больше истинного среднего значения. И наоборот, если всегда округлять вниз, то среднее значение множества округленных чисел будет немного меньше истинного среднего значения. При округлении до четного числа в большинстве реальных ситуаций такого статистического смещения можно избежать. Здесь в 50 % случаев округление производится вверх и в 50 % случаев вниз.

Округление до четного можно применять, даже когда оно выполняется не до целого числа – нужно просто обращать внимание на четность или нечетность наименее значимой цифры. Например, предположим, что нужно округлить десятичные числа до ближайшей сотой. 1,2349999 будет округлено до 1,23, а 1,2350001 до 1,24, независимо от режима округления, потому что эти числа не находятся в середине диапазона от 1,23 до 1,24. С другой стороны, до 1,24 будут округлены оба числа, 1,2350000 и 1,2450000, потому что оба находятся посередине между границами округления и в обоих случаях результат имеет четную младшую цифру.

Подобным же образом округление до четного применимо к двоичным дробным числам. Мы считаем нулевой наименее значимый бит четным, а единичный – нечетным. Вообще говоря, режим округления имеет значение при наличии комбинации битов в форме  $XX \dots X.YY \dots Y100 \dots$ , где  $X$  и  $Y$  обозначают произвольные значения битов и крайний правый  $Y$  – это позиция, до которой необходимо выполнить округление. Комбинации битов только такой формы обозначают значения, находящиеся посередине между возможными результатами. В качестве примера рассмотрим задачу округления значений до ближайшей четверти (до 2 бит справа от двоичной точки).  $10.00011_2 (2\frac{3}{32})$  можно округлить до  $10.00_2 (2)$ , а  $10.00110_2 (2\frac{3}{16})$  – до  $10.01_2 (2\frac{1}{4})$ , потому что эти значения не находятся посередине между двумя возможными значениями. Мы можем округлить  $10.11100_2 (2\frac{7}{8})$  до  $11.00_2 (3)$ , а  $10.10100_2 (2\frac{5}{8})$  до  $10.10_2 (2\frac{1}{2})$ , потому что эти значения находятся посередине между двумя возможными значениями, и предпочтительнее, чтобы наименее значимый бит был нулевой.

#### Упражнение 2.50 (решение в конце главы)

Покажите, как следующие двоичные дробные значения будут округлены до ближайшей половины (до одного бита справа от двоичной точки) в соответствии с правилом округления до четного. В каждом случае приводите числовые значения до и после округления.

1.  $10.010_2$
2.  $10.011_2$
3.  $10.110_2$
4.  $11.001_2$

**Упражнение 2.51 (решение в конце главы)**

В упражнении 2.46 мы видели, что программное обеспечение, управляющее комплексом «Patriot», аппроксимирует число 0,1 как  $x = 0,00011001100110011001100_2$ . Теперь представьте, что вместо этого представления используется режим округления до четного для определения аппроксимации  $x'$  числа 0.1 с 23 битами справа от двоичной точки.

1. Как выглядит такое двоичное представление  $x'$ ?
2. Какое приближительное десятичное значение даст выражение  $x' - 0,1$ ?
3. Насколько сильно отклонилось бы вычисленное время после 100 часов работы?
4. Насколько далеким от истинного получился бы прогноз местоположения ракеты «Скад»?

**Упражнение 2.52 (решение в конце главы)**

Взгляните на следующие два 7-разрядных представления с плавающей точкой, основанных на формате IEEE. Ни одно из них не имеет знакового бита – они могут представлять только неотрицательные числа.

1. Формат А
  - показатель имеет длину  $k = 3$  бита; смещение показателя равно 3;
  - дробная часть имеет длину  $n = 4$  бита.
2. Формат В
  - показатель имеет длину  $k = 4$  бита; смещение показателя равно 7;
  - дробная часть имеет длину  $n = 3$  бита.

Ниже представлено несколько комбинаций битов в формате А, и ваша задача – преобразовать их в ближайшие значения в формате В. При необходимости применяйте правило округления до четного. Также укажите числовые значения, соответствующие комбинациям битов в формате А и в формате В. Используйте целые (например, 17) или дробные числа (например, 17/64).

Формат А		Формат В	
Биты	Значение	Биты	Значение
011 0000	1	0111 000	1
101 1110	_____	_____	_____
010 1001	_____	_____	_____
110 1111	_____	_____	_____
000 0001	_____	_____	_____

### 2.4.5. Операции с плавающей точкой

Стандарт IEEE устанавливает простое правило определения результатов таких арифметических операций, как сложение или умножение. Результатом некоторой операции  $\odot$  с числами с плавающей точкой  $x$  и  $y$ , представляющими вещественные значения, должно быть  $Round(x \odot y)$  – округленная величина точного результата операции. На практике же разработчики таких операций применяют разнообразные хитроумные трюки, чтобы избежать выполнения точных расчетов, потому что высокая точность необходима только для гарантии получения правильного округленного результата. Когда один из аргументов является особым значением, например  $-0$ ,  $\infty$  или  $NaN$ , то стандарт

определяет вполне разумные правила. Например, выражение  $1/-0$  по определению должно давать в результате  $-\infty$ , а выражение  $1/+0$  должно давать  $+\infty$ .

Одной из сильных сторон метода определения поведения операций с плавающей точкой в стандарте IEEE является его независимость от конкретной аппаратной и программной реализации. То есть абстрактные математические свойства этого метода можно рассматривать, не заботясь о фактической реализации.

Выше уже упоминалось, что сложение целых без знака или в дополнительном коде образует абелеву группу. Сложение вещественных чисел тоже образует абелеву группу, но мы должны учитывать влияние, оказываемое округлением на его свойства. Пусть  $x \text{ } ^f y$  – это  $\text{Round}(x + y)$ . Эта операция определена для всех значений  $x$  и  $y$ , хотя она может давать в результате бесконечность из-за переполнения, несмотря на то что  $x$  и  $y$  – вещественные числа. Данная операция коммутативна  $x \text{ } ^f y = y \text{ } ^f x$  для всех значений  $x$  и  $y$ .

С другой стороны, она не ассоциативна. Например, применительно к числам с плавающей точкой одинарной точности выражение  $(3.14+1e10)-1e10$  даст  $0.0$ : значение  $3.14$  будет утеряно из-за округления. С другой стороны, выражение  $3.14+(1e10-1e10)$  даст  $3.14$ . Как и в абелевой группе, большинство значений с плавающей точкой имеют аддитивные инверсии, т. е.  $x \text{ } ^f -x = 0$ . Исключениями являются бесконечности (потому что  $+\infty - \infty = NaN$ ) и  $NaN$ , так как  $NaN \text{ } ^f x = NaN$  для любого  $x$ .

Отсутствие ассоциативности при сложении значений с плавающей точкой – это наиболее важное групповое свойство. Оно влечет важные следствия как для программ научных вычислений, так и для компиляторов. Например, предположим, что компилятору предложено скомпилировать следующий фрагмент исходного кода:

```
x = a + b + c;  
y = b + c + d;
```

Компилятор может попытаться сэкономить на операциях сложения с плавающей точкой, сгенерировав такой код:

```
t = b + c;  
x = a + t;  
y = t + d;
```

Однако подобный подход к вычислениям может дать иной результат для  $x$ , нежели оригинал, потому что в нем используется ассоциация, отличная от ассоциации операций сложения. В большинстве программ разница будет весьма незначительной, если вообще заметной. К сожалению, компиляторы не могут знать, чем готов поступиться пользователь: производительностью или точным поведением программы-оригинала. Поэтому компиляторы, как правило, очень консервативны и избегают любой оптимизации, которая может оказать даже самое незначительное влияние на функциональность.

С другой стороны, сложение с плавающей точкой удовлетворяет следующему свойству монотонности: если  $a \geq b$ , то  $x \text{ } ^f a \geq x \text{ } ^f b$  для любых значений  $a$ ,  $b$  и  $x$ , отличных от  $NaN$ . Это свойство вещественного (и целочисленного) сложения не относится ни к сложению чисел без знака, ни к сложению чисел в дополнительном коде.

Умножение чисел с плавающей точкой тоже подчиняется многим свойствам, которые обычно ассоциируются с умножением. Пусть  $x \text{ } ^* y = \text{Round}(x \times y)$ . Эта операция близка к математическому умножению (хотя может дать в результате бесконечность или  $NaN$ ), она коммутативна и в качестве единичного элемента умножения имеет  $1.0$ . С другой стороны, она не ассоциативна из-за возможности переполнения или потери точности из-за округления. Например, применительно к числам с плавающей точкой одинарной точности выражение  $(1e20*1e20)*1e-20$  даст в результате  $+\infty$ , а выражение  $1e20*(1e20*1e-20)$  даст  $1e20$ . Кроме того, умножение с плавающей точкой не является дистрибутивным по сложению. Например, применительно к числам с плавающей точ-

кой одинарной точности выражение  $1e20*(1e20-1e20)$  даст в результате  $0.0$ , тогда как выражение  $1e20*1e20-1e20*1e20$  даст  $NaN$ .

С другой стороны, умножение с плавающей точкой удовлетворяет следующим свойствам монотонности для любых значений  $a, b$  и  $c$ , отличных от  $NaN$ :

$$\begin{aligned} a \geq b \text{ и } c \geq 0 &\Rightarrow a^{*f} c \geq b^{*f} c \\ a \geq b \text{ и } c \leq 0 &\Rightarrow a^{*f} c \leq b^{*f} c \end{aligned}$$

Помимо этого, также гарантируется, что  $a^{*f} a \geq 0$  при условии  $a \neq NaN$ . Как уже отмечалось, ни одно из этих свойств монотонности не выполняется для умножения целых без знака или в дополнительном коде.

Такой недостаток ассоциативности и дистрибутивности есть предмет серьезной озабоченности в среде разработчиков компиляторов и приложений для научных расчетов, потому что решение даже такой, на первый взгляд, простой задачи, как написание кода, определяющего факт пересечения двух прямых в трехмерном пространстве, может превратиться в серьезную проблему.

## 2.4.6. Значения с плавающей точкой в C

В языке C имеются два типа данных для представления значений с плавающей точкой: `float` и `double`. На машинах, поддерживающих стандарт IEEE, эти типы данных соответствуют представлениям с плавающей точкой одинарной и двойной точности. Кроме того, в этих машинах используется режим округления до четного значения. К сожалению, стандарт C не требует использования стандарта IEEE, поэтому не существует ни стандартных методов смены режима округления, ни способов получения особых значений, таких как  $-0$ ,  $+\infty$ ,  $-\infty$  или  $NaN$ . В большинстве систем для доступа к этим особенностям предусмотрена комбинация заголовочных файлов (`.h`) и библиотек процедур, однако в каждой системе имеются свои тонкости. Например, компилятор GNU GCC определяет константы `INFINITY` (представляющий  $+\infty$ ) и `NAN` (представляющий  $NaN$ ), которые становятся доступными, если в файле с исходным кодом присутствует такая комбинация директив:

```
#define _GNU_SOURCE 1
#include <math.h>
```

### Упражнение 2.53 (решение в конце главы)

Закончите определения макросов, определяющих следующие значения двойной точности:  $+\infty$ ,  $-\infty$  и  $-0$ :

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
```

При этом нельзя использовать заголовочные файлы (такие как `math.h`), однако можно использовать тот факт, что наибольшее конечное число, которое можно представить с использованием двойной точности, имеет значение, близкое к  $1,8 \times 10^{308}$ .

При преобразовании значений между типами `int`, `float` и `double` программа изменяет числовые значения и битовые представления следующим образом (предполагается, что тип `int` имеет 32-разрядный размер):

- из `int` в `float` переполнения не происходит, но число может быть округлено;
- из `int` или `float` в `double` может быть сохранено точное числовое значение, потому что `double` имеет больший диапазон представимых значений и большую точность (т. е. число значимых разрядов);

- из `double` в `float` может произойти переполнение до  $+\infty$  или  $-\infty$  из-за меньшего диапазона представимых значений. Также значение может быть округлено из-за меньшей точности;
- из `float` или `double` в `int` значение будет округлено в сторону нуля. Например, 1.999 будет преобразовано в 1, а  $-1.999$  – в  $-1$ . Также может произойти переполнение. Для этого случая стандарт C не устанавливает фиксированного результата. Intel-совместимые микропроцессоры используют комбинацию битов `[10...00]` ( $TMin_w$  для размера слова  $w$ ) для представления *неопределенного целого* значения. Любое преобразование из числа с плавающей точкой в целое число, когда нельзя получить разумное целочисленное приближение, дает это значение. Таким образом, выражение `(int)+1e10` дает  $-21483648$ , создавая отрицательное значение из положительного.

#### Упражнение 2.54 (решение в конце главы)

Предположим, что переменные  $x$ ,  $f$  и  $d$  имеют типы `int`, `float` и `double` соответственно. Их значения произвольны, за исключением того, что ни  $f$ , ни  $d$  не равны ни  $+\infty$ , ни  $-\infty$ , ни `NaN`. Докажите для каждого из следующих выражений на языке C, что они всегда будут истинны (т.е. давать в результате 1), либо задайте переменным такие значения, что выражение будет ложным (т.е. давать в результате 0).

1. `x == (int)(double)x`
2. `x == (int)(float)x`
3. `d == (double)(float)d`
4. `f == (float)(double)f`
5. `f == -(-f)` F. `1.0/2 == 1/2.0`
6. `d*d >= 0.0`
7. `(f+d)-f == d`

## 2.5. Итоги

Компьютеры кодируют информацию в виде битов, организованных в последовательности байтов. Для представления целых и вещественных чисел и символьных строк используются различные виды кодировок. В разных моделях компьютеров применяются разные правила кодирования чисел и упорядочивания байтов данных.

Язык C проектировался с учетом широкого диапазона различных реализаций в том, что касается длин слов и представлений чисел. Машины с 64-разрядным размером слова получают все большее распространение, заменяя 32-разрядные машины, которые доминировали на рынке на протяжении 30 лет. Поскольку 64-разрядные машины могут также запускать программы, скомпилированные для 32-разрядных машин, мы сосредоточились на различии между 32- и 64-разрядными программами, а не машинами. Преимущество 64-разрядных программ заключается в том, что они могут выходить за рамки адресного пространства 32-разрядных программ, ограниченного размером 4 Гбайт.

В большинстве машин для представления целых со знаком используется дополнительный код, а для представления чисел с плавающей точкой – стандарт IEEE 754. Понимание этих представлений на уровне битов, а также понимание математических свойств арифметических операций важно для написания программ, работающих корректно во всем диапазоне числовых значений.

В отношении преобразований целых значений со знаком и без знака одного размера большинство реализаций C придерживаются соглашения о том, что комбинация битов не должна меняться. На компьютере, использующем представление в дополнительном



коде, такое поведение характеризуется функциями  $T2U_w$  и  $U2T_w$  для  $w$ -разрядных значений. Неявное преобразование в языке C часто дает результаты, которых многие программисты просто не ожидают и которые приводят к многочисленным программным ошибкам.

Из-за конечной длины числовых представлений компьютерная арифметика обладает свойствами, существенно отличающимися от традиционной целочисленной и вещественной арифметики. Конечная длина может приводить к переполнению чисел, когда они выходят за рамки диапазона представимых значений. Значения с плавающей точкой могут терять значимость, когда оказываются настолько близкими к 0.0, что превращаются в ноль.

Арифметика с конечными целыми числами, реализованная в C, как и в большинстве других языков программирования, обладает некоторыми особыми свойствами, если сравнивать ее с истинной целочисленной арифметикой. Например, из-за переполнения результатом выражения  $x*x$  может быть отрицательное число. Несмотря на это, арифметика без знака и арифметика в дополнительном коде удовлетворяют многим свойствам целочисленной арифметики, включая ассоциативность, коммутативность и дистрибутивность. Это позволяет компиляторам применять множество оптимизаций. Например, при замене выражения  $7*x$  на  $(x<<3)$ -х используются свойства ассоциативности, коммутативности и дистрибутивности наряду с взаимосвязью между сдвигом и умножением на степень двойки.

Мы рассмотрели несколько интересных способов использования комбинаций операций на уровне битов и арифметических операций. Например, мы видели, что согласно арифметике в дополнительном коде  $-x+1$  равно  $-x$ . В качестве еще одного примера предположим, что нужна комбинация битов в форме  $[0, \dots, 0, 1, \dots, 1]$ , состоящая из  $w - k$  нулей, за которыми следует  $k$  единиц. Такие комбинации битов необходимы для операций маскирования. Эту комбинацию на языке C можно создать выражением  $(1<<k)-1$ , использующим то свойство, что нужная комбинация битов имеет числовое значение  $2^k - 1$ . Например, выражение  $(1<<8)-1$  генерирует комбинацию битов  $0xFF$ .

Представления чисел с плавающей точкой аппроксимируют вещественные числа путем кодирования их в формате  $x \times 2^y$ . Наиболее распространенное представление чисел с плавающей точкой определено стандартом IEEE 754. Он допускает несколько разных степеней точности, из которых наиболее широко используется представление с одинарной (32 бита) и двойной (64 бита) точностью. Стандарт IEEE также определяет представления особых значений: плюс и минус бесконечность и не число.

Арифметику с плавающей точкой следует использовать с осторожностью, потому что диапазон и точность представимых значений весьма ограничены, а также потому что они не подчиняются общим математическим свойствам, например ассоциативности.

## Библиографические заметки

В справочных руководствах по языку C [45, 61] рассматриваются свойства разных типов данных и операций. Из них только в книге Стила (Steele) и Харбисона (Harbison) [45] охватываются новые функции, добавленные в ISO C99. И похоже, что пока нет книг, описывающих новые возможности ISO C11. В стандарте C не определены такие подробности, как точные длины слов или способы представления чисел. Они опущены намеренно, чтобы C можно было реализовать на как можно большем количестве моделей компьютеров. Существует несколько книг, специально написанных для программистов на C [59, 74], в которых содержатся предупреждения о проблемах, связанных с переполнением, неявным преобразованием в целые без знака, и некоторые другие тонкости, упомянутые в данной главе. В этих книгах также представлены полезные советы по выбору имен для переменных, стилям программирования и тестированию. В книгах, посвященных языку Java (мы рекомендуем приобрести нашу книгу, написанную в соав-

торстве с Джеймсом Гослингом (James Gosling), создателем данного языка [5]), описываются форматы данных и арифметические операции, поддерживаемые в Java.

В большинстве книг по логическому проектированию [58, 116] имеется раздел, в котором рассматриваются форматы представления данных и арифметические операции. В этих книгах описываются различные способы реализации арифметических вычислений. Книга Овертона (Overton), посвященная арифметике с плавающей точкой IEEE [82], приводит подробное описание этого формата, а также его свойств с точки зрения программиста, занимающегося разработкой вычислительных приложений.

#### **Ariane 5: цена переполнения чисел с плавающей точкой**

Преобразование больших чисел с плавающей точкой в целые является основным источником ошибок в программировании. Известны случаи, когда подобные ошибки имели катастрофические последствия, например первый запуск ракеты Ariane 5, состоявшийся 4 июня 1996 года. Через 37 секунд после старта ракета отклонилась от курса, переломилась и взорвалась. Стоимость находившихся на ее борту спутников связи составляла порядка 500 млн долларов.

В ходе проведенного расследования [73, 33] выяснилось, что компьютер, управляющий инерционной системой навигации, отправил неверные данные на компьютер, управляющий соплами. Вместо отправки контрольной информации о ходе полета была отправлена диагностическая комбинация битов, сообщающая о переполнении при преобразовании 64-разрядного числа с плавающей точкой в 16-разрядное целое со знаком.

Переполнившееся значение отражало горизонтальную скорость ракеты, которая могла в пять раз превышать скорость, развитую предыдущей ракетой Ariane 4. При разработке программного обеспечения для нее числовые значения были тщательно проанализированы и сделан вывод о том, что горизонтальная скорость никогда не переполнит 16-разрядное число. К сожалению, в Ariane 5 эта часть программы была просто использована повторно, без проверки допущений, на которых она базировалась.

## **Домашние задания**

### **Упражнение 2.55 ♦**

Скомпилируйте и запустите код, использующий `show_bytes` (файл `show_bytes.c`) на различных машинах, к которым имеется доступ. Определите порядок следования байтов на этих машинах.

### **Упражнение 2.56 ♦**

Попробуйте вызывать `show_bytes` с разными значениями.

### **Упражнение 2.57 ♦**

Напишите процедуры `show_short`, `show_long` и `show_double`, которые выводят байтовые представления значений типов `short`, `long` и `double` соответственно. Попробуйте вызывать их на разных компьютерах.

### **Упражнение 2.58 ♦♦**

Напишите процедуру `is_little_endian`, которая при компиляции и выполнении на компьютере с обратным (`little endian`) порядком следования байтов будет возвращать 1, а на компьютере с прямым (`big endian`) порядком следования байтов – 0. Эта программа должна работать на компьютерах любого типа, независимо от длины используемых в них слов.

### Упражнение 2.59 ♦♦

Напишите выражение на C, которое будет давать слово, состоящее из наименее значимого байта  $x$  и остальных байтов из  $y$ . Для операндов  $x=0x89ABCDEF$  и  $y=0x76543210$  выражение должно дать результат  $0x765432EF$ .

### Упражнение 2.60 ♦♦

Пусть байты в  $w$ -разрядном слове нумеруются от 0 (младший байт) до  $w/8 - 1$  (старший байт). Напишите на C функцию со следующей сигнатурой, возвращающую целое без знака, в котором  $i$ -й байт в аргументе  $x$  заменен байтом  $b$ :

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

Вот несколько примеров, показывающих, как должна работать функция:

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0xAB345678
```

## Правила представления целых чисел на битовом уровне

В некоторых из следующих задач мы искусственно ограничим, какие программные конструкции вы можете использовать, чтобы помочь вам лучше понять битовые, логические и арифметические операции C. При ответе на эти проблемы ваш код должен следовать этим правилам:

- допущения:
  - ♦ целые числа представлены в дополнительном коде;
  - ♦ сдвиг вправо целых со знаком выполняется арифметически;
  - ♦ тип данных `int` имеет длину  $w$  бит. В некоторых заданиях вам будет предложено использовать конкретное значение  $w$ , в остальных случаях ваш код должен работать при любых значениях  $w$ , кратных 8. Для вычисления  $w$  можно использовать выражение `sizeof(int)<<3`;
- запрещено использовать:
  - ♦ условные выражения (`if` или `?:`), циклы, операторы `switch`, библиотечные функции и макросы;
  - ♦ деление, деление по модулю и умножение;
  - ♦ операторы отношений (`<`, `>`, `<=` и `>=`);
- разрешено использовать:
  - ♦ все битовые и логические операции;
  - ♦ сдвиг влево и вправо, но только на величину от 0 до  $w - 1$ ;
  - ♦ сложение и вычитание;
  - ♦ проверку на равенство (`==`) и неравенство (`!=`); в некоторых задачах использование этих операторов будет явно запрещено;
  - ♦ целочисленные константы `INT_MIN` и `INT_MAX`;
  - ♦ приведение типов `int` и `unsigned` явно или неявно.

Даже с этими правилами вы должны стараться сделать свой код легко читаемым, выбирая описательные имена переменных и используя комментарии для описания логики ваших решений. Ниже приводится пример кода, извлекающего старший байт из целочисленного аргумента  $x$ :

```
/* Возвращает старший байт из x */
int get_msb(int x) {
    /* Величина сдвига на w-8 */
    int shift_val = (sizeof(int)-1)<<3;
```

```

/* Арифметический сдвиг */
int xright = x >> shift_val;
/* Обнулить все байты, кроме младшего */
return xright & 0xFF;
}

```

### Упражнение 2.61 ♦♦

Используя только битовые и логические операции, напишите выражения на C, дающие 1 для условий, описанных ниже, и 0 в противном случае. Представьте, что `x` имеет тип `int`.

1. Хотя бы один бит в `x` равен 1.
2. Хотя бы один бит в `x` равен 0.
3. Хотя бы один бит в младшем байте в `x` равен 1.
4. Хотя бы один бит в старшем байте в `x` равен 0.

### Упражнение 2.62 ♦♦♦

Напишите функцию `int_shifts_are_arithmetic()`, которая возвращает 1, если вызывается на машине, где для типа `int` сдвиг вправо выполняется арифметически, и 0 в противном случае. Функция должна правильно работать на машине с любой длиной слова. Протестируйте написанную функцию на нескольких машинах.

### Упражнение 2.63 ♦♦♦

Допишите следующие функции. Функция `srl` выполняет логический сдвиг вправо, используя результат арифметического сдвига вправо (представленный значением `xsra`). Вам запрещается использовать операции сдвига вправо и деления. Функция `sra` выполняет арифметический сдвиг вправо, используя результат логического сдвига вправо (представленный значением `x srl`). Вам запрещается использовать операции сдвига вправо и деления. Для определения числа битов `w` в типе `int` можно использовать выражение `8*sizeof(int)`. Величина сдвига `k` может находиться в диапазоне от 0 до `w - 1`.

```

unsigned srl(unsigned x, int k) {
    /* Получить результат арифметического сдвига */
    unsigned xsra = (int) x >> k;
    .
    .
    .
    .
    .
}

int sra(int x, int k) {
    /* Получить результат логического сдвига */
    int xsrl = (unsigned) x >> k;
    .
    .
    .
    .
    .
}

```

**Упражнение 2.64** ♦

Напишите следующую функцию:

```
/* Возвращает 1, если хотя бы один нечетный бит в x равен 1;
   0 в противном случае. Предполагается, что w = 32 */
int any_odd_one(unsigned x);
```

Функция должна следовать правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше, но вам разрешено предположить, что тип данных `int` имеет размер `w = 32` бита.

**Упражнение 2.65** ♦♦♦♦

Напишите следующую функцию:

```
/* Возвращает 1, если x содержит нечетное количество 1;
   0 в противном случае. Предполагается, что w = 32 */
int odd_ones(unsigned x);
```

Функция должна следовать правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше, но вам разрешено предположить, что тип данных `int` имеет размер `w = 32` бита.

Функция должна содержать не более 12 арифметических, битовых и логических операций.

**Упражнение 2.66** ♦♦♦

Напишите следующую функцию:

```
/*
 * Генерирует маску, оставляющую самый старший единичный бит в x.
 * Предполагается, что w = 32.
 * Примеры: 0xFF00 -> 0x8000 и 0x6600 -> 0x4000.
 * Если x = 0, то функция должна вернуть 0.
 */
int leftmost_one(unsigned x);
```

Функция должна следовать правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше, но вам разрешено предположить, что тип данных `int` имеет размер `w = 32` бита.

Функция должна содержать не более 15 арифметических, битовых и логических операций.

*Подсказка:* сначала преобразуйте `x` в битовый вектор в форме `[0 ... 011 ... 1]`.

**Упражнение 2.67** ♦♦

Вам дано задание написать процедуру `int_size_is_32()`, возвращающую 1 при выполнении на машине с размером типа `int` в 32 бита, и 0 в противном случае. Вам запрещено использовать оператор `sizeof`. Вот первая версия функции:

```
1 /* На некоторых машинах следующий код работает некорректно */
2 int bad_int_size_is_32() {
3     /* Установить старший бит (msb) на 32-разрядной машине */
4     int set_msb = 1 << 31;
5     /* Сдвинуть старший бит за границы 32-разрядного слова */
6     int beyond_msb = 1 << 32;
7
8     /* переменная set_msb не равна нулю, если размер слова >= 32
9     переменная beyond_msb равна нулю, если размер слова <= 32 */
```

```

10     return set_msb && !beyond_msb;
11 }

```

Однако на 32-разрядном SUN SPARK эта функция вернула 0. Следующее сообщение компилятора указывает на проблему:

```

warning: left shift count >= width of type
(предупреждение: счетчик сдвига влево >= ширины типа)

```

1. В чем написанный код не соответствует стандарту C?
2. Модифицируйте код так, чтобы он корректно выполнялся на любом компьютере с размером типа `int` не меньше 32 бит.
3. Модифицируйте код так, чтобы он корректно выполнялся на любом компьютере с размером типа `int` не меньше 16 бит.

### Упражнение 2.68 ♦♦

Напишите функцию со следующим прототипом:

```

/*
 * Генерирует маску, оставляющую n младших бит
 * Примеры: n = 6 --> 0x3F, n = 17 --> 0xFFFF
 * Предполагается, что 1 <= n <= w
 */
int lower_one_mask(int n);

```

Функция должна следовать правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше. Будьте осторожны со случаем  $n = w$ .

### Упражнение 2.69 ♦♦♦

Напишите функцию со следующим прототипом:

```

/*
 * Выполняет сдвиг влево с переносом старших битов в младшие.
 * Предполагается, что 0 <= n < w
 * Примеры для x = 0x12345678 и w = 32:
 *   n=4 -> 0x23456781, n=20 -> 0x67812345
 */
unsigned rotate_left(unsigned x, int n);

```

Функция должна следовать правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше. Будьте осторожны со случаем  $n = 0$ .

### Упражнение 2.70 ♦♦

Напишите функцию со следующим прототипом:

```

/*
 * Возвращает 1, если x является n-разрядным числом в дополнительном коде;
 * 0 в противном случае.
 * Предполагается, что 1 <= n <= w
 */
int fits_bits(int x, int n);

```

Функция должна следовать правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше.

### Упражнение 2.71 ♦

Вы устроились на работу в компанию, занимающуюся реализацией набора процедур для работы со структурой данных, которая упаковывает четыре однобайтных целых со знаком в 32-разрядный тип `unsigned`. Байты в слове пронумерованы от 0 (младший) до 3 (старший). Вам дано задание реализовать функцию со следующим прототипом для машины, использующей арифметику в дополнительном коде и выполняющей сдвиг вправо арифметически:

```
/* Определение типа данных, упаковывающего 4 байта в тип unsigned */
typedef unsigned packed_t;

/* Извлекает байт из слова. Возвращает его как целое со знаком */
int xbyte(packed_t word, int bytenum);
```

То есть функция должна извлечь указанный байт и расширить его знак до 32-разрядного типа `int`.

Ваш предшественник (уволенный за некомпетентность) написал следующую версию функции:

```
/* Некорректная версия xbyte */
int xbyte(packed_t word, int bytenum)
{
    return (word >> (bytenum << 3)) & 0xFF;
}
```

1. Что неправильно в этом коде?
2. Напишите правильную реализацию функции, использующую только операции сдвига вправо и влево с одним вычитанием.

### Упражнение 2.72 ♦♦

Вам дано задание написать функцию, которая будет копировать целое число `val` в буфер `buf`, но только если в буфере достаточно места.

Вот код, который вы написали:

```
/* Копирует целое число в буфер, если в нем есть место */
/* ВНИМАНИЕ: этот код содержит ошибку! */
void copy_int(int val, void *buf, int maxbytes) {
    if (maxbytes - sizeof(val) >= 0)
        memcpy(buf, (void *) &val, sizeof(val));
}
```

Этот код использует библиотечную функцию `memcpy`. В данном случае, когда требуется просто скопировать значение типа `int`, это выглядит немного странно, но мы хотели проиллюстрировать подход, обычно используемый для копирования более крупных структур данных.

В ходе тестирования кода вы обнаруживаете, что он всегда копирует число в буфер, даже если `maxbytes` имеет слишком маленькое значение.

1. Объясните, почему проверка условия в коде всегда успешна.  
*Подсказка:* оператор `sizeof` возвращает значение типа `size_t`.
2. Покажите, как можно переписать проверку, чтобы она выполнялась правильно.

### Упражнение 2.73 ♦♦

Напишите функцию со следующим прототипом:

```
/* Насыщающее сложение с учетом пределов TMin и TMax */
int saturating_add(int x, int y);
```

Вместо переполнения, как это происходит при обычном сложении целых в дополнительном коде, насыщающее сложение возвращает *TMax*, если наступает положительное переполнение, и *TMin*, если наступает отрицательное переполнение. Насыщающая арифметика обычно используется в программах, выполняющих цифровую обработку сигналов.

Функция должна следовать правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше.

### Упражнение 2.74 ♦♦

Напишите функцию со следующим прототипом:

```
/* Определяет возможность вычитания аргументов без переполнения */
int tsub_ok(int x, int y);
```

Функция должна возвращать 1, если выражение  $x - y$  не вызовет переполнения.

### Упражнение 2.75 ♦♦♦

Предположим, что необходимо рассчитать  $2w$ -разрядное представление  $x \cdot y$ , где  $x$  и  $y$  – значения без знака, на машине с  $w$ -разрядным типом данных `unsigned`. Младшие  $w$  бит произведения можно получить с помощью выражения  $x * y$ , поэтому потребуется только процедура с прототипом

```
unsigned unsigned_high_prod(unsigned x, unsigned y);
```

которая вычисляет старшие  $w$  бит произведения  $x \cdot y$  переменных без знака.

Можно использовать библиотечную функцию с прототипом:

```
int signed_high_prod(int x, int y);
```

которая вычисляет старшие  $w$  бит произведения  $x \cdot y$  переменных со значениями в дополнительном коде. Напишите код, вызывающий эту процедуру, для реализации функции с аргументами без знака. Обоснуйте правильность своего решения.

*Подсказка:* обратите внимание на связь между произведением  $x \cdot y$  со знаком и произведением  $x' \cdot y'$  без знака в выводе уравнения (2.18).

### Упражнение 2.76 ♦

Вот как выглядит объявление библиотечной функции `calloc`:

```
void *calloc(size_t nmemb, size_t size);
```

Согласно документации: «Функция `calloc` выделяет память для массива из `nmemb` элементов с размером `size` байт каждый. Выделенная память очищается – заполняется нулевыми байтами. Если `nmemb` или `size` равны нулю, `calloc` возвращает `NULL`».

Напишите реализацию `calloc`, которая выделяет память вызовом `malloc` и очищает память с помощью `memset`. В вашем коде не должно быть никаких уязвимостей из-за арифметического переполнения, и он должен работать правильно независимо от размера типа данных `size_t`.

Для справки функции `malloc` и `memset` имеют следующие объявления:

```
void *malloc(size_t size);
void *memset(void *s, int c, size_t n);
```

### Упражнение 2.77 ♦♦

Предположим, вам поручили написать код для умножения целочисленной переменной  $x$  на различные постоянные множители  $K$ . Для большей эффективности нужно использовать только операции  $+$ ,  $-$  и  $\ll$ . Напишите выражения умножения для следующих значений  $K$ , используя не больше трех операций в одном выражении.



1.  $K = 17$ .
2.  $K = -7$ .
3.  $K = 60$ .
4.  $K = -112$ .

### Упражнение 2.78 ♦♦

Напишите функцию со следующим прототипом:

```
/* Деление на степень 2. Предполагается, что  $0 \leq k < w-1$  */
int divide_power2(int x, int k);
```

Функция должна вычислять  $x/2^k$  с корректным округлением и следовать правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше.

### Упражнение 2.79 ♦♦

Напишите функцию `mul3div4`, которая принимает целочисленный аргумент  $x$  и вычисляет  $3x/4$ , следуя правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше. Ваш код должен учитывать тот факт, что вычисление  $3x$  может вызвать переполнение.

### Упражнение 2.80 ♦♦♦

Напишите функцию `threefourths`, которая принимает целочисленный аргумент  $x$  и вычисляет  $\frac{3}{4}x$  с округлением в сторону нуля. Она не должна вызывать переполнения. Ваша функция должна следовать правилам, перечисленным в разделе «Правила представления целых чисел на битовом уровне» выше.

### Упражнение 2.81 ♦♦

Напишите выражения на языке C, генерирующие следующие ниже комбинации битов, где  $a^k$  представляет  $k$  повторений символа  $a$ . Предполагается, что тип данных имеет размер  $w$  бит. Ваш код может ссылаться на параметры  $j$  и  $k$ , представляющие значения  $j$  и  $k$ , но не должен использовать параметр, представляющий  $w$ .

1.  $1^{w-k}0^k$
2.  $0^{w-k-j}1^k0^j$

### Упражнение 2.82 ♦

Программы выполняются на машине, где значения типа `int` имеют 32-разрядное представление в дополнительном коде и сдвиг вправо выполняется арифметически. Значения типа `unsigned` тоже имеют 32-разрядное представление.

Мы сгенерировали произвольные целые значения  $x$  и  $y$  и преобразовали их в значения без знака:

```
/* Создать несколько произвольных значений */
int x = random();
int y = random();

/* Преобразовать в целые без знака */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

Для каждого из следующих выражений необходимо указать, *всегда ли* данное выражение дает 1. Если да, то опишите математические принципы, на которых основано

ваше суждение. Если нет, то приведите примеры аргументов, при которых выражения дают 0.

1.  $(x < y) == (-x > -y)$
2.  $((x + y) < 4) + y - x == 17 * y + 15 * x$
3.  $\sim x + \sim y + 1 == \sim(x + y)$
4.  $(ux - uy) == -(\text{unsigned})(y - x)$
5.  $((x >> 2) << 2) <= x$

### Упражнение 2.83 ♦♦

Рассмотрим числа, имеющие двоичное представление, состоящее из бесконечной строки вида  $0.yuuuu\dots$ , где  $y$  –  $k$ -разрядная последовательность. Например,  $\frac{1}{3}$  имеет двоичное представление  $0.01010101\dots$  ( $y = 01$ ), тогда как  $\frac{1}{5}$  имеет двоичное представление  $0.001100110011\dots$  ( $y = 0011$ ).

1. Пусть  $Y = B2U_k(y)$ , т. е. число, имеющее двоичное представление  $y$ . Выведите формулу в терминах  $Y$  и  $k$  для значения, представленного бесконечной строкой. *Подсказка:* изучите влияние сдвига двоичной точки на  $k$  позиций вправо.
2. Каково числовое значение строки для следующих значений  $y$ ?
  - 001;
  - 1001;
  - 000111.

### Упражнение 2.84 ♦

Подставьте возвращаемое значение в следующую процедуру, которая сравнивает первый аргумент со вторым и возвращает 1, если первый аргумент не больше второго. Предполагается, что функция `f2u` возвращает 32-разрядное целое без знака, имеющее то же битовое представление, что и аргумент с плавающей точкой. Можно также предположить, что ни один из аргументов не является *NaN*. Две разновидности нулей,  $+0$  и  $-0$ , считаются равными.

```
int float_le(float x, float y) {
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);

    /* Получить знаковые биты */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    /* Подставьте выражение, использующее только ux, uy, sx и sy */
    return _____;
}
```

### Упражнение 2.85 ♦

Для формата представления значений с плавающей точкой с  $k$ -разрядным полем порядка и  $n$ -разрядной дробной частью напишите формулы вычисления порядка  $E$ , мантиссы  $M$ , дробной части  $f$  и значения  $V$  для следующих чисел. Помимо этого, опишите битовое представление.

1. Число 7.0.
2. Наибольшее нечетное целое число, которое можно представить точно.
3. Обратную величину наименьшего положительного нормализованного значения.

Упражнение 2.86 ♦

Процессоры, совместимые с Intel, также поддерживают формат представления значений с плавающей точкой «повышенной точности» с размером 80 бит, из которых 1 бит отводится под знак,  $k = 15$  бит для поля порядка, один бит для поля *целая часть* и  $n = 63$  бита для поля дробной части. Бит целой части является явной копией подразумеваемого бита в формате представления чисел с плавающей точкой IEEE. То есть он равен 1 для нормализованных значений и 0 для ненормализованных. Заполните следующую таблицу приблизительными значениями некоторых «интересных» чисел в данном формате:

Описание	Повышенная точность	
	Значение	Десятичное значение
Наименьшее положительное ненормализованное	_____	_____
Наименьшее положительное нормализованное	_____	_____
Наибольшее нормализованное	_____	_____

Этот формат можно использовать в программах на языке C, скомпилированных для Intel-совместимых машин, путем объявления данных с типом `long double`. Однако он заставляет компилятор генерировать код с использованием устаревших инструкций для 8087, выполняющих операции с плавающей точкой. Получившаяся в результате программа, скорее всего, будет работать намного медленнее, чем при использовании типов данных `float` и `double`.

Упражнение 2.87 ♦

Версия стандарта IEEE арифметики с плавающей точкой от 2008 года, получившая название IEEE 754-2008, включает 16-разрядный формат представления чисел с плавающей точкой «половинной точности». Первоначально он разрабатывался компаниями, занимающимися компьютерной графикой, и предназначался для хранения данных, в которых требуется более высокий динамический диапазон, чем позволяют получить 16-разрядные целые. Этот формат имеет 1 знаковый бит, 5 бит порядка ( $k = 5$ ) и 10 бит дробной части ( $n = 10$ ). Смещение показателя составляет  $2^{5-1} - 1 = 15$ .

Заполните пустые ячейки в таблице ниже, следуя инструкциям для каждого столбца.

*Шестнадцатеричное значение:* четыре шестнадцатеричные цифры, описывающие форму представления указанного числа.

*M:* значение мантиссы. Это должно быть число в форме  $x$  или  $\frac{x}{y}$ , где  $x$  – целое число, а  $y$  – целая степень 2. Например:  $0, \frac{67}{64}, \frac{1}{256}$ .

*E:* целое значение показателя степени.

*V:* представленное числовое значение. Используйте обозначение  $x$  или  $x \times 2^z$ , где  $x$  и  $z$  – целые числа.

*D:* числовое значение (возможно, приблизительное), напечатанное с использованием спецификатора формата `%f` функции `printf`.

Например, для представления числа  $\frac{7}{8}$ :  $s = 0$ ,  $M = \frac{7}{4}$  и  $E = -1$ . Соответственно, поле показателя будет содержать  $01110_2$  (десятичное значение  $15 - 1 = 14$ ) и поле мантиссы –  $110000000_2$ , или в шестнадцатеричном представлении  $3B00$ . Числовое значение 0.875. Ячейки с прочерками заполнять не нужно.

Описание	Шестнадцатеричное значение	<i>M</i>	<i>E</i>	<i>V</i>	<i>D</i>
–0	_____	_____	_____	–0	–0,0
Наименьшее значение > 2	_____	_____	_____	512	_____
512	_____	_____	_____	_____	512,0
Наибольшее ненормализованное	_____	_____	_____	_____	_____
–∞	_____	–	–	–∞	–∞
Число с шестнадцатеричным представлением 3BB0	3BB0	_____	_____	_____	_____

### Упражнение 2.88 ♦♦

Взгляните на следующие два 9-разрядных представления чисел с плавающей точкой, основанных на формате IEEE.

#### 1. Формат А

- 1 бит знака;
- $k = 5$  бит показателя. Смещение показателя равно 15;
- $n = 3$  бита дробной части.

#### 2. Формат В

- 1 бит знака;
- $k = 4$  бита показателя. Смещение показателя равно 7;
- $n = 4$  бита дробной части.

В следующей таблице дается несколько комбинаций битов в формате А. Ваша задача: преобразовать их в ближайшее значение в формате В. При необходимости округление должно производиться в сторону  $+\infty$ . Кроме того, укажите числовые значения, соответствующие комбинациям битов в форматах А и В. Укажите их целочисленные (например, 17) или дробные (например,  $17/64$  или  $17/2^6$ ) значения.

Формат А		Формат В	
Биты	Числовое значение	Биты	Числовое значение
1 01111 001	$-\frac{9}{8}$	1 0111 0010	$-\frac{9}{8}$
0 10110 011	_____	_____	_____
1 00111 010	_____	_____	_____
0 00000 111	_____	_____	_____
1 11100 000	_____	_____	_____
0 10111 100	_____	_____	_____

### Упражнение 2.89 ♦

Программы выполняются на машине, где значения типа `int` имеют 32-разрядное представление в дополнительном коде. Значения типа `float` представлены в 32-разрядном формате IEEE, а значения типа `double` – в 64-разрядном формате IEEE.

Генерируются произвольные целые значения  $x$ ,  $y$ ,  $z$  и преобразуются в значения типа `double` следующим образом:

```
/* Создать несколько произвольных значений */
int x = random();
int y = random();
int z = random();

/* Преобразовать в double */
double dx = (double) x;
double dy = (double) y;
double dz = (double) z;
```

Для каждого из следующих выражений на C необходимо указать, всегда или нет данное выражение дает 1. Если да, то опишите математические принципы, на которых основано ваше суждение. Если нет, то приведите примеры аргументов, при которых выражения дают 0. Заметьте, что для проверки ответов нельзя пользоваться машиной IA32 с компилятором GCC, потому что в этом случае будет применяться 80-разрядное представление повышенной точности как для `float`, так и для `double`.

1. `(float) x == (float) dx`
2. `dx - dy == (double) (x-y)`
3. `(dx + dy) + dz == dx + (dy + dz)`
4. `(dx * dy) * dz == dx * (dy * dz)`
5. `dx / dx == dz / dz`

## Упражнение 2.90 ♦

Вам дано задание написать функцию на C для расчета представления с плавающей точкой числа  $2^x$ . Понятно, что наилучшим способом будет прямое конструирование IEEE-представления результата с одинарной точностью. Если  $x$  имеет слишком маленькую величину, то процедура должна вернуть 0.0, если слишком большую, то процедура должна вернуть  $+\infty$ . Заполните пропущенные места в листинге, чтобы получить корректный результат. Предполагается, что функция `u2f` возвращает значение с плавающей точкой, имеющее битовое представление, идентичное битовому представлению его аргумента без знака.

```
float fpwr2(int x)
{
    /* Показатель и дробная часть результата */
    unsigned exp, frac;
    unsigned u;

    if (x < _____) {
        /* Слишком маленькое, вернуть 0.0 */
        exp = _____;
        frac = _____;
    } else if (x < _____) {
        /* Ненормализованный результат */
        exp = _____;
        frac = _____;
    } else if (x < _____) {
        /* Нормализованный результат. */
        exp = _____;
        frac = _____;
    } else {
        /* Слишком большое, вернуть +∞ */
        exp = _____;
```

```

        frac = _____;
    }

    /* Упаковать exp и frac в 32-разрядное представление */
    u = exp << 23 | frac;
    /* Вернуть как float */
    return u2f(u);
}

```

### Упражнение 2.91 ♦

Приблизительно в 250 году до н. э. греческий математик Архимед доказал, что  $\frac{223}{71} < \pi < \frac{22}{7}$ . Имей он компьютер и стандартную библиотеку `<math.h>`, он смог бы определить, что аппроксимация числа  $\pi$  значением с плавающей точкой одинарной точности имеет шестнадцатеричное представление `0x40490FDB`. Разумеется, это только аппроксимация, потому что  $\pi$  — это иррациональное число.

1. Какое числовое значение имеет дробная часть этой величины с плавающей точкой?
2. Как выглядит в битовом представлении дробная часть числа  $\frac{22}{7}$ ?  
*Подсказка: смотри упражнение 2.83.*
3. В каком двоичном разряде (относительно двоичной точки) эти две аппроксимации расходятся?

## Правила представления чисел с плавающей точкой на битовом уровне

В следующих упражнениях вам будет предлагаться написать код, обрабатывающий числа с плавающей точкой и работающий непосредственно с их представлениями на битовом уровне. Ваш код должен точно соответствовать соглашениям арифметики с плавающей точкой IEEE, включая использование режима округления до четного, когда требуется округление.

С этой целью мы определим тип данных `float_bits` как эквивалент `unsigned`:

```

/* Для доступа к битовому представлению чисел с плавающей точкой */
typedef unsigned float_bits;

```

Вместо типа `float` вы будете использовать в своем коде тип `float_bits`. Вы также можете использовать типы `int` и `unsigned`, включая константы и операции без знака и со знаком. Вам запрещается использовать какие-либо объединения, структуры или массивы. И что особенно важно, вам запрещается использовать любые типы данных, операции или константы с плавающей точкой. Вместо этого ваш код должен выполнять манипуляции на уровне битов, реализующие указанные операции с плавающей точкой.

Следующая функция иллюстрирует использование этих правил. Для аргумента  $f$  она возвращает  $\pm 0$ , если  $f$  имеет ненормализованное представление (с сохранением знака  $f$ ), и  $f$  в противном случае.

```

/* Если f имеет ненормализованное представление, то возвращает 0, иначе f */
float_bits float_denorm_zero(float_bits f) {
    /* Разложить битовое представление на компоненты */
    unsigned sign = f >> 31;
    unsigned exp  = f >> 23 & 0xFF;
    unsigned frac = f & 0x7FFFFFFF;
    if (exp == 0) {
        /* Ненормализованное. Записать 0 в дробную часть */

```

```

    frac = 0;
}
/* Собрать битовые представления воедино */
return (sign << 31) | (exp << 23) | frac;
}

```

### Упражнение 2.92 ♦♦

Следуя правилам представления чисел с плавающей точкой на битовом уровне, напишите функцию со следующим прототипом:

```

/* Вычисляет  $-f$ . Если  $f$  -- это NaN, то возвращает  $f$ . */
float_bits float_negate(float_bits f);

```

Для числа  $f$  с плавающей точкой эта функция вычисляет  $-f$ . Если  $f$  равно NaN, то функция должна просто вернуть  $f$ .

Проверьте свою функцию, применив ее ко всем  $2^{32}$  возможным значениям аргумента  $f$ , и сравните результаты с теми, что были бы получены с использованием стандартных операций с плавающей точкой на вашем компьютере.

### Упражнение 2.93 ♦♦

Следуя правилам представления чисел с плавающей точкой на битовом уровне, напишите функцию со следующим прототипом:

```

/* Вычисляет  $|f|$ . Если  $f$  -- это NaN, то возвращает  $f$ . */
float_bits float_absval(float_bits f);

```

Для числа  $f$  с плавающей точкой эта функция вычисляет  $|f|$ . Если  $f$  равно NaN, то функция должна просто вернуть  $f$ .

Проверьте свою функцию, применив ее ко всем  $2^{32}$  возможным значениям аргумента  $f$ , и сравните результаты с теми, что были бы получены с использованием стандартных операций с плавающей точкой на вашем компьютере.

### Упражнение 2.94 ♦♦♦

Следуя правилам представления чисел с плавающей точкой на битовом уровне, напишите функцию со следующим прототипом:

```

/* Вычисляет  $2 \cdot f$ . Если  $f$  -- это NaN, то возвращает  $f$ . */
float_bits float_twice(float_bits f);

```

Для числа  $f$  с плавающей точкой эта функция вычисляет  $2.0 \cdot f$ . Если  $f$  равно NaN, то функция должна просто вернуть  $f$ .

Проверьте свою функцию, применив ее ко всем  $2^{32}$  возможным значениям аргумента  $f$ , и сравните результаты с теми, что были бы получены с использованием стандартных операций с плавающей точкой на вашем компьютере.

### Упражнение 2.95 ♦♦♦

Следуя правилам представления чисел с плавающей точкой на битовом уровне, напишите функцию со следующим прототипом:

```

/* Вычисляет  $0.5 \cdot f$ . Если  $f$  -- это NaN, то возвращает  $f$ . */
float_bits float_half(float_bits f);

```

Для числа  $f$  с плавающей точкой эта функция вычисляет  $0.5 \cdot f$ . Если  $f$  равно NaN, то функция должна просто вернуть  $f$ .

Проверьте свою функцию, применив ее ко всем  $2^{32}$  возможным значениям аргумента  $f$ , и сравните результаты с теми, что были бы получены с использованием стандартных операций с плавающей точкой на вашем компьютере.

## Упражнение 2.96 ♦♦♦♦

Следуя правилам представления чисел с плавающей точкой на битовом уровне, напишите функцию со следующим прототипом:

```
/*
 * Вычисляет (int) f.
 * Если преобразование вызывает переполнение или если f -- это NaN,
 * то возвращает 0x80000000
 */
int float_f2i(float_bits f);
```

Для числа  $f$  с плавающей точкой эта функция вычисляет  $(\text{int}) f$ . Функция должна выполнять округление в сторону нуля. Если  $f$  невозможно представить в виде целого числа (например, число слишком большое или  $f$  имеет значение  $NaN$ ), то функция должна вернуть  $0x80000000$ .

Проверьте свою функцию, применив ее ко всем  $2^{32}$  возможным значениям аргумента  $f$ , и сравните результаты с теми, что были бы получены с использованием стандартных операций с плавающей точкой на вашем компьютере.

## Упражнение 2.97 ♦♦♦♦

Следуя правилам представления чисел с плавающей точкой на битовом уровне, напишите функцию со следующим прототипом:

```
/* Вычисляет (float) i */
float_bits float_i2f(int i);
```

Для аргумента  $i$  функция должна вычислить битовое представление  $(\text{float}) i$ .

Проверьте свою функцию, применив ее ко всем  $2^{32}$  возможным значениям аргумента  $f$ , и сравните результаты с теми, что были бы получены с использованием стандартных операций с плавающей точкой на вашем компьютере.

## Решения упражнений

### Решение упражнения 2.1

Понимание связи между шестнадцатеричным и двоичным форматами особенно важно при исследовании программ на машинном уровне. В тексте приводится метод преобразований между этими форматами, однако для его освоения требуется определенный практический навык.

1.  $0x39A7F8$  – в двоичное:

Шестнадцатеричное	3	9	A	7	F	8
Двоичное	0011	1001	1010	0111	1111	1000

2. Двоичное 1100100101111011 – в шестнадцатеричное:

Двоичное	1100	1001	0111	1011
Шестнадцатеричное	C	9	7	B

3.  $0xD5E4C$  – в двоичное:

Шестнадцатеричное	D	5	E	4	C
Двоичное	1101	0101	1110	0100	1100

4. Двоичное 1001101110011110110101 – в шестнадцатеричное:

Двоичное	10	0110	1110	0111	1011	0101
Шестнадцатеричное	2	6	E	7	B	5



## Решение упражнения 2.2

Это упражнение дает шанс задуматься о степенях двойки и их шестнадцатеричных представлениях.

Степень	2 <sup>n</sup> (десятичное)	2 <sup>n</sup> (шестнадцатеричное)
9	512	0x200
19	524 288	0x80000
14	16 384	0x4000
16	65 536	0x10000
17	131 072	0x20000
5	32	0x20
7	128	0x80

## Решение упражнения 2.3

Это упражнение дает шанс попробовать выполнить преобразования между шестнадцатеричным и десятичным представлениями некоторых небольших чисел. Для больших чисел гораздо удобнее и надежнее пользоваться калькулятором или программой преобразования.

Десятичное	Двоичное	Шестнадцатеричное
0	0000 0000	0x00
$167 = 10 \cdot 16 + 7$	1010 0111	0xA7
$62 = 3 \cdot 16 + 14$	0011 1110	0x3E
$188 = 11 \cdot 16 + 12$	1011 1100	0xBC
$3 \cdot 16 + 7 = 55$	0011 0111	0x37
$8 \cdot 16 + 8 = 136$	1000 1000	0x88
$15 \cdot 16 + 3 = 243$	1111 0011	0xF3
$5 \cdot 16 + 2 = 82$	0101 0010	0x52
$10 \cdot 16 + 12 = 172$	1010 1100	0xAC
$14 \cdot 16 + 7 = 231$	1110 0111	0xE7

## Решение упражнения 2.4

Когда вы приступите к отладке программ в машинном коде, обнаружится много случаев, когда пригодится простая шестнадцатеричная арифметика. Вы всегда можете преобразовать числа в десятичные значения, выполнить арифметические операции и преобразовать их обратно, однако возможность работать непосредственно с шестнадцатеричными значениями более эффективна и информативна.

1.  $0x503c + 0x8 = 0x5044$ . Сложение 8 с шестнадцатеричной цифрой с дает 4 с переносом 1 в следующий разряд.
2.  $0x503c - 0x40 = 0x4ffc$ . Вычитание 4 из 3 в позиции второй цифры требует займа из третьей. Поскольку это цифра 0, то заем необходимо произвести и из четвертой позиции.
3.  $0x503c + 64 = 0x507c$ . Десятичное значение 64 ( $2^6$ ) равно шестнадцатеричному значению  $0x40$ .
4.  $0x50ea - 0x503c = 0xae$ . Чтобы вычесть шестнадцатеричное с (десятичное 12) из шестнадцатеричного а (десятичное 10), займем 16 из второй цифры, в результате чего получим шестнадцатеричное е (десятичное 14). Затем переходим ко второй цифре и вычитаем 3 из шестнадцатеричного d (десятичное 13), в результате чего получим шестнадцатеричное а (десятичное 10).

## Решение упражнения 2.5

Это упражнение поможет вам проверить понимание байтового представления данных и двух различных способов упорядочивания байтов.

1. Обратный порядок (little-endian): 21  
Прямой порядок (big-endian): 87
2. Обратный порядок (little-endian): 21 43  
Прямой порядок (big-endian): 87 65
3. Обратный порядок (little-endian): 21 43 65  
Прямой порядок (big-endian): 87 65 43

Напомним, что `show_bytes` перечисляет последовательности байтов, начиная с младшего адреса и перемещаясь в сторону старших адресов. На машине с обратным (little-endian) порядком следования байтов `show_bytes` будет выводить байты в порядке от наименее значимого к наиболее значимому. На машине с прямым (big-endian) порядком следования байтов `show_bytes` будет выводить байты в порядке от наиболее значимого к наименее значимому.

## Решение упражнения 2.6

Это упражнение – еще один шанс попрактиковаться в преобразовании из шестнадцатеричного счисления в двоичное. Оно также заставляет задуматься об особенностях представления целых чисел и чисел с плавающей точкой. Эти вопросы подробно рассматриваются далее в этой главе.

1. Используя запись представленного в тексте примера, запишем две следующие строки:

```

0 0 3 5 9 1 4 1
00000000001101011001000101000001
*****
4 A 5 6 4 5 0 4
01001010010101100100010100000100
```

2. Сдвинув второе слово на две позиции относительно первого, получаем совпадение в 21 разряде.
3. Все биты целого числа оказываются встроенными в число с плавающей точкой, за исключением наиболее значимого бита, имеющего значение 1. Это случай для примера, приведенного в тексте. Кроме этого, число с плавающей точкой имеет некоторые ненулевые старшие биты, не совпадающие с битами целого числа.

## Решение упражнения 2.7

Он выведет: 61 62 63 64 65 66. Напомним, что библиотечная функция `strlen` не учитывает завершающий нулевой символ, поэтому `show_bytes` выведет только шесть символов, включая последний символ «f».

## Решение упражнения 2.8

Это упражнение призвано помочь освоить булевы операции.

Операция	Результат	Операция	Результат
a	[01101001]	a & b	[01000001]
b	[01010101]	a   b	[01111101]
~a	[10010110]	a ^ b	[00111100]
~b	[10101010]		

## Решение упражнения 2.9

Это упражнение иллюстрирует использование булевой алгебры для описания и обоснования рассуждений о реальных системах. Эта цветовая алгебра идентична булевой по битовым векторам с длиной 3.

1. Цвета дополняются дополнением величин  $R$ ,  $G$  и  $B$ . Отсюда можно сделать вывод, что Белый является дополнением Черного, Желтый – Синего, Алый – Зеленого, а Голубой – Красного.
2. Булевы операции выполняются на основе битовых векторов, представляющих цвета. Соответственно, получаем:

$$\begin{array}{lll}
 \text{Синий (001)} & | & \text{Зеленый (010)} = \text{Голубой (011)} \\
 \text{Желтый (110)} & \& \text{Голубой (011)} = \text{Зеленый (010)} \\
 \text{Красный (010)} & \wedge & \text{Алый (101)} = \text{Синий (001)}
 \end{array}$$

## Решение упражнения 2.10

Данная процедура основана на свойствах ассоциативности и коммутативности операции ИСКЛЮЧАЮЩЕЕ ИЛИ (EXCLUSIVE-OR) и том факте, что  $a \wedge a = 0$  для любого  $a$ .

Шаг	*x	*y
Начальное состояние	a	b
Шаг 1	a	$a \wedge b$
Шаг 2	$a \wedge (a \wedge b) = (a \wedge a) \wedge b = b$	$a \wedge b$
Шаг 3	b	$b \wedge (a \wedge b) = (b \wedge b) \wedge a = a$

Смотри упражнение 2.11, где рассматривается случай, когда эта функция терпит неудачу.

## Решение упражнения 2.11

Это упражнение иллюстрирует тонкую и интересную особенность нашей процедуры перестановки на месте.

1. Обе переменные, `first` и `last`, получают значение  $k$ , поэтому фактически будет выполнена попытка поменять местами средний элемент сам с собой.
2. В этом случае оба аргумента,  $x$  и  $y$ , функции `inplace_swap` будут указывать на одно и то же место. Вычисляя  $x \wedge y$ , мы получаем 0. Затем сохраняем 0 в средний элемент массива, и последующие шаги сохраняют этот элемент равным 0. Теперь ясно видно, что наши рассуждения в упражнении 2.10 неявно предполагали, что  $x$  и  $y$  обозначают разные местоположения.
3. Достаточно просто заменить проверку в строке 4 в функции `reverse_array` на `first < last`, потому что нет необходимости менять местами средний элемент с самим собой.

## Решение упражнения 2.12

Ответом являются выражения:

1.  $x \& 0xFF$
2.  $x \wedge \sim 0xFF$
3.  $x | 0xFF$

Это типичные выражения для выполнения низкоуровневых битовых операций. Выражение  $\sim 0xFF$  создает маску, в которой 8 младших бит равны нулю, а все остальные – единице. Обратите внимание, что такая маска создается независимо от длины слова.

В противоположность этому выражение `0xFFFFF00` будет давать правильный результат, только если тип `int` имеет размер 32 бита.

### Решение упражнения 2.13

Это упражнение поможет вам задуматься о связи между логическими операциями и типичными способами применения операций маскирования. Вот полный код:

```
/* Объявления функций, реализующих операции bis и bic */
int bis(int x, int m);
int bic(int x, int m);

/* Вычислить x|y, используя только функции bis и bic */
int bool_or(int x, int y) {
    int result = bis(x,y);
    return result;
}
/* Вычислить x^y, используя только функции bis и bic */
int bool_xor(int x, int y) {
    int result = bis(bic(x,y), bic(y,x));
    return result;
}
```

Операция `bis` эквивалентна булевой операции ИЛИ (OR) – бит устанавливается в `z`, если соответствующий бит установлен в `x` или в `m`. Операция `bic(x, m)` эквивалентна `x & ~m`; нам нужно, чтобы результат был равен 1, только когда соответствующий бит в `x` равен 1, а в `m` равен 0.

Учитывая это, можно реализовать операцию `|` одним вызовом `bis`. Для реализации `^` мы воспользовались свойством:

$$x \wedge y = (x \& \sim y) | (\sim x \& y).$$

### Решение упражнения 2.14

Это упражнение подчеркивает связь в языке C между битовыми и логическими операциями. Распространенной ошибкой является использование битовой операции, когда предполагается логическая, и наоборот.

Выражение	Значение	Выражение	Значение
<code>x &amp; y</code>	0x20	<code>x &amp;&amp; y</code>	0x01
<code>x   y</code>	0x7F	<code>x    y</code>	0x01
<code>~x   ~y</code>	0xDF	<code>!x    !y</code>	0x00
<code>x &amp; !y</code>	0x00	<code>x &amp;&amp; ~y</code>	0x01

### Решение упражнения 2.15

Выражение будет иметь вид: `!(x ^ y)`.

То есть `x ^ y` будет равно нулю, если и только если каждый бит в `x` совпадает с соответствующим битом в `y`. После этого используется способность оператора `!` определять наличие ненулевых битов в слове.

Нет никаких веских причин использовать это выражение вместо простого сравнения `x == y`, кроме как для демонстрации некоторых нюансов логических и битовых операций.

### Решение упражнения 2.16

Целью этого упражнения является развитие понимания различных операций сдвига.

x		x << 3		Логический		Арифметический	
				x >> 2		x >> 2	
Шестнадцатеричное	Двоичное	Двоичное	Шестнадцатеричное	Двоичное	Шестнадцатеричное	Двоичное	Шестнадцатеричное
0xC3	[11000011]	[00011000]	0x18	[00110000]	0x30	[11110000]	0xF0
0x75	[01110101]	[10101000]	0xA8	[00011101]	0x1D	[00011101]	0x1D
0x87	[10000111]	[00111000]	0x38	[00100001]	0x21	[11100001]	0xE1
0x66	[01100110]	[00110000]	0x30	[00011001]	0x19	[00011001]	0x19

### Решение упражнения 2.17

Вообще говоря, упражнения на примерах со словами маленькой длины помогают лучше понять компьютерную арифметику.

Значения без знака соответствуют указанным в табл. 2.2. Для значений в дополнительном коде шестнадцатеричные значения от 0 до 7 имеют нулевой старший бит, дающий неотрицательные значения, тогда как шестнадцатеричные цифры от 8 до F имеют единичный старший бит, дающий отрицательное значение.

Шестнадцатеричное	Двоичное		
$\vec{x}$		$B2U_w(\vec{x})$	$B2T_w(\vec{x})$
0xE	[1110]	$2^3 + 2^2 + 2^1 = 14$	$-2^3 + 2^2 + 2^1 = -2$
0x0	[0000]	0	0
0x5	[0101]	$2^2 + 2^0 = 5$	$2^2 + 2^0 = 5$
0x8	[1000]	$2^3 = 8$	$-2^3 = -8$
0xD	[1101]	$2^3 + 2^2 + 2^0 = 13$	$-2^3 + 2^2 + 2^0 = -3$
0xF	[1111]	$2^3 + 2^2 + 2^1 + 2^0 = 15$	$-2^3 + 2^2 + 2^1 + 2^0 = -1$

### Решение упражнения 2.18

Для 32-разрядного слова любое значение, состоящее из восьми шестнадцатеричных цифр, начинающееся с одной из цифр в диапазоне от 8 до f, представляет отрицательное число. Числа, начинающиеся со строки символов f, – достаточно распространенное явление, потому что отрицательные числа имеют единицы в старших разрядах. Однако нужно быть внимательным. Например, число 0x8048337 имеет только семь цифр. Добавление ведущего нуля дает 0x08048337 – положительное число.

```

4004d0: 48 81 ec e0 02 00 00 sub $0x2e0,%rsp      A. 736
4004d7: 48 8b 44 24 a8 mov -0x58(%rsp),%rax B. -88
4004dc: 48 03 47 28 add 0x28(%rdi),%rax C. 40
4004e0: 48 89 44 24 d0 mov %rax,-0x30(%rsp) D. -48
4004e5: 48 8b 44 24 78 mov 0x78(%rsp),%rax E. 120
4004ea: 48 89 87 88 00 00 00 mov %rax,0x88(%rdi) F. 136
4004f1: 48 8b 84 24 f8 01 00 mov 0x1f8(%rsp),%rax G. 504
4004f8: 00
4004f9: 48 03 44 24 08 add 0x8(%rsp),%rax
4004fe: 48 89 84 24 c0 00 00 mov %rax,0xc0(%rsp) H. 192
400505: 00
400506: 48 8b 44 d4 b8 mov -0x48(%rsp,%rdx,8),%rax I. -72

```

### Решение упражнения 2.19

Функции *T2U* и *U2T* очень своеобразны с математической точки зрения. Важно понимать, как они себя ведут.

Это упражнение решается переупорядочиванием строк в решении упражнения 2.17 в соответствии со значениями в дополнительном коде, а затем подставляя беззнаковое

значение, служащее результатом применения функции. Мы добавили в ответ шестнадцатеричные значения, чтобы сделать этот процесс более конкретным (см. табл.).

### Решение упражнения 2.20

Данное упражнение проверяет понимание уравнения 2.5.

В первых четырех записях значения  $x$  – отрицательные,  $T2U_4(x) = x + 2^4$ . В оставшихся двух записях значения  $x$  неотрицательны и  $T2U_4(x) = x$ .

$\vec{x}$ (шестн.)	$x$	$T2U_4(x)$
0x8	-8	8
0xD	-3	13
0xE	-2	14
0xF	-1	15
0x0	0	0
0x5	5	5

### Решение упражнения 2.21

Это упражнение закрепляет понимание связи между представлениями в дополнительном коде и без знака, а также влияние правил языка C. Напомним, что  $TMin_{32} = -2\ 147\ 483\ 648$  и при приведении к значению без знака превращается в  $2\ 147\ 483\ 648$ . Кроме этого, если какой-либо из операндов является числом без знака, то другой операнд будет приведен к значению без знака до сравнения.

Выражение	Тип	Результат
$-2147483647 - 1 == 2147483648U$	Без знака	1
$-2147483647 - 1 < 2147483648$	Со знаком	1
$-2147483647 - 1U < 2147483648$	Без знака	0
$-2147483647 - 1 < -2147483648$	Со знаком	1
$-2147483647 - 1U < -2147483648$	Без знака	1

### Решение упражнения 2.22

Это упражнение служит конкретной демонстрацией, как расширение знака сохраняет числовое значение представления в дополнительном коде.

- $[1011] \quad -2^3 + 2^1 + 2^0 = -8 + 2 + 1 = -5$
- $[11011] \quad -2^4 + 2^3 + 2^1 + 2^0 = -16 + 8 + 2 + 1 = -5$
- $[111011] \quad -2^5 + 2^4 + 2^3 + 2^1 + 2^0 = -32 + 16 + 8 + 2 + 1 = -5$

### Решение упражнения 2.23

Выражения в этих функциях представляют типичные программные «идиомы» извлечения значений из слова, упакованных в несколько битовых полей. Они используют свойства различных операций сдвига, такие как дополнение нулями и расширение знака. Обратите особое внимание на порядок операций преобразования и сдвига. В `fun1` сдвиги применяются к беззнаковой переменной `word` и, следовательно, выполняются логически. В `fun2` сдвиги применяются после приведения `word` к типу `int` и, следовательно, выполняются арифметически.

- | $w$        | $\text{fun1}(w)$ | $\text{fun2}(w)$ |
|------------|------------------|------------------|
| 0x00000076 | 0x00000076       | 0x00000076       |
| 0x87654321 | 0x00000021       | 0x00000021       |
| 0x000000C9 | 0x000000C9       | 0xFFFFFC9        |
| 0xEDCBA987 | 0x00000087       | 0xFFFFF87        |

- Функция `fun1` извлекает значение из 8 младших бит аргумента, получая целое число в диапазоне от 0 до 255. Функция `fun2` тоже извлекает значение из 8 младших бит аргумента, но выполняет расширение знака. В результате получается число в диапазоне от -128 до 127.

### Решение упражнения 2.24

Эффект усечения числа без знака достаточно понятен, однако это не относится к числам в дополнительном коде. Данное упражнение помогает исследовать свойства усечения, используя слова маленькой длины.

Шестнадцатеричное число		Без знака		Дополнительный код	
Исходное	Усеченное	Исходное	Усеченное	Исходное	Усеченное
0	0	0	0	0	0
2	2	2	2	2	2
9	1	9	1	-7	1
B	3	11	3	-5	3
F	7	15	7	-1	-1

Согласно уравнению 2.9, усечение величины без знака просто определяет остаток от деления по модулю 8. Однако усечение величины со знаком дает немного более сложный эффект. В соответствии с уравнением 2.10 сначала определяется остаток от деления по модулю 8. В результате для аргументов от 0 до 7, а также для аргументов от -8 до -1 получаются значения от 0 до 7. Затем к этим остаткам применяется функция  $U2T_8$ , в результате чего получаются два повторения последовательностей от 0 до 3 и от -4 до -1.

### Решение упражнения 2.25

Упражнение показывает, насколько прост механизм появления ошибок из-за неявного преобразования значения со знаком в значение без знака. Кажется вполне естественным передать параметр `length` в виде числа без знака, потому что вряд ли кому-то придет в голову использовать отрицательную длину. Критерий остановки вычислений `i <= length-1` тоже представляется вполне естественным. Однако их объединение дает совершенно неожиданный результат!

Поскольку параметр `length` не имеет знака, вычисление `0 - 1` производится с использованием арифметики чисел без знака, что эквивалентно сложению по модулю. В результате получается `UMax`. Сравнение `<=` тоже применяется к значениям без знака, и поскольку любое число меньше или равно `UMax`, данное сравнение всегда возвращает истинный результат! В результате в какой-то момент код предпринимает попытку обратиться к несуществующим элементам массива `a`.

Этот код можно исправить, объявив `length` с типом `int` или изменив проверку в цикле `for` на `i < length`.

### Решение упражнения 2.26

Этот пример демонстрирует тонкую особенность беззнаковой арифметики, а также тот факт, что иногда мы используем беззнаковую арифметику, не осознавая этого. Это может привести к очень коварным ошибкам.

1. В каких случаях эта функция даст неверный результат? Функция вернет 1, если `s` короче `t`.
2. Объясните, как возникает этот неверный результат. Поскольку `strlen` возвращает беззнаковый результат, вычитание и сравнение выполняются с использованием беззнаковой арифметики. Когда `s` короче `t`, разность `strlen(s) - strlen(t)` должна быть отрицательной, но вместо получается большое число без знака, которое больше 0.
3. Покажите, как исправить код, чтобы он работал надежно. Нужно заменить проверку на:

```
return strlen(s) > strlen(t);
```

### Решение упражнения 2.27

Эта функция является прямой реализацией правил определения переполнения при сложении целых без знака.

```
/* Определяет, можно ли сложить аргументы без переполнения */
int uadd_ok(unsigned x, unsigned y);
    unsigned sum = x+y;
    return sum >= x;
}
```

### Решение упражнения 2.28

Упражнение является простой демонстрацией арифметики по модулю 16. Самым простым решением будет преобразование шестнадцатеричной комбинации в десятичное значение без знака. Для ненулевых значений  $x$  необходимо найти  $(-\frac{1}{4}x) + x = 16$ , а затем преобразовать дополненную величину обратно в шестнадцатеричную форму.

x		-¼x	
Шестнадцатеричное	Десятичное	Шестнадцатеричное	Десятичное
0	0	0	0
5	5	11	В
8	8	8	8
D	13	3	3
F	15	1	1

### Решение упражнения 2.29

Это упражнение направлено на понимание сложения целых в дополнительном коде.

x	y	x + y	x + ½y	Случай
-12	-15	-27	5	1
[10100]	[10001]	[100101]	[00101]	
-8	-8	-16	-16	2
[11000]	[11000]	[110000]	[10000]	
-9	8	-1	-1	2
[10111]	[01000]	[111111]	[11111]	
2	5	7	7	3
[00010]	[00101]	[000111]	[00111]	
12	4	16	-16	4
[01100]	[00100]	[010000]	[10000]	

### Решение упражнения 2.30

Эта функция является прямой реализацией правил определения переполнения при сложении целых в дополнительном коде.

```
/* Определяет, можно ли сложить аргументы без переполнения */
int tadd_ok(int x, int y);
    int sum = x+y;
    int neg_over = x < 0 && y < 0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum < 0;
    return !neg_over && !pos_over;
}
```

### Решение упражнения 2.31

Ваш коллега мог бы узнать, изучив раздел 2.3.2, что сложение целых в дополнительном коде образует абелеву группу, поэтому выражение  $(x + y) - x$  будет давать в результате  $y$ ,



независимо от того, имеет место переполнение или нет, а выражение  $(x + y) - y$  всегда будет давать в результате  $x$ .

### Решение упражнения 2.32

Эта функция дает верный результат, кроме случая, когда  $y$  равно  $TMin$ . В этом случае мы имеем значение  $-y$ , которое также равно  $TMin$ , и поэтому вызов `tadd_ok` будет указывать на переполнение, когда  $x$  отрицательно, и на отсутствие переполнения, когда  $x$  неотрицательно. На самом деле верно и обратное: вызов `tsub_ok(x, TMin)` должен давать 0, если  $x$  отрицательно, и 1, если неотрицательно.

Из этого упражнения следует важный урок:  $TMin$  следует включать в качестве одного из случаев в любую процедуру тестирования функции.

### Решение упражнения 2.33

Это упражнение помогает понять тонкости отрицания целых в дополнительном коде с использованием слов маленькой длины.

Для  $w = 4$  имеем  $TMin_4 = -8$ . Поэтому  $-8$  является своей собственной аддитивной инверсией, тогда как к другим величинам применяется целочисленное отрицание.

x		$-x$	
Шестнадцатеричное	Десятичное	Шестнадцатеричное	Десятичное
0	0	0	0
5	5	-5	В
8	-8	-8	8
D	-3	3	3
F	-1	1	1

### Решение упражнения 2.34

Это упражнение на проверку понимания умножения целых в дополнительном коде.

Режим	x		y		x · y		Усеченное x · y	
Без знака	4	[100]	5	[101]	20	[010100]	4	[100]
В дополнительном коде	-4	[100]	-3	[101]	12	[001100]	-4	[100]
Без знака	2	[010]	7	[111]	14	[001110]	6	[110]
В дополнительном коде	2	[010]	-1	[111]	-2	[111110]	-2	[110]
Без знака	6	[110]	6	[110]	36	[100100]	4	[100]
В дополнительном коде	-2	[110]	-2	[110]	4	[000100]	-4	[100]

### Решение упражнения 2.35

Проверить эту функцию со всеми возможными значениями  $x$  и  $y$  нереально. Даже имея возможность выполнять 10 млрд проверок в секунду, для тестирования всех комбинаций, когда тип данных `int` имеет длину 32 бита, потребовалось бы более 58 лет. С другой стороны, функцию вполне можно протестировать, если использовать тип данных `short` или `char`.

Но есть более принципиальный подход, вытекающий из предложенного набора аргументов.

1. Мы знаем, что  $x \cdot y$  можно записать как  $2w$ -разрядное число в дополнительном коде. Пусть  $u$  обозначает число без знака, представленное младшими  $w$  битами, а  $v$  – число в дополнительном коде, представленное старшими  $w$  битами. Тогда, основываясь на уравнении 2.3, можно заметить, что  $x \cdot y = v2^w + u$ .

Мы также знаем, что  $u = T2U_w(p)$ , поскольку эти два числа, без знака и в дополнительном коде, образованы одной и той же комбинацией битов. Поэтому, согласно уравнению 2.6, мы можем записать  $u = p + p_{w-1}2^w$ , где  $p_{w-1}$  – это старший бит  $p$ .

Полагая  $t = v + p_{w-1}$ , имеем  $x \cdot y = p + t2^w$ .

Когда  $t = 0$ , мы имеем  $x \cdot y = p$ ; переполнения нет. При  $t \neq 0$  имеем  $x \cdot y \neq p$ ; переполнение есть.

2. По определению целочисленное деление  $p$  на ненулевое  $x$  дает частное  $q$  и остаток  $r$  такие, что  $p = x \cdot q + r$  и  $|r| < |x|$ . (Здесь мы используем абсолютные значения, потому что знаки  $x$  и  $r$  могут различаться. Например, деление  $-7$  на  $2$  дает частное  $-3$  и остаток  $-1$ .)
3. Предположим, что  $q = y$ . Тогда мы имеем  $x \cdot y = x \cdot y + r + t2^w$ . Отсюда видно, что  $r + t2^w = 0$ . Но  $|r| < |x| \leq 2^w$ , поэтому это тождество может выполняться, только если  $t = 0$ , и в этом случае  $r = 0$ .

Предположим, что  $r = t = 0$ . Тогда мы имеем  $x \cdot y = x \cdot q$ , откуда следует, что  $y = q$ .

Когда  $x$  равно 0, умножение не приводит к переполнению, и поэтому наш код дает надежный способ проверить, вызовет ли переполнение умножение целых в дополнительном коде.

### Решение упражнения 2.36

Имея 64-разрядный тип, можно выполнить умножение без переполнения, а затем проверить, меняется ли значение при преобразовании произведения в 32-разрядное целое:

```
1 /* Определяет, можно ли умножить указанные аргументы
2    без переполнения */
3 int tmult_ok(int x, int y) {
4     /* Вычислить произведение без переполнения */
5     int64_t pll = (int64_t) x*y;
6     /* Проверить изменение результата после преобразования в тип int */
7     return pll == (int) pll;
8 }
```

Обратите внимание, что приведение в правой части в строке 5 имеет решающее значение. Если записать его так:

```
int64_t pll = x * y;
```

то произведение будет вычислено как 32-разрядное значение (возможно, с переполнением) с последующим расширением знака до 64 бит.

### Решение упражнения 2.37

1. Это изменение никак не поможет. Несмотря на то что значение `asize` будет вычислено точно, вызов `malloc` приведет к его преобразованию в 32-разрядное целое без знака и возникнут те же условия переполнения.
2. Так как `malloc` принимает в аргументе 32-разрядное целое без знака, она не сможет выделить блок размером более  $2^{32}$  байт, поэтому нет смысла пытаться выделить или скопировать такой объем памяти. Вместо этого функция должна прекратить работу и вернуть `NULL`, как показано ниже (см. строку 9):

```
1 uint64_t required_size = ele_cnt * (uint64_t) ele_size;
2 size_t request_size = (size_t) required_size;
3 if (required_size != request_size)
4     /* Переполнение неизбежно. Прервать работу */
5     return NULL;
6 void *result = malloc(request_size);
7 if (result == NULL)
8     /* malloc потерпела неудачу */
9     return NULL;
```

### Решение упражнения 2.38

В главе 3 будет показано много примеров с инструкцией LEA. Данная инструкция предназначена для поддержки арифметики указателей, однако компилятор C часто использует ее для выполнения умножения на небольшие константы.

Для каждого значения  $k$  можно рассчитать две кратные величины:  $2^k$  (когда  $b$  равно 0) и  $2^k + 1$  (когда  $b$  равно  $a$ ). Следовательно, можно рассчитать кратные 1, 2, 3, 4, 5, 8 и 9.

### Решение упражнения 2.39

Выражение просто превращается в  $-(x \ll m)$ . Для демонстрации допустим, что слово имеет такой размер, чтобы  $n = w - 1$ . В форме В указано, что мы должны вычислить  $(x \ll w) - (x \ll m)$ , но сдвиг  $x$  влево на  $w$  позиций даст значение 0.

### Решение упражнения 2.40

Это упражнение требует от вас опробовать уже описанные оптимизации, а также проявить немного собственной изобретательности.

К	Сдвиг	Сложений/вычитаний	Выражение
6	2	1	$(x \ll 2) + (x \ll 1)$
31	1	1	$(x \ll 5) - x$
-6	2	1	$(x \ll 1) - (x \ll 3)$
55	2	2	$(x \ll 6) - (x \ll 3) - x$

Обратите внимание, что в четвертом случае используется модифицированная версия формы В. Мы можем рассматривать комбинацию битов [110111] как серию из 6 единиц с нулем в середине, и поэтому мы применяем правило для формы В, но затем вычитаем член, соответствующий нулевому биту.

### Решение упражнения 2.41

Если предположить, что сложение и вычитание имеют одинаковую производительность, то компилятор должен выбрать форму А, когда  $n = m$ , любую из форм, когда  $n = m + 1$ , и форму В, когда  $n > m + 1$ .

Вот краткое обоснование этого правила. Предположим сначала, что  $m > 0$ . Когда  $n = m$ , форма А требует только одного сдвига, а форма В – двух сдвигов и вычитания. Когда  $n = m + 1$ , обе формы требуют двух сдвигов и либо сложения, либо вычитания. Когда  $n > m + 1$ , форма В требует только двух сдвигов и одного вычитания, тогда как форма А требует  $n - m + 1 > 2$  сдвигов и  $n - m > 1$  сложений. В случае  $m = 0$  мы получаем на один сдвиг меньше в обеих формах, А и В, поэтому выбор между ними определяется теми же правилами.

### Решение упражнения 2.42

Единственная сложность здесь – вычислить смещение без каких-либо проверок и условных операций. Для этого можно воспользоваться уловкой, заключающейся в том, что выражение  $x \gg 31$  генерирует слово со всеми единицами, если  $x$  отрицательно, и всеми нулями в противном случае. Маскируя соответствующие биты, мы получаем желаемое значение смещения.

```
int div16(int x) {
    /* Вычислить смещение, которое будет 0 (x >= 0) или 15 (x < 0) */
    int bias = (x >> 31) & 0xF;
    return (x + bias) >> 4;
}
```

### Решение упражнения 2.43

Мы обнаружили, что многие испытывают трудности при выполнении этого упражнения, работая непосредственно с ассемблерным кодом. Задача становится более простой после показанного преобразования в функцию `optarith`.

Здесь можно заметить, что  $M$  получает значение 31;  $x * M$  вычисляется как  $(x \ll 5) - x$ .  
Значение  $N$  равно 8; значение смещения 7 добавляется, когда  $y$  отрицательное и выполняется сдвиг вправо на 3 позиции.

### Решение упражнения 2.44

Эти «головоломки на C» наглядно демонстрируют необходимость понимания программистами свойств компьютерной арифметики:

1.  $(x > 0) \parallel (x-1 < 0)$

*Ложно.* Пусть  $x$  равно  $-2\,147\,483\,648$  ( $TMin_{32}$ ). Тогда  $x-1$  будет равно  $2\,147\,483\,647$  ( $TMax_{32}$ ).

2.  $(x \& 7) != 7 \parallel (x \ll 29 < 0)$

*Истинно.* Если  $(x \& 7) != 7$  даст 0, тогда должен быть бит  $x_2$ , равный 1. При сдвиге влево на 29 получится знаковый разряд (бит).

3.  $(x * x) >= 0$

*Ложно.* Когда  $x$  равно  $65\,535$  ( $0xFFFF$ ),  $x*x$  равно  $-131\,071$  ( $0xFFFF0001$ ).

4.  $x < 0 \parallel -x <= 0$

*Истинно.* Если  $x$  имеет неотрицательное значение, тогда  $-x$  будет неположительным значением.

5.  $x > 0 \parallel -x >= 0$

*Ложно.* Пусть  $x$  равно  $-2\,147\,483\,648$  ( $TMin_{32}$ ). Тогда получается, что  $x$  и  $-x$  – обе отрицательные величины.

6.  $x+y == uy+ux$

*Истинно.* Сложение значений в дополнительном коде и без знака ведет себя одинаково на уровне битов, и в обоих случаях сложение коммутативно.

7.  $x*~y + uy*ux == -x$

*Истинно.*  $~y$  равно  $-y-1$ .  $uy*ux$  равно  $x*y$ . То есть левая сторона эквивалентна выражению  $x*-y-x+x*y$ .

### Решение упражнения 2.45

Понимание дробного двоичного представления является важным шагом на пути к пониманию представления с плавающей точкой. Данное упражнение дает возможность опробовать несколько простых примеров.

Дробное значение	Двоичное представление	Десятичное представление
$\frac{1}{4}$	0,01	0,25
$\frac{3}{4}$	0,11	0,75
$\frac{25}{16}$	1,1001	1,5625
$\frac{43}{16}$	10,1101	2,6875
$\frac{9}{8}$	1,011	1,125
$\frac{47}{8}$	101,111	5,825
$\frac{51}{16}$	11,0011	3,1875



Биты	$e$	$E$	$2^E$	$f$	$M$	$2^E \times M$	$V$	Десятичное
0 10 11	2	1	2	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$	$\frac{7}{2}$	3.5
0 11 00	–	–	–	–	–	–	$\infty$	–
0 11 01	–	–	–	–	–	–	NaN	–
0 11 10	–	–	–	–	–	–	NaN	–
0 11 11	–	–	–	–	–	–	NaN	–

### Решение упражнения 2.48

Шестнадцатеричное значение  $0x359141$  эквивалентно двоичному значению

[1101011001000101000001].

Показатель степени формируется путем добавления смещения 127 к 21, что дает 148 (двоичное [10010100]). Объединим это с полем знака 0, чтобы получить двоичное представление

[01001010010101100100010100000100].

Можно заметить, что совпадение между двумя этими представлениями соответствует младшим битам целого числа, до наиболее значимого бита, равного единице, соответствующей 21 старшему биту дробной части:

```

0 0 3 5 9 1 4 1
0000000001101011001000101000001
*****
4 A 5 6 4 5 0 4
01001010010101100100010100000100

```

### Решение упражнения 2.49

Данное упражнение помогает задуматься о том, какие числа нельзя представить точно в формате с плавающей точкой.

1. Данное число имеет двоичное представление 1, за которым следует  $n$  нулей, за которыми следует единица, что дает  $2^{n+1} + 1$ .
2. Для  $n = 23$  значение будет равно  $2^{24} + 1 = 16\,777\,217$ .

### Решение упражнения 2.50

Выполнение округления вручную помогает закрепить понимание идеи округления двоичных чисел до четного.

Исходные значения		Округленные значения	
$10.010_2$	$2\frac{1}{4}$	10,0	2
$10.011_2$	$2\frac{3}{8}$	10,1	$2\frac{1}{2}$
$10.110_2$	$2\frac{3}{4}$	11,0	3
$11.001_2$	$3\frac{1}{8}$	11,0	3



2. `x == (int)(float) x`

*Нет.* Например, когда `x` равно `TMax`.

3. `d == (double)(float) d`

*Нет.* Например, когда `d` равно `1e40`, мы получим  $+\infty$  с правой стороны.

4. `f == (float)(double) f`

*Да,* потому что тип `double` имеет более высокую точность и диапазон представимых значений, чем тип `float`.

5. `f == -(-f)`

*Да,* потому что отрицание чисел с плавающей точкой производится простым изменением знакового бита.

6. `1.0/2 == 1/2.0`

*Да,* потому что оба операнда, числитель и знаменатель, преобразуются в представление с плавающей точкой перед делением.

7. `d*d >= 0.0`

*Да,* хотя при определенных значениях `d` может произойти переполнение до  $+\infty$ .

8. `(f+d)-f == d`

*Нет.* Например, когда `f` равно `1.0e20` и `d` равно `1.0`, выражение `f+d` будет округлено до `1.0e20`, из-за чего выражение слева даст значение `0.0`, тогда как выражение справа даст `1.0`.



## Представление программ на машинном уровне

- 3.1. Историческая перспектива.
- 3.2. Кодирование программ.
- 3.3. Форматы данных.
- 3.4. Доступ к информации.
- 3.5. Арифметические и логические операции.
- 3.6. Управление.
- 3.7. Процедуры.
- 3.8. Распределение памяти под массивы и доступ к массивам.
- 3.9. Разнородные структуры данных.
- 3.10. Комбинирование управления с данными в машинном коде.
- 3.11. Программный код операций с плавающей точкой.
- 3.12. Итоги.

Библиографические заметки.

Домашние задания.

Решения упражнений.

**К**омпьютеры выполняют *машинный код* – последовательность байтов, представляющих низкоуровневые операции, которые манипулируют данными, управляют памятью, читают и записывают данные на запоминающие устройства и пересылают данные по сети. Компилятор генерирует машинный код, выполняя серию этапов, опираясь на правила языка программирования, набор инструкций целевой машины и соглашений, принятых в операционной системе. Компилятор GCC C генерирует на выходе *ассемблерный код* – текстовое представление машинного кода, – содержащий последовательность программных инструкций. Затем GCC вызывает *ассемблер* и *компоновщика*, чтобы преобразовать ассемблерный код в машинный. В этой главе мы подробно исследуем машинный код и его удобочитаемое представление в виде ассемблерного кода.

При программировании на языках высокого уровня, таких как C и тем более Java, мы защищены от деталей реализации наших программ на машинном уровне. С другой стороны, разрабатывая программу на языке ассемблера (как мы делали это на заре развития вычислительной техники), программист должен точно указать, какие низкоуровневые инструкции должна использовать программа для вычислений. В большинстве случаев намного эффективней и надежней работать с высокоуровневыми абстракциями, которые предоставляют языки высокого уровня. Контроль соответствия типов,

предлагаемый компилятором, позволяет обнаружить многие программные ошибки и гарантирует, что все ссылки и операции с данными будут выполнены корректно. Программный код, который генерируют современные оптимизирующие компиляторы, по меньшей мере так же эффективен, как и код, написанный высококвалифицированным программистом вручную на языке ассемблера. Более того, программу, написанную на высокоуровневом языке, можно скомпилировать и выполнить на машинах различных типов, в то время как ассемблерный код в значительной степени машинно ориентированный.

Так зачем же тратить время на изучение машинного кода? Даже при наличии оптимизирующих компиляторов способность читать и понимать программы на языке ассемблера является существенным преимуществом и признаком высокой квалификации в среде серьезных программистов. Если вызывать компилятор с соответствующими параметрами командной строки, то он сгенерирует файл с ассемблерным кодом. Читая программу на языке ассемблера, можно понять, какими возможностями обладает компилятор, и выявить неэффективные фрагменты программы. Как вы узнаете в главе 5, программисты, стремящиеся обеспечить максимальную производительность наиболее важных разделов программ, предпринимают попытки использовать различные варианты исходного кода, каждый раз компилируя и исследуя полученный ассемблерный код, чтобы оценить, насколько эффективной будет выполняемая программа. Более того, имеют место случаи, когда уровни абстракции, обеспечиваемые языками высокого уровня, скрывают информацию о поведении программы во время выполнения, которое мы также должны оценить. Например, когда мы пишем многопоточные программы, используя пакет поддержки потоков выполнения, как показано в главе 12, важно знать, какие данные используются разными потоками совместно, а какие индивидуально, как и в какие моменты происходят обращения к совместно используемым данным. Эта информация видна на уровне машинного кода. Еще один пример: наши программы могут быть атакованы множеством способов и позволять вредоносным программам проникать в систему за счет использования нюансов хранения управляющей информации во время выполнения. Многие атаки основаны на эффекте переполнения буфера и затирания управляющей информации и за счет этого получения контроля над системой. Чтобы знать, как возникают эти уязвимости и как от них защититься, необходимо знать, как выглядит программа в машинном коде. Если раньше программисты изучали машинный код, чтобы уметь писать программы на языке ассемблера, то в настоящее время знание машинного кода необходимо им, чтобы читать и понимать код, сгенерированный компиляторами.

В этой главе мы изучим особенности конкретного языка ассемблера и увидим, как программы на языке C компилируются в эту форму машинного кода. Чтение ассемблерного кода, сгенерированного компилятором, требует других навыков, отличных от навыков, необходимых для разработки программ на языке ассемблера. Мы должны уметь замечать преобразования, выполняемые типичными компиляторами, превращающими конструкции на языке C в машинный код. Например, вычисления, выраженные в программе на языке C, оптимизирующие компиляторы могут переупорядочивать, удалять ненужные вычисления, заменять медленные операции более быстрыми и даже заменять рекурсивные вычисления на итеративные. Понимание взаимосвязей между исходным и сгенерированным ассемблерным кодом часто может оказаться трудной задачей, во многом подобной составлению мозаики, когда настоящая картинка слегка отличается от той, что изображена на коробке. Это как бы некоторая форма *обратного проектирования* (reverse engineering) – попытка понять процесс, в результате которого система была построена путем изучения системы и всех этапов ее построения, двигаясь в обратном порядке. В рассматриваемом случае система является программой на языке ассемблера, сгенерированной машиной, а не созданной человеком. Это упрощает задачу анализа, потому что сгенерированный код во многом подчиняется четко определен-

ным правилам, и есть возможность проводить эксперименты, заставляя компилятор генерировать разный программный код. В данном обзоре мы представим множество примеров и упражнений, иллюстрирующих различные аспекты языка ассемблера и компиляторов. Это именно тот случай, когда понимание деталей является предпосылкой для понимания более глубоких фундаментальных понятий. Те, кто говорят: «Я понимаю общие принципы, и знать подробности совсем необязательно», – заблуждаются. Обязательно найдите время на изучение примеров, решение упражнений и сравнение ваших решений с теми, которые будут предложены нами.

Наш обзор основан на машинном языке для архитектуры x86-64, т. е. для большинства процессоров, используемых в современных портативных и настольных компьютерах, а также в суперкомпьютерах, установленных в крупных центрах обработки данных. Этот язык развивался на протяжении многих лет, начиная с первого 16-разрядного процессора корпорации Intel, созданного в 1978 году, до 32-разрядного, а затем и до 64-разрядного. В ходе этой эволюции добавлялись новые возможности, позволяющие эффективнее использовать доступную полупроводниковую технологию и удовлетворять потребности рынка. Основная масса разработок была начата в компании Intel, но ее конкурент – компания Advanced Micro Devices (AMD) – тоже внесла важный вклад. В результате получилась довольно своеобразная архитектура с некоторыми возможностями, которые имеют смысл только с исторической точки зрения. Она также полна особенностями, обеспечивающими обратную совместимость, которые не используются современными компиляторами и операционными системами. Мы же сосредоточимся на подмножестве особенностей, используемых GCC и Linux. Это поможет нам избежать встречи с немалыми сложностями и многими загадочными особенностями архитектуры x86-64.

Свой технический обзор мы начнем с того, что покажем связь между исходным кодом на языке C, ассемблерным и объектным кодами. Затем перейдем к исследованию особенностей архитектуры x86-64, начав с представления данных, манипулирования ими и реализации управления. Мы покажем, как реализованы управляющие конструкции языка C, такие как операторы `if`, `while` и `switch`. Затем расскажем, как реализуются процедуры, в частности как используется стек для передачи данных и управления между процедурами, а также как хранятся локальные переменные. Далее мы рассмотрим такие структуры данных, как массивы, структуры и объединения, и их реализацию на машинном уровне. Получив эти базовые знания в области программирования на машинном уровне, мы сможем приступить к исследованию проблем, связанных с выходом за пределы адресного пространства и с уязвимостями к атакам злоумышленников, использующих метод переполнения буфера. Мы закончим эту часть технического обзора несколькими рекомендациями по использованию отладчика GDB с целью исследования поведения программ машинного уровня. В завершение главы поговорим о машинном представлении программного кода, включающего данные и операции с плавающей запятой.

Не так давно компьютерная индустрия перешла с 32-разрядных на 64-разрядные архитектуры. 32-разрядная архитектура может использовать только около 4 Гбайт ( $2^{32}$  байт) оперативной памяти. При резком падении цен на память и увеличении наших потребностей и объемов обрабатываемых данных стало экономически целесообразным и технически желательным расширить это ограничение. Современные 64-разрядные машины могут использовать до 256 Тбайт ( $2^{48}$  байт) оперативной памяти, и этот предел легко можно увеличить до 16 Эбайт ( $2^{64}$  байт). Конечно, сейчас трудно представить себе машину с таким объемом оперативной памяти, но в свое время и 4 Гбайт казались просто гигантским объемом, когда 32-разрядные машины только стали появляться в 1970-х и 1980-х годах.

Основное внимание в нашем обзоре мы будем уделять программам на машинном языке, которые генерируются при компиляции программ на C и других подобных язы-

ках программирования, ориентированных на современные операционные системы. Как следствие мы не будем описывать многие особенности x86-64, обусловленные unsupported поддержкой программ, написанных для первых микропроцессоров, когда большая часть кода писалась вручную и программистам приходилось бороться с ограниченным диапазоном адресов 16-разрядных машин.

#### **ASM:IA32** Программирование для архитектуры IA32

Архитектура IA32, 32-разрядная предшественница x86-64, была представлена компанией Intel в 1985 году. Она оставалась наиболее предпочтительной машинной архитектурой в течение нескольких десятилетий. Большинство микропроцессоров x86, продаваемых в настоящее время, и большинство операционных систем, установленных на машинах с ними, предназначены для работы с архитектурой x86-64. Однако они также могут выполнять программы IA32 в режиме обратной совместимости. В результате многие прикладные программы по-прежнему основаны на архитектуре IA32. Кроме того, многие существующие системы не могут выполнять код x86-64 из-за ограничений оборудования или системного программного обеспечения. IA32 продолжает оставаться важным машинным языком. Впоследствии вы обнаружите, что, имея опыт работы с x86-64, вы с легкостью сможете освоить машинный язык IA32.

### 3.1. Историческая перспектива

Линейка процессоров Intel, известная под общим названием x86, прошла долгий путь эволюционного развития. Все началось с выпуска одного из первых 16-разрядных микропроцессоров, при разработке которого проектировщикам пришлось пойти на многие компромиссы, обусловленные ограниченными возможностями технологии интегральных схем того времени. С тех пор он претерпел существенные изменения, вбирая в себя технические достижения, обеспечившие высокую производительность и поддержку более совершенных операционных систем.

В следующем далее списке перечислены некоторые модели процессоров Intel и их основные свойства в порядке возрастания производительности. Мы добавили в список число транзисторов в этих процессорах как показатель того, насколько увеличивалась их сложность. В этом списке символ «К» означает 1000 ( $10^3$ ), «М» – 1 000 000 ( $10^6$ ) и «Г» – 1 000 000 000 ( $10^9$ ).

8086 (1978, 29 К транзисторов), один из первых 16-разрядных однокристальных микропроцессоров. Версия 8088 процессора 8086 с 8-разрядной внешней шиной представляет собой центральное звено оригинальных персональных компьютеров корпорации IBM. Корпорация IBM заключила договор с небольшой по тем временам компанией Microsoft на разработку операционной системы MS-DOS. Первые модели выпускались с объемом оперативной памяти 32 768 байт и с двумя накопителями на гибких магнитных дисках (без жестких дисков). В плане архитектуры эти машины имели ограниченное адресное пространство 655 360 байт (длина адреса тогда составляла 20 разрядов, что позволяло адресовать до 1 048 576 байт), при этом операционная система резервировала 393 216 байт для собственных нужд. В 1980 году Intel представила арифметический сопроцессор 8087 (45 К транзисторов) для выполнения операций с плавающей точкой, способный интегрироваться с процессором 8086 или 8088. Сопроцессор 8087 определил модель арифметики с плавающей точкой для линейки x86, которую часто называют «x87»;

80286 (1982, 134 К транзисторов). Реализовано несколько режимов адресации (ныне устаревших). Этот процессор послужил базой для персонального компьютера IBM PC-AT, исходной платформы для MS Windows;

i386 (1985, 275 К транзисторов). Архитектура расширена до 32 разрядов. Добавлена модель плоской адресации, используемая операционной системой Linux и новыми версиями семейства операционных систем Windows. Это была первая машина, которая была способна поддерживать операционную систему Unix;

i486 (1989, 1.2 М транзисторов). Увеличена производительность, в кристалл процессора интегрирован блок вычислений с плавающей точкой, однако система команд не претерпела существенных изменений;

Pentium (1993, 3.1 М транзисторов). Увеличена производительность, однако система команд была расширена незначительно;

Pentium Pro (1995, 5.5 М транзисторов). Совершенно новая конструкция процессора, известная в профессиональных кругах как микроархитектура P6. В систему команд добавлен класс команд «условного перемещения»;

Pentium/MMX (1997, 4.5 М транзисторов). Добавлен новый класс инструкций, позволяющий манипулировать целочисленными векторами. Каждый элемент данных мог иметь длину 1, 2 или 4 байта. Каждый вектор состоял из 64 бит;

Pentium II (1997, 7 М транзисторов). Продолжение развития микроархитектуры P6;

Pentium III (1999, 8.2 М транзисторов). Введен еще один класс команд – SSE (Streaming SIMD Extensions – потоковые расширения SIMD) – для выполнения операций с целочисленными векторами или данными с плавающей точкой. Каждый элемент данных мог иметь длину 1, 2 или 4 байта, элементы данных упаковываются в 128-разрядный вектор. Более поздние версии этого процессора имели до 24 М транзисторов из-за внедрения кеш-памяти второго уровня;

Pentium 4 (2000, 42 М транзисторов). Расширены наборы инструкций SSE и SSE2, добавлена поддержка новых типов данных (включая числа с плавающей точкой двойной точности), а также 144 новые инструкции. Благодаря этим усовершенствованиям компиляторы получили возможность использовать инструкции SSE вместо инструкций x87 при компиляции операций с плавающей точкой;

Pentium 4E (2004, 125 М транзисторов). Добавлена гиперпоточность – метод одновременного выполнения двух программ на одном процессоре, а также EM64T – реализация компании Intel 64-разрядного расширения для IA32, разработанного в Advanced Micro Devices (AMD), которое мы называем x86-64;

Core 2 (2006, 291 М транзисторов). Возврат к микроархитектуре, аналогичной P6. Первый *многоядерный* микропроцессор Intel, в котором несколько процессоров размещено на одном кристалле. Не поддерживал гиперпоточность;

Core i7, Nehalem (2008, 781 М транзисторов). Добавлена поддержка гиперпоточности и многоядерности. Первоначальная версия поддерживала одновременное выполнение двух программ на каждом ядре и до четырех ядер на каждом кристалле;

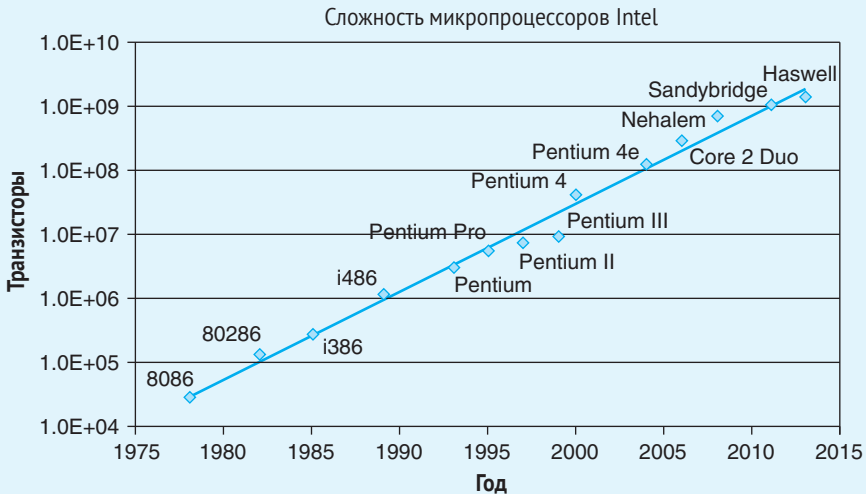
Core i7, Sandy Bridge (2011, 1.17 G транзисторов). Добавлено расширение AVX набора инструкций SSE для поддержки данных, упакованных в 256-битные векторы;

Core i7, Haswell (2013, 1.4 G транзисторов). Набор инструкций AVX расширен до AVX2, добавлено множество других инструкций и форматов инструкций.

Каждый очередной процессор разрабатывался с учетом требования к обратной совместимости – способности выполнять программы, скомпилированные для более ранних версий процессоров. Как мы увидим далее, в системе команд существует мно-

жество странных артефактов, обусловленных этим эволюционным наследием. Intel несколько раз меняла название своей линейки процессоров, в том числе *IA32*, обозначающее «Intel Architecture 32-bit», и *Intel64*, обозначающее 64-разрядное расширение архитектуры IA32, которое мы ныне называем x86-64. Эта серия процессоров в разговорной речи часто называется «x86», что соответствует соглашению о присвоении имен, действовавшему до появления модели i486.

### Закон Мура



Если построить график возрастания числа транзисторов в разных процессорах Intel, в зависимости от года выпуска, и воспользоваться логарифмическим масштабом по вертикальной оси, то можно увидеть феноменальный рост. Проведя прямую линию по точкам, соответствующим данным, мы увидим, что число транзисторов увеличивается за год примерно на 37 %, то есть число транзисторов удваивается каждые 26 месяцев. Этот рост сохраняется на протяжении десятилетий истории существования микропроцессоров x86.

В 1965 году Гордон Мур (Gordon Moore), основатель компании Intel Corporation, предсказал, опираясь на существовавшие тогда технологии изготовления микросхем (которые в то время позволяли разместить на одном кристалле примерно 64 транзистора), что число транзисторов будет удваиваться каждый год в течение следующих 10 лет. Это предсказание известно как закон Мура. Как выяснилось позже, предсказание оказалось несколько оптимистичным и в то же время недальновидным. На протяжении своей 50-летней истории полупроводниковая промышленность оказалась способной удваивать число транзисторов каждые 18 месяцев.

Подобные экспоненциальные темпы роста имеют место и в других компьютерных технологиях, включая емкость дисковой и полупроводниковой памяти. Такие поразительные темпы роста стали основной движущей силой компьютерной революции.

В течение многих лет несколько компаний изготавливали процессоры, совместимые с процессорами компании Intel и способные исполнять те же программы в машинном коде. Самой заметной среди них была компания Advanced Micro Devices (AMD). На протяжении многих лет стратегия компании AMD заключалась в том, чтобы идти след в след за компанией Intel и производить более дешевые процессоры, хотя и несколько



меньшей производительности. Но начиная с 2002 года она начала более агрессивно конкурировать с Intel, когда первой выпустила серийный микропроцессор, преодолевший барьер тактовой частоты в 1 ГГц, и представила архитектуру x86-64 – широко распространенное 64-разрядное расширение для архитектуры IA32 компании Intel. Хотя в своем обзоре мы будем говорить о процессорах Intel, тем не менее все сказанное нами в равной степени применимо к совместимым процессорам, производимым конкурентами Intel.

Основные сложности архитектуры x86 не касаются тех, кто исследует программы для операционной системы Linux, генерируемые компилятором GCC. Модель памяти, использованная в оригинальном процессоре 8086 и его расширении 80286, устарела с появлением i386. Модель вычислений с плавающей точкой x87 устарела с появлением инструкций SSE2. И несмотря на наличие артефактов исторической эволюции x86 в программах x86-64, многие из самых загадочных особенностей x86 никак не проявляются.

## 3.2. Программный код

Предположим, что мы написали программу в языке C и сохранили ее в двух файлах: `p1.c` и `p2.c`. В Unix этот программный код можно скомпилировать командой

```
linux> gcc -Og -o p p1.c p2.c
```

Команда `gcc` – это компилятор GCC языка C. Поскольку этот компилятор используется в Linux по умолчанию, мы можем его вызвать просто как `cc`. Параметр `-Og`<sup>1</sup> требует от компилятора применить оптимизацию, порождающую машинный код, близко соответствующий структуре оригинального кода на C. При использовании более высоких уровней оптимизации генерируется настолько сильно трансформированный код, что уловить его связь с исходной программой становится очень трудно. По этой причине в процессе знакомства с инструментами мы будем использовать уровень оптимизации `-Og`, а затем покажем, что происходит с кодом при увеличении уровня оптимизации. На практике более высокие уровни оптимизации (например, `-O1` или `-O2`) считаются более удачным выбором, с точки зрения производительности программ.

Команда GCC фактически вызывает целый комплекс программ в определенной последовательности, которые превращают исходный код в выполняемую программу. Сначала *препроцессор* языка C расширяет исходный, подключая все заголовочные файлы, указанные в директивах `#include`, и подставляет все макросы, объявленные в директивах `#define`. Затем *компилятор* генерирует версии ассемблерного кода с именами `p1.s` и `p2.s` для двух исходных файлов. Далее *ассемблер* преобразует ассемблерный код в двоичный *объектный код* в файлах `p1.o` и `p2.o`. И наконец, *редактор связей* (или *компоновщик*) объединяет эти два объектных файла с реализациями стандартных функций из библиотеки (таких как `printf`) и генерирует выполняемый код, сохраняя его в файле `p` (как определено параметром командной строки `-o p`). Выполняемый код – это вторая форма машинного кода, которую мы рассмотрим и которая выполняется процессором. Связь между этими различными формами машинного кода и процессом компоновки более подробно описана в главе 7.

### 3.2.1. Машинный код

Как рассказывалось в разделе 1.9.3, компьютерные системы используют несколько форм абстракции, скрывая детали реализации за счет использования более простой аб-

<sup>1</sup> Этот уровень оптимизации был введен в GCC версии 4.8. Более ранние версии GCC, а также компиляторы, отличные от GNU, не распознают этот параметр. Для них лучшим выбором, когда требуется сгенерировать код, следующий исходной структуре программы, является использование уровня оптимизации 1 (указывается с помощью параметра командной строки `-O1`).

страктной модели. Две из них особенно важны для программирования на машинном уровне. Во-первых, формат и поведение машинной программы зависят от *архитектуры набора команд* (Instruction Set Architecture, ISA), определяющей состояние процессора, формат инструкций и влияние каждой из этих инструкций на состояние. Большинство архитектур, включая x86-64, описывают поведение программы так, будто каждая инструкция выполняется последовательно, причем выполнение каждой предыдущей инструкции завершается до начала выполнения следующей. Устройство процессора намного сложнее; он может выполнять множество инструкций одновременно, но при этом гарантирует, что общее поведение соответствует последовательному выполнению операций ISA. Во-вторых, адреса памяти, используемые машинной программой, являются *виртуальными*, благодаря чему обеспечивается представление памяти как очень большого массива байтов. Фактическая реализация памяти включает комбинацию нескольких аппаратных запоминающих устройств и программного обеспечения операционной системы, как описано в главе 9.

Компилятор выполняет почти всю работу в общей последовательности действий, преобразуя программы, выраженные в форме относительно абстрактной модели выполнения языка C, в элементарные инструкции, которые выполняет процессор. Представление в ассемблерном коде очень близко к машинному коду. Его главная особенность в том, что он имеет более удобочитаемый текстовый формат, чем двоичный формат выполняемого кода. Умение понимать ассемблерный код и как он соотносится с исходным кодом является ключом к пониманию особенностей выполнения программ компьютерами.

Машинный код для x86-64 сильно отличается от исходного кода на C. Он позволяет видеть элементы состояния процессора, которые обычно скрыты от программиста на C:

- *счетчик инструкций* (в архитектуре x86-64 его обозначает мнемоника `%rip`) указывает адрес следующей команды в памяти, подлежащей исполнению;
- *блок целочисленных регистров* содержит 16 именованных ячеек для хранения 64-разрядных значений. В этих регистрах могут храниться адреса (соответствующие указателям языка C) или целые числа. Некоторые регистры используются для хранения наиболее важных элементов состояния программы, тогда как другие – для хранения временных данных, таких как аргументы и локальные переменные процедур, а также значения, возвращаемые функциями;
- *регистры кодов условий* (флагов) хранят информацию о последних выполненных арифметических и логических инструкциях. Эти регистры используются для реализации управления данными и потоком выполнения, например операторов `if` или `while`;
- *набор векторных регистров*, каждый из которых может хранить или несколько целых чисел, или чисел с плавающей точкой.

Если язык C предоставляет модель, которая позволяет объявлять и размещать в памяти объекты разных типов, то машинный код рассматривает память как большой массив байтов. Агрегатные типы данных в языке C, такие как массивы и структуры, представлены в машинном коде как непрерывные последовательности байтов. Даже в отношении скалярных типов данных машинный код не отличает целые со знаком от целых без знака, указатели разных типов и даже указатели от целых чисел.

Память программы содержит выполняемый машинный код, некоторую информацию, необходимую операционной системе, стек программы, используемый для управления вызовами процедур и возврата из них, а также блоки памяти, которые распределяет пользователь (например, используя библиотечную функцию `malloc`). Как отмечалось выше, адресация памяти производится с использованием виртуальных адресов. В любой конкретный момент времени допустимыми считаются только ограниченные под-



диапазоны виртуальных адресов. Например, виртуальные адреса в архитектуре x86-64 представлены 64-битными словами. В текущих реализациях старшие 16 бит должны быть установлены в ноль, поэтому можно адресовать только байты в диапазоне адресов от 0 до  $2^{48}$  или 256 Тбайт. Типичные программы используют от нескольких мегабайтов до нескольких гигабайтов памяти. Этим виртуальным адресным пространством управляет операционная система, которая преобразует виртуальные адреса в физические адреса в памяти процессора. 32-разрядные адреса могут потенциально охватывать диапазон значений.

#### Постоянно меняющиеся формы сгенерированного кода

В нашем обзоре мы покажем код, сгенерированный конкретной версией GCC и с определенными параметрами командной строки. Если вы будете компилировать код на своем компьютере, то, скорее всего, вы будете использовать другой компилятор или другую версию GCC и, следовательно, получать другой машинный код. Сообщество разработчиков программного обеспечения с открытым исходным кодом, поддерживающее GCC, постоянно меняет генератор кода, пытаясь добиться создания более эффективного кода, в соответствии с изменениями в руководствах, предоставленных производителями микропроцессоров.

Главная цель, которую мы преследуем, предоставляя в этой книге примеры кода, – научить вас анализировать ассемблерный код и сопоставлять его с конструкциями в языках программирования высокого уровня. Вам придется адаптировать эти методы к стилю кода, созданного вашим конкретным компилятором.

Одна машинная инструкция может исполнить только очень простую операцию. Например, складывать два числа, находящихся в регистрах, передавать данные между памятью и регистром или совершать условный переход по адресу новой инструкции. Задача компилятора заключается в том, чтобы сгенерировать последовательность инструкций, реализующих различные программные конструкции, такие как вычисление значений арифметических выражений, циклы или вызовы процедур и возврат из них.

### 3.2.2. Примеры кода

Представим себе, что мы пишем программный код на языке C и помещаем в файл `mstore.c` определение следующей функции:

```
long mult2(long, long);

void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

Чтобы увидеть соответствующий ассемблерный код, сгенерированный компилятором C, добавим в командную строку параметр `-S`:

```
linux> gcc -Og -S mstore.c
```

По этой команде GCC запустит компилятор, сгенерирует ассемблерный код, сохранит его в файле `mstore.s` и на этом остановится. (В обычной ситуации он далее вызывал бы ассемблер, чтобы тот сгенерировал файл с объектным кодом.)

Файл с ассемблерным кодом содержит различные объявления, включая следующие строки:

```
multstore:
```

```

pushq %rbx
movq %rdx, %rbx
call mult2
movq %rax, (%rbx)
popq %rbx
ret

```

Каждая строка с отступом в этом коде соответствует одной машинной инструкции. Например, инструкция `pushq` кладет содержимое регистра `%rbx` на стек программы. Вся информация об именах локальных переменных или типах данных на этом уровне отсутствует.

#### Как увидеть представление программы в байтах?

Чтобы увидеть двоичный объектный код (например, в файле `mstore`), можно воспользоваться *дисассемблером* (описывается далее), который покажет, что программный код процедуры содержит 14 байт. Затем можно запустить отладчик GNU GDB и открыть с его помощью файл `mstore.o` командой:

```
(gdb) x/14xb mstore
```

потребовав показать (параметр `x`) 14 байт (параметр `b`) в шестнадцатеричном формате (еще один параметр `x`). Далее, в разделе 3.10.2, вы узнаете, что отладчик GDB имеет много полезных возможностей, помогающих анализировать программы в машинном коде.

Если при компиляции использовать параметр командной строки `-c`, то GCC не только скомпилирует программу, но и вызовет ассемблер, чтобы получить объектный код:

```
linux> gcc -Og -c mstore.c
```

В результате будет сгенерирован файл с объектным кодом `mstore.o` в двоичном формате, из-за чего его нельзя просматривать непосредственно. В файле `mstore.o`, содержащем 1368 байт, находится последовательность из 14 байт, имеющая следующее шестнадцатеричное представление:

```
53 48 89 d3 e8 00 00 00 00 48 89 03 5b c3
```

Это объектный код, соответствующий ассемблерным инструкциям, приведенным выше. Основной урок, который вы должны вынести из всего вышесказанного, заключается в том, что программа, которую фактически исполняет машина, является всего лишь последовательностью байтов, представляющей серию инструкций. Машина практически не имеет никакой информации об исходном коде, на основе которого были получены эти инструкции.

Для исследования содержимого файлов с машинным кодом особенно ценным может оказаться класс программ, известных как *дисассемблеры*. Эти программы генерируют из машинного кода программный код на языке ассемблера. В системах Linux эту роль может играть программа `OBJDUMP` (от «object dump» – «вывод объектного кода»), если передать ей параметр командной строки `-d`:

```
linux> objdump -d mstore.o
```

Результат (мы только добавили нумерацию строк слева и комментарии справа) показан ниже:

Результат дисассемблирования функции `multstore` в двоичном файле `mstore.o`

```
1 0000000000000000 <multstore>:
```

Смещение    Байты

Эквивалент на языке ассемблера

2	0:	53	push %rbx
3	1:	48 89 d3	mov %rdx,%rbx
4	4:	e8 00 00 00 00	callq 9 <multstore+0x9>
5	9:	48 89 03	mov %rax, (%rbx)
6	c:	5b	pop %rbx
7	d:	c3	retq

Слева можно видеть 14 шестнадцатеричных значений байтов из последовательности, представленной выше, разбитых на группы, содержащие от 1 до 5 байт. Каждая из этих групп представляет одну инструкцию, эквивалент которой на языке ассемблера показан справа.

Следует отметить несколько особенностей машинного кода и дизассемблированного представления:

- инструкции x86-64 могут иметь длину от 1 до 15 байт. Кодирование инструкций продумано так, что часто используемые инструкции и инструкции с малым числом операндов имеют меньшую длину, чем редко используемые инструкции или инструкции с большим числом операндов;
- формат инструкций спроектирован так, что начиная с определенной позиции байты однозначно декодируются в машинные инструкции. Например, только инструкция `pushq %rbx` начинается с байта со значением 53;
- дизассемблер генерирует ассемблерный код, основываясь исключительно на последовательностях байтов в объектном файле. Он не требует доступа к исходному коду или к версии программы на языке ассемблера;
- дизассемблер использует соглашения об именовании команд, несколько отличные от тех, которые использует формат GCC, когда генерирует ассемблерный код. В рассматриваемом примере он опускает суффикс `q` во многих инструкциях. Эти суффиксы обозначают размеры и в большинстве случаев могут не использоваться. И наоборот, дизассемблер добавляет суффикс `q` в инструкции `call` и `ret`. И снова эти суффиксы можно просто опустить.

Чтобы получить фактический выполняемый код, файлы с объектным кодом необходимо обработать компоновщиком, при этом один из этих файлов обязательно должен содержать функцию `main`. Предположим, что в файле `main.c` определена следующая функция:

```
#include <stdio.h>

void multstore(long, long, long *);

int main() {
    long d;
    multstore(2, 3, &d);
    printf("2 * 3 --> %ld\n", d);
    return 0;
}

long mult2(long a, long b) {
    long s = a * b;
    return s;
}
```

В таком случае мы можем сгенерировать выполняемый файл `prog` программы следующим образом:

```
linux> gcc -Og -o prog main.c multstore.c
```

Размер файла `prog` вырос до 8655 байт, потому что он содержит не только машинный код двух процедур, но и информацию, используемую для запуска и завершения программы, а также для взаимодействия с операционной системой.

Файл `prog` можно дизассемблировать:

```
linux> objdump -d prog
```

Дизассемблер извлечет различные последовательности кода, в том числе следующую:

Результат дизассемблирования функции `multstore` в двоичном файле `prog`

```

1  0000000000400540 <multstore>:
2  400540: 53                push %rbx
3  400541: 48 89 d3         mov %rdx,%rbx
4  400544: e8 42 00 00 00   callq 40058b <mult2>
5  400549: 48 89 03         mov %rax, (%rbx)
6  40054c: 5b              pop %rbx
7  40054d: c3              retq
8  40054e: 90              nop
9  40054f: 90              nop
```

Этот код практически идентичен полученному при дизассемблировании файла `mstore.o`. Главное отличие – другие адреса в колонке слева. Компоновщик изменил местоположение этого кода, в результате чего адреса инструкций изменились. Второе отличие – компоновщик заполнил адрес вызываемой функции `mult2` в инструкции `callq` (строка 4). Одна из задач компоновщика – сопоставить вызовы функций с их местоположениями. И последнее отличие – появились две дополнительные строки кода (строки 8–9). Эти инструкции не влияют на выполнение программы, так как они находятся после инструкции возврата (строка 7). Они были добавлены, чтобы выровнять код функции по границе 16 байт, дабы оптимальнее (с точки зрения производительности памяти) разместить следующий блок кода.

### 3.2.3. Замечание по форматированию

Человеку трудно читать ассемблерный код, сгенерированный компилятором GCC. С одной стороны, он содержит информацию, которая для нас не представляет интереса. С другой стороны, в нем нет никакой информации с описанием программы или которая помогла бы понять, как она работает. Например, предположим, что мы выполнили следующую команду:

```
linux> gcc -Og -S mstore.c
```

чтобы сгенерировать файл `mstore.s`. Вот полное содержимое этого файла:

```

.file "010-mstore.c"
.text
.globl multstore
.type multstore, @function
multstore:
    pushq %rbx
    movq %rdx, %rbx
    call mult2
    movq %rax, (%rbx)
    popq %rbx
    ret
.size multstore, .-multstore
.ident "GCC: (Ubuntu 4.8.1-2ubuntu1-12.04) 4.8.1"
.section .note.GNU-stack,"",@progbits
```

Все строки, начинающиеся с точки, – это директивы, управляющие работой ассемблера и компоновщика. В большинстве случаев мы будем их игнорировать. С другой стороны, здесь нет разъяснительных комментариев, описывающих, что представляют собой те или иные инструкции и как они связаны с исходным кодом.

### Форматы ассемблерного кода ATT и Intel

В нашем обсуждении мы демонстрируем ассемблерный код, используя формат ATT (названный в честь AT&T – компании, которая много лет управляла Bell Laboratories), который по умолчанию используют GCC, OBJDUMP и другие инструменты. С другой стороны, инструменты, производимые Microsoft и иными компаниями, а также Intel в своей документации, оформляют ассемблерный код в формате intel. Эти два формата различаются по многим параметрам. Например, мы можем сгенерировать ассемблерный код функции multstore в формате Intel с помощью GCC, выполнив такую команду:

```
linux> gcc -Og -S -masm=intel mstore.c
```

и получить следующий результат:

```
multstore:
    push    rbx
    mov     rbx, rdx
    call    mult2
    mov     QWORD PTR [rbx], rax
    pop     rbx
    ret
```

Вот основные отличия между форматами ATT и Intel:

- в формате Intel отсутствуют суффиксы, обозначающие размер: вместо pushq и movq мы видим инструкции push и mov;
- в формате Intel не используется символ % перед именами регистров, например вместо %rbx используется имя rbx;
- в формате Intel иначе описываются ссылки на местоположения в памяти, например QWORD PTR [rbx], а не (%rbx);
- в формате Intel, в инструкциях с несколькими операндами, операнды перечисляются в обратном порядке. Это может сбивать с толку при переключении между двумя форматами.

Мы не будем использовать формат Intel в нашем обсуждении, но он обязательно встретится вам в документации Intel и Microsoft.

Чтобы сделать ассемблерный код более понятным, представим его в форме без директив, с номерами строк и дополнительными пояснениями. Вот как выглядит версия нашего примера с комментариями:

```
void multstore(long x, long y, long *dest)
x в %rdi, y в %rsi, dest в %rdx
1  multstore:
2      pushq    %rbx                Сохранить %rbx на стеке
3      movq     %rdx, %rbx          Скопировать dest в %rbx
4      call     mult2              Вызвать mult2(x, y)
5      movq     %rax, (%rbx)        Сохранить результат по адресу *dest
6      popq     %rbx               Восстановить %rbx
7      ret                    Вернуть управление
```

Обычно мы будем показывать только строки кода, имеющие отношение к обсуждению. Все строки пронумерованы, чтобы на них было проще ссылаться в тексте, а справа добавлены краткие пояснения действий, выполняемых инструкциями, и как эти инст-

рукции связаны с исходным кодом на С. Именно так программисты обычно оформляют код программ на языке ассемблера, который они пишут.

Мы также будем приводить дополнительные сведения в приложениях в интернете для поклонников машинного языка. В одном из приложений в интернете описывается машинный код архитектуры IA32. Опыт работы с x86-64 упрощает изучение IA32. В другой врезке мы кратко опишем способы включения ассемблерного кода в программы на языке С. В некоторых ситуациях программистам приходится прибегать к ассемблерному коду, чтобы получить доступ к низкоуровневым возможностям машины. Один из способов получить такой доступ – написать группу функций на ассемблере и объединить их с функциями на С на этапе компоновки. Второй способ – использовать поддержку GCC для встраивания ассемблерного кода непосредственно в программы на С.

#### **Приложение в интернете ASM:EASM.** Комбинирование ассемблерного кода с программами на языке С

Компилятор С прекрасно справляется с преобразованием вычислений в программе в машинный код, но на машинном уровне доступны некоторые особенности, недоступные из программы на С. Например, каждый раз, когда процессор x86-64 выполняет арифметическую или логическую операцию, он устанавливает 1-битный *флаг условия*, который называется PF (от «parity flag» – флаг четности), когда младшие 8 бит в результате имеют четное число единиц, и 0 в противном случае. Чтобы получить эту информацию на языке С, требуется выполнить как минимум семь операций сдвига, маскирования и ИСКЛЮЧАЮЩЕГО ИЛИ (см. упражнение 2.65). Даже притом что микропроцессор вычисляет эту информацию как часть каждой арифметической или логической операции, программа на С лишена возможности напрямую определять значение флага PF. Но эту задачу легко решить, включив в программу несколько ассемблерных инструкций.

Есть два способа внедрить ассемблерный код в программы на С. Первый: написать функцию целиком на языке ассемблера, поместить ее в отдельный файл и позволить ассемблеру и компоновщику объединить ее с программным кодом на С. Второй: использовать поддержку *встраивания ассемблерного кода*, имеющуюся в GCC, позволяющую с помощью директивы `asm` вставлять короткие фрагменты кода на ассемблере прямо в программу на С. Преимущество этого подхода в том, что он сводит к минимуму объем машинно зависимого кода.

Конечно, включение ассемблерного кода в программу на С делает код зависимым от определенного класса машин (например, x86-64), и поэтому его следует использовать, только когда желаемое действие можно выполнить лишь так и никак иначе.

### 3.3. Форматы данных

Вследствие своего происхождения как 16-разрядной архитектуры, которая затем расширилась до 32-разрядной архитектуры, Intel использует термин «слово» для обозначения 16-разрядного типа данных. По этой причине 32-разрядные числа иногда называют «двойными словами», а 64-разрядные – «четверными словами». В табл. 3.1 перечислены представления простых типов данных языка С в архитектуре x86-64. Стандартные значения `int` хранятся в виде двойных слов (32 бита). Указатели (тип `char*` в табл. 3.1) хранятся как 8-байтные четверные слова, как и следовало ожидать для 64-разрядной архитектуры. Тип данных `long` в x86-64 занимает 64 бита, охватывая довольно широкий диапазон значений. В большинстве примеров кода в этой главе используются указатели и тип данных `long`, занимающие четыре слова. Набор инструкций x86-64 также включает полный набор инструкций для работы с байтами, словами и двойными словами.

**Таблица 3.1.** Размеры типов данных на C в архитектуре x86-64.  
В 64-разрядной архитектуре указатели занимают 8 байт

Объявление на C	Тип данных Intel	Суффикс в языке ассемблера	Размер (в байтах)
char	Байт	b	1
short	Слово	w	2
int	Двойное слово	l	4
long	Четверное слово	q	8
char *	Четверное слово	q	8
float	Одинарной точности	s	4
double	Двойной точности	l	8

Числа с плавающей точкой представлены двумя основными форматами: с одинарной точностью (4 байта), соответствующим типу `float` в языке C, и с двойной точностью (8 байт), соответствующим типу `double`. Микропроцессоры семейства x86 традиционно реализуют все операции с плавающей точкой с использованием специального 80-разрядного (10-байтного) формата представления чисел с плавающей точкой (см. упражнение 2.86). Этот формат можно указать в программах на C, объявив переменную с типом `long double`. Однако мы не рекомендуем использовать данный формат. Он непреносим на другие классы машин и часто реализован без использования высокопроизводительного оборудования, в отличие от операций с плавающей точкой с одинарной и двойной точностью.

Как показано в табл. 3.1, большинство ассемблерных инструкций, генерируемых GCC, имеют односимвольный суффикс, обозначающий размер операнда. Например, инструкция перемещения данных имеет четыре варианта: `movb` (перемещение байта), `movw` (перемещение слова), `movl` (перемещение двойного слова) и `movq` (перемещение четверного слова). Суффикс `l` используется для обозначения двойных слов, потому что 32-разрядные числа считаются «длинными словами» (long words). В ассемблерном коде суффикс `l` используется для обозначения 4-байтных целых, а также 8-байтных чисел с плавающей точкой двойной точности. Здесь нет никакой двусмысленности, потому что операции с плавающей точкой выполняются совершенно другим набором инструкций и регистров.

### 3.4. Доступ к информации

Центральное процессорное устройство (Central Processing Unit, CPU) с архитектурой x86-64 имеет набор из 16 *регистров общего назначения*, способных хранить 64-разрядные значения. Эти регистры используются для хранения целочисленных данных, а также указателей. На рис. 3.1 показана диаграмма с этими 16 регистрами. Имена всех регистров начинаются с `%r`, но в остальном они следуют различным соглашениям об именах, обусловленных историей развития набора инструкций. Оригинальная модель 8086 имела восемь 16-разрядных регистров, которые на рис. 3.1 показаны с именами от `%ax` до `%sp`. Каждый из них имел свое назначение, и поэтому их имена отражали их предназначения. При расширении архитектуры до IA32 размеры этих регистров были увеличены до 32 разрядов, и им были присвоены имена от `%eax` до `%esp`. С расширением архитектуры до x86-64 размеры первоначальных восьми регистров увеличились до 64 разрядов, и они получили имена от `%rax` до `%rsp`. К восьми оригинальным регистрам также были добавлены еще восемь регистров, которые в соответствии с новым соглашением об именовании получили имена от `%r8` до `%r15`.

63	31	15	7	0	
%rax	%eax	%ax	%al		Возвращаемое значение
%rbx	%ebx	%bx	%bl		Сохраняется вызываемым
%rcx	%ecx	%cx	%cl		4-й аргумент
%rdx	%edx	%dx	%dl		3-й аргумент
%rsi	%esi	%si	%sil		2-й аргумент
%rdi	%edi	%di	%dil		1-й аргумент
%rbp	%ebp	%bp	%bpl		Сохраняется вызывающим
%rsp	%esp	%sp	%spl		Указатель стека
%r8	%r8d	%r8w	%r8b		5-й аргумент
%r9	%r9d	%r9w	%r9b		6-й аргумент
%r10	%r10d	%r10w	%r10b		Сохраняется вызывающим
%r11	%r11d	%r11w	%r11b		Сохраняется вызывающим
%r12	%r12d	%r12w	%r12b		Сохраняется вызывающим
%r13	%r13d	%r13w	%r13b		Сохраняется вызывающим
%r14	%r14d	%r14w	%r14b		Сохраняется вызывающим
%r15	%r15d	%r15w	%r15b		Сохраняется вызывающим

**Рис. 3.1.** Целочисленные регистры. К младшим частям всех 16 регистров можно обращаться как к байтам, словам (16 разрядов), двойным словам (32 разряда) и четверным словам (64 разряда)

Как показывают вложенные прямоугольники на рис. 3.1, инструкции могут работать с данными разного размера, хранящимися в младших байтах 16 регистров. Операции с отдельными байтами манипулируют младшими байтами, 16-разрядные операции – двумя младшими байтами, 32-разрядные операции – четырьмя младшими байтами, 64-разрядные операции – целыми регистрами.

В следующих разделах мы покажем ряд инструкций для копирования и создания 1-, 2-, 4- и 8-байтных значений. Когда инструкция, генерирующая результат меньше 8 байт, сохраняет его в регистре, используются два соглашения в отношении неиспользуемых байтов: инструкции, генерирующие 1- или 2-байтный результат, оставляют оставшиеся байты без изменений. Инструкции, генерирующие 4-байтный результат,



записывают 0 в старшие 4 байта регистра. Последнее соглашение было принято при расширении IA32 до x86-64.

Как показывают комментарии справа на рис. 3.1, разные регистры играют разные роли в типичных программах. Самым уникальным является указатель стека `%rsp`, хранящий адрес вершины стека во время выполнения. Имеется несколько специализированных инструкций, читающих и изменяющих этот регистр. Остальные 15 регистров более универсальные. Некоторые инструкции специально используют конкретные регистры. Что еще более важно, набор стандартных соглашений о программировании определяет, как должны использоваться регистры для управления стеком, передачи аргументов функциям, возврата значений из функций и хранения локальных и временных данных. Мы рассмотрим эти соглашения далее в этой главе и, в частности, в разделе 3.7, где описывается реализация процедур.

### 3.4.1. Спецификаторы операндов

Большая часть инструкций принимает один или несколько *операндов*, описывающих исходные значения, с которыми выполняется операция, и место, куда следует поместить результат. Архитектура x86-64 поддерживает несколько форматов операторов (табл. 3.2). Исходные значения могут быть представлены константами, храниться в регистрах или в памяти. Результаты могут помещаться в регистр или в память. То есть операнды можно разделить на три категории. Первая категория – *непосредственные операнды*, представлены константами. В формате АТТ такие операнды начинаются с символа `$`, за которым следует целое число в стандартной нотации языка C, например `$-577` или `$0x1F`. Разные инструкции поддерживают разные диапазоны непосредственных значений; ассемблер автоматически выбирает наиболее компактное представление. Вторая категория – *регистр*, в данном случае подразумевается, что значение хранится в указанном регистре, либо в 8-, 4-, 2- или 1-байтной части одного из шестнадцати регистров – 64, 32, 16 или 8 бит соответственно. В табл. 3.2 мы используем нотацию  $r_a$  для обозначения произвольного регистра  $a$ , а его значение обозначается как  $R[r_a]$ , где  $R$  рассматривается как массив, индексируемый идентификаторами регистров.

**Таблица 3.2.** Формы операндов. Операндами могут быть непосредственные значения (константы), значения в регистрах или в памяти. Коэффициент  $s$  может иметь значение 1, 2, 4 или 8

Категория	Форма	Значение операнда	Название
Непосредственное значение	$\$Imm$	$Imm$	Непосредственное значение
Регистр	$r_a$	$R[r_a]$	Регистр
Память	$Imm$	$M[Imm]$	Абсолютный адрес
Память	$(r_a)$	$M[R[r_a]]$	Косвенный адрес
Память	$Imm(r_b)$	$M[Imm + R[r_b]]$	База + смещение
Память	$(r_b, r_i)$	$M[R[r_b] + R[r_i]]$	Индекс
Память	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Индекс
Память	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Нормированный индекс
Память	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Нормированный индекс
Память	$(r_b, r_i, s)$	$M[R[r_b] + R[r_i] \cdot s]$	Нормированный индекс
Память	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Нормированный индекс

Третья категория операндов – ссылка на *память*, определяющая адрес ячейки в памяти, полученный в результате соответствующих вычислений. Этот адрес часто называется *эффективным адресом* (effective address). Мы рассматриваем память как большой массив байтов и используем обозначение  $M_b[Addr]$  для представления ссылки на  $b$ -байтное значение, хранящееся в памяти, начиная с адреса  $Addr$ . Далее для простоты мы будем опускать индекс  $b$ .

Как показано в табл. 3.2, существует несколько режимов адресации, благодаря чему мы имеем несколько форм ссылок на ячейки памяти. Наиболее типичная форма показана в последней строке в табл. 3.2, обозначенная синтаксисом  $Imm(r_b, r_i, s)$ . Такая ссылка состоит из четырех компонентов: непосредственного смещения  $Imm$ , регистра с базовым адресом  $r_b$ , индексного регистра  $r_i$  и масштабного коэффициента  $s$ , который может принимать значения 1, 2, 4 и 8. Базовый и индексный регистры должны быть 64-разрядными регистрами. Эффективный адрес вычисляется по формуле  $Imm + R[r_b] + R[r_i] \cdot s$ . Эту общую форму ссылки часто можно видеть при работе с массивами. Все другие формы являются лишь частными случаями этой общей формы, в которых опущены те или иные компоненты. Как мы увидим далее, более сложные формы адресации удобно использовать для ссылки на элементы массивов и структур.

### Упражнение 3.1 (решение в конце главы)

Предположим, что следующие значения хранятся в памяти по указанным адресам и в регистрах:

Адрес	Значение	Регистр	Значение
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Заполните следующую таблицу, подставив значения указанных операндов:

Операнд	Значение
%rax	_____
0x104	_____
\$0x108	_____
(%rax)	_____
4(%rax)	_____
9(%rax,%rdx)	_____
260(%rcx,%rdx)	_____
0xFC(%rcx,4)	_____
(%rax,%rdx,4)	_____

## 3.4.2. Инструкции перемещения данных

К числу наиболее часто используемых относятся также инструкции, копирующие данные из одного места в другое. Универсальность обозначений операндов позволяет простым инструкциям перемещения данных выполнить то, для чего на многих машинах потребуется последовательность из нескольких инструкций. Мы представим ряд различных инструкций перемещения данных, различающихся типами операндов

источников и приемников, выполняемыми преобразованиями и другими побочными эффектами, которые они могут иметь. При этом мы сгруппируем инструкции в классы, в каждом из которых инструкции выполняют одну и ту же операцию, но с операндами разных размеров.

**Особенности перемещения данных в зависимости от выбора регистра-приемника**

Как было отмечено выше, есть два разных соглашения, касающихся изменения старших байтов инструкциями перемещения данных в зависимости от разрядности выбранного регистра-приемника. Различие между этими соглашениями иллюстрирует следующий код:

1	movabsq	\$0x0011223344556677, %rax	%rax = 0011223344556677
2	movb	\$(-1), %al	%rax = 00112233445566FF
3	movw	\$(-1), %ax	%rax = 001122334455FFFF
4	movl	\$(-1), %eax	%rax = 00000000FFFFFFFF
5	movq	\$(-1), %rax	%rax = FFFFFFFFFFFFFFFF

В следующем обсуждении все значения записаны в шестнадцатеричной нотации. В примере выше инструкция в строке 1 инициализирует регистр %rax комбинацией байтов 0011223344556677. Остальные инструкции используют в качестве источника непосредственное значение -1. Напомним, что в шестнадцатеричном представлении число -1 имеет вид FF...F, где количество цифр F в два раза больше количества байтов в представлении числа. Таким образом, инструкция movb (строка 2) запишет FF в младший байт регистра %rax, а инструкция movw (строка 3) запишет FFFF в два младших байта, при этом обе инструкции оставят старшие байты без изменений. Инструкция movl (строка 4) запишет FFFFFFFF в младшие четыре байта, но, кроме того, запишет 00000000 в старшие четыре байта. Наконец, инструкция movq (строка 5) запишет FFFFFFFFFFFFFFFF, заполнив все байты регистра.

В табл. 3.3 представлен список наиболее простых инструкций перемещения данных, относящихся к классу MOV. Эти инструкции копируют данные из одного местоположения в другое без всяких преобразований. Класс содержит четыре инструкции: movb, movw, movl и movq. Все эти инструкции производят одинаковый эффект и отличаются только размерами копируемых данных: 1, 2, 4 и 8 байт соответственно.

**Таблица 3.3.** Простые инструкции перемещения данных

Инструкция		Эффект	Описание
MOV	$S, D$	$D \leftarrow S$	Перемещение
movb			Перемещение байта
movw			Перемещение слова
movl			Перемещение двойного слова
movq			Перемещение четверного слова
movabsq	$I, R$	$R \leftarrow I$	Перемещение абсолютного четверного слова

Операнд-источник представляет значение – непосредственное или хранящееся в регистре либо в памяти. Операнд-приемник представляет регистр или адрес ячейки в памяти. Архитектура x86-64 вводит ограничение, согласно которому оба операнда в одной инструкции не могут быть ссылками на ячейки памяти. Чтобы скопировать значение из одной ячейки памяти в другую, нужно выполнить две инструкции: первая загружает значение в регистр, а вторая выгружает это значение из регистра в ячейку па-

мяти. Как показано на рис. 3.1, операнды этих инструкций, представляющие регистры, могут быть любыми именованными частями любого из 16 регистров, при этом размер операнда-регистра должен соответствовать размеру, обозначаемому последним символом в имени инструкции (b, w, l или q). В большинстве случаев инструкции MOV изменяют только определенные байты регистра или ячейки памяти, соответствующие операнду-приемнику. Единственным исключением является инструкция `movl`, которая, получая полный регистр в качестве приемника, запишет 0 в старшие 4 байта. Это исключение возникает из соглашения, принятого в x86-64, согласно которому любая инструкция, генерирующая 32-разрядное значение для регистра, также записывает 0 в старшую половину регистра.

Следующие примеры использования инструкции MOV демонстрируют пять различных сочетаний типов операндов-источников и операндов-приемников. Напомним, что операнд-источник указывается первым, а операнд-приемник – вторым.

1	<code>movl \$0x4050,%eax</code>	Непосредственное значение -- Регистр, 4 байта
2	<code>movw %bp,%sp</code>	Регистр -- Регистр, 2 байта
3	<code>movb (%rdi,%rcx),%al</code>	Память -- Регистр, 1 байт
4	<code>movb \$-17,(%rsp)</code>	Непосредственное значение -- Память, 1 байт
5	<code>movq %rax,-12(%rbp)</code>	Регистр -- Память, 8 байт

Последняя инструкция в табл. 3.3 предназначена для работы с 64-разрядными непосредственными значениями. Обычная инструкция `movq` может принимать только 32-разрядные значения в качестве операнда-источника, такие как целые числа в дополнительном коде. Она расширяет знаковый разряд, чтобы получить 64-разрядное значение, и сохраняет результат в операнде-приемнике. Команда `movabsq` может принимать произвольные 64-разрядные непосредственные значения в качестве операнда-источника, но в роли операнда-приемника допускает указывать только регистры.

В табл. 3.4 и 3.5 показаны два класса инструкций перемещения данных, предназначенных для копирования данных из операнда-источника меньшего размера в операнд-приемник большего размера. Операндом-источником во всех этих инструкциях может быть регистр или местоположение в памяти, а операндом-приемником – только регистр. Инструкции в классе MOVZ заполняют нулями оставшиеся байты в операнде-приемнике, а инструкции в классе MOVS расширяют знак, копируя самый старший бит операнда-источника. Обратите внимание, что имя каждой инструкции имеет двухсимвольный суффикс, обозначающий размеры. Первый определяет размер операнда-источника, а второй – размер операнда-приемника. Как видите, в каждом классе имеется три инструкции, охватывающие все случаи 1- и 2-байтных размеров операнда-источника и 2- и 4-байтных размеров операнда-приемника, с учетом только тех случаев, конечно, когда размер операнда-приемника превышает размер операнда-источника.

Обратите внимание на отсутствие в табл. 3.4 инструкции, перемещающей 4-байтное исходное значение в 8-байтный приемник. Логично было бы предположить, что такая инструкция будет называться `movzlw`, но ее не существует. По этой причине такой вид перемещения данных можно реализовать с использованием инструкции `movl` с регистром в роли операнда-приемника. Этот метод использует свойство инструкции, генерирующей 4-байтное значение в регистре-приемнике, согласно которому старшие 4 байта заполняются нулями. В противном случае перемещение в 64-разрядные приемники для всех трех типов операндов-источников производится с расширением знака, при этом для двух типов источников меньшего размера поддерживается перемещение с расширением нулями.

В табл. 3.5 также представлена инструкция `cltq`. Она не имеет операндов и всегда перемещает данные из регистра `%eax` и в регистр `%rax` с расширением знакового разряда. Как следствие она дает тот же эффект, что и инструкция `movslq %eax, %rax`, но имеет более компактное представление в коде.

**Таблица 3.4.** Инструкции перемещения, расширяющие данные нулями.  
Операндом-источником во всех этих инструкциях может быть регистр или местоположение в памяти, а операндом-приемником – только регистр

Инструкция	Эффект	Описание
MOVZ <i>S, R</i>	$R \leftarrow S$ (с расширением нулями)	Перемещает данные с расширением нулями
movzbw		Перемещает байт в слово с расширением нулями
movzbl		Перемещает байт в двойное слово с расширением нулями
movzwl		Перемещает слово в двойное слово с расширением нулями
movzbq		Перемещает байт в четверное слово с расширением нулями
movzwq		Перемещает слово в четверное слово с расширением нулями

**Таблица 3.5.** Инструкции перемещения, расширяющие знаковый разряд.  
Операндом-источником во всех этих инструкциях может быть регистр или местоположение в памяти, а операндом-приемником – только регистр

Инструкция	Эффект	Описание
MOVS <i>S, R</i>	$R \leftarrow S$ (с расширением знакового разряда)	Перемещает данные с расширением знакового разряда
movsbw		Перемещает байт в слово с расширением знакового разряда
movsbl		Перемещает байт в двойное слово с расширением знакового разряда
movswl		Перемещает слово в двойное слово с расширением знакового разряда
movsbq		Перемещает байт в четверное слово с расширением знакового разряда
movswq		Перемещает слово в четверное слово с расширением знакового разряда
movslq		Перемещает двойное слово в четверное слово с расширением знакового разряда
cltq	$\%rax \leftarrow \%eax$ (с расширением знакового разряда)	Перемещает $\%eax$ в $\%rax$ с расширением знакового разряда

### Упражнение 3.2 (решение в конце главы)

Для каждой из следующих ассемблерных инструкций определите соответствующий суффикс, опираясь на операнды. (Например, `mov` можно записать как `movb`, `movw`, `movl` или `movq`.)

```

mov__    %eax, (%rsp)
mov__    (%rax), %dx
mov__    $0xFF, %bl
mov__    (%rsp,%rdx,4), %dl
mov__    (%rdx), %rax
mov__    %dx, (%rax)

```

**Упражнение 3.3 (решение в конце главы)**

Все следующие инструкции заставляют ассемблер сгенерировать сообщение об ошибке при их компиляции. Объясните причину для каждой строки.

```
movb    $0xF, (%ebx)
movl    %rax, (%rsp)
movw    (%rax), 4(%rsp)
movb    %al, %sl
movq    %rax, $0x123
movl    %eax, %rdx
movb    %si, 8(%rbp)
```

**Сравнение команд перемещения байтов**

Следующие примеры иллюстрируют, как разные инструкции изменяют или не изменяют старшие байты в операнде-приемнике. Обратите внимание, что три инструкции перемещения байта – `movb`, `movsbq` и `movzbq` – имеют некоторые отличия между собой:

1	<code>movabsq \$0x0011223344556677, %rax</code>	<code>%rax = 0011223344556677</code>
2	<code>movb \$0xAA, %dl</code>	<code>%dl = AA</code>
3	<code>movb %dl, %al</code>	<code>%rax = 00112233445566AA</code>
4	<code>movsbq %dl, %rax</code>	<code>%rax = FFFFFFFF000000AA</code>
5	<code>movzbq %dl, %rax</code>	<code>%rax = 00000000000000AA</code>

В следующем обсуждении все значения записаны в шестнадцатеричной нотации. Первые две инструкции в примере выше инициализируют регистры `%rax` и `%dl` значениями `0011223344556677` и `AA` соответственно. Все остальные инструкции копируют младший байт из `%rdx` в младший байт `%rax`. Инструкция `movb` (строка 3) не изменяет другие байты. Инструкция `movsbq` (строка 4) записывает в остальные 7 байт либо `FF`, либо `00`, в зависимости от значения знакового разряда в байте-источнике. Так как шестнадцатеричная цифра `A` представляет двоичное значение `1010`, расширение знака вызывает запись `FF` во все старшие байты. Инструкция `movzbq` (строка 5) всегда обнуляет 7 других байт.

**3.4.3. Примеры перемещения данных**

В качестве примера кода, использующего инструкции перемещения данных, рассмотрим процедуру обмена данными, показанную в листинге 3.1. В этом листинге приводится исходный код на C и ассемблерный код, сгенерированный GCC.

Как можно видеть в листинге 3.1 (b), функция `exchange` реализуется всего тремя инструкциями: двумя инструкциями перемещения данных (`movq`) и одной инструкцией возврата в точку вызова функции (`ret`). Детали вызова функций и возврата из них мы рассмотрим в разделе 3.7, а пока лишь упомянем, что аргументы передаются функциям в регистрах. Комментарии в ассемблерном коде отмечают это. Функция возвращает значение, сохраняя его в регистре `%rax` или в одной из младших частей этого регистра.

**Листинг 3.1.** Код функции `exchange` на языках C и ассемблера.  
Регистры `%rdi` и `%rsi` хранят параметры `x` и `y` соответственно

```
(a) Код на C
long exchange(long *xp, long y)
{
    long x = *xp;
```

```

    *xp = y;
    return x;
}

```

(b) Код на ассемблере

```

long exchange(long *xp, long y)
xp в %rdi, y в %rsi
1  exchange:
2  movq    (%rdi), %rax    Извлечь x из xp. Сохранить как возвращаемое значение.
3  movq    %rsi, (%rdi)    Сохранить y в xp.
4  ret                     Вернуться.

```

### Новичок в C? Некоторые примеры указателей

Функция `exchange` (листинг 3.1(a)) является хорошей иллюстрацией использования указателей в языке C. Аргумент `xp` – это указатель на длинное целое, а `y` – само длинное целое. Инструкция

```
long x = *xp;
```

указывает, что программа должна прочесть значение, хранящееся в ячейке, на которую ссылается `xp`, и запомнить его в локальной переменной с именем `x`. Эта операция считывания называется *разыменованием* указателя. Оператор `*` в языке C выполняет разыменование указателя. Инструкция

```
*xp = y;
```

выполняет обратное действие – она записывает значение параметра `y` в ячейку памяти, на которую ссылается `xp`. Это еще одна форма разыменования указателя (а следовательно, и оператора `*`), только здесь он обозначает операцию записи, потому что находится в левой части выражения присваивания. Вот пример выполнения функции `exchange`:

```

long a = 4;
long b = exchange(&a, 3);
printf("a = %ld, b = %ld\n", a, b);

```

Этот код выведет:

```
a = 3, b = 4
```

Оператор `&` в языке C (называется «взятие адреса переменной») *создает* указатель, в данном случае – указатель на ячейку, где хранится значение переменной `a`. Функция `exchange` затирает значение в `a`, сохраняя в ней число 3, и возвращает ее прежнее значение 4. Обратите внимание, как, получая указатель, функция `exchange` может изменить данные, хранящиеся в удаленной ячейке.

Когда процедура начинает выполнение, ее параметры – `xp` и `y` – находятся в регистрах `%rdi` и `%rsi` соответственно. Инструкция в строке 2 читает `x` из памяти и сохраняет значение в регистре `%rax`, прямо реализуя операцию `x = *xp` в программе на C. Позже регистр `%rax` будет использоваться для возврата результата из функции, то есть функция вернет `x`. Инструкция в строке 3 записывает `y` в ячейку памяти, на которую ссылается параметр `xp`, то есть по адресу, хранящемуся в регистре `%rdi`, что является прямой реализацией операции `*xp = y`. Этот пример показывает, как инструкции `mov` могут использоваться для чтения из памяти в регистр (строка 2) и для записи из регистра в память (строка 3).

Стоит отметить две особенности этого ассемблерного кода. Во-первых, мы видим, что «указатели» в языке C – это просто адреса. Разыменование указателя предполагает

копирование адреса в регистр, а затем использование этого регистра для ссылки на память. Во-вторых, локальные переменные, такие как `x`, часто хранятся в регистрах, а не в ячейках памяти. Доступ к регистрам выполняется намного быстрее, чем к памяти.

#### Упражнение 3.4 (решение в конце главы)

Допустим, у нас есть две переменные, `sp` и `fp`, объявленные, как показано ниже:

```
src_t *sp;
dest_t *dp;
```

где `src_t` и `dest_t` – это типы данных, объявленные с помощью `typedef`. Мы хотим использовать соответствующую пару инструкций перемещения данных для реализации операции

```
*dp = (dest_t) *sp;
```

Предположим, что значения `sp` и `fp` хранятся в регистрах `%rdi` и `%rsi` соответственно. Для каждой записи в следующей таблице покажите две инструкции, реализующие указанное перемещение. Первая инструкция должна читать данные из памяти, выполнять соответствующее преобразование и устанавливать соответствующую часть регистра `%rax`. Вторая инструкция должна записать соответствующую часть `%rax` в память. В обоих случаях такими частями могут быть `%rax`, `%eax`, `%ax` или `%al`.

Напомним, что при преобразовании, включая изменение размера и изменение между целыми со знаком и без знака в C, изменение размера должно выполняться первым (раздел 2.2.6).

src_t	dest_t	Инструкция
long	long	movq (%rdi), %rax movq %rax, (%rsi)
char	int	_____
		_____
char	unsigned	_____
		_____
unsigned char	long	_____
		_____
int	char	_____
		_____
unsigned	unsigned char	_____
		_____
char	short	_____
		_____

#### Упражнение 3.5 (решение в конце главы)

Вам дана следующая информация: функция с прототипом

```
void decode1(long *xp, long *yp, long *zp);
```

компилируется в ассемблерный код

```
void decode1(long *xp, long *yp, long *zp)
xp в %rdi, yp в %rsi, zp в %rdx
```



```
decode1:
    movq    (%rdi), %r8
    movq    (%rsi), %rcx
    movq    (%rdx), %rax
    movq    %r8, (%rsi)
    movq    %rcx, (%rdx)
    movq    %rax, (%rdi)
    ret
```

Параметры `xr`, `ur` и `zr` хранятся в регистрах `%rdi`, `%rsi` и `%rdx` соответственно. Напишите код `decode1` на C, эквивалентный приведенному ассемблерному коду.

3.4.4. Вталкивание данных в стек и выталкивание из стека

Две последние инструкции перемещения данных (табл. 3.6) используются для вталкивания данных в программный стек и выталкивания их из стека. Как мы увидим далее, стек играет важную роль в вызовах процедур. Стек – это структура данных, позволяющая добавлять и удалять значения, но только в соответствии с принципом «последним пришел – первым ушел». Добавление данных в стек производится с помощью операции *втталкивания* – *push*, а удаление – с помощью операции *выталкивания* – *pop*, при этом всегда выталкивается значение, которое было добавлено в стек последним. Стек может быть реализован как массив, добавление и удаление элементов которого всегда производится с одного конца. Этот конец называется *вершиной* стека. В архитектуре x86-64 программный стек хранится в некоторой области памяти. Как показано на рис. 3.2, стек растет вниз, поэтому верхний элемент стека имеет наименьший адрес среди всех других элементов. (По соглашению мы рисуем стеки в перевернутом виде, поэтому «вершина» стека показана внизу.) Указатель стека `%rsp` хранит адрес элемента на вершине стека.

Таблица 3.6. Инструкции *push* и *pop*

Инструкция	Эффект	Описание
<code>pushq S</code>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Вталкивает четверное слово в стек
<code>popq D</code>	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	Выталкивает четверное слово из стека

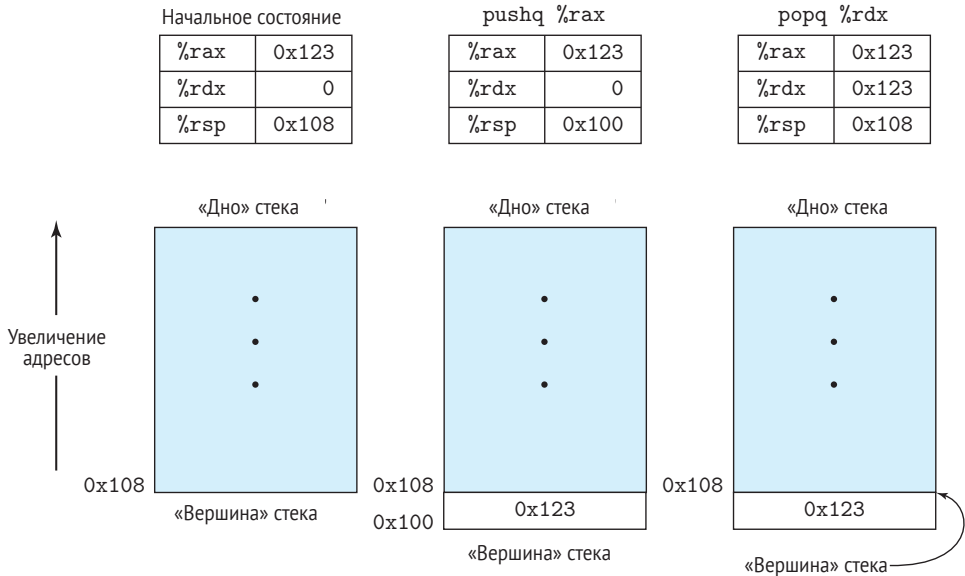
Инструкция `pushq` вталкивает данные в стек, а инструкция `popq` выталкивает их. Каждая из этих инструкций принимает единственный операнд – источник данных для вталкивания и приемник для выталкивания.

Вталкивание четверного слова в стек включает предварительное уменьшение указателя стека на 8, а затем запись значения по новому адресу вершины стека.

Таким образом результат выполнения инструкции `pushq %rbp` эквивалентен результату выполнения двух следующих инструкций:

```
subq $8,%rsp      Уменьшит значение указателя стека
movq %rbp,(%rsp)  Сохранит %rbp на стеке
```

за исключением того, что инструкция `pushq` представлена в машинном коде одним байтом, а пара указанных выше инструкций требует 8 байт. Два первых столбца на рис. 3.2 показывают результат выполнения инструкции `pushq %rax`, когда `%rsp` хранит значение `0x108` и `%rax` хранит значение `0x123`. Сначала значение `%rsp` уменьшается на 8 и получает новое значение `0x100`, а затем значение `0x123` сохраняется в памяти по адресу `0x100`.



**Рис. 3.2.** Иллюстрация работы стека. По соглашению мы рисуем стеки в перевернутом виде, поэтому «вершина» стека показана внизу. В архитектуре x86-64 стеки растут в сторону меньших адресов, поэтому вталкивание в стек включает уменьшение указателя стека (регистр %rsp) и сохранение вталкиваемого значения в памяти, а выталкивание включает чтение из памяти и увеличение указателя стека

Выталкивание четверного слова выполняется путем чтения значения с вершины стека и последующего увеличения значения указателя стека на 8. Соответственно, результат выполнения инструкции `popq %rax` эквивалентен результату выполнения двух следующих инструкций:

```
movq (%rsp),%rax    Прочитает данные со стека в %rax
addq $8,%rsp        Увеличит указатель стека
```

Третий столбец на рис. 3.2 показывает результат выполнения инструкции `popq %rdx` сразу после выполнения команды `pushq`. Значение 0x123 будет считано из памяти и помещено в регистр %rdx. Значение регистра %rsp вновь увеличится до значения 0x108. Как показано на рис. 3.2, значение 0x123 будет продолжать оставаться в ячейке памяти с адресом 0x100, пока не будет затерто другой инструкцией вталкивания в стек. В то же время считается, что вершина стека всегда находится по адресу, хранящемуся в регистре %rsp.

Поскольку стек находится в одной памяти с программным кодом и другими данными, то сама программа может обращаться к любым позициям в стеке, используя стандартные методы адресации памяти. Например, предположим, что на вершине стека хранится четверное слово, тогда инструкция `movq 8(%rsp),%rdx` скопирует второе четверное слово из стека в регистр %rdx.

## 3.5. Арифметические и логические операции

В табл. 3.7 перечислены некоторые операции с целыми числами и логическими значениями в архитектуре x86-64. Большинство операций представлены в виде классов инструкций, потому что они могут иметь разные варианты в зависимости от размеров

операндов. (Только `leaq` не имеет вариантов, зависящих от размера.) Например, класс инструкций `ADD` включает четыре инструкции сложения: `addb`, `addw`, `addl` и `addq`, – складывающие байты, слова, двойные слова и четверные слова соответственно. На самом деле каждый из перечисленных классов включает инструкции, оперирующие данными этих четырех размеров. Кроме того, операции разделены на четыре группы: загрузка эффективного адреса, унарные, бинарные (двухместные) и сдвиги. *Бинарные операции* имеют два операнда, а *унарные* – только один. Операнды указываются с использованием той же нотации, что и в разделе 3.4.

**Таблица 3.7.** Целочисленные арифметические операции. Инструкция загрузки эффективного адреса (`leaq`) часто используется для выполнения простых арифметических действий. Остальные – более типичные унарные или бинарные операции. Для обозначения арифметического (`arithmic`) и логического (`logical`) сдвигов вправо используются формы записи  $\gg_A$  и  $\gg_L$  соответственно. Обратите внимание на неочевидный порядок следования операндов в ассемблерном коде в формате АТТ

Инструкция		Эффект	Описание
<code>leaq</code>	$S, D$	$D \leftarrow \&S$	Загрузка эффективного адреса
<code>INC</code>	$D$	$D \leftarrow D + 1$	Инкремент
<code>DEC</code>	$D$	$D \leftarrow D - 1$	Декремент
<code>NEG</code>	$D$	$D \leftarrow -D$	Отрицание
<code>NOT</code>	$D$	$D \leftarrow \neg D$	Дополнение
<code>ADD</code>	$S, D$	$D \leftarrow D + S$	Сложение
<code>SUB</code>	$S, D$	$D \leftarrow D - S$	Вычитание
<code>IMUL</code>	$S, D$	$D \leftarrow D * S$	Умножение
<code>XOR</code>	$S, D$	$D \leftarrow D \wedge S$	ИСКЛЮЧАЮЩЕЕ ИЛИ
<code>OR</code>	$S, D$	$D \leftarrow D \vee S$	ИЛИ
<code>AND</code>	$S, D$	$D \leftarrow D \& S$	И
<code>SAL</code>	$k, D$	$D \leftarrow D \ll k$	Сдвиг влево
<code>SHL</code>	$k, D$	$D \leftarrow D \ll k$	Сдвиг влево (то же, что и <code>SAL</code> )
<code>SAR</code>	$k, D$	$D \leftarrow D \gg_A k$	Арифметический сдвиг вправо
<code>SHR</code>	$k, D$	$D \leftarrow D \gg_L k$	Логический сдвиг вправо

### 3.5.1. Загрузка эффективного адреса

Инструкция загрузки эффективного адреса `leaq` фактически является вариантом инструкции `movq`. Она имеет форму инструкции, читающей данные из памяти в регистр, но при этом вообще не ссылается на память. Ее первый операнд имеет вид ссылки на ячейку памяти, однако вместо чтения из указанного адреса она копирует эффективный адрес в операнд-приемник. Мы показали этот эффект в табл. 3.7 с использованием оператора взятия адреса `&S` из языка C. Эту инструкцию можно использовать для формирования указателей с целью последующего обращения к памяти. Кроме того, ее можно использовать для компактного описания обычных арифметических операций. Например, если регистр `%rdx` содержит значение  $x$ , то инструкция `leaq 7(%rdx,%rdx,4), %rax` запишет в регистр `%rax` результат вычисления выражения  $5x + 7$ . Компиляторы часто

используют хитроумные трюки с использованием `leaq`, которые не имеют ничего общего с эффективным вычислением адресов. Операнд-приемник должен быть регистром.

### Упражнение 3.6 (решение в конце главы)

Пусть регистр `%rax` хранит значение  $x$ , а регистр `%rcx` – значение  $y$ . Заполните следующую таблицу формулами, показывающими значения, которые будут сохранены в регистре `%rdx` каждой из ассемблерных инструкций.

Инструкция	Результат
<code>leaq 6(%rax), %rdx</code>	_____
<code>leaq (%rax,%rcx), %rdx</code>	_____
<code>leaq (%rax,%rcx,4), %rdx</code>	_____
<code>leaq 7(%rax,%rax,8), %rdx</code>	_____
<code>leaq 0xA(,%rcx,4), %rdx</code>	_____
<code>leaq 9(%rax,%rcx,2), %rdx</code>	_____

В качестве иллюстрации использования `leaq` в скомпилированном коде рассмотрим следующую программу на C:

```
long scale(long x, long y, long z) {
    long t = x + 4 * y + 12 * z;
    return t;
}
```

Арифметические операции внутри функции компилируются в последовательность из трех инструкций `leaq`, описываемых в комментариях справа:

```
long scale(long x, long y, long z)
x в %rdi, y в %rsi, z в %rdx
scale:
leaq  (%rdi,%rsi,4), %rax      x + 4*y
leaq  (%rdx,%rdx,2), %rdx      z + 2*z = 3*z
leaq  (%rax,%rdx,4), %rax      (x+4*y) + 4*(3*z) = x + 4*y + 12*z
ret
```

Способность инструкции `leaq` выполнять сложение и ограниченные формы умножения часто используется для вычисления простых арифметических выражений, таких как в этом примере.

### Упражнение 3.7 (решение в конце главы)

Взгляните на следующий код, в котором пропущено вычисляемое выражение:

```
long scale2(long x, long y, long z) {
    long t = _____;
    return t;
}
```

В результате компиляции фактической функции GCC дает следующий ассемблерный код:

```
long scale2(long x, long y, long z)
x в %rdi, y в %rsi, z в %rdx
scale2:
leaq  (%rdi,%rdi,4), %rax
leaq  (%rax,%rsi,2), %rax
```

```
leaq  (%rax,%rdx,8), %rax
ret
```

Подставьте пропущенное выражение в исходный код на С.

### 3.5.2. Унарные и бинарные операции

Вторая группа операций – это унарные (одноместные) операции, принимающие единственный операнд, который служит одновременно и источником, и приемником. Этим операндом может быть регистр или местоположение в памяти. Например, инструкция `incq (%rsp)` увеличивает значение на вершине стека. Такой синтаксис напоминает операторы инкремента (`++`, увеличение на 1) и декремента (`--`, уменьшение на 1) в языке С.

Третья группа операций – бинарные (двухместные) операции, в которых второй операнд используется одновременно как источник и приемник. Этот синтаксис напоминает комбинированные операторы присваивания в языке С, такие как `+=`. Но обратите внимание, что операнд-источник указывается первым, а операнд-приемник – вторым. Это условие выглядит несколько странным для некоммутативных операций. Например, инструкция `subq %rax, %rdx` уменьшит значение регистра `%rdx` на величину, хранящуюся в регистре `%rax`. (Эту инструкцию следует читать так: «вычесть `%rax` из `%rdx`».) Первым операндом может быть непосредственное значение, регистр или ячейка памяти. Вторым операндом может быть регистр или ячейка памяти. Однако, как и в случае с инструкциями `MOV`, оба операнда не могут быть ячейками памяти.

#### Упражнение 3.8 (решение в конце главы)

Предположим, что в памяти по указанным адресам и в указанных регистрах хранятся следующие значения:

Адрес	Значение	Регистр	Значение
0x100	0xFF	%rax	0x100
0x108	0xAB	%rcx	0x1
0x110	0x13	%rdx	0x3
0x118	0x11		

Заполните следующую таблицу и покажите результат выполнения приведенных в ней инструкций, указав регистры и ячейки памяти, которые будут изменены, а также значения в них:

Инструкция	Приемник	Значение
<code>addq %rcx, (%rax)</code>	_____	_____
<code>subq %rdx, 8(%rax)</code>	_____	_____
<code>imulq \$16, (%rax, %rdx, 8)</code>	_____	_____
<code>incq 16(%rax)</code>	_____	_____
<code>decq %rcx</code>	_____	_____
<code>subq %rdx, %rax</code>	_____	_____

### 3.5.3. Операции сдвига

Завершающая группа операций – операции сдвига, в которых величина сдвига определяется первым, а сдвигаемое значение – вторым операндом. Сдвиг вправо может быть арифметическим или логическим. В разных инструкциях величина сдвига может опре-

делиться константой или однобайтным регистром %cl. (Необычность этих инструкций в том, что они позволяют использовать в роли операнда только этот конкретный регистр.) Использование одного байта для представления величины сдвига позволяет закодировать величину сдвига в диапазоне от 0 до  $2^8 - 1 = 255$ . В архитектуре x86-64 инструкция сдвига оперирует значениями данных с длиной  $w$  бит и определяет величину сдвига по младшим  $m$  битам регистра %cl, где  $2^m = w$ . Старшие биты просто игнорируются. Так, например, когда регистр %cl имеет шестнадцатеричное значение 0xFF, инструкция salb выполнит сдвиг на 7, инструкция salw – на 15, sall – на 31, а salq – на 63 позиции.

В табл. 3.7 показаны две инструкции сдвига влево: SAL и SHL. Обе дают одинаковый результат, заполняя разряды справа нулями. Команды сдвига вправо различаются тем, что SAR выполняет арифметический сдвиг (заполнение производится копиями знакового разряда), а SHR выполняет логический сдвиг (заполняет биты слева нулями). Роль операнда-приемника в операции сдвига может играть либо регистр, либо ячейка памяти. Мы обозначили две разные операции сдвига вправо в табл. 3.7 как  $\gg_A$  (arithmetic – арифметический) и  $\gg_L$  (logical – логический).

### Упражнение 3.9 (решение в конце главы)

Предположим, что мы решили написать код на языке ассемблера, соответствующий следующей функции на C:

```
long shift_left4_rightn(long x, long n)
{
    x <<= 4;
    x >>= n;
    return x;
}
```

Следующий код – это фрагмент на языке ассемблера, который выполняет сдвиг и оставляет результат в регистре %rax. Здесь опущены две ключевые инструкции. Параметры  $x$  и  $n$  хранятся в регистрах %rdi и %rsi соответственно.

```
long shift_left4_rightn(long x, long n)
x в %rdi, n в %rsi

shift_left4_rightn:
    movq    %rdi, %rax      Получает x
                        x <<= 4
    -----
    movl    %esi, %ecx      Получает n (4 байта)
    -----
                        x >>= n
```

Вставьте недостающие инструкции, соответствующие комментариям справа. Сдвиг вправо выполняется арифметически.

## 3.5.4. Обсуждение

Как вы могли убедиться, большинство инструкций из перечисленных в табл. 3.7 можно использовать для арифметических вычислений и с числами без знака, и с числами в дополнительном коде. Только для сдвига приходится применять инструкции, различающие данные со знаком и без знака. Это одна из особенностей, которая делает арифметику в дополнительном коде предпочтительным способом выполнения операций с числами со знаком.

В листинге 3.2 показан пример функции, которая выполняет арифметические операции, и ее реализация на языке ассемблера. Аргументы  $x$ ,  $y$  и  $z$  функции хранятся в регистрах %rdi, %rsi и %rdx соответственно. Ассемблерные инструкции точно соответствуют

строкам в исходном коде на C. Строка 2 вычисляет значение  $x^y$ . В строках 3 и 4 вычисляется выражение  $z*48$  с помощью комбинации инструкций `leaq` и сдвига. Строка 5 объединяет значения `t1` и `0x0F0F0F0F`, выполняя поразрядную операцию И. Заключительное вычитание производится в строке 6. Поскольку приемником в инструкции вычитания является регистр `%rax`, его результат и будет значением, возвращаемым функцией.

### Листинг 3.2. Код арифметической функции на C и ассемблере

(a) Код на C

```
long arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

(b) Код на ассемблере

```
long arith(long x, long y, long z)
x в %rdi, y в %rsi, z в %rdx
1  arith:
2  xorq    %rsi, %rdi          t1 = x ^ y
3  leaq    (%rdx,%rdx,2), %rax  3*z
4  salq    $4, %rax            t2 = 16 * (3*z) = 48*z
5  andl    $252645135, %edi     t3 = t1 & 0x0F0F0F0F
6  subq    %rdi, %rax          t2 - t3
7  ret
```

В ассемблерном коде в листинге 3.2 последовательность сменяющих друг друга значений в регистре `%rax` соответствует результатам вычисления выражений  $3*z$ ,  $z*48$  и `t4` (возвращаемое значение). Как правило, компиляторы генерируют код, который использует разные регистры для разных значений и перемещает эти значения между регистрами.

### Упражнение 3.10 (решение в конце главы)

В следующем варианте функции из листинга 3.2(a) выражения были опущены:

```
long arith2(long x, long y, long z)
{
    long t1 = _____;
    long t2 = _____;
    long t3 = _____;
    long t4 = _____;
    return t4;
}
```

Для этой функции был сгенерирован следующий ассемблерный код:

```
long arith2(long x, long y, long z)
x в %rdi, y в %rsi, z в %rdx
arith2:
    orq    %rsi, %rdi
```

```

sarq    $3, %rdi
notq    %rdi
movq    %rdx, %rax
subq    %rdi, %rax
ret

```

Опираясь на этот ассемблерный код, восстановите выражения в коде на С.

### Упражнение 3.11 (решение в конце главы)

В ассемблерном коде часто можно встретить инструкции, такие как

```
xorq %rdx, %rdx
```

причем в коде на С операции ИСКЛЮЧАЮЩЕЕ ИЛИ отсутствуют.

1. Объясните действие этой конкретной инструкции ИСКЛЮЧАЮЩЕЕ ИЛИ и какую полезную операцию она реализует.
2. Есть ли более простой способ выразить эту операцию на языке ассемблера?
3. Сравните количество байтов, необходимых для кодирования этих двух разных реализаций одной и той же операции.

## 3.5.5. Специальные арифметические операции

Как мы видели в разделе 2.3, умножение двух 64-разрядных целых чисел со знаком или без знака может дать результат, для представления которого требуется 128 бит. Набор инструкций x86-64 обеспечивает ограниченную поддержку операций с 128-разрядными (16-байтными) числами. Следуя соглашению об именовании: слово (2 байта), двойное слово (4 байта) и четверное слово (8 байт), Intel определяет 16-байтный элемент данных как *восьмерное слово* (*oct word*). В табл. 3.8 перечислены инструкции, поддерживающие создание полного 128-разрядного произведения двух 64-разрядных чисел, а также целочисленное деление.

**Таблица 3.8.** Специальные арифметические операции. Эти операции поддерживают полное 128-разрядное умножение и деление как для чисел со знаком, так и для чисел без знака. Пара регистров %rdx и %rax рассматривается как образующая одно 128-разрядное восьмерное слово

Инструкция	Эффект	Описание
imulq <i>S, D</i>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Полное умножение со знаком
mulq <i>S</i>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Полное умножение без знака
cqto	$R[\%rdx]:R[\%rax] \leftarrow R[\%rax]$ (с расширением знака)	Преобразование в восьмерное слово
idivq <i>S</i>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Деление со знаком
divq <i>S</i>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Деление без знака

Инструкция `imulq` имеет две разные формы. Одна форма, показанная в табл. 3.7, входит в класс инструкций `IMUL`. Она представляет операцию умножения «двух операн-



дов» и генерирует 64-разрядное произведение двух 64-разрядных операндов, реализуя операции  $\ast_{64}^u$  и  $\ast_{64}^s$ , описанные в разделах 2.3.4 и 2.3.5. (Напомним, что при усечении произведения до 64 разрядов умножение без знака и умножение со знаком показывают одинаковое поведение на битовом уровне.)

Кроме того, набор инструкций x86-64 включает две разные инструкции умножения с одним операндом для вычисления полного 128-разрядного произведения двух 64-разрядных значений – одну для умножения целых без знака (`mulq`) и одну для умножения целых в дополнительном коде (`imulq`). Для обеих этих инструкций один аргумент должен находиться в регистре `%rax`, а другой передается в операнде-источнике. Результат сохраняется в регистрах `%rdx` (старшие 64 бита) и `%rax` (младшие 64 бита). Несмотря на то что имя `imulq` используется для обозначения двух разных операций умножения, ассемблер способен определить, какая из них имела в виду, по количеству операндов.

Следующий пример кода на C демонстрирует создание 128-разрядного произведения умножением двух 64-разрядных чисел без знака `x` и `y`:

```
#include <inttypes.h>

typedef unsigned __int128 uint128_t;

void store_uprod(uint128_t *dest, uint64_t x, uint64_t y) {
    *dest = x * (uint128_t) y;
}
```

В этой программе мы явно объявили `x` и `y` 64-разрядными числами, используя определения из заголовка `inttypes.h`, являющегося частью расширения стандарта C. К сожалению, этот стандарт не предусматривает 128-разрядных значений. Поэтому мы полагаемся на поддержку 128-разрядных целых чисел, реализованную в GCC в виде типов с именами `__int128`. Наш код использует тип данных `uint128_t`, имя которого было выбрано в соответствии с шаблоном именования других типов в `inttypes.h`. Код указывает, что полученное произведение должно храниться в 16 байтах, на которые ссылается указатель `dest`.

Ассемблерный код, сгенерированный компилятором GCC для этой функции, выглядит так:

```
void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
dest в %rdi, x в %rsi, y в %rdx
1 store_uprod:
2 movq    %rsi, %rax      Скопировать x в регистр-множимое
3 mulq    %rdx            Умножить на y
4 movq    %rax, (%rdi)     Сохранить младшие 8 байт в dest
5 movq    %rdx, 8(%rdi)    Сохранить старшие 8 байт в dest+8
6 ret
```

Обратите внимание, что для сохранения произведения потребовались две инструкции `movq`: одна для сохранения младших 8 байт (строка 4) и одна – старших 8 байт (строка 5). Так как код генерируется для машины с обратным порядком следования байтов (`little-endian`), то старшие байты сохраняются в старших адресах, как показывает спецификатор адреса `8(%rdi)`.

В предыдущей таблице арифметических операций (табл. 3.7) отсутствуют операции деления или деления по модулю. Эти операции выполняются инструкциями деления с одним операндом, подобными инструкциям умножения с одним операндом. Инструкция деления со знаком `idivq` принимает 128-разрядное делимое в регистрах `%rdx` (старшие 64 бита) и `%rax` (младшие 64 бита). Делитель передается инструкции в операнде. Частное сохраняется в регистре `%rax`, а остаток – в регистре `%rdx`.

В большинстве приложений 64-разрядного деления делимое передается как 64-разрядное значение. Значение делимого нужно сохранить в регистре `%rax`. Затем во все биты `%rdx` записать нули (арифметика без знака) или знаковый бит из `%rax` (арифметика со знаком). Последнюю операцию можно выполнить с помощью инструкции `cqto`<sup>2</sup>. Эта инструкция не принимает операндов и просто извлекает знаковый бит из `%rax` и копирует его во все биты в `%rdx`.

В качестве иллюстрации реализации деления в архитектуре x86-64 ниже приводится функция на C, вычисляющая частное и остаток от деления двух 64-разрядных чисел со знаком:

```
void remdiv(long x, long y,
            long *qp, long *rp) {
    long q = x/y;
    long r = x%y;
    *qp = q;
    *rp = r;
}
```

Она компилируется в следующий ассемблерный код:

```
void remdiv(long x, long y, long *qp, long *rp)
x в %rdi, y в %rsi, qp в %rdx, в in %rcx
1 remdiv:
2 movq %rdx, %r8          Скопировать qp
3 movq %rdi, %rax          Переместить x в младшие 8 байт делимого
4 cqto                    Расширить знак в старшие 8 байт делимого
5 idivq %rsi              Разделить на y
6 movq %rax, (%r8)         Сохранить частное в qp
7 movq %rdx, (%rcx)        Сохранить остаток в rp
8 ret
```

В этом коде аргумент `qp` сначала нужно сохранить в другом регистре (строка 2), потому что регистр с аргументом `%rdx` потребуется для выполнения операции деления. Строки 3–4 подготавливают делимое, копируя и расширяя знак `x`. После деления частное, находящееся в регистре `%rax`, сохраняется в `qp` (строка 6), а остаток, находящийся в регистре `%rdx`, сохраняется в `rp` (строка 7).

Деление без знака выполняется инструкцией `divq`. Обычно перед ней регистр `%rdx` заранее сбрасывается в ноль.

### Упражнение 3.12 (решение в конце главы)

Взгляните на следующую функцию, вычисляющую частное и остаток от деления двух беззнаковых 64-разрядных чисел:

```
void uremdiv(unsigned long x, unsigned long y,
             unsigned long *qp, unsigned long *rp) {
    unsigned long q = x/y;
    unsigned long r = x%y;
    *qp = q;
    *rp = r;
}
```

Измените ассемблерный код, показанный для функции деления со знаком, чтобы он соответствовал этой функции.

<sup>2</sup> В документации Intel эта инструкция называется `cqo`. Это один из немногих случаев, когда имя инструкции в формате АТТ не совпадает с именем в формате Intel.

## 3.6. Управление

До этого момента мы рассматривали *линейный код*, когда инструкции выполняются по порядку, одна за другой. Некоторые конструкции языка C, такие как условные операторы, циклы и операторы выбора `switch`, выполняются нелинейно, в зависимости от проверок некоторых условий. Язык ассемблера предоставляет два основных механизма для реализации нелинейной логики управления: они проверяют значения данных и затем изменяют направление потока выполнения или потока данных, в зависимости от результатов этих проверок.

Управление потоком выполнения в зависимости от данных является более распространенным подходом к реализации условного поведения, поэтому мы сначала рассмотрим его. Обычно операторы языка C и инструкции машинного кода выполняются *последовательно* в том порядке, в котором они следуют в программе. Порядок выполнения машинных инструкций можно изменить с помощью инструкции *перехода* в различные части программы, возможно по результатам некоторой проверки. Компилятор должен сгенерировать последовательности инструкций, основанные на этом низкоуровневом механизме, для реализации управляющих конструкций языка C.

Далее мы сначала рассмотрим два способа реализации условных операторов, а затем опишем методы представления циклов и оператора `switch`.

### 3.6.1. Флаги условий

Наряду с целочисленными регистрами центральный процессор поддерживает набор одноразрядных регистров с *флагами условий* (или *флагами состояний*), описывающих атрибуты последних арифметических и логических операций. Затем, опираясь на эти флаги, можно выполнять условные переходы. Вот наиболее часто используемые коды условий:

CF (Carry Flag): флаг переноса. Устанавливается, когда последняя операция вызвала перенос самого старшего двоичного разряда. Используется для обнаружения переполнения при выполнении операций с числами без знака;

ZF (Zero Flag): признак нуля. Устанавливается, когда в результате последней операции был получен ноль;

SF (Sign Flag): флаг знака. Устанавливается, когда в результате последней операции получена отрицательная величина;

OF (Overflow Flag): признак переполнения. Устанавливается, когда последняя операция с числами в дополнительном коде вызвала переполнение – как положительное, так и отрицательное.

Например, предположим, что мы использовали одну из инструкций `ADD` для вычисления выражения, аналогичного оператору присваивания на языке C `t = a + b`, где переменные `a`, `b` и `t` являются целыми числами. Вот как будут установлены флаги, в зависимости от следующих условий:

CF (unsigned) <code>t &lt; (unsigned) a</code>	Переполнение без знака
ZF <code>(t == 0)</code>	Ноль
SF <code>(t &lt; 0)</code>	Отрицательная величина
OF <code>(a &lt; 0 == b &lt; 0) &amp;&amp; (t &lt; 0 != a &lt; 0)</code>	Переполнение со знаком

Инструкция `leaq` не меняет никаких флагов, потому что предназначена для вычисления адресов. С другой стороны, все команды, перечисленные в табл. 3.7, устанавливают те или иные флаги. Для логических операций, таких как `XOR`, флаги переноса и переполнения сбрасываются в 0. Операции сдвига устанавливают флаг переноса

равным последнему биту, сдвинутому за пределы операнда-приемника, а флаг переполнения меняют только при сдвиге на 1 позицию, в соответствии с правилами, зависящими от типа сдвига. По причинам, в которые мы не будем углубляться, инструкции INC и DEC устанавливают флаги переполнения и нуля, но оставляют флаг переноса без изменений.

В дополнение к инструкциям, перечисленным в табл. 3.7, которые влияют на флаги, есть еще два класса инструкций (имеющих 8-, 16-, 32- и 64-разрядную форму), устанавливающих флаги, но не изменяющих значений других регистров; они перечислены в табл. 3.9. Инструкции CMP устанавливают флаги в соответствии с различиями между операндами. Они действуют подобно инструкциям SUB, но не изменяют операнд-приемник, а только устанавливают флаги. В формате АТТ операнды указываются в обратном порядке, из-за чего программный код трудно читать. Эти инструкции устанавливают флаг нуля, если операнды равны. Другие флаги можно использовать для определения отношения между операндами. Инструкции TEST действуют подобно инструкциям AND, но не изменяют операнд-приемник, а только устанавливают флаги. Обычно в этих инструкциях дважды используется один и тот же операнд (например, `testq %rax, %rax`, чтобы узнать, хранит ли `%rax` ноль, отрицательное или положительное значение) либо один из операндов является маской, показывающей, какой разряд проверить.

**Таблица 3.9.** Инструкции сравнения и проверки. Эти инструкции устанавливают флаги без изменения любых других регистров

Инструкция		Основана на	Описание
CMP	$S_1, S_2$	$S_2 - S_1$	Сравнение
cmpb			Сравнение байтов
cmpw			Сравнение слов
cmpl			Сравнение двойных слов
cmpq			Сравнение четверных слов
TEST	$S_1, S_2$	$S_1 \& S_2$	Проверка
testb			Проверка байта
testw			Проверка слова
testl			Проверка двойного слова
testq			Проверка четверного слова

### 3.6.2. Доступ к флагам

Вместо прямого чтения для доступа к флагам используются три наиболее распространенных метода: (1) установить однобайтовое значение равным 0 или 1 в зависимости от некоторой комбинации флагов, (2) выполнить условный переход к другой части программы или (3) выполнить условную передачу данных. Инструкции, реализующие первый метод (табл. 3.10), устанавливают однобайтное значение равным 0 или 1 в зависимости от некоторой комбинации флагов. Этот класс инструкций мы называем SET; они отличаются друг от друга в зависимости от комбинаций учитываемых флагов, которые определяются различными суффиксами в именах инструкций. Важно понимать, что суффиксы в этих инструкциях обозначают разные условия, а не разные размеры операндов. Например, инструкции `setl` и `setb` обозначают «установить, если меньше» (`set less`) и «установить, если ниже» (`set below`), а не «установить длинное слово» или «установить байт».

**Таблица 3.10.** Инструкции SET. Каждая инструкция устанавливает однобайтное значение равным 0 или 1 в зависимости от комбинации флагов. Некоторые инструкции имеют «синонимы» – альтернативные названия одной и той же машинной инструкции

Инструкция	Синоним	Эффект	Набор проверяемых флагов
sete	<i>D</i> setz	$D \leftarrow ZF$	Равно / ноль
setne	<i>D</i> setnz	$D \leftarrow \sim ZF$	Не равно / не ноль
sets	<i>D</i>	$D \leftarrow SF$	Отрицательное
setns	<i>D</i>	$D \leftarrow \sim SF$	Неотрицательное
setg	<i>D</i> setnle	$D \leftarrow \sim (SF \wedge OF) \ \& \ \sim ZF$	Больше (сравнение со знаком >)
setge	<i>D</i> setnl	$D \leftarrow \sim (SF \wedge OF)$	Больше или равно (сравнение со знаком >=)
setl	<i>D</i> setnge	$D \leftarrow SF \wedge OF$	Меньше (сравнение со знаком <)
setle	<i>D</i> setng	$D \leftarrow (SF \wedge OF) \mid ZF$	Меньше или равно (сравнение со знаком <=)
seta	<i>D</i> setnbe	$D \leftarrow \sim CF \ \& \ \sim ZF$	Выше (сравнение без знака >)
setae	<i>D</i> setnb	$D \leftarrow \sim CF$	Выше или равно (сравнение без знака >=)
setb	<i>D</i> setnae	$D \leftarrow CF$	Ниже (сравнение без знака <)
setbe	<i>D</i> setna	$D \leftarrow CF \mid ZF$	Ниже или равно (сравнение без знака <=)

Инструкция SET принимает в качестве операнда-приемника либо младшую однобайтную часть регистра (рис. 3.1), либо однобайтную ячейку памяти и записывает в него значение 0 или 1. Чтобы получить 32- или 64-разрядный результат, нужно очистить старшие биты. Вот как выглядит типичная последовательность инструкций для вычисления выражения  $a < b$  на языке C, где обе переменные,  $a$  и  $b$ , имеют тип `long`:

```
int comp(data_t a, data_t b)
a в %rdi, b в %rsi
1  comp:
2      cmpq   %rsi, %rdi      Сравнить a:b
3      setl   %al             Записать 0 или 1 в младший байт %eax
4      movzbl %al, %eax       Очистить остальные байты в %eax (и в %rax)
5      ret
```

Обратите внимание на порядок операндов в инструкции `cmpq` (строка 2). Несмотря на то что операнды перечислены в порядке `%rsi` ( $b$ ), затем `%rdi` ( $a$ ), в действительности сравнение выполняется между  $a$  и  $b$  (то есть виртуально вычитет  $a$  из  $b$ ). Напомним также, что, как отмечалось в разделе 3.4.2, инструкция `movzbl` (строка 4) очищает не только старшие 3 байта в `%eax`, но также старшие 4 байта во всем регистре `%rax`.

Некоторые базовые машинные инструкции могут иметь несколько возможных имен, которые мы называем «синонимами». Например, имена `setg` («установить, если больше») и `setnle` («установить, если не меньше или равно») относятся к одной и той же машинной инструкции. Компиляторы и дизассемблеры по собственному усмотрению выбирают, какие имена использовать.

Несмотря на то что все арифметические операции устанавливают флаги условий, описание различных инструкций SET относится к случаю, когда выполняется инструкция сравнения, устанавливая флаги в соответствии с результатом выражения  $t = a - b$ . В частности, пусть  $a$ ,  $b$  и  $t$  – целые числа в дополнительном коде, хранящиеся в переменных  $a$ ,  $b$  и  $t$  соответственно, тогда  $t = a - {}^w_t b$ , где  $w$  зависит от размеров  $a$  и  $b$ .

Рассмотрим инструкцию `sete` («установить, если равно»). Если  $a = b$ , то мы получим  $t = 0$  и, соответственно, будет установлен флаг нуля. Аналогично рассмотрим проверку сравнения чисел со знаком, выполняемую инструкцией `setl` («установить, если мень-

ше»). Если переполнения не возникает (флаг OF сброшен в 0), то мы имеем  $a < b$ , когда  $a - {}^t_w b < 0$  (флаг SF установлен в 1), и  $a \geq b$ , когда  $a - {}^t_w b \geq 0$  (флаг SF сброшен в 0). С другой стороны, когда возникает переполнение, мы имеем  $a < b$ , когда  $a - {}^t_w b > 0$  (отрицательное переполнение) и  $a > b$ , когда  $a - {}^t_w b < 0$  (положительное переполнение). Переполнения не может быть, когда  $a = b$ . Таким образом, когда OF установлен в 1, условие  $a < b$  будет выполняться только тогда, когда SF установлен в 0. Соответственно, операция ИСКЛЮЧАЮЩЕЕ ИЛИ с флагами переполнения и знака позволяет проверить выполнение условия  $a < b$ . Другие операции сравнения чисел со знаком основаны на других комбинациях SF ^ OF и ZF.

Теперь перейдем к сравнению целых без знака; пусть  $a$  и  $b$  – целые числа без знака, хранящиеся в переменных  $a$  и  $b$ . При вычислении  $t = a - b$  флаг переноса будет установлен инструкцией CMP, когда  $a - b < 0$ , и поэтому операции сравнения целых без знака используют комбинацию флагов переноса и нуля.

Важно отметить, как машинный код различает или не различает значения со знаком и без знака. В отличие от C, он не знает тип каждого значения. И практически всегда использует одни и те же инструкции для обоих случаев, потому что многие арифметические операции имеют одинаковое поведение на уровне битов и для чисел без знака, и для чисел в дополнительном коде. В некоторых случаях требуются разные инструкции для выполнения операций с числами со знаком и без знака, например разные версии сдвига вправо, инструкции деления и умножения и, соответственно, проверки разных комбинаций флагов условий.

### Упражнение 3.13 (решение в конце главы)

Код на C

```
int comp(data_t a, data_t b) {
    return a COMP b;
}
```

содержит обобщенную операцию сравнения аргументов  $a$  и  $b$ , где  $\text{data\_t}$  – это тип данных аргументов, определяется (через typedef) как один из целочисленных типов данных, перечисленных в табл. 3.1. Сравнение COMP определяется через #define.

Предположим, что  $a$  находится в некоторой части  $\%rdi$ ,  $b$  – в некоторой части  $\%rsi$ . Для каждой из следующих последовательностей инструкций определите, какие типы данных  $\text{data\_t}$  и какие сравнения COMP могут заставить компилятор сгенерировать такой код. (Правильных ответов может быть несколько; вы должны перечислить их все.)

```
1.cmpl %esi, %edi
   setl %al
2.cmpw %si, %di
   setge %al
3.cmpb %sil, %dil
   setbe %al
4.cmpq %rsi, %rdi
   setne %al
```

### Упражнение 3.14 (решение в конце главы)

Код на C

```
int test(data_t a) {
    return a TEST 0;
}
```

содержит обобщенную операцию сравнения аргумента `a` со значением 0, и мы можем задать тип данных аргумента, объявив `data_t` с помощью `typedef`, и характер сравнения, объявив `TEST` с помощью `#define`. Следующие последовательности инструкций реализуют сравнение, где `a` хранится в некоторой части регистра `%rdi`. Для каждой последовательности определите, какой тип данных `data_t` и какое сравнение `TEST` могут заставить компилятор сгенерировать такой код. (Правильных ответов может быть несколько; вы должны перечислить их все.)

```
1.testq %rdi, %rdi
   setge %al
2.testw %di, %di
   sete %al
3.testb %dil, %dil
   seta %al
4.testl %edi, %edi
   setle %al
```

### 3.6.3. Инструкции перехода

В обычных условиях инструкции выполняются одна за другой в том порядке, в каком они записаны. Команда *перехода* может передать управление в совершенно новую позицию в программе. Места, куда передается управление, в ассемблерном коде обычно обозначены *метками*. Рассмотрим следующую последовательность инструкций на языке ассемблера:

<code>movq \$0,%rax</code>	Записать 0 в <code>%rax</code>
<code>jmp .L1</code>	Перейти к метке <code>.L1</code>
<code>movq (%rax),%rdx</code>	Разыменование пустого указателя (пропускается)
<code>.L1:</code>	
<code>popq %rdx</code>	Переход к требуемой точке

Инструкция `jmp .L1` заставляет программу миновать инструкцию `movq` и продолжить выполнение с инструкции `popq`. В процессе компиляции файла ассемблер определяет адреса всех инструкций с метками и подставляет их в соответствующие инструкции перехода.

В табл. 3.11 перечислены различные инструкции перехода. Инструкция `jmp` выполняет безусловный переход. Переход может быть *прямым*, когда адрес перехода указан непосредственно в коде самой программы, или *косвенным*, когда адрес перехода извлекается из какого-нибудь регистра или из ячейки памяти. Прямые переходы записываются на языке ассемблера с указанием метки, обозначающей адреса перехода, такой как `.L1` в листинге выше. Косвенные переходы записываются со звездочками (\*), за которыми следует описатель операнда в формате, как показано в табл. 3.2. Например, инструкция

```
jmp %rax
```

использует значение, хранящееся в регистре `%rax` как адрес перехода, а инструкция

```
jmp *(%rax)
```

извлекает адрес перехода из ячейки памяти, на которую ссылается регистр `%rax`.

Другие инструкции перехода в табл. 3.11 – *условные*. Они либо выполняют переход, либо продолжают выполнение со следующей за ними инструкции в зависимости от состояния флагов. Имена этих инструкций и условия, при которых выполняется переход, соответствуют именам и условиям инструкций SET. Подобно инструкциям SET, некоторые инструкции перехода имеют несколько имен. Условные переходы могут быть только прямыми.

**Таблица 3.11.** Инструкции перехода. Эти инструкции выполняют переход в позицию, отмеченную меткой, когда выполняется определенное условие. Некоторые инструкции имеют «синонимы», альтернативные имена одной и той же машинной инструкции

Инструкция	Синоним	Условие перехода	Описание
jmp <i>Label</i>		1	Прямой безусловный переход
jmp <i>*Операнд</i>		1	Косвенный безусловный переход
je <i>Метка</i>	jz	ZF	Равно / ноль
jne <i>Метка</i>	jnz	~ZF	Не равно / не ноль
js <i>Метка</i>		SF	Отрицательное
jns <i>Метка</i>		~SF	Неотрицательное
jg <i>Метка</i>	jnl	~SF ^ OF) & ~ZF	Больше (сравнение со знаком >)
jge <i>Метка</i>	jnl	~(SF ^ OF)	Больше или равно (сравнение со знаком >=)
jl <i>Метка</i>	jnge	SF ^ OF	Меньше (сравнение со знаком <)
jle <i>Метка</i>	jng	(SF ^ OF)   ZF	Меньше или равно (сравнение со знаком <=)
ja <i>Метка</i>	jnb	~CF & ~ZF	Выше (сравнение без знака >)
jae <i>Метка</i>	jnb	~CF	Выше или равно (сравнение без знака >=)
jb <i>Метка</i>	jnae	CF	Ниже (сравнение без знака <)
jbe <i>Метка</i>	jna	CF   ZF	Ниже или равно (сравнение без знака <=)

### 3.6.4. Кодирование инструкций перехода

В этой книге мы редко будем вдаваться в обсуждение тонкостей формата машинного кода. Но, с другой стороны, понимание особенностей кодирования целевых адресов в инструкциях перехода пригодится нам, когда мы перейдем к изучению процесса связывания в главе 7. Кроме того, это понимание поможет нам интерпретировать вывод дизассемблера. В ассемблерном коде адреса переходов представляются в виде символических меток. Ассемблер, а позже и компоновщик генерируют для них соответствующие целевые адреса. Существует несколько способов обозначения целевых адресов, но чаще других используются *адреса относительно счетчика команд* (PC relative). То есть целевой адрес вычисляется как разность между адресом целевой инструкции (к которой выполняется переход) и адресом инструкции, которая следует непосредственно за инструкцией перехода. Подобного рода смещения можно представить в виде одного, двух или четырех байт. Второй способ заключается в подстановке 4-байтного «абсолютного» адреса, указывающего точно в точку перехода. Ассемблер и компоновщик сами выбирают наиболее подходящий способ представления адресов перехода.

В качестве примера относительной адресации рассмотрим код функции на языке ассемблера, полученный в результате компиляции файла `branch.c`. В нем имеются две инструкции перехода: `jmp` в строке 2 выполняет переход вперед к старшему адресу и `jg` в строке 7 выполняет переход назад, к младшему адресу.

```

1  movq %rdi, %rax
2  jmp .L2
3  .L3:
4  sarq %rax
5  .L2:
6  testq %rax, %rax
7  jg .L3
8  rep; ret
```



Вот дизассемблированная версия файла .o, сгенерированная дизассемблером:

```

1   0: 48 89 f8      mov    %rdi,%rax
2   3: eb 03          jmp    8 <loop+0x8>
3   5: 48 d1 f8      sar    %rax
4   8: 48 85 c0      test   %rax,%rax
5  b: 7f f8          jg     5 <loop+0x5>
6  d: f3 c3      repz  retq

```

В комментариях справа, сгенерированных дизассемблером, целевые адреса перехода обозначены как 0x8 (в инструкции перехода в строке 2) и 0x5 (в инструкции перехода в строке 5). Все числа дизассемблер приводит в шестнадцатеричном виде. Однако, глядя на байтовый код инструкций, можно заметить, что целевой адрес первой инструкции перехода кодируется (вторым байтом) как 0x03. Сложив его с числом 0x5 – адресом инструкции, следующей далее, мы получаем целевой адрес перехода 0x8 – адрес инструкции в строке 4.

### Что делают инструкции `per` и `repz`?

В строке 8 в ассемблерном коде, показанном выше, присутствует комбинация инструкций `per; ret`. В дизассемблированном коде (строка 6) она отображается как `repz retq`. Можно предположить, что `repz` является синонимом инструкции `per`, так же как `retq` является синонимом для `ret`. В описании инструкции `per` в документации Intel и AMD говорится, что она обычно используется для реализации повторяющихся операций [3, 51]. Здесь это кажется совершенно неуместным. Ответ на эту загадку можно увидеть в рекомендациях AMD для разработчиков компиляторов [1]. Они рекомендуют использовать комбинацию `per; ret`, чтобы избежать превращения местоположения инструкции `ret` в целевой адрес для инструкции условного перехода. Без инструкции `per` инструкция `jg` (строка 7 в ассемблерном коде) выполнила бы переход к инструкции `ret` при невыполнении условия. Согласно документации AMD, их процессоры содержат ошибку в блоке предсказания целевого адреса инструкции `ret` для инструкции перехода. Команда `per` служит здесь формой инструкции «нет операции», и добавление ее в качестве пункта назначения для перехода не меняет поведения кода, за исключением того, что выполнение на процессорах AMD ускоряется. Мы можем спокойно игнорировать любые инструкции `per` и `repz`, которые увидим в остальных примерах в этой книге.

Аналогично целевой адрес во второй инструкции перехода представлен однобайтным значением 0xf8 (десятичное число –8) в дополнительном коде. Прибавив его к 0xd (десятичное число 13), адресу инструкции в строке 6, получаем 0x5, адрес инструкции в строке 3.

Как показывают эти примеры, при выполнении относительной адресации за точкой отсчета принимается адрес инструкции, следующей за инструкцией перехода, а не самой инструкции перехода. Это соглашение восходит к ранним реализациям, когда процессор обновлял программный счетчик перед выполнением инструкции.

Ниже показана дизассемблированная версия программы после компоновки:

```

1 4004d0: 48 89 f8      mov    %rdi,%rax
2 4004d3: eb 03          jmp    4004d8 <loop+0x8>
3 4004d5: 48 d1 f8      sar    %rax
4 4004d8: 48 85 c0      test   %rax,%rax
5 4004db: 7f f8          jg     4004d5 <loop+0x5>
6 4004dd: f3 c3      repz  retq

```

Инструкции были перемещены в другой адрес, но целевые адреса в строках 2 и 5 остались прежними. Благодаря поддержке относительной адресации инструкции могут кодироваться более компактно (требуя всего 2 байта), а объектный код может смещаться в памяти без изменений.

### Упражнение 3.15 (решение в конце главы)

В следующих фрагментах двоичного кода, сгенерированного дизассемблером, некоторые данные заменены символами X. Ответьте на следующие вопросы, касающиеся этих инструкций.

Определите адрес, куда выполнит переход инструкция `je`. (Здесь не требуется ничего знать об инструкции `callq`.)

```
4003fa: 74 02          je      XXXXXX
4003fc: ff d0         callq   %rax
```

Определите адрес, куда выполнит переход следующая инструкция `je`.

```
40042f: 74 f4          je      XXXXXX
400431: 5d             pop     %rbp
```

Определите адреса инструкций `ja` и `pop`.

```
XXXXXX: 77 02          ja      400547
XXXXXX: 5d             pop     %rbp
```

В следующем фрагменте используется относительный целевой адрес перехода в 4-байтной форме. Байты располагаются в обратном порядке (little-endian). Определите абсолютный адрес перехода.

```
4005e8: e9 73 ff ff    jmpq   XXXXXXX
4005ed: 90             nop
```

Инструкции перехода дают возможность реализовать условное выполнение (`if`) и различные конструкции циклов.

## 3.6.5. Реализация условного ветвления потока управления

Условные операторы и выражения в языке C реализуются с использованием различных комбинаций условных и безусловных переходов. (Как будет показано в разделе 3.6.6, некоторая условная обработка может быть реализована с использованием передачи данных по условию вместо передачи управления.) Например, в листинге 3.3 (a) показана функция на языке C, которая вычисляет абсолютное значение разности двух чисел<sup>3</sup>. Функция также имеет побочный эффект: она увеличивает один из двух счетчиков, хранящихся в глобальных переменных `lt_cnt` и `ge_cnt`. Компилятор GCC генерирует код на языке ассемблера, представленный в листинге 3.3 (c). Наша попытка переложить машинную версию обратно на язык C (в виде функции `gotodiff_se`) показана в листинге 3.3 (b). Она использует оператор `goto` языка C, похожий на инструкцию безусловного перехода в языке ассемблера. Использование операторов `goto` считается дурным тоном в программировании, потому что они затрудняют чтение и отладку кода. Мы применяем этот оператор только для воссоздания программ на языке C, описывающих управляющую логику программ на языке ассемблера. Мы будем называть такие программы на C «goto-программами».

<sup>3</sup> Фактически функция может вернуть отрицательное значение, если при вычитании произойдет переполнение. Но наша цель здесь другая – продемонстрировать машинный код, а не надежную реализацию.

**Листинг 3.3.** Компиляция условных операторов: (а) процедура `absdiff_se` на С содержит оператор `if-else`, в (с) показан сгенерированный ассемблерный код, а в (b) – процедура `gotodiff_se` на С, воссозданная на основе ассемблерного кода

(а) Оригинальный код на С

```
long lt_cnt = 0;
long ge_cnt = 0;

long absdiff_se(long x, long y)
{
    long result;
    if (x < y) {
        lt_cnt++;
        result = y - x;
    }
    else {
        ge_cnt++;
        result = x - y;
    }
    return result;
}
```

(b) Эквивалентная версия с оператором `goto`

```
1 long gotodiff_se(long x, long y)
2 {
3     long result;
4     if (x >= y)
5         goto x_ge_y;
6     lt_cnt++;
7     result = y - x;
8     return result;
9 x_ge_y:
10    ge_cnt++;
11    result = x - y;
12    return result;
13 }
```

(с) Сгенерированный ассемблерный код

```
long absdiff_se(long x, long y)
x в %rdi, y в %rsi
1 absdiff_se:
2     cmpq    %rsi, %rdi          Сравнить x:y
3     jge     .L2                 Если >=, перейти к x_ge_y
4     addq    $1, lt_cnt(%rip)    lt_cnt++
5     movq    %rsi, %rax
6     subq    %rdi, %rax          результат = y - x
7     ret                                Вернуть
8 .L2:
9     addq    $1, ge_cnt(%rip)    x_ge_y:
                                ge_cnt++
10    movq    %rdi, %rax
11    subq    %rsi, %rax          результат = x - y
12    ret                                Вернуть
```

В `goto`-версии (листинг 3.3 (b)) оператор `goto x_ge_y` в строке 5 выполняет переход к метке `x_ge_y` в строке 9 (если выполняется условие  $x \geq y$ ). Начиная с этой точки выполняются вычисления, которые соответствуют ветке `else` в оригинальной функции `absdiff_se`, и полученный результат возвращается вызывающей программе. Если условие  $x \geq y$  не выполняется, то функция `absdiff_se` выполнит вычисления в ветке `if` и затем точно так же вернет полученный результат.

Реализация на языке ассемблера (листинг 3.3 (с)) сначала сравнивает два операнда (строка 2), устанавливая флаги условий. Если результат сравнения показывает, что  $x$  меньше  $y$ , то выполняется переход к блоку, начинающемуся с адреса 8, который увеличивает глобальную переменную `ge_cnt`, вычисляет разность  $y-x$  и возвращает полученный результат. В противном случае продолжается выполнение кода, начиная со строки 4, который увеличивает глобальную переменную `lt_cnt`, вычисляет разность  $y-x$  и возвращает полученный результат. Как видите, поток управления в ассемблерном коде, сгенерированном при компиляции функции `absdiff_se`, очень похож на `goto`-версию `gotodiff_se`.

### Описание машинного кода на языке C

В листинге 3.3 показан пример трансляции управляющих конструкций языка C в машинный код. Листинг содержит пример функции на C (a) и версию ассемблерного кода с комментариями, сгенерированного компилятором GCC (c). В нем также показана версия на C, которая воспроизводит структуру кода на ассемблере (b). Эти версии были созданы в порядке (a), (c) и (b), однако мы советуем исследовать их в порядке (a), (b) и (c). То есть представление машинного кода на языке C поможет вам понять ключевые моменты и сам ассемблерный код.

Вот как выглядит синтаксис оператора `if-else` в языке C в общем виде:

```
if (условное-выражение)
    инструкция-then
else
    инструкция-else
```

где *условное-выражение* – это целочисленное выражение, возвращающее 0 (интерпретируется как «ложь») или ненулевое значение (интерпретируется как «истина»). Выполняется только одна из веток (*инструкция-then* или *инструкция-else*).

Реализация этой общей формы на языке ассемблера придерживается следующей формы, выраженной на языке C, для описания управляющей логики программы:

```
t = условие-выражение;
if (!t)
    goto false;
инструкция-then
goto done;
false:
инструкция-else
done;
```

То есть компилятор генерирует отдельные блоки кода для *инструкции-else* и *инструкции-then*. Он вставляет условный и безусловный переходы, чтобы гарантировать выполнение нужного блока.

### Упражнение 3.16 (решение в конце главы)

Для кода на языке C

```
void cond(long a, long *p)
{
    if (p && a > *p)
        *p = a;
}
```

GCC сгенерировал следующий ассемблерный код:

```
void cond(long a, long *p)
a в %rdi, p в %rsi
cond:
testq %rsi, %rsi
je .L1
cmpq %rdi, (%rsi)
jge .L1
movq %rdi, (%rsi)
```

```
.L1:
    rep; ret
```

1. Напишите goto-версию на C, которая выполняет те же вычисления и воспроизводит управляющую логику программы на ассемблере подобно тому, как показано в листинге 3.3 (b). Возможно, вы сочтете полезным снабдить эту версию комментариями, как мы делали это в наших примерах.
2. Объясните, почему код на языке ассемблера содержит два условных перехода, тогда как код на языке C – лишь один оператор if.

### Упражнение 3.17 (решение в конце главы)

Альтернативное правило для трансляции операторов if в goto-код выглядит следующим образом:

```
    t = условное-выражение;
    if (t)
        goto true;
    инструкция-else
    goto done;
true:
    инструкция-then
done:
```

1. Напишите goto-версию функции absdiff\_se, руководствуясь этим альтернативным правилом.
2. Сможете ли вы найти хоть одну причину, почему следовало бы предпочесть то или иное правило?

### Упражнение 3.18 (решение в конце главы)

Для кода на языке C

```
long test(long x, long y, long z) {
    long val = _____;
    if ( _____ ) {
        if ( _____ )
            val = _____;
        else
            val = _____;
    } else if ( _____ )
        val = _____;
    return val;
}
```

GCC сгенерировал следующий ассемблерный код:

```
long test(long x, long y, long z)
x в %rdi, y в %rsi, z в %rdx
test:
    leaq    (%rdi,%rsi), %rax
    addq    %rdx, %rax
    cmpq    $-3, %rdi
    jge     .L2
    cmpq    %rdx, %rsi
    jge     .L3
    movq    %rdi, %rax
```

```

    imulq    %rsi, %rax
    ret
.L3:
    movq     %rsi, %rax
    imulq    %rdx, %rax
    ret
.L2:
    cmpq     $2, %rdi
    jle      .L4
    movq     %rdi, %rax
    imulq    %rdx, %rax
.L4:
    rep; ret

```

Добавьте недостающие выражения в код на C.

### 3.6.6. Реализация условного ветвления потока данных

Передача *управления* по условию – это обычный способ реализации условных операций, когда программа следует по одному пути, когда условие выполняется, и по другому, когда оно не выполняется. Этот механизм прост и универсален, но на современных процессорах он может быть очень неэффективным.

Альтернативная стратегия – условная передача *данных*. В этом подходе вычисляются обе условные операции, а затем из двух результатов выбирается один, в зависимости от выполнения некоторого условия. Эта стратегия имеет смысл только в ограниченных случаях, но зато ее можно реализовать с помощью простой инструкции *условного перемещения*, которая лучше соответствует характеристикам производительности современных процессоров. Здесь мы исследуем эту стратегию и ее реализацию в архитектуре x86-64.

В листинге 3.4 (а) показан пример кода, который может быть скомпилирован с использованием условного перемещения. Функция вычисляет разность абсолютных значений своих аргументов  $x$  и  $y$ , как и в предыдущем примере (листинг 3.3). Однако в предыдущем примере имели место побочные эффекты в ветвях в виде изменения значений `lt_cnt` или `ge_cnt`, а эта версия просто вычисляет значение, которое должна вернуть функция.

**Листинг 3.4.** Компиляция условных операторов с использованием условного присваивания: (а) функция `absdiff` на C содержит условное выражение, в (с) показан сгенерированный ассемблерный код, а в (b) – процедура `cmovdiff` на C, воссозданная на основе ассемблерного кода

(а) Оригинальный код на C

```

long absdiff(long x, long
y)
{
    long result;
    if (x < y)
        result = y - x;
    else
        result = x - y;
    return result;
}

```

(b) Реализация с использованием условного присваивания

```

1 long cmovdiff(long x, long y)
2 {
3     long rval = y-x;
4     long eval = x-y;
5     long ntest = x >= y;
6     /* Строка ниже реализуется
7        одной инструкцией: */
8     if (ntest) rval = eval;
9     return rval;
10 }

```

(с) Сгенерированный ассемблерный код

```

long absdiff(long x, long y)
x в %rdi, y в %rsi
1  absdiff:
2      movq    %rsi, %rax
3      subq    %rdi, %rax      rval = y-x
4      movq    %rdi, %rdx
5      subq    %rsi, %rdx      eval = x-y
6      cmpq    %rsi, %rdi      Сравнить x:y
7      cmovge  %rdx, %rax      Если >=, rval = eval
8      ret          Вернуть rval

```

Для этой функции GCC сгенерировал ассемблерный код, показанный в листинге 3.4 (с), повторяющий приблизительно форму функции `cmovdiff` на языке С, показанной в листинге 3.4 (b). В версии на С мы видим, что она вычисляет оба выражения,  $y-x$  и  $x-y$ , значения которых сохраняются в переменных `rval` и `eval` соответственно. Затем она сравнивает  $x$  и  $y$ , и если значение  $x$  больше и равно значению  $y$ , то копирует `eval` в `rval` перед возвратом `rval`. Ассемблерный код в листинге 3.4 (с) следует той же логике. Ключ в том, что условное присваивание (строка 8) в `cmovdiff` реализует единственная инструкция `cmovge` (строка 7) в ассемблерном коде. Она копирует содержимое регистра-источника в регистр-получатель, только если инструкция `cmpq` в строке 6 сообщает, что одно значение больше или равно другому (как указывает суффикс `ge`).

Чтобы понять, почему код, основанный на условной передаче данных, может преуспеть по эффективности код, основанный на условной передаче управления (как в листинге 3.3), нужно изучить особенности работы современных процессоров. Как будет показано в главах 4 и 5, процессоры достигают высокой производительности за счет *конвейерной обработки*, когда обработка инструкции производится в несколько этапов, каждый из которых выполняет одну небольшую часть требуемых операций (например, выборку инструкции из памяти, определение типа инструкции, чтение данных из памяти, выполнение арифметической операции, запись результата в память и обновление программного счетчика). Этот подход обеспечивает высокую производительность за счет одновременного выполнения разных этапов обработки последовательности инструкций, например одновременно с этапом вычисления арифметической операции в одной инструкции может выполняться этап выборки следующей инструкции. Для этого процессор должен иметь возможность прогнозировать последовательность выполнения инструкций, чтобы конвейер был заполнен инструкциями, подлежащими выполнению. Обнаружив условный переход (называемый «ветвлением»), процессор не может определить, в каком направлении пойдет выполнение, пока не оценит условие ветвления. Процессоры используют сложную *логику предсказания переходов*, чтобы попытаться угадать, будет ли выполнен условный переход. Пока ему удастся угадывать (современные микропроцессоры добиваются успеха примерно в 90 % случаев), конвейер команд будет заполнен инструкциями. С другой стороны, в случае ошибки предсказания перехода процессору приходится отбросить большую часть выполненной работы по обработке последующих инструкций и начинать заполнять конвейер инструкциями с правильного места. Как мы увидим далее, такие ошибки предсказания могут повлечь за собой серьезный штраф, скажем 15–30 тактов, потраченных впустую, что влечет серьезное снижение производительности программы.

Для эксперимента мы измерили производительность двух версий функции `absdiff` с разными реализациями условного ветвления на процессоре Intel Haswell. В типичном приложении результат проверки  $x < y$  крайне непредсказуем, поэтому даже самый

сложный алгоритм прогнозирования ветвлений будет правильно угадывать только в 50 % случаев. Кроме того, вычисления, выполняемые в каждой из двух ветвей, требуют только одного такта. Как следствие штраф за неправильное предсказание составляет доминирующую величину в производительности этой функции. Мы выяснили, что для выполнения версии функции с условными переходами требуется около 8 тактов, когда ветвление легко предсказуемо, и около 17,50 тактового цикла, когда ветвление является случайным. Из этого можно сделать вывод, что штраф за неверное предсказание перехода составляет около 19 тактов, то есть для выполнения функции требуется от 8 до 27 тактов, в зависимости от правильности прогнозирования ветвления.

С другой стороны, для выполнения версии с условным перемещением требуется около 8 тактов, независимо от проверяемых данных. Поток управления не зависит от данных, и это позволяет процессору поддерживать конвейер постоянно заполненным.

#### Как определить величину штрафа?

Предположим, что вероятность неверного предсказания равна  $p$ , время выполнения кода при правильном предсказании равно  $T_{\text{OK}}$  и штраф за ошибочное предсказание равен  $T_{\text{MP}}$ . Тогда среднее время выполнения кода является функцией от  $p$ :

$$T_{\text{avg}}(p) = (1 - p)T_{\text{OK}} + p(T_{\text{OK}} + T_{\text{MP}}) = T_{\text{OK}} + pT_{\text{MP}}$$

Нам даны  $T_{\text{OK}}$  и  $T_{\text{ran}}$ , среднее время, когда  $p = 0,5$ , и нам нужно определить  $T_{\text{MP}}$ . Подставив известные значения в уравнение, получаем

$$T_{\text{ran}} = T_{\text{avg}}(0,5) = T_{\text{OK}} + 0,5T_{\text{MP}} \text{ откуда } T_{\text{MP}} = 2(T_{\text{ran}} - T_{\text{OK}}).$$

То есть для  $T_{\text{OK}} = 8$  и  $T_{\text{ran}} = 17,5$  мы получаем  $T_{\text{MP}} = 19$ .

#### Упражнение 3.19 (решение в конце главы)

Для выполнения на более старой модели процессора нашему коду требовалось около 16 тактов, когда ветвления были легко предсказуемы, и около 31 такта, когда ветвления происходили случайно.

1. Вычислите примерный штраф за ошибку предсказания.
2. Сколько тактов потребуется для выполнения функции в случае ошибки прогнозирования ветвления?

В табл. 3.12 показаны некоторые инструкции условного перемещения, доступные в архитектуре x86-64. Каждая имеет два операнда: операнд-источник  $S$  (регистр или ячейка памяти) и операнд-приемник  $R$  (только регистр). По аналогии с инструкциями SET (раздел 3.6.2) и переходов (раздел 3.6.3), результат выполнения этих инструкций зависит от значений флагов состояния. Исходное значение читается из памяти или из регистра, но копируется в приемник, только если выполняется указанное условие.

Источник и приемник могут иметь размер 16, 32 или 64 бита. Однобайтные условные перемещения не поддерживаются. В отличие от безусловных инструкций, где длина операнда явно закодирована в имени инструкции (например, movb и movl), ассемблер может определять длину операнда инструкции условного перемещения по имени регистра-приемника, благодаря чему одно и то же имя инструкции может использоваться с операндами всех поддерживаемых размеров.

В отличие от условных переходов, процессор может выполнять инструкции условного перемещения без предсказания результата проверки. Он просто читает исходное значение (возможно, из памяти), проверяет флаг условия, а затем либо обновляет



регистр-приемник, либо оставляет его как есть. Мы рассмотрим реализацию условных перемещений в главе 4.

**Таблица 3.12.** Инструкции условного перемещения. Эти инструкции копируют содержимое источника  $S$  в приемник  $R$ , когда выполняется определенное условие. Некоторые инструкции имеют «синонимы», альтернативные имена одной и той же машинной инструкции

Инструкция	Синоним	Условие перемещения	Описание	
cmovz	$S, R$	ZF	Равно / ноль	
cmovnz	$S, R$	$\sim$ ZF	Не равно / не ноль	
cmovs	$S, R$	SF	Отрицательное	
cmovns	$S, R$	$\sim$ SF	Неотрицательное	
cmovg	$S, R$	cmovnl	$\sim(SF \wedge OF) \ \& \ \sim ZF$	Больше (сравнение со знаком >)
cmovge	$S, R$	cmovnl	$\sim(SF \wedge OF)$	Больше или равно (сравнение со знаком >=)
cmovl	$S, R$	cmovnge	SF $\wedge$ OF	Меньше (сравнение со знаком <)
cmovle	$S, R$	cmovng	(SF $\wedge$ OF)   ZF	Меньше или равно (сравнение со знаком <=)
cmova	$S, R$	cmovnbe	$\sim$ CF & $\sim$ ZF	Выше (сравнение без знака >)
cmovae	$S, R$	cmovnb	$\sim$ CF	Выше или равно (сравнение без знака >)
cmovb	$S, R$	cmovnae	CF	Ниже (сравнение без знака <)
cmovbe	$S, R$	cmovna	CF   ZF	Ниже или равно (сравнение без знака <=)

Чтобы понять, как условные операции могут быть реализованы с использованием условной передачи данных, рассмотрим следующую общую форму условного выражения и присваивания:

$v = \text{условное-выражение} ? \text{выражение-then} : \text{выражение-else};$

Стандартный способ компиляции этого выражения с использованием условной передачи управления будет иметь следующую форму:

```
if (!условное-выражение)
    goto false;
v = выражение-then;
goto done;
false:
    v = выражение-else;
done:
```

Этот код содержит две последовательности кода: одна вычисляет *выражение-then*, а другая – *выражение-else*. Для вычисления только одной из последовательностей используется комбинация из условного и безусловного переходов.

Код, использующий условное перемещение, вычисляет оба выражения, *выражение-then* и *выражение-else*, причем окончательный результат выбирается на основе оценки *условного-выражения*. Это можно описать следующим абстрактным кодом:

```
v = выражение-then;
ve = выражение-else;
t = условное-выражение;
if (!t) v = ve;
```

Последний оператор в этой последовательности реализуется условным перемещением – значение *ve* копируется в *v*, только если не выполняется условие *t*.

Не все условные выражения можно скомпилировать с использованием условного перемещения. Что особенно важно, абстрактный код, который мы показали выше, вычисляет не только *выражение-then*, но также *выражение-else*, независимо от результата проверки. Если одно из этих двух выражений может вызвать ошибку или имеет побочный эффект, это может привести к недопустимому поведению. Так обстоит дело с нашим предыдущим примером (листинг 3.3). На самом деле мы специально добавили побочные эффекты в этот пример, чтобы заставить GCC реализовать эту функцию с использованием условных переходов.

В качестве еще одной иллюстрации рассмотрим следующую функцию на C:

```
long cread(long *xp) {
    return (xp ? *xp : 0);
}
```

На первый взгляд она кажется хорошим кандидатом для компиляции с использованием условного перемещения, чтобы вернуть 0, когда указатель имеет значение NULL, как показано в следующем ассемблерном коде:

```
long cread(long *xp)
Неправильная реализация функции cread
xp в регистре %rdi
1  cread:
2      movq    (%rdi), %rax    v = *xp
3      testq   %rdi, %rdi     Проверить xp
4      movl    $0, %edx       Установить ve = 0
5      cmovle  %rdx, %rax     Если xp==0, то v = ve
6      ret                                Вернуть v
```

Однако эта реализация неверна, потому что разыменование `xp` инструкцией `movq` (строка 2) происходит, даже если условие не выполняется, вызывая ошибку разыменования пустого указателя. Вместо этого данный код должен компилироваться с использованием ветвления управления.

Кроме того, использование условного перемещения не всегда улучшает эффективность кода. Например, если оценка *выражения-then* или *выражения-else* требует значительных вычислений, тогда это время будет потрачено впустую, если соответствующее условие не выполнится. Компиляторы должны учитывать затраты на вычисление выражений и оценивать потенциальное снижение производительности из-за неправильного предсказания перехода. По правде говоря, у компиляторов действительно не так много информации, чтобы принять надежное решение; например, они не знают, насколько хорошо ветвление будет следовать предсказаниям. Наши эксперименты с GCC показывают, что он использует условные перемещения, только когда два выражения можно вычислить легко и быстро, например с использованием одной инструкции сложения. Опираясь на свой опыт, мы можем сказать, что GCC часто использует условные переходы, даже когда стоимость неверного предсказания перехода превышает стоимость более сложных вычислений.

В заключение можно сказать, что условная передача данных предлагает альтернативную стратегию реализации условных операций. Она применима в ограниченных случаях, но эти случаи довольно распространены и прекрасно подходят для современных процессоров.

### Упражнение 3.20 (решение в конце главы)

В следующей функции на C мы опустили определение операции `OP`:

```
#define OP _____ /* Неизвестный оператор */
```

```
long arith(long x) {
    return x OP 8;
}
```

При компиляции GCC сгенерировал следующий ассемблерный код:

```
long arith(long x)
x в %rdi
arith:
    leaq    7(%rdi), %rax
    testq   %rdi, %rdi
    cmovns  %rdi, %rax
    sarq    $3, %rax
    ret
```

1. Определите операцию OP.
2. Добавьте в код комментарии, поясняющие его работу.

### Упражнение 3.21 (решение в конце главы)

Для следующей функции на C

```
long test(long x, long y) {
    long val = _____;
    if ( _____ ) {
        if ( _____ )
            val = _____;
        else
            val = _____;
    } else if ( _____ )
        val = _____;
    return val;
}
```

GCC сгенерировал такой ассемблерный код:

```
long test(long x, long y)
x в %rdi, y в %rsi
test:
    leaq    0(,%rdi,8), %rax
    testq   %rsi, %rsi
    jle     .L2
    movq    %rsi, %rax
    subq    %rdi, %rax
    movq    %rdi, %rdx
    andq    %rsi, %rdx
    cmpq    %rsi, %rdi
    cmovge  %rdx, %rax
    ret
.L2:
    addq    %rsi, %rdi
    cmpq    $-2, %rsi
    cmovle  %rdi, %rax
    ret
```

Добавьте недостающие выражения в код на C.

### 3.6.7. Циклы

Язык C предлагает несколько конструкций циклов: `do-while`, `while` и `for`. Соответствующих им конструкций в языке ассемблера нет. Вместо них для реализации циклов используются различные комбинации проверок условий и переходов. GCC и другие компиляторы генерируют код циклов, опираясь на два основных типа циклов. Мы охватим эти два типа в процессе изучения трансляции циклов, начав с цикла `do-while` и двигаясь к более сложным циклам с более сложной реализацией.

#### Циклы `do-while`

Общая форма цикла `do-while` имеет следующий вид:

```
do
    тело-цикла
while (условное-выражение);
```

Назначение цикла состоит в том, чтобы многократно выполнять *тело-цикла*, вычислять значение *условного-выражения* и продолжать цикл, если результат *условного-выражения* не равен нулю. Обратите внимание, что *тело-цикла* выполняется по меньшей мере один раз.

Эту общую форму можно преобразовать в условные выражения и операторы `goto` следующим образом:

```
loop:
    тело-цикла
    t = условное-выражение;
    if (t)
        goto loop;
```

То есть в каждой итерации программа вычисляет *тело-цикла*, а затем *условное-выражение*. Если проверка завершается успешно, то программа переходит к следующей итерации.

**Листинг 3.5.** Код версии `do-while` программы вычисления факториала. Условный переход обеспечивает циклическое выполнение программы

(a) Оригинальный код на C

```
long fact_do(long n)
{
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

(b) Эквивалентная `goto`-версия

```
long fact_do_goto(long n)
{
    long result = 1;
loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Соответствующий ассемблерный код

```
long fact_do(long n)
n в %rdi
1 fact_do:
2 movl    $1, %eax      Установить result = 1
3 .L2:
4 imulq   %rdi, %rax     Вычислить result *= n
5 subq    $1, %rdi       Уменьшить n
6 cmpq    $1, %rdi       Сравнить n:1
```

7	jg        .L2	Если >, перейти к loop
8	ret; ret	Вернуть результат

Например, в листинге 3.5 показана реализация процедуры для вычисления факториала аргумента (записывается как  $n!$ ) с использованием цикла `do-while`. Эта функция верно вычисляет результат только для  $n > 0$ :

### Упражнение 3.22 (решение в конце главы)

1. Определите максимальное значение  $n$ , для которого можно вычислить  $n!$  с 32-разрядным типом `int`.
2. То же самое для 64-разрядного типа.

`goto`-версия в листинге 3.5 (b) показывает, как цикл превращается в низкоуровневую комбинацию проверок и условных переходов. После инициализации результата `result` программа начинает цикл. Сначала она выполняет тело цикла, включающее инструкцию изменения переменных `result` и `n`. Затем проверяет условие  $n > 1$ , и, если условие выполняется, возвращается в начало цикла. В листинге 3.5 (c) показан ассемблерный код, из которого была воссоздана `goto`-версия. Инструкция условного перехода `jg` (строка 7) является ключевой в реализации цикла. Она определяет, продолжать итерацию или выйти из цикла.

### Воссоздание циклов

Ключом к пониманию связи между сгенерированным ассемблерным кодом и оригинальным исходным кодом является поиск соответствия между значениями программы и регистрами. Это было просто сделать для цикла, показанного в листинге 3.5, но для более сложных программ эта задача может оказаться гораздо сложнее. Компилятор C часто меняет порядок вычислений, из-за чего некоторые переменные в коде на C не имеют аналогов в машинном коде, а в машинный код вводятся новые значения, которых нет в исходном коде. Более того, компилятор часто пытается минимизировать использование регистров, сохраняя несколько значений программы в один и тот же регистр.

Процесс, который мы описали на примере `fact_do`, представляет лишь общую стратегию воссоздания циклов. Посмотрите, как регистры инициализируются перед циклом, как изменяются и проверяются внутри цикла и как используются после цикла. Каждая такая операция дает подсказку, которую можно использовать для решения головоломки. Будьте готовы к неожиданным преобразованиям, часть из которых явно относится к случаям оптимизации кода компилятором, но иногда очень трудно объяснить, почему компилятор выбрал именно эту стратегию.

Для воссоздания кода на C по ассемблерному коду, такому как в листинге 3.5 (c), требуется определить, какие регистры используются для хранения тех или иных значений в программе. В данном случае выяснить это довольно просто: мы знаем, что значение  $n$  передается в функцию в регистре `%rdi`. Также мы видим, что регистр `%rax` инициализируется значением 1 (строка 2). (Несмотря на то что инструкция `movl` принимает в роли операнда-приемника регистр `%eax`, она также установит в 0 старшие 4 байта регистра `%rax`.) Далее, в строке 4, в этот регистр записывается результат умножения. Кроме того, поскольку `%rax` используется для возврата значения функции, он часто выбирается для хранения возвращаемых значений. Отсюда можно заключить, что `%rax` хранит значение `result`.

**Упражнение 3.23 (решение в конце главы)**

Для кода на языке C

```
long dw_loop(long x) {
    long y = x*x;
    long *p = &x;
    long n = 2*x;
    do {
        x += y;
        (*p)++;
        n--;
    } while (n > 0);
    return x;
}
```

GCC сгенерировал следующий ассемблерный код:

```
long dw_loop(long x)
x первоначально хранится в %rdi
1  dw_loop:
2      movq    %rdi, %rax
3      movq    %rdi, %rcx
4      imulq   %rdi, %rcx
5      leaq    (%rdi,%rdi), %rdx
6      .L2:
7      leaq    1(%rcx,%rax), %rax
8      subq    $1, %rdx
9      testq   %rdx, %rdx
10     jg      .L2
11     rep; ret
```

1. Какие регистры используются для хранения значений переменных x, y и n?
2. Как компилятор избавился от необходимости в переменной-указателе p и от разыменования указателя, подразумеваемого выражением (\*p)++?
3. Добавьте комментарии в ассемблерный код, описывающие работу программы, как это сделано в листинге 3.5 (с).

## Циклы while

В общем случае синтаксис цикла while имеет следующий вид:

```
while (условное-выражение)
    тело-цикла
```

Он отличается от цикла do-while тем, что сначала вычисляется *условное-выражение*, определяющее условие продолжения цикла, то есть существует возможность прекращения цикла до выполнения *тела-цикла*. Есть несколько способов преобразовать цикл while в машинный код, два из которых используются компилятором GCC. Оба основаны на том же шаблоне, что применяется для трансляции циклов do-while, но отличаются реализацией начальной проверки.

Первый способ, который мы называем *переходом в середину*, выполняет начальную проверку, производя безусловный переход к проверке в конце цикла. Это можно выразить в виде следующего шаблона в форме goto-кода:

```
goto test;
loop:
    тело-цикла
```

```
test:
    t = условное-выражение;
    if (t)
        goto loop;
```

Для примера в листинге 3.6 (а) показана реализация функции вычисления факториала с использованием цикла `while`. Эта функция правильно вычисляет  $0! = 1$ . Соседняя с ней функция `fact_while_jm_goto` (листинг 3.6 (b)) – это `goto`-версия на языке C, воссозданная из ассемблерного кода, сгенерированного компилятором GCC с параметром оптимизации `-Og`. Сравнив `goto`-версию `fact_while_jm_goto` с `goto`-версией `fact_do_goto` (листинг 3.5 (b)), легко заметить, насколько они похожи, отличаясь только наличием оператора `goto test` в первой из них перед циклом, который заставляет программу сначала выполнить проверку `n` перед изменением значения `result` или `n`. В листинге 3.6 (c) показан фактически сгенерированный ассемблерный код.

**Листинг 3.6.** Код на C и на ассемблере `while`-версии функции вычисления факториала с использованием перехода к проверке условия перед началом цикла. Функция на C `fact_while_jm_goto` иллюстрирует работу ассемблерного кода

(а) Оригинальный код на C

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

(b) Эквивалентная `goto`-версия

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
loop:
    result *= n;
    n = n-1;
test:
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Соответствующий ассемблерный код

```
long fact_while(long n)
n в %rdi
fact_while:
    movl    $1, %eax           Установить result = 1
    jmp     .L5                перейти к test
.L6:
    imulq   %rdi, %rax         Вычислить result *= n
    subq    $1, %rdi           Уменьшить n
.L5:
    cmpq    $1, %rdi           Сравнить n:1
    jg      .L6                Если >, перейти к loop
    rep; ret                   Вернуть результат
```

### Упражнение 3.24 (решение в конце главы)

Для следующей программы на C

```
long loop_while(long a, long b)
{
    long result = _____;
    while ( _____ ) {
```

```

        result = _____;
        a = _____;
    }
    return result;
}

```

компилятор GCC, запущенный с параметром `-Og`, сгенерировал такой ассемблерный код:

```

long loop_while(long a, long b)
a в %rdi, b в %rsi
1 loop_while:
2  movl    $1, %eax
3  jmp     .L2
4  .L3:
5  leaq    (%rdi,%rsi), %rdx
6  imulq   %rdx, %rax
7  addq    $1, %rdi
8  .L2:
9  cmpq    %rsi, %rdi
10 jl     .L3
11 rep; ret

```

Как видите, компилятор использовал способ перехода к середине (инструкция `jmp` в строке 3), чтобы перейти к проверке условия, начинающейся с метки `.L2`. Заполните недостающие фрагменты в коде на C.

Второй способ, который мы называем *защищенным-do*, заключается в преобразовании кода в цикл `do-while` и использовании условного ветвления, чтобы пропустить тело цикла, если начальная проверка завершилась неудачей. GCC следует этой стратегии при компиляции с более высокими уровнями оптимизации, например с параметром командной строки `-O1`. Этот способ можно выразить в общей форме в виде следующего шаблона преобразования цикла `while` в цикл `do-while`:

```

t = условное-выражение;
if (!t)
    goto done;
do
    тело-цикла
while (условное-выражение);
done:

```

Вот как можно представить этот шаблон в виде `goto`-кода:

```

t = условное-выражение;
if (!t)
    goto done;
loop:
    тело-цикла
    t = условное-выражение;
    if (t)
        goto loop;
done:

```

Используя эту стратегию, компилятор может оптимизировать начальную проверку, например определив, что условие всегда будет выполняться.

В качестве примера в листинге 3.7 показан код на C той же функции вычисления факториала, что и в листинге на рис. 3.6, но теперь демонстрируется результат компиляции с параметром `-O1`. В листинге 3.7 (с) показан фактически сгенерированный ассемблерный код, а в листинге 3.7 (b) воссозданная на его основе `goto`-версия на языке C. Судя



по этой goto-версии, цикл будет пропущен, если начальное значение  $n \leq 1$ . Сам цикл имеет ту же структуру, что и цикл, созданный для do-while-версии (листинг 3.6). Однако обратите внимание на одну интересную особенность: проверка условия продолжения цикла (строка 9 в ассемблерном коде) изменилась с  $n > 1$  в оригинальном коде на C на  $n \neq 1$ . Компилятор определил, что цикл может начаться только при условии  $n > 1$  и что в результате уменьшения  $n$  обязательно достигнет значения 1, прежде чем получит значение меньше 1. Следовательно, проверка  $n \neq 1$  будет эквивалентна проверке  $n < 1$ .

**Листинг 3.7.** Код на C и на ассемблере while-версии функции вычисления факториала с использованием способа трансляции «защищенный-do». Функция на C `fact_while_gd_goto` иллюстрирует работу ассемблерного кода

(a) Оригинальный код на C

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

(b) Эквивалентная goto-версия

```
long fact_while_gd_goto(long n)
{
    long result = 1;
    if (n <= 1)
        goto done;
loop:
    result *= n;
    n = n-1;
    if (n != 1)
        goto loop;
done:
    return result;
}
```

(c) Соответствующий ассемблерный код

```
long fact_while(long n)
n в %rdi
1 fact_while:
2   cmpq   $1, %rdi      Сравнить n:1
3   jle    .L7           Если <=, перейти к done
4   movl   $1, %eax      Установить result = 1
5   .L6:
6   imulq  %rdi, %rax     Вычислить result *= n
7   subq   $1, %rdi      Уменьшить n
8   cmpq   $1, %rdi      Сравнить n:1
9   jne    .L6           Если !=, перейти к loop
10  rep; ret             Вернуть результат
11  .L7:
12  movl   $1, %eax      done:
13  ret                 Вычислить result = 1
```

### Упражнение 3.25 (решение в конце главы)

Для следующей программы на C

```
long loop_while2(long a, long b)
{
    long result = ____;
    while ( ____ ) {
        result = ____;
        b = ____;
    }
}
```

```
    return result;
}
```

компилятор GCC, запущенный с параметром -O1, сгенерировал такой ассемблерный код:

```
a в %rdi, b в %rsi
1 loop_while2:
2  testq    %rsi, %rsi
3  jle      .L8
4  movq     %rsi, %rax
5  .L7:
6  imulq    %rdi, %rax
7  subq     %rdi, %rsi
8  testq    %rsi, %rsi
9  jg       .L7
10 rep; ret
11 .L8:
12 movq     %rsi, %rax
13 ret
```

Как видите, компилятор использовал способ защищенный-do, сгенерировав инструкцию `jle` в строке 3, чтобы пропустить цикл, если начальная проверка завершится неудачей. Заполните недостающие фрагменты в коде на C. Обратите внимание, что структура управления в ассемблерном коде не совсем соответствует структуре в goto-коде, полученном путем прямого воссоздания кода на C в соответствии с нашими правилами. В частности, в нем присутствуют две разные инструкции `ret` (строки 10 и 13). Однако вы должны восстановить недостающие фрагменты в коде на C, чтобы он имел поведение, эквивалентное поведению ассемблерного кода.

### Упражнение 3.26 (решение в конце главы)

Имеется функция `fun_a` со следующей структурой:

```
long fun_a(unsigned long x) {
    long val = 0;
    while ( ... ) {
        .
        .
        .
    }
    return ...;
}
```

Компилятор GCC сгенерировал такой ассемблерный код:

```
long fun_a(unsigned long x)
x в %rdi
1 fun_a:
2  movl     $0, %eax
3  jmp      .L5
4  .L6:
5  xorq     %rdi, %rax
6  shrq     %rdi          Сдвиг вправо на одну позицию
7  .L5:
8  testq    %rdi, %rdi
9  jne      .L6
10 andl     $1, %eax
11 ret
```

Воссоздайте версию на С из этого кода, а затем:

1. Определите, какой способ трансляции цикла использовался.
2. Опираясь на ассемблерную версию, восстановите недостающие фрагменты в коде на С.
3. Опишите простым языком, что вычисляет эта функция.

## Циклы for

В общем случае синтаксис цикла `for` имеет следующий вид:

```
for (выражение-инициализации; условное-выражение; выражение-обновления)
    тело-цикла
```

Стандарт языка С устанавливает (с одним исключением, о котором рассказывается в упражнении 3.29), что поведение такого цикла можно описать с помощью цикла `while`, как показано ниже:

```
выражение-инициализации;
while (условное-выражение) {
    тело-цикла
    выражение-обновления;
}
```

То есть сначала программа вычисляет *выражение-инициализации*. Затем входит в цикл, где сначала вычисляет *условное-выражение*, и если условие выполняется, то выполняет *тело-цикла* и потом *выражение-обновления*.

GCC генерирует ассемблерный код, следуя одной из двух стратегий трансляции цикла `while` в зависимости от уровня оптимизации. Сначала рассмотрим `goto`-версию с переходом в середину:

```
    выражение-инициализации;
    goto test;
loop:
    тело-цикла
    выражение-обновления;
test:
    t = условное-выражение;
    if (t)
        goto loop;
```

а вот версия, полученная в результате применения стратегии защищенного-`do`:

```
    выражение-инициализации;
    t = условное-выражение;
    if (!t)
        goto done;
loop:
    тело-цикла
    выражение-обновления;
    t = условное-выражение;
    if (t)
        goto loop;
done:
```

В качестве примера рассмотрим функцию вычисления факториала, использующую цикл `for`:

```
long fact_for(long n)
{
    long i;
```

```

    long result = 1;
    for (i = 2; i <= n; i++)
        result *= i;
    return result;
}

```

Как видите, естественным способом реализации функции вычисления факториала с помощью цикла `for` является последовательное умножение множимого на числа от 2 до  $n$ , поэтому эта функция сильно отличается от той, что мы реализовали с использованием циклов `while` и `do-while`.

Вот как идентифицируются различные компоненты цикла `for` в этом коде:

<i>выражение-инициализации</i>	<code>i = 2</code>
<i>условное-выражение</i>	<code>i &lt;= n</code>
<i>выражение-обновления</i>	<code>i++</code>
<i>тело-цикла</i>	<code>result *= i;</code>

Подставляя эти компоненты в шаблон преобразования цикла `for` в цикл `while`, получаем:

```

long fact_for_while(long n)
{
    long i = 2;
    long result = 1;
    while (i <= n) {
        result *= i;
        i++;
    }
    return result;
}

```

Применяя способ трансляции циклов `while` с переходом в середину, получаем следующую `goto`-версию кода:

```

long fact_for_jm_goto(long n)
{
    long i = 2;
    long result = 1;
    goto test;
loop:
    result *= i;
    i++;
test:
    if (i <= n)
        goto loop;
    return result;
}

```

Как оказывается, ассемблерный код, созданный компилятором GCC с параметром командной строки `-Og`, следует этому шаблону:

<code>long fact_for(long n)</code>	
<code>n в %rdi</code>	
<code>fact_for:</code>	
<code>movl \$1, %eax</code>	Установить result = 1
<code>movl \$2, %edx</code>	Установить i = 2
<code>jmp .L8</code>	перейти к test
<code>.L9:</code>	loop:
<code>imulq %rdx, %rax</code>	Вычислить result *= i
<code>addq \$1, %rdx</code>	Увеличить i

<pre>.L8:   cmpq    %rdi, %rdx   jle     .L9   rep; ret</pre>	<pre>test:   Сравнить i:n   Если &lt;=, перейти к loop   Вернуть результат</pre>
---	--

**Упражнение 3.27 (решение в конце главы)**

Напишите goto-версию функции `fact_for`, сначала преобразовав ее в версию с циклом `while`, а затем применив способ трансляции с защищенным-`do`.

Из вышесказанного следует вывод, что все три формы циклов в языке C – `do-while`, `while` и `for` – можно транслировать с помощью простой стратегии, генерируя код с одним или несколькими условными ветвлениями. Условная передача управления – это основной механизм преобразования циклов в машинный код.

**Упражнение 3.28 (решение в конце главы)**

Имеется функция `fun_b` со следующей структурой:

```
long fun_b(unsigned long x) {
    long val = 0;
    long i;
    for ( ... ; ... ; ... ) {
        ...
    }
    return val;
}
```

Компилятор GCC сгенерировал следующий ассемблерный код:

```
long fun_b(unsigned long x)
x в %rdi
1  fun_b:
2      movl    $64, %edx
3      movl    $0, %eax
4      .L10:
5      movq    %rdi, %rcx
6      andl    $1, %ecx
7      addq    %rax, %rax
8      orq     %rcx, %rax
9      shrq    %rdi                Сдвиг вправо на 1
10     subq    $1, %rdx
11     jne     .L10
12     rep; ret
```

Воссоздайте версию на C из этого кода, а затем:

- 1) опираясь на ассемблерную версию, восстановите недостающие фрагменты в коде на C;
- 2) объясните, почему отсутствует начальная проверка условия перед циклом и начальный переход к проверке в конце цикла;
- 3) опишите простым языком, что вычисляет эта функция.

**Упражнение 3.29 (решение в конце главы)**

Оператор `continue` в языке C заставляет программу перейти в конец текущей итерации цикла. При использовании оператора `continue` правило преобразования цикла `for` в цикл `while` требует некоторой доработки. Например, рассмотрим следующий код на C:

```

/* Пример цикла for с оператором continue */
/* Вычисляет сумму нечетных чисел между 0 и 9 */
long sum = 0;
long i;
for (i = 0; i < 10; i++) {
    if (i & 1)
        continue;
    sum += i;
}

```

1. Что получится, если просто применить правило преобразования цикла for в цикл while, описанное выше? Какую ошибку допустит такой код?
2. Как можно заменить оператор continue оператором goto, чтобы гарантировать правильное преобразование цикла for в цикл while?

### 3.6.8. Оператор switch

Операторы switch обеспечивают возможность множественного ветвления в зависимости от значения целочисленного индекса. Они особенно эффективны при использовании проверок, когда возможно большое количество разных результатов. Они не только делают более читаемым программный код на C, но также позволяют повысить эффективность реализации за счет использования структуры данных, получившей название *таблица переходов*. Таблица переходов – это массив, в котором  $i$ -й элемент содержит адрес кода, реализующего действия, которые программа должна выполнить, когда индекс принимает значение  $i$ . Программа ссылается на таблицу переходов, чтобы определить адрес для инструкции перехода. Преимущество использования таблицы переходов перед длинной последовательностью операторов if-else заключается в том, что время выполнения оператора выбора не зависит от числа переходов. GCC выбирает метод трансляции оператора switch в зависимости от числа вариантов и от размера шага, разделяющего эти варианты. Таблицы переходов используются, когда имеется достаточно большое число случаев (например, четыре и более) и они охватывают небольшой диапазон значений.

В листинге 3.8 (а) представлен пример оператора switch языка C. Этот пример имеет ряд интересных особенностей, включая метки вариантов, которые не охватывают непрерывные диапазоны (здесь отсутствуют метки для вариантов 101 и 105), также есть варианты с несколькими метками (например, 104 и 106) и варианты, которые *проваливаются* (fall through) в другие варианты (например, вариант 102), потому что соответствующий им программный код не заканчивается оператором break.

В листинге 3.9 показана программа на ассемблере, полученная при компиляции функции switch\_eg. Ее поведение иллюстрирует воссозданная версия процедуры switch\_eg\_impl на языке C в листинге 3.8 (b). Код использует расширенную поддержку таблиц переходов в компиляторе GCC, отсутствующую в стандарте C. Массив jt содержит 7 записей, каждая из которых хранит адрес соответствующего блока кода. Эти адреса определяются метками в коде и обозначаются в элементах в jt с помощью префиксов &&. (Напомним, что оператор & создает указатель на значение данных. Для поддержки этого расширения авторы GCC создали новый оператор &&, создающий указатель на блок кода.) Мы рекомендуем изучить процедуру switch\_eg\_impl и провести параллели с ассемблерным кодом.

Оригинальный код на C включает варианты со значениями 100, 102–104 и 106, но переменная-переключатель n может хранить любое целое число. Компилятор сначала сдвигает диапазон в границы от 0 до 6, вычитая 100 из n, тем самым создавая новую переменную, которую мы называли index в воссозданной версии на C. Это помогает упрос-

тить реализацию ветвления, обрабатывая `index` как *значение без знака*, с учетом того, что отрицательные числа в дополнительном коде отображаются в большие положительные числа в представлении без знака. В результате появляется возможность проверить, попадает ли значение `index` в пределы диапазона 0–6, сравнив его с числом 6. В коде на C и ассемблере имеется пять точек, куда можно выполнить переход в зависимости от значения `index`. Это `loc_A` (в ассемблерном коде ей соответствует метка `.L3`), `loc_B` (`.L5`), `loc_C` (`.L6`), `loc_D` (`.L7`) и `loc_def` (`.L8`), последняя из которых представляет вариант по умолчанию. Каждая из этих меток идентифицирует блок кода, реализующий один из вариантов. В обоих языках, C и ассемблере, программа сравнивает переменную `index` с числом 6 и переходит к блоку кода для варианта по умолчанию, если ее значений оказывается больше.

Ключевым этапом выполнения оператора `switch` является доступ к адресу блока кода через таблицу переходов. Это происходит в строке 16 в воссозданной версии на C, где оператор `goto` ссылается на таблицу переходов `jt`. Этот *вычисляемый goto* поддерживается компилятором GCC как расширение языка C. В версии на ассемблере аналогичная операция выполняется в строке 5, где операнд инструкции `jmp` имеет префикс `*`, указывающий на косвенный переход, а сам операнд ссылается на ячейку памяти с индексом в регистре `%rsi`. (В разделе 3.8 мы покажем, как ссылки на массивы транслируются в машинный код.)

**Листинг 3.8.** Пример оператора `switch` и его трансляция на расширенный язык C. Полученная версия иллюстрирует создание таблицы переходов `jt` и порядок ее использования. Такие таблицы поддерживаются компилятором GCC как расширение языка C

(a) Оператор `switch`

```
void switch_eg(long x, long n,
               long *dest)
{
    long val = x;

    switch (n) {

        case 100:
            val *= 13;
            break;

        case 102:
            val += 10;
            /* проваливается далее */

        case 103:
            val += 11;
            break;

        case 104:
        case 106:
            val *= val;
            break;

        default:
            val = 0;
    }
    *dest = val;
}
```

(b) Результат трансляции на расширенный язык C

```
1 void switch_eg_impl(long x, long n,
2                     long *dest)
3 {
4     /* Таблица указателей на блоки кода */
5     static void *jt[7] = {
6         &loc_A, &loc_def, &loc_B,
7         &loc_C, &loc_D, &loc_def,
8         &loc_D
9     };
10    unsigned long index = n - 100;
11    long val;
12
13    if (index > 6)
14        goto loc_def;
15    /* Множественное ветвление */
16    goto *jt[index];
17
18 loc_A: /* Вариант 100 */
19     val = x * 13;
20     goto done;
21 loc_B: /* Вариант 102 */
22     x = x + 10;
23     /* проваливается далее */
24 loc_C: /* Вариант 103 */
25     val = x + 11;
26     goto done;
27 loc_D: /* Варианты 104, 106 */
28     val = x * x;
29     goto done;
```

```

30 loc_def: /* Вариант по умолчанию */
31     val = 0;
32 done:
33     *dest = val;
34 }

```

**Листинг 3.9.** Реализация на ассемблере оператора switch из листинга 3.22

```

void switch_eg(long x, long n, long *dest)
x в %rdi, n в %rsi, dest в %rdx
1  switch_eg:
2      subq    $100, %rsi                Вычислить index = n-100
3      cmpq    $6, %rsi                 Сравнить index:6
4      ja      .L8                       Если >, перейти к loc_def
5      jmp     *.L4(,%rsi,8)             перейти к *jt[index]
6  .L3:                                loc_A:
7      leaq    (%rdi,%rdi,2), %rax       3 * x
8      leaq    (%rdi,%rax,4), %rdi       val = 13 * x
9      jmp     .L2                       перейти к done
10 .L5:                                loc_B:
11     addq    $10, %rdi                 x = x + 10
12 .L6:                                loc_C:
13     addq    $11, %rdi                 val = x + 11
14     jmp     .L2                       перейти к done
15 .L7:                                loc_D:
16     imulq   %rdi, %rdi                 val = x * x
17     jmp     .L2                       перейти к done
18 .L8:                                loc_def:
19     movl    $0, %edi                  val = 0
20 .L2:                                done:
21     movq    %rdi, (%rdx)              *dest = val
22     ret                                  Возврат

```

В нашей goto-версии на C таблица переходов объявляется как массив из семи элементов, каждый из которых является указателем на точку в коде. Эти элементы соответствуют индексам 0–6, которые, в свою очередь, соответствуют значениям n от 100 до 106. Обратите внимание, что повторяющиеся варианты в таблице переходов обрабатываются простым дублированием одной и той же метки (loc\_D в элементах 4 и 6), а отсутствующие варианты – дублированием метки для варианта по умолчанию (loc\_def в элементах 1 и 5).

В ассемблерном коде таблица переходов объявляется следующим образом (мы добавили дополнительные комментарии для ясности):

```

1  .section      .rodata
2  .align 8      Выравнивать адреса по границам, кратным 8
3  .L4:
4  .quad .L3     Вариант 100: loc_A
5  .quad .L8     Вариант 101: loc_def
6  .quad .L5     Вариант 102: loc_B
7  .quad .L6     Вариант 103: loc_C
8  .quad .L7     Вариант 104: loc_D
9  .quad .L8     Вариант 105: loc_def
10 .quad .L7     Вариант 106: loc_D

```

Согласно этим объявлениям, в сегменте с именем .rodata (read-only data – данные только для чтения) должна находиться последовательность из семи четверных (8-байт-



ных) слов, значение каждого из которых определяет адрес указанной метки в ассемблерном коде (например, .L3). Метка .L4 отмечает начало этой последовательности. Ее адрес служит базой для косвенного перехода (строка 5).

Различные блоки кода (в коде на C соответствующие меткам от `loc_A` до `loc_D` и `loc_def`) реализуют разные ветви в операторе `switch`. В большинстве своем они просто вычисляют значение `val` и переходят в конец функции. Точно так же блоки кода на ассемблере вычисляют значение для регистра `%rdi` и переходят к метке .L2. Только блок кода для варианта 102 в исходном коде на C не следует этому шаблону, «проваливаясь» в блок для варианта 103. В коде на ассемблере этот блок начинается с метки .L5 – в нем отсутствует инструкция `jmp`, вследствие чего продолжается выполнение кода из следующего блока. Точно так же в `switch_eg_impl` отсутствует оператор `goto` в конце блока, начинающегося с метки `loc_B`.

Этот код достоин тщательного изучения, но главное, что следует понять, – использование таблицы переходов позволяет очень эффективно реализовать множественное ветвление. В нашем случае программа может перейти к любой из пяти различных точек в коде, обратившись к таблице переходов только один раз. Даже если бы у нас был оператор `switch` с сотнями ветвей, его точно так же можно было бы реализовать с использованием единственного обращения к таблице переходов.

### Упражнение 3.30 (решение в конце главы)

В следующей функции на C мы опустили тело оператора `switch`. В ней метки вариантов не охватывают непрерывный диапазон, а некоторые варианты отмечены несколькими метками.

```
void switch2(long x, long *dest) {
    long val = 0;
    switch (x) {
        .
        .    Тело оператора switch опущено
        .
    }
    *dest = val;
}
```

Для этой функции компилятор GCC сгенерировал ассемблерный код, начальная часть которого показана ниже. Переменная `x` хранится в `%rdi`.

```
void switch2(long x, long *dest)
x в %rdi
1  switch2:
2      addq    $1, %rdi
3      cmpq    $8, %rdi
4      ja      .L2
5      jmp     *.L4(,%rdi,8)
```

Также он сгенерировал следующую таблицу переходов:

```
1  .L4:
2      .quad   .L9
3      .quad   .L5
4      .quad   .L6
5      .quad   .L7
6      .quad   .L2
7      .quad   .L7
8      .quad   .L8
9      .quad   .L2
10     .quad   .L5
```

Опираясь на имеющуюся информацию, ответьте на следующие вопросы:

1. Определите значения меток в теле оператора switch.
2. Какие варианты в коде на C имеют несколько меток?

### Упражнение 3.31 (решение в конце главы)

Для функции switcher со следующей структурой:

```
void switcher(long a, long b, long c, long *dest)
{
    long val;
    switch(a) {
        case ____:          /* Вариант A */
            c = ____;
            /* проваливается далее */
        case ____:          /* Вариант B */
            val = ____;
            break;
        case ____:          /* Вариант C */
        case ____:          /* Вариант D */
            val = ____;
            break;
        case ____:          /* Вариант E */
            val = ____;
            break;
        default:
            val = ____;
    }
    *dest = val;
}
```

компилятор GCC сгенерировал ассемблерный код и таблицу переходов, показанные в листинге 3.10.

Заполните недостающие фрагменты в коде на C. За исключением порядка следования меток C и D, есть только один способ расположить разные варианты в шаблоне.

### Листинг 3.10. Ассемблерный код и таблица переходов для упражнения 3.31

(a) Код

```
void switcher(long a, long b, long c, long *dest)
a в %rdi, b в %rsi, c в %rdx, dest в %rcx
1  switcher:
2      cmpq    $7, %rdi
3      ja     .L2
4      jmp     *.L4(,%rdi,8)
5      .section .rodata
6      .L7:
7      xorq    $15, %rsi
8      movq    %rsi, %rdx
9      .L3:
10     leaq    112(%rdx), %rdi
11     jmp     .L6
12     .L5:
13     leaq    (%rdx,%rsi), %rdi
```

```

14  salq    $2, %rdi
15  jmp     .L6
16  .L2:
17  movq    %rsi, %rdi
18  .L6:
19  movq    %rdi, (%rcx)
20  ret

```

(b) Таблица переходов

```

1  .L4:
2  .quad    .L3
3  .quad    .L2
4  .quad    .L5
5  .quad    .L2
6  .quad    .L6
7  .quad    .L7
8  .quad    .L2
9  .quad    .L5

```

### 3.7. Процедуры

Процедуры – это важная абстракция программного обеспечения. Они позволяют упаковать код, реализующий некоторые действия с указанным набором аргументов и, возможно, возвращающий некоторое значение. Такую процедуру можно вызывать из разных точек в программе. Хорошо спроектированное программное обеспечение использует процедуры в качестве механизма абстракции, скрывая детали реализации некоторых действий, обеспечивая при этом четкое и ясное определение интерфейса для вычисления определенных значений и влияния на состояние программы. Процедуры имеют множество обликов на разных языках программирования – функции, методы, подпрограммы, обработчики и т. д., но все они имеют общий набор характеристик.

Существует множество различных аспектов, которые необходимо учитывать в реализации поддержки процедур на машинном уровне. В целях обсуждения предположим, что процедура *P* вызывает процедуру *Q*, которая выполняется и возвращает управление процедуре *P*. Эти действия включают использование одного или нескольких из следующих механизмов:

- *передача управления.* При вызове *Q* в счетчик инструкций нужно записать ее начальный адрес, а при возврате – записать адрес инструкции в *P*, следующей за инструкцией вызова *Q*;
- *передача данных.* Процедура *P* должна иметь возможность передать один или несколько параметров в *Q*, а *Q* должна иметь возможность вернуть некоторое значение обратно в *P*;
- *выделение и освобождение памяти.* Процедуре *Q* может потребоваться выделить память для локальных переменных при вызове, а затем освободить ее перед возвратом.

Реализация процедур в архитектуре x86-64 основана на комбинации специальных инструкций и ряда соглашений об использовании машинных ресурсов, таких как регистры и оперативная память. В результате значительных усилий, направленных на минимизацию накладных расходов, связанных с вызовом процедур, была выработана минималистская стратегия, использующая ровно столько из вышеперечисленных

механизмов, сколько требуется для каждой конкретной процедуры. Далее мы шаг за шагом создадим различные механизмы, сначала описав передачу управления, затем передачу данных и, наконец, способы управления памятью.

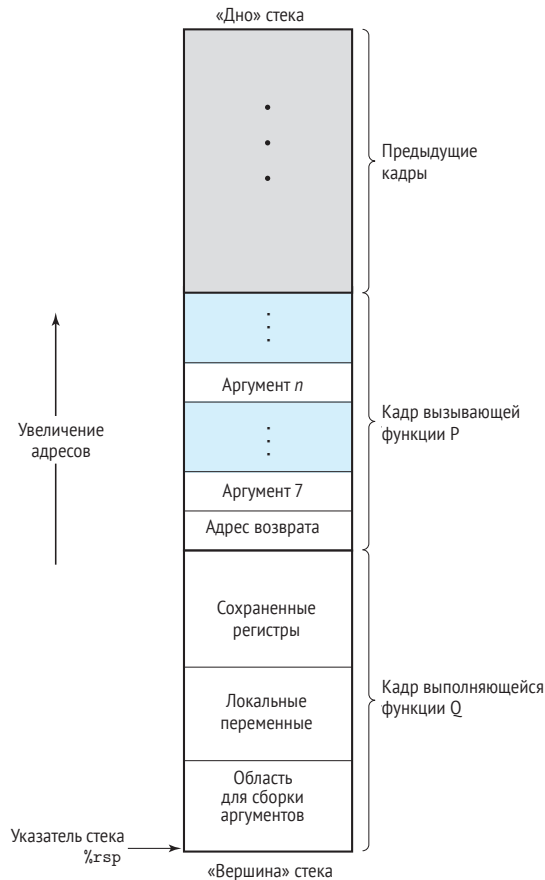
### 3.7.1. Стек времени выполнения

Ключевой особенностью механизма вызова процедур в языке С и в большинстве других языков является возможность использования механизма управления памятью, действующего по принципу «последним пришел – первым ушел», который поддерживается такой структуры данных, как стек. Рассматривая наш пример процедуры Р, вызывающей процедуру Q, можно заметить, что пока Q выполняется, Р вместе со всеми остальными процедурами в цепочке вызовов до Р временно приостанавливается. Пока Q выполняется, только ей нужна возможность выделения места для своих локальных переменных или настройки вызова другой процедуры. С другой стороны, когда Q возвращается, любую выделенную ею локальную память можно освободить. Следовательно, программа может управлять памятью, необходимой процедурам, используя стек, причем стек и регистры программы могут хранить информацию, необходимую для передачи управления и данных, а также для распределения памяти. Когда Р вызывает Q, информация об управлении и данных добавляется на вершину стека и удаляется, когда Р возвращает управление.

Как уже отмечалось в разделе 3.4.4, стек в архитектуре x86-64 растет в сторону меньших адресов, а указатель стека `%rsp` указывает на элемент на вершине стека. Данные могут сохраняться на стеке и извлекаться из стека с помощью инструкций `pushq` и `popq`. Пространство для данных без начального значения можно выделить в стеке, просто уменьшив указатель стека на соответствующую величину. Точно так же пространство можно освободить, увеличив указатель стека.

Когда в архитектуре x86-64 процедуре требуется дополнительная память для сохранения данных, не умещающихся в регистрах, она выделяет место в стеке. Эта область называется *кадром стека* процедуры. На рис. 3.3 показана общая структура стека во время выполнения программы, включая его деление на кадры. Кадр текущей выполняемой процедуры всегда находится на вершине стека. Когда процедура Р вызывает процедуру Q, она помещает *адрес возврата* в стек, указывая место внутри Р, с которого программа должна продолжить выполнение после возврата из Q. Мы считаем, что адрес возврата является частью кадра стека Р, потому что он содержит состояние, относящееся к Р. Код процедуры Q выделяет пространство, необходимое для ее кадра стека, путем переноса текущей границы стека. В этом пространстве он может сохранять значения регистров, выделять место для локальных переменных и настраивать аргументы для процедур, которые будут вызываться далее. Большинство процедур имеют кадры стека фиксированного размера, выделяемые в начале процедуры. Однако некоторые процедуры изменяют размеры своих кадров. Этот вопрос обсуждается в разделе 3.10.5. Процедура Р может передавать в регистрах до шести целочисленных значений (то есть указателей и целых чисел), но если Q требует больше аргументов, то Р может сохранить их в ее кадре стека до вызова.

В интересах экономии места и времени процедуры в архитектуре x86-64 размещают на стеке только те части кадра, которые им действительно необходимы. Например, число аргументов во многих процедурах не превышает шести, и их все можно передать в регистрах. Соответственно, некоторые элементы кадра стека, показанные на рис. 3.3, могут быть опущены. В действительности многим функциям вообще не нужен свой кадр стека, например когда все локальные переменные умещаются в регистрах, и функция не вызывает никаких других функций (такие процедуры иногда называют *листовыми*, следуя аналогии древовидной структуры вызовов процедур). Например, ни одна из функций, рассмотренных нами до сих пор, не требовала кадров стека.



**Рис. 3.3.** Общая структура кадра стека. Стек может использоваться для передачи аргументов и возвращаемой информации, для сохранения регистров и для локальных переменных. Элементы кадра могут быть опущены, когда они не нужны

### 3.7.2. Передача управления

Чтобы передать управление из функции P в функцию Q, достаточно записать в счетчик инструкций (Program Counter, PC) начальный адрес функции Q. Однако, когда позже придет время выйти из Q, процессор должен знать, куда в P вернуть управление. В архитектуре x86-64 эта информация записывается при вызове процедуры Q инструкцией `call Q`. Эта инструкция помещает адрес A в стек и записывает в PC адрес начала Q. Адрес A называется *адресом возврата* и вычисляется как адрес инструкции, следующей сразу за инструкцией `call`. Соответствующая инструкция `ret` извлекает адрес A из стека и записывает его в PC.

Ниже представлен обобщенный синтаксис инструкций `call` и `ret`:

Инструкция	Описание
<code>call</code> <i>Метка</i>	Вызов процедуры
<code>call</code> <i>*Операнд</i>	Вызов процедуры
<code>ret</code>	Возврат из вызова процедуры

(В ассемблерных листингах, генерируемых программой OBJDUMP, эти инструкции называются `callq` и `retq`. Дополнительный суффикс `q` просто подчеркивает, что это версии инструкций вызова и возврата для архитектуры x86-64, а не IA32. В ассемблерном коде для x86-64 обе версии могут использоваться как взаимозаменяемые.)

Инструкция `call` имеет цель, определяющую адрес инструкции, с которой должно начаться выполнение вызываемой процедуры. Подобно переходам, вызовы могут быть прямыми или косвенными. В ассемблерном коде цель прямого вызова задается меткой, а цель косвенного вызова отмечается звездочкой «\*», за которой следует описатель операнда в одном из форматов, описанных в табл. 3.2.

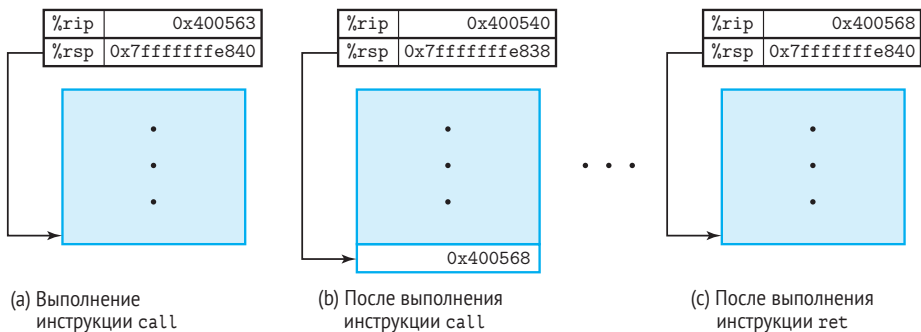
Рисунок 3.4 иллюстрирует выполнение инструкций `call` и `ret` в функциях `multstore` и `main`, представленных в разделе 3.2.2. Ниже показаны фрагменты дизассемблированного кода этих двух функций:

```

Начало функции multstore
1  0000000000400540 <multstore>:
2  400540: 53                push    %rbx
3  400541: 48 89 d3          mov     %rdx,%rbx
...
Возврат из функции multstore
4  40054d: c3               retq
...
Вызов multstore из main
5  400563: e8 d8 ff ff ff   callq   400540 <multstore>
6  400568: 48 8b 54 24 08   mov     0x8(%rsp),%rdx

```

Здесь инструкция `call`, находящаяся по адресу `0x400563` в функции `main`, вызывает функцию `multstore`. Состояние стека и регистров указателя стека `%rsp` и счетчика инструкций `%rip`, соответствующее этому моменту, показано на рис. 3.4 (а). Инструкция `call` помещает на стек адрес возврата `0x400568` и переходит к первой инструкции в функции `multstore` по адресу `0x400540` (рис. 3.4 (б)). Выполнение функции `multstore` продолжается до тех пор, пока не будет достигнута инструкция `ret` по адресу `0x40054d`. Эта инструкция выталкивает значение `0x400568` из стека и переходит по этому адресу, возобновляя выполнение `main` с инструкции, следующей сразу за инструкцией `call` (рис. 3.4 (с)).



**Рис. 3.4.** Иллюстрация действия инструкций `call` и `ret`. Инструкция `call` передает управление в начало вызываемой функции, а инструкция `ret` возвращается к инструкции, следующей за инструкцией вызова

Рассмотрим более подробный пример передачи управления между процедурами. В листинге 3.11 показан дизассемблированный код двух функций, `top` и `leaf`, а также часть кода функции `main`, которая вызывает `top`. Все инструкции сопровождаются метками L1-L2 (в `leaf`), T1-T4 (в `top`) и M1-M2 (в `main`).

**Листинг 3.11.** Дизассемблированный код, иллюстрирующий вызовы процедур и возвраты из них

```

Дизассемблированный код функции leaf(long y)
y в %rdi
1 000000000400540 <leaf>:
2   400540: 48 8d 47 02      lea    0x2(%rdi),%rax   L1: y+2
3   400544: c3              retq                      L2: Возврат

4 000000000400545 <top>:

Дизассемблированный код функции top(long x)
x в %rdi
5   400545: 48 83 ef 05      sub    $0x5,%rdi       T1: x-5
6   400549: e8 f2 ff ff ff   callq  400540 <leaf>    T2: Вызов leaf(x-5)
7   40054e: 48 01 c0         add    %rax,%rax       T3: Удвоить результат
8   400551: c3              retq                      T4: Возврат

...
Вызов top из функции main
9   40055b: e8 e5 ff ff ff   callq  400545 <top>     M1: Вызов top(100)
10  400560: 48 89 c2         mov    %rax,%rdx       M2: Возобновление

```

В табл. 3.12 показана подробная трассировка выполнения кода, в ходе которого main вызывает top(100), а top вызывает leaf(95). Функция leaf возвращает 97 в top, которая, в свою очередь, возвращает 194 в main. Первые три столбца в табл. 3.12 описывают выполняемую инструкцию – ее метку, адрес и тип. Следующие четыре столбца показывают состояние программы *перед* выполнением инструкции, включая содержимое регистров %rdi, %rax и %rsp, а также значение на вершине стека. Внимательно изучите содержимое этой таблицы, потому что она демонстрирует, насколько важную роль играет стек в поддержке вызовов процедур и возвратов из них.

**Таблица 3.12.** Трассировка выполнения примера

Инструкция			Состояние (к началу)				Описание
Метка	PC	Инструкция	%rdi	%rax	%rsp	*%rsp	
M1	0x40055b	callq	100	–	0x7fffffff820	–	Вызов top(100)
T1	0x400545	sub	100	–	0x7fffffff818	0x400560	Вход в top
T2	0x400549	callq	95	–	0x7fffffff818	0x400560	Вызов leaf(95)
L1	0x400540	lea	95	–	0x7fffffff810	0x40054e	Вход в leaf
L2	0x400544	retq	–	97	0x7fffffff810	0x40054e	Возврат 97 из leaf
T3	0x40054e	add	–	97	0x7fffffff818	0x400560	Возобновление top
T4	0x400551	retq	–	194	0x7fffffff818	0x400560	Возврат 194 из top
M2	0x400560	mov	–	194	0x7fffffff820	–	Возобновление main

Инструкция L1 в leaf записывает в %rax значение 97, которое должно быть возвращено. Затем инструкция L2 выполняет возврат. Она выталкивает 0x40054e из стека. После записи этого значения в счетчик инструкций PC управление передается инструкции T3 в top. Программа успешно завершила вызов leaf и вернулась в top.





### 3.7.3. Передача данных

Помимо передачи управления при вызове процедур и возврате из них, могут также передаваться данные в виде аргументов и возвращаемых значений. В x86-64 большая часть данных передается через регистры. Например, мы уже видели множество примеров функций, в которых аргументы передаются в регистрах `%rdi`, `%rsi` и др., а значения возвращаются в регистре `%rax`. Когда процедура *P* вызывает процедуру *Q*, то *P* должна сначала скопировать аргументы в соответствующие регистры. Точно так же, когда *Q* возвращает управление, процедура *P* может получить доступ к возвращаемому значению в регистре `%rax`. В этом разделе мы рассмотрим эти соглашения более подробно.

В архитектуре x86-64 через регистры можно передать до шести целочисленных аргументов (т. е. целых чисел и указателей). В табл. 3.13 описывается, какие аргументы и в каких регистрах передаются, с учетом размеров передаваемых данных. Аргументы помещаются в регистры в соответствии с их порядком в списке аргументов. Аргументы с размерами меньше 64 бит можно получить, обратившись к соответствующей части 64-разрядного регистра. Например, если первый аргумент имеет размер 32 бита, то его можно получить, обратившись к регистру `%edi`.

**Таблица 3.13.** Распределение регистров для передачи аргументов в функции. Регистры используются в указанном порядке и именуются в соответствии с размерами аргументов

Размер операнда (бит)	Номер аргумента					
	1	2	3	4	5	6
64	<code>%rdi</code>	<code>%rsi</code>	<code>%rdx</code>	<code>%rcx</code>	<code>%r8</code>	<code>%r9</code>
32	<code>%edi</code>	<code>%esi</code>	<code>%edx</code>	<code>%ecx</code>	<code>%r8d</code>	<code>%r9d</code>
16	<code>%di</code>	<code>%si</code>	<code>%dx</code>	<code>%cx</code>	<code>%r8w</code>	<code>%r9w</code>
8	<code>%dil</code>	<code>%sil</code>	<code>%dl</code>	<code>%cl</code>	<code>%r8b</code>	<code>%r9b</code>

Когда функция принимает более шести целочисленных аргументов, то 7-й и все последующие аргументы передаются через стек. Предположим, что процедура *P* вызывает процедуру *Q*, которая принимает *n* целочисленных аргументов, причем  $n > 6$ . В этом случае *P* должна выделить кадр стека достаточно большой, чтобы поместить в него аргументы с 7-го по *n*-й, как показано на рис. 3.3. Она помещает аргументы 1–6 в соответствующие регистры, а аргументы с 7-го по *n*-й сохраняет на стеке в обратном порядке так, что 7-й аргумент оказывается на вершине стека. При передаче параметров через стек все размеры данных округляются до кратных восьми. Разместив аргументы, программа выполняет инструкцию `call` для передачи управления процедуре *Q*. Процедура *Q* может получить свои аргументы из регистров и, если их больше 6, еще и из стека. Если *Q*, в свою очередь, вызывает некоторую функцию, принимающую больше шести аргументов, то она тоже может выделить для них место в своем кадре стека – в области, обозначенной на рис. 3.3 как «Область для сборки аргументов».

Для примера рассмотрим функцию `proc` на языке C, показанную в листинге 3.12 (а). Эта функция принимает восемь аргументов, включая целые числа разного размера (8, 4, 2 и 1 байт), а также разные типы указателей, каждый из которых имеет размер 8 байт.

**Листинг 3.12.** Пример функции, принимающей несколько аргументов разных типов. Аргументы 1–6 передаются через регистры, а аргументы 7–8 – через стек

(а) Код на C

```
void proc(long a1, long *a1p,
          int a2, int *a2p,
```

```

        short a3, short *a3p,
        char a4, char *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}

```

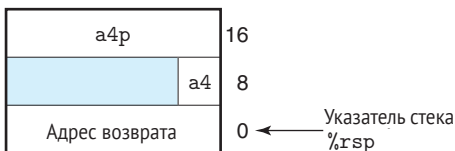
(b) Сгенерированный ассемблерный код

```

void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
Аргументы передаются в таком порядке:
a1 в %rdi      (64 бита)
a1p в %rsi     (64 бита)
a2 в %edx      (32 бита)
a2p в %rcx     (64 бита)
a3 в %r8w      (16 бит)
a3p в %r9      (64 бита)
a4 в %rsp+8    ( 8 бит)
a4p в %rsp+16  (64 бита)
1 proc:
2 movq    16(%rsp), %rax    Получить a4p (64 бита)
3 addq    %rdi, (%rsi)     *a1p += a1 (64 бита)
4 addl    %edx, (%rcx)     *a2p += a2 (32 бита)
5 addw    %r8w, (%r9)      *a3p += a3 (16 бит)
6 movl    8(%rsp), %edx     Получить a4 ( 8 бит)
7 addb    %dl, (%rax)      *a4p += a4 ( 8 бит)
8 ret                                Возврат

```

Ассемблерный код, сгенерированный в результате компиляции функции `proc`, показан в листинге 3.12 (b). Первые шесть аргументов передаются через регистры и последние два – через стек, как показано на диаграмме на рис. 3.5. На этой диаграмме показано состояние стека во время выполнения `proc`. Здесь видно, что при вызове процедуры на стек был помещен адрес возврата. Соответственно, два последних аргумента находятся в позициях 8 и 16 относительно указателя стека. В коде также можно заметить разные версии инструкции `ADD`, соответствующие размерам операндов: `addq` для `a1` (long), `addl` для `a2` (int), `addw` для `a3` (short) и `addb` для `a4` (char). Обратите внимание, что инструкция `movl` в строке 6 читает из памяти 4 байта, а следующая за ней инструкция `addb` использует только младший байт.



**Рис. 3.5.** Структура кадра стека функции `proc`. Аргументы `a4` и `a4p` передаются через стек

### Упражнение 3.33 (решение в конце главы)

Функция `proctrob` на языке C принимает четыре аргумента: `u`, `a`, `v` и `b`. Все они являются либо числами со знаком, либо указателями на числа со знаком, при этом сами числа имеют разные размеры. Вот как выглядит тело функции:

```

*u += a;
*v += b;
return sizeof(a) + sizeof(b);

```

В архитектуре x86-64 она компилируется в следующий ассемблерный код:

```

1  procprob:
2      movslq  %edi, %rdi
3      addq    %rdi, (%rdx)
4      addb    %sil, (%rcx)
5      movl    $6, %eax
6      ret

```

Определите действительный порядок и типы четырех параметров. Это упражнение имеет два правильных решения.

### 3.7.4. Локальные переменные на стеке

Большинство примеров процедур, которые мы видели до сих пор, не требовали дополнительной памяти для размещения локальных переменных и все необходимое им хранили в регистрах. Однако иногда локальные переменные приходится сохранять в памяти. Вот несколько типичных примеров таких случаев:

- недостаточно регистров для хранения всех локальных переменных;
- к локальной переменной применяется оператор взятия адреса &, и, следовательно, должна иметься возможность сгенерировать этот адрес;
- некоторые локальные переменные могут быть массивами или структурами и, как следствие, должны быть доступны по ссылкам на массив или структуру. Мы обсудим эту возможность, когда будем обсуждать размещение в памяти массивов и структур.

Обычно процедура выделяет память в своем кадре, уменьшая указатель стека. Это приводит к появлению в кадре стека области, подписанной на рис. 3.3 как «Локальные переменные».

Рассмотрим, например, оператор взятия адреса. Пусть есть две функции, показанные в листинге 3.13 (а). Функция `swap_add` меняет местами два значения, на которые ссылаются указатели `xp` и `yp`, а также возвращает сумму этих значений. Функция `caller` создает указатели на локальные переменные `arg1` и `arg2` и передает их в `swap_add`. В листинге 3.13 (б) показано, как `caller` использует кадр стека для хранения этих локальных переменных. Сначала код в `caller` уменьшает указатель стека на 16, тем самым выделяя на стеке 16 байт. Если обозначить значение указателя стека как  $S$ , то можно увидеть, что код вычисляет выражение `&arg2` как  $S + 8$  (строка 5), `&arg1` как  $S$  (строка 6). Отсюда можно сделать вывод, что локальные переменные `arg1` и `arg2` хранятся в кадре стека со смещениями 0 и 8 относительно указателя стека. Когда вызов `swap_add` завершается, то `caller` извлекает два значения из стека (строки 8–9), вычисляет их разницу и умножает ее на значение, возвращаемое функцией `swap_add` в регистре `%rax` (строка 10). В заключение функция освобождает свой кадр, увеличивая указатель стека на 16 (строка 11). Как показывает этот пример, стек предоставляет простой механизм выделения памяти для локальных переменных, когда это необходимо, и ее освобождения по завершении функции.

**Листинг 3.13.** Пример определения и вызова процедуры. Вызываемый код должен выделить место в кадре стека из-за наличия операторов взятия адреса

(а) Код на C функций `swap_add` и `caller`

```

long swap_add(long *xp, long *yp)
{

```

```

    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}

```

(b) Ассемблерный код, сгенерированный для функции caller

```

long caller()
1 caller:
2     subq    $16, %rsp           Выделить 16 байт в кадре стека
3     movq    $534, (%rsp)       Сохранить 534 в arg1
4     movq    $1057, 8(%rsp)     Сохранить 1057 в arg2
5     leaq    8(%rsp), %rsi      Вычислить &arg2 для второго аргумента
6     movq    %rsp, %rdi         Вычислить &arg1 для первого аргумента
7     call    swap_add           Вызвать swap_add(&arg1, &arg2)
8     movq    (%rsp), %rdx       Получить arg1
9     subq    8(%rsp), %rdx      Вычислить diff = arg1 - arg2
10    imulq   %rdx, %rax         Вычислить sum * diff
11    addq    $16, %rsp          Освободить память в кадре стека
12    ret                                Возврат

```

В качестве более сложного примера рассмотрим функцию `call_proc` в листинге 3.14, иллюстрирующую многие аспекты использования стека в архитектуре x86-64. Несмотря на большой объем этого примера, его стоит внимательно изучить. В нем демонстрируется функция, которая должна выделять память в стеке для локальных переменных, а также вызывать другую функцию с 8 аргументами (листинг 3.12). Функция `call_proc` создает кадр стека, как показано на рис. 3.6.

**Листинг 3.14.** Пример вызова функции `proc` из листинга 3.12. Этот код создает кадр стека

(a) Код на C функции `call_proc`

```

long call_proc()
{
    long x1 = 1; int x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}

```

(b) Сгенерированный ассемблерный код

```

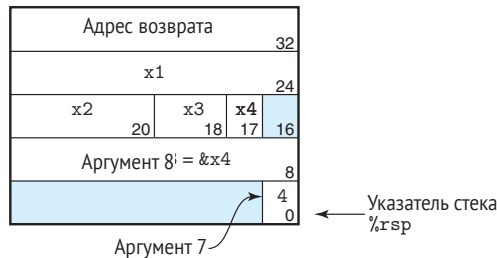
long call_proc()
1 call_proc:
    Подготовка аргументов для вызова proc
2     subq    $32, %rsp          Выделить кадр стека с размером 32 байта

```

```

3  movq    $1, 24(%rsp)    Сохранить 1 в &x1
4  movl    $2, 20(%rsp)    Сохранить 2 в &x2
5  movw    $3, 18(%rsp)    Сохранить 3 в &x3
6  movb    $4, 17(%rsp)    Сохранить 4 в &x4
7  leaq    17(%rsp), %rax   Создать &x4
8  movq    %rax, 8(%rsp)    Сохранить &x4 в 8-м аргументе
9  movl    $4, (%rsp)       Сохранить 4 в 7-м аргументе
10 leaq    18(%rsp), %r9    Сохранить &x3 в 6-м аргументе
11 movl    $3, %r8d         Сохранить 3 в 5-м аргументе
12 leaq    20(%rsp), %rcx   Сохранить &x2 в 4-м аргументе
13 movl    $2, %edx         Сохранить 2 в 3-м аргументе
14 leaq    24(%rsp), %rsi   Сохранить &x1 во 2-м аргументе
15 movl    $1, %edi         Сохранить 1 в 1-м аргументе
    Вызов proc
16 call    proc
    Извлечение результатов из памяти
17 movslq   20(%rsp), %rdx   Извлечь x2 и преобразовать в long
18 addq     24(%rsp), %rdx   Вычислить x1+x2
19 movswl   18(%rsp), %eax   Извлечь x3 и преобразовать в int
20 movsbl   17(%rsp), %ecx   Извлечь x4 и преобразовать в int
21 subl     %ecx, %eax       Вычислить x3-x4
22 cltq     Преобразовать в long
23 imulq    %rdx, %rax       Вычислить (x1+x2) * (x3-x4)
24 addq     $32, %rsp        Освободить память в кадре стека
25 ret      Return

```



**Рис. 3.6.** Кадр стека функции `call_proc`. Кадр стека содержит локальные переменные, а также два аргумента для передачи функции `proc`

Глядя на ассемблерный код `call_proc` (листинг 3.14 (b)), можно заметить, что большая часть кода (строки 2–15) подготавливает аргументы для вызова функции `proc`. Сюда входят настройка кадра стека для локальных переменных и параметров функции, а также загрузка аргументов в регистры. Как показано на рис. 3.6, локальные переменные `x1–x4` размещены в стеке и имеют разные размеры. Если выразить их местоположения через смещение относительно указателя стека, то они занимают байты 24–31 (`x1`), 20–23 (`x2`), 18–19 (`x3`) и 17 (`x4`). Указатели на эти местоположения генерируются инструкциями `leaq` (строки 7, 10, 12 и 14). Аргументы 7-й (со значением 4) и 8-й (указатель на `x4`) хранятся в стеке со смещениями 0 и 8 относительно указателя стека.

### 3.7.5. Локальные переменные в регистрах

Набор регистров действует как единый ресурс, совместно используемый всеми процедурами. И хотя в каждый конкретный момент времени активной может быть только одна процедура, мы должны сделать так, чтобы при вызове одной процедуры (*вызыва-*

емой) из другой (вызывающей) вызываемая процедура не затирает значения некоторых регистров, которые вызывающая процедура планирует использовать в дальнейшем. По этой причине архитектура x86-64 устанавливает набор правил по использованию регистров, которые должны соблюдаться всеми процедурами, включая библиотечные.

По соглашению регистры `%rbx`, `%rbp` и `%r12-%r15` относятся к категории регистров, *сохраняемых вызываемым* (callee saved). Когда процедура Q вызывается процедурой P, она должна *сохранить* значения этих регистров и гарантировать, что они будут иметь те же значения после возврата из Q в P. Процедура Q может просто не изменять значения этих регистров или сохранить их в стеке, использовать для своих нужд, а затем восстановить со стека перед возвратом. Сохранение регистров на стеке приводит к созданию области в кадре, которая на рис. 3.3 подписана как «Сохраненные регистры». В соответствии с этим соглашением процедура P может записать свое значение в регистр, сохраняемый вызываемой процедурой (разумеется, предварительно сохранив в своем кадре прежнее значение, доставшееся от вышестоящей процедуры), вызывать Q, а затем использовать значение в этом регистре, не опасаясь, что оно будет изменено.

Все остальные регистры, за исключением указателя стека `%rsp`, классифицируются как регистры, *сохраняемые вызывающим* (caller saved). Это означает, что они могут быть изменены любой функцией. В контексте примера, когда процедура P вызывает процедуру Q, под «сохраняемыми вызывающим» подразумевается, что если P использует эти регистры для хранения локальных данных, то она (как вызывающая сторона) должна сохранить эти регистры перед вызовом Q, потому что та может свободно изменять такие регистры.

В качестве примера рассмотрим функцию P, показанную в листинге 3.15 (а). Она дважды вызывает функцию Q. Перед первым вызовом она должна сохранить значение `x` для использования в дальнейшем. Точно так же перед вторым вызовом она должна сохранить значение, вычисленное вызовом `Q(y)`. В листинге 3.15 (b) мы видим, что код, сгенерированный компилятором GCC, использует два регистра, сохраняемых вызываемым: `%rbp`, используемый для хранения `x`, и `%rbx`, используемый для хранения результата `Q(y)`. Начиная выполнение, функция P сохраняет эти два регистра на стеке (строки 2–3). Далее она копирует аргумент `x` в `%rbp` перед первым вызовом Q (строка 5) и затем копирует результат первого вызова в `%rbx` перед вторым вызовом Q (строка 8). В конце (строки 13–14) функция P восстанавливает значения двух регистров, сохраняемых вызываемым, выталкивая их из стека. Обратите также внимание, что они выталкиваются в обратном порядке, чтобы учесть принцип работы стека «последним пришел – первым ушел».

**Листинг 3.15.** Код, демонстрирующий использование регистров, сохраняемых вызываемым. Значение `x` должно сохраниться после первого вызова, а значение `Q(y)` должно сохраняться после второго

(a) Вызывающая функция

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

(b) Сгенерированный ассемблерный код вызывающей функции

```
long P(long x, long y)
x в %rdi, y в %rsi
1 P:
2 pushq    %rbp                Сохранить %rbp
```

3	pushq	%rbx	Сохранить %rbx
4	subq	\$8, %rsp	Передвинуть границу кадра стека
5	movq	%rdi, %rbp	Сохранить x
6	movq	%rsi, %rdi	Переместить y в первый аргумент
7	call	Q	Вызвать Q(y)
8	movq	%rax, %rbx	Сохранить результат
9	movq	%rbp, %rdi	Переместить x в первый аргумент
10	call	Q	Вызвать Q(x)
11	addq	%rbx, %rax	Сложить результаты Q(y) и Q(x)
12	addq	\$8, %rsp	Вернуть границу кадра стека на место
13	popq	%rbx	Восстановить %rbx
14	popq	%rbp	Восстановить %rbp
15	ret		

### Упражнение 3.34 (решение в конце главы)

Взгляните на функцию P, которая использует локальные переменные с именами a0–a8. Она вызывает функцию Q, передавая значения этих переменных в качестве аргументов. GCC сгенерировал следующий код для первой части функции P:

```

long P(long x)
x в %rdi
1 P:
2   pushq   %r15
3   pushq   %r14
4   pushq   %r13
5   pushq   %r12
6   pushq   %rbp
7   pushq   %rbx
8   subq    $24, %rsp
9   movq    %rdi, %rbx
10  leaq     1(%rdi), %r15
11  leaq     2(%rdi), %r14
12  leaq     3(%rdi), %r13
13  leaq     4(%rdi), %r12
14  leaq     5(%rdi), %rbp
15  leaq     6(%rdi), %rax
16  movq     %rax, (%rsp)
17  leaq     7(%rdi), %rdx
18  movq     %rdx, 8(%rsp)
19  movl     $0, %eax
20  call     Q
...
```

1. Определите, какие локальные переменные сохраняются в регистрах, сохраняемых вызываемым.
2. Определите, какие локальные переменные сохраняются в стеке.
3. Объясните, почему программа не смогла сохранить все локальные переменные в регистрах, сохраняемых вызываемым.

## 3.7.6. Рекурсивные процедуры

Соглашения об использовании регистров и стека, описанные выше, позволяют процедурам рекурсивно вызывать самих себя. Для каждого вызова на стеке выделяется

отдельное пространство, поэтому локальные переменные многих незавершенных вызовов никак не мешают друг другу. Более того, принцип работы стека обеспечивает правильное выделение локальной памяти при вызове процедуры и ее освобождение при возвращении.

В листинге 3.16 показан программный код на рекурсивной функции вычисления факториала. Как видите, сгенерированный ассемблерный код использует регистр %rbx для хранения параметра n, предварительно сохраняя его значение в стеке (строка 2) и затем восстанавливая перед возвратом (строка 11). Благодаря особенностям работы стека и соглашениям о сохранении регистров мы можем быть уверены, что когда рекурсивный вызов rfact(n-1) вернет результат (строка 9), то (1) результат вызова будет храниться в регистре %rax и (2) значение аргумента n будет храниться в регистре %rbx. Умножение этих двух значений даст нам желаемый результат.

**Листинг 3.16.** Код рекурсивной функции вычисления факториала. Стандартные механизмы обработки процедур прекрасно подходят для реализации рекурсивных функций

(a) Код на C

```
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

(b) Сгенерированный ассемблерный код

```
long rfact(long n)
n в %rdi
1 rfact:
2 pushq %rbx           Сохранить %rbx
3 movq %rdi, %rbx      Сохранить n в регистре, сохраняемом вызываемым
4 movl $1, %eax        Записать 1 в возвращаемое значение
5 cmpq $1, %rdi        Сравнить n:1
6 jle .L35             Если <=, перейти к done
7 leaq -1(%rdi), %rdi   Вычислить n-1
8 call rfact           Вызвать rfact(n-1)
9 imulq %rbx, %rax      Умножить результат на n
10 .L35:               done:
11 popq %rbx           Восстановить %rbx
12 ret                 Возврат
```

Из этого примера видно, что рекурсивный вызов функции выполняется так же, как любой другой вызов. Стек обеспечивает такую механику работы, согласно которой каждый вызов функции получает свое место на стеке для хранения информации о состоянии (адрес возврата и регистров, сохраняемых вызываемым). При необходимости стек также может хранить локальные переменные. Механика работы стека естественным образом соответствует механике вызова функций и возврата из них. Этот метод реализации вызовов функций и возвратов работает даже в более сложных случаях, включая взаимную рекурсию (например, когда процедура P вызывает Q, а та, в свою очередь, вызывает P).



**Упражнение 3.35 (решение в конце главы)**

Для функции на C

```
long rfun(unsigned long x) {
    if ( _____ )
        return _____;
    unsigned long nx = _____;
    long rv = rfun(nx);
    return _____;
}
```

GCC сгенерировал следующий ассемблерный код:

```
long rfun(unsigned long x)
x in %rdi
1  rfun:
2  pushq    %rbx
3  movq     %rdi, %rbx
4  movl     $0, %eax
5  testq    %rdi, %rdi
6  je       .L2
7  shrq     $2, %rdi
8  call     rfun
9  addq     %rbx, %rax
10 .L2:
11 popq     %rbx
12 ret
```

1. Какое значение хранит функция `rfun` в регистре `%rbx`, который должен сохраняться вызываемым?
2. Заполните недостающие выражения в коде на C.

## 3.8. Распределение памяти под массивы и доступ к массивам

Массивы в языке C являются одним из способов объединения скалярных данных в более крупные типы. Язык C использует простейшие реализации массивов, поэтому они легко и просто транслируются в машинный код. Одна из интересных особенностей языка C заключается в возможности генерировать указатели на элементы массива и выполнять арифметические операции с ними. В ассемблерном коде эти указатели транслируются в вычисления соответствующих адресов.

Оптимизирующие компиляторы весьма успешно упрощают операции вычисления адресов с использованием индексов массивов, что серьезно усложняет определение соответствия между машинным кодом и кодом на языке C.

### 3.8.1. Базовые принципы

Рассмотрим объявление типа данных  $T$ , в котором используется целочисленная константа  $N$ :

$$T\ A[N];$$

Обозначим начальный адрес массива как  $x_A$ . Это объявление имеет два следствия. Во-первых, выделяется непрерывная область памяти размером  $L \cdot N$ , где  $L$  – размер

(в байтах) типа данных  $T$ . Во-вторых, вводится в употребление идентификатор  $A$ , который можно использовать как указатель на начало массива. Значение этого указателя равно  $x_A$ . Элементы массива доступны по целочисленным индексам в диапазоне от 0 до  $N - 1$ . Элемент массива с индексом  $i$  будет храниться по адресу  $x_A + L \cdot i$ .

Рассмотрим, например, такие объявления:

```
char   A[12];
char   *B[8];
int     C[6];
double *D[5];
```

Эти объявления генерируют массивы со следующими параметрами:

Массив	Размер элемента	Общий размер	Начальный адрес	Элемент $i$
A	1	12	$x_A$	$x_A + i$
B	8	64	$x_B$	$x_B + 8i$
C	4	24	$x_C$	$x_C + 4i$
D	8	40	$x_D$	$x_D + 8i$

Массив A состоит из 12 однобайтных элементов (char). Массив C состоит из 6 целочисленных значений, каждое из которых занимает 4 байта. Массивы B и D хранят указатели, соответственно, для каждого элемента в них требуется 8 байт.

Инструкции ссылки на память в архитектуре x86-64 спроектированы так, что упрощают доступ к массивам. Например, предположим, что E – это массив значений типа int, и нам нужно получить значение элемента  $E[i]$ , при этом начальный адрес E находится в регистре %rdx, а  $i$  – в регистре %rcx. Тогда инструкция

```
movl (%rdx,%rcx,4),%eax
```

вычислит адрес  $x_E + 4i$ , прочитает значение элемента по этому адресу и сохранит его в регистре %eax. Допустимые коэффициенты масштабирования – 1, 2, 4 и 8 – охватывают размеры всех простых типов данных.

### Упражнение 3.36 (решение в конце главы)

Рассмотрим следующие объявления:

```
short   S[7];
short   *T[3];
short   **U[6];
int      V[8];
double  *W[4];
```

Заполните пропуски в приводимой далее таблице, описывающей размеры элементов, общие размеры и адрес элемента  $i$  для каждого из указанных массивов:

Массив	Размер элемента	Общий размер	Начальный адрес	Элемент $i$
S	_____	_____	$x_S$	_____
T	_____	_____	$x_T$	_____
U	_____	_____	$x_U$	_____
V	_____	_____	$x_V$	_____
W	_____	_____	$x_W$	_____

### 3.8.2. Арифметика указателей

Язык C поддерживает арифметические операции с указателями, при этом вычисленное значение масштабируется в соответствии с типом данных, на которые ссылается указатель. Иначе говоря, если  $p$  – это указатель на данные типа  $T$ , а значением  $p$  является  $x_p$ , то выражение  $p+i$  имеет значение  $x_p + L \cdot i$ , где  $L$  – размер типа данных  $T$ .

Унарные операторы `&` и `*` позволяют генерировать и разыменовывать указатели. То есть для выражения *Expr*, обозначающего некоторый конкретный объект, `&Expr` дает указатель с адресом объекта. Для выражения *AExpr*, обозначающего адрес, `*AExpr` дает значение, хранящееся по этому адресу. Следовательно, выражения *Expr* и `*&Expr` эквивалентны. Операцию индексирования можно применять как к массивам, так и к указателям. Ссылка на элемент массива `A[i]` идентична выражению `*(A+i)`. Оно вычисляет адрес  $i$ -го элемента массива, а затем извлекает соответствующий элемент из памяти.

#### Упражнение 3.37 (решение в конце главы)

Предположим, что  $x_p$ , адрес целочисленного массива с элементами типа `short`, и целочисленный индекс  $i$  хранятся в регистрах `%edx` и `%ecx` соответственно. Для каждого из следующих выражений укажите его тип, формулу вычисления значения и реализацию на языке ассемблера. Результат должен быть помещен в регистр `%rax`, если это указатель, и в регистр `%eax`, если это целое число типа `short`.

Выражение	Тип	Значение	Ассемблерный код
<code>S+1</code>	_____	_____	_____
<code>S[3]</code>	_____	_____	_____
<code>&amp;S[i]</code>	_____	_____	_____
<code>S[4*i+1]</code>	_____	_____	_____
<code>S+i-5</code>	_____	_____	_____

Возвращаясь к предыдущему примеру, предположим, что начальный адрес массива целых чисел  $E$  и целочисленный индекс  $i$  хранятся в регистрах `%rdx` и `%rcx` соответственно. Ниже приводится несколько выражений, в которых фигурирует массив  $E$ . Мы также покажем код на ассемблере, реализующий каждое из этих выражений, результаты которых запоминаются в регистре `%eax` (для данных) или `%rax` (для указателей).

Выражение	Тип	Значение	Ассемблерный код
<code>E</code>	<code>int *</code>	$x_E$	<code>movq %rdx,%rax</code>
<code>E[0]</code>	<code>int</code>	$M[x_E]$	<code>movl (%rdx),%eax</code>
<code>E[i]</code>	<code>int</code>	$M[x_E + 4i]$	<code>movl (%rdx,%rcx,4),%eax</code>
<code>&amp;E[2]</code>	<code>int *</code>	$x_E + 8$	<code>leaq 8(%rdx),%rax</code>
<code>E+i-1</code>	<code>int *</code>	$x_E + 4i - 4$	<code>leaq -4(%rdx,%rcx,4),%rax</code>
<code>*(E+i-3)</code>	<code>int</code>	$M[x_E + 4i - 12]$	<code>movl -12(%rdx,%rcx,4),%eax</code>
<code>&amp;E[i]-E</code>	<code>long</code>	$i$	<code>movq %rcx,%rax</code>

В этих примерах мы видим, что выражения, возвращающие значения массива, имеют тип `int` и, следовательно, включают 4-байтные операции (например, `movl`) и регистры (например, `%eax`). Выражения, возвращающие указатели, имеют тип `int *` и, следо-

вательно, включают 8-байтные операции (например, `leaq`) и регистры (например, `%rax`). Последний пример показывает, что можно вычислить разность двух указателей в одной и той же структуре данных, в результате чего данные имеют тип `long` и значение, равное разности двух адресов, деленной на размер типа данных.

### 3.8.3. Вложенные массивы

Общие принципы размещения массивов в памяти и ссылок на них остаются в силе, даже для массивов. Например, объявление

```
int A[5][3];
```

эквивалентно объявлению

```
typedef int row3_t[3];
row3_t A[5];
```

Тип данных `row3_t` определяется как массив из трех целых чисел. Массив `A` содержит пять таких элементов, каждый из которых требует 12 байт для хранения трех целых чисел. То есть общий размер массива равен  $4 \cdot 5 \cdot 3 = 60$  байт.

Массив `A` также можно рассматривать как двумерный массив с пятью строками и тремя столбцами, элементы которого обозначаются как `A[0][0]` – `A[4][2]`. Элементы массива упорядочены в памяти *по строкам*, то есть сначала следуют все элементы строки 0, которую можно обозначить как `A[0]`, за ней следуют все элементы строки 1 (`A[1]`) и т. д., как показано на рис. 3.7.

Строка	Элемент	Адрес
A[0]	A[0][0]	$x_A$
	A[0][1]	$x_A + 4$
	A[0][2]	$x_A + 8$
A[1]	A[1][0]	$x_A + 12$
	A[1][1]	$x_A + 16$
	A[1][2]	$x_A + 20$
A[2]	A[2][0]	$x_A + 24$
	A[2][1]	$x_A + 28$
	A[2][2]	$x_A + 32$
A[3]	A[3][0]	$x_A + 36$
	A[3][1]	$x_A + 40$
	A[3][2]	$x_A + 44$
A[4]	A[4][0]	$x_A + 48$
	A[4][1]	$x_A + 52$
	A[4][2]	$x_A + 56$

Рис. 3.7. Элементы массива, упорядоченные по строкам

Этот порядок является следствием вложенного объявления. Рассматривая `A` как массив из пяти элементов, каждый из которых сам является массивом из трех значений `int`, мы сначала индексируем строки `A[0]`, затем `A[1]` и т. д.

Для доступа к элементам многомерных массивов компилятор генерирует код, вычисляющий смещение желаемого элемента, а затем использует одну из инструкций

MOV, передавая ей начальный адрес массива в качестве базового адреса и смещение (возможно, масштабированное) в качестве индекса. В общем случае для массива, объявленного как

$$TD[R][C];$$

элемент  $D[i][j]$  располагается в памяти по адресу

$$\&D[i][j] = x_0 + L(C \cdot i + j), \quad (3.1)$$

где  $L$  – размер типа данных  $T$  в байтах. Например, рассмотрим массив  $A$  из  $5 \times 3$  целых чисел, который мы определили выше. Предположим, что  $x_A$ ,  $i$  и  $j$  находятся в регистрах  $\%rdi$ ,  $\%rsi$  и  $\%rdx$  соответственно. Тогда элемент массива  $A[i][j]$  можно скопировать в регистр  $\%eax$  следующим образом:

	$A$ в $\%rdi$ , $i$ в $\%rsi$ и $j$ в $\%rdx$	
1	leaq $(\%rsi, \%rsi, 2)$ , $\%rax$	Вычислить $3i$
2	leaq $(\%rdi, \%rax, 4)$ , $\%rax$	Вычислить $x_A + 12i$
3	movl $(\%rax, \%rdx, 4)$ , $\%eax$	Прочитать из $M[x_A + 12i + 4j]$

Как можно заметить, этот код вычисляет адрес элемента как  $x_A + 12i + 4j = x_A + 4(3i + j)$ , используя возможности масштабирования и сложения в адресной арифметике x86-64.

### Упражнение 3.38 (решение в конце главы)

Взгляните на следующий исходный код, где  $M$  и  $N$  – константы, объявленные с помощью директивы `#define`:

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

В процессе компиляции этой программы GCC сгенерировал следующий ассемблерный код:

```
long sum_element(long i, long j)
i в %rdi, j в %rsi
1 sum_element:
2 leaq 0(,%rdi,8), %rdx
3 subq %rdi, %rdx
4 addq %rsi, %rdx
5 leaq (%rsi,%rsi,4), %rax
6 addq %rax, %rdi
7 movq Q(,%rdi,8), %rax
8 addq P(,%rdx,8), %rax
9 ret
```

Используйте свои навыки воссоздания исходного кода на C и определите значения  $M$  и  $N$  на основе этого ассемблерного кода.

## 3.8.4. Массивы фиксированных размеров

Компилятор C обладает массой возможностей оптимизации программ, работающих с многомерными массивами фиксированных размеров. Далее мы продемонстрируем некоторые оптимизации, выполняемые компилятором GCC при компиляции с фла-

гом -01. Например, предположим, что мы объявили тип данных `fix_matrix` как массив целых чисел фиксированного размера  $16 \times 16$ :

```
#define N 16
typedef int fix_matrix[N][N];
```

(Этот пример демонстрирует эффективный прием программирования: всякий раз, когда программа использует некоторую константу, задающую размер массива или буфера, то лучше всего определить ее имя через объявление `#define`, а затем последовательно использовать это имя вместо числового значения. Тогда, если когда-нибудь возникнет потребность изменить значение константы, это можно будет сделать, просто изменив объявление `#define`.) Код в листинге 3.17 (а) вычисляет элементы с индексами  $i, k$  произведения матриц  $A$  и  $B$ , то есть скалярное произведение строки  $i$  из  $A$  на столбец  $k$  в  $B$ . Произведение вычисляется по формуле  $\sum_{0 \leq j < N} a_{ij} \cdot b_{jk}$ . GCC сгенерировал код, который мы затем переложили на C (функция `fix_prod_ele_opt` в листинге 3.17 (б)). В этом коде можно видеть несколько интересных оптимизаций. Компилятор избавился от целочисленного индекса  $j$  и преобразовал все ссылки на массив в операции разыменования указателя. Для этого он (1) сгенерировал указатель, который мы назвали `Aptr`, ссылающийся на последовательные элементы в строке  $i$  массива  $A$ , (2) сгенерировал указатель, который мы назвали `Bptr`, ссылающийся на последовательные элементы в столбце  $k$  массива  $B$ , и (3) сгенерировал указатель, который мы назвали `Bend`, равный значению `Bptr`, которое получится, когда придет время завершить цикл. Начальное значение для `Aptr` – это адрес первого элемента строки  $i$  в  $A$ , который на языке C задается выражением `&A[i][0]`. Начальное значение для `Bptr` – это адрес первого элемента столбца  $k$  в  $B$ , который на языке C задается выражением `&B[0][k]`. Значение для `Bend` – это адрес  $(n + 1)$ -го элемента в столбце  $j$  в  $B$ , который на языке C задается выражением `&B[N][k]`.

**Листинг 3.17.** Оригинальный и оптимизированный код для вычисления элемента  $i, k$  матричного произведения для массивов фиксированной длины. Компилятор выполняет эти оптимизации автоматически

(а) Оригинальный код на C

```
/* Вычисляет элементы i,k произведения матриц фиксированного размера */
int fix_prod_ele (fix_matrix A, fix_matrix B, long i, long k) {
    long j;
    int result = 0;

    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];

    return result;
}
```

(б) Оптимизированный код на C

```
1 /* Вычисляет элементы i,k произведения матриц фиксированного размера */
2 int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i, long k) {
3     int *Aptr = &A[i][0]; /* Указывает на элементы в строке i массива A */
4     int *Bptr = &B[0][k]; /* Указывает на элементы в столбце k массива B */
5     int *Bend = &B[N][k]; /* Когда Bptr достигнет этого значения, итерации
6                             прекращаются */
7     int result = 0;
8     do {
9         result += *Aptr * *Bptr; /* Добавить следующее произведение в сумму */
10        Aptr++; /* Переместить Aptr на следующий элемент */
11        Bptr++; /* Переместить Bptr на следующий элемент */
12    } while (Bptr != Bend); /* Проверить достижение конечной точки */
```

```

13 return result;
14 }

```

Ниже приводится фактический ассемблерный код, сгенерированный компилятором GCC для функции `fix_prod_ele`. Здесь используются четыре регистра: `%eax` хранит результат, `%rdi` хранит `Aptr`, `%rcx` хранит `Bptr` и `%rsi` хранит `Bend`.

```

int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i, long k)
A в %rdi, B в %rsi, i в %rdx, k в %rcx

1 fix_prod_ele:
2 salq    $6, %rdx          Вычислить 64 * i
3 addq    %rdx, %rdi        Вычислить Aptr = xA + 64i = &A[i][0]
4 leaq    (%rsi,%rcx,4), %rcx  Вычислить Bptr = xB + 4k = &B[0][k]
5 leaq    1024(%rcx), %rsi    Вычислить Bend = xB + 4k + 1024 = &B[N][k]
6 movl    $0, %eax          Установить result = 0
7 .L7:                      loop:
8 movl    (%rdi), %edx        Прочитать *Aptr
9 imull   (%rcx), %edx        Умножить на *Bptr
10 addl   %edx, %eax          Прибавить к result
11 addq   $4, %rdi            Увеличить Aptr++
12 addq   $64, %rcx           Увеличить Bptr += N
13 cmpq   %rsi, %rcx         Сравнить Bptr:Bend
14 jne    .L7                Если !=, перейти к loop
15 rep; ret                  Возврат

```

### Упражнение 3.39 (решение в конце главы)

Используя уравнение 3.1, объясните, почему вычисления начальных значений для `Aptr`, `Bptr` и `Bend` в коде C (листинг 3.17 (b), строки 3–5) правильно описывают вычисления в ассемблерном коде, сгенерированном для `fix_prod_ele` (строки 3–5).

### Упражнение 3.40 (решение в конце главы)

Следующий код на C присваивает диагональным элементам одного из наших массивов фиксированного размера значение `val`:

```

/* Присваивает всем диагональным элементам значение val */
void fix_set_diag(fix_matrix A, int val) {
    long i;
    for (i = 0; i < N; i++)
        A[i][i] = val;
}

```

При компиляции с уровнем оптимизации `-O1` GCC сгенерировал следующий ассемблерный код:

```

void fix_set_diag(fix_matrix A, int val)
A в %rdi, val в %rsi
1 fix_set_diag:
2 movl    $0, %eax
3 .L13:
4 movl    %esi, (%rdi,%rax)
5 addq    $68, %rax
6 cmpq    $1088, %rax
7 jne     .L13
8 rep; ret

```

Воссоздайте код на C в виде функции `fix_set_diag_opt`, использующий оптимизации, подобные тем, что использованы в ассемблерном коде, подобно тому, как это было сделано в листинге 3.17 (b). Используйте выражения, содержащие параметр  $N$  вместо целочисленной константы, чтобы код продолжал работать правильно после переопределения  $N$ .

### 3.8.5. Массивы переменных размеров

Традиционно язык C поддерживает многомерные массивы, размеры которых (за исключением, возможно, первого измерения) становятся известными на этапе компиляции. Во многих приложениях, однако, требуются массивы переменных размеров, которые приходится размещать в памяти с помощью таких функций, как `malloc` или `calloc`. Раньше при работе с подобными массивами приходилось явно отображать многомерные массивы в одномерные с помощью индексирования по строкам, как показано в уравнении 3.1. Стандарт ISO C99 ввел возможность выражать измерения массивов, вычисляемые при выделении памяти.

В языке C массив переменного размера можно объявить

```
int A[выражение1][выражение2]
```

как локальную переменную или как аргумент функции, а затем определять размеры массива путем вычисления выражений *выражение1* и *выражение2* в момент обработки объявления. Например, мы можем написать функцию для доступа к элементу  $i, j$  массива  $n \times n$  следующим образом:

```
int var_ele(long n, int A[n][n], long i, long j) {
    return A[i][j];
}
```

Параметр  $n$  должен предшествовать параметру  $A[n][n]$ , чтобы компилятор мог вычислить размеры массива.

Для этой функции GCC генерирует следующий код:

```
int var_ele(long n, int A[n][n], long i, long j)
n в %rdi, A в %rsi, i в %rdx, j в %rcx
1 var_ele:
2  imulq    %rdx, %rdi          Вычислить  $n \cdot i$ 
3  leaq     (%rsi,%rdi,4), %rax  Вычислить  $x_A + 4(n \cdot i)$ 
4  movl     (%rax,%rcx,4), %eax  Прочитать из  $M[x_A + 4(n \cdot i) + 4j]$ 
5  ret
```

Как отмечается в комментариях, этот код вычисляет адрес элемента  $i, j$  как  $x_A + 4(n \cdot i) + 4j = x_A + 4(n \cdot i + j)$ . Вычисление адреса выполняется так же, как при работе с массивами фиксированного размера (раздел 3.8.3), за исключением того, что (1) из-за дополнительного параметра  $n$  меняется порядок использования регистров и (2) для вычисления  $n \cdot i$  используется инструкция умножения (строка 2), а не `leaq`. Таким образом, обращение к массивам переменного размера требует лишь небольшого обобщения, по сравнению с массивами фиксированного размера. При работе с массивами переменного размера приходится использовать инструкцию умножения, чтобы масштабировать  $i$  по  $n$  вместо серии сдвигов и сложений. В некоторых процессорах такое умножение может привести к значительному снижению производительности, но в данном случае оно неизбежно.

Когда ссылки на массивы переменного размера используются в цикле, компилятор часто может оптимизировать вычисление индекса, учитывая закономерности в организации доступа. Например, в листинге 3.18 (a) показан код C, вычисляющий эле-



мент  $i, k$  произведения двух массивов  $n \times n$  –  $A$  и  $B$ . Для этой функции GCC сгенерировал ассемблерный код, который мы переложили на язык C (листинг 3.18 (b)). Стиль этого кода отличается от оптимизированного кода, выполняющего перемножение массивов фиксированного размера (листинг 3.17), но это скорее следствие выбора, сделанного компилятором, а не фундаментальное требование двух разных функций. Код в листинге 3.18 (b) сохраняет переменную цикла  $j$  для определения момента завершения цикла и для индексации массива, состоящего из элементов  $i$ -й строки в  $A$ .

**Листинг 3.18.** Оригинальный и оптимизированный код для вычисления элемента  $i, k$  матричного произведения для массивов переменной длины. Компилятор выполняет эти оптимизации автоматически

(a) Оригинальный код на C

```
1 /* Вычисляет элементы i,k произведения матриц переменного размера */
2 int var_prod_ele(long n, int A[n][n], int B[n][n], long i, long k) {
3     long j;
4     int result = 0;
5
6     for (j = 0; j < n; j++)
7         result += A[i][j] * B[j][k];
8
9     return result;
10 }
```

(b) Оптимизированный код на C

```
/* Вычисляет элементы i,k произведения матриц переменного размера */
int var_prod_ele_opt(long n, int A[n][n], int B[n][n], long i, long k) {
    int *Arow = A[i];
    int *Bptr = &B[0][k];
    int result = 0;
    long j;
    for (j = 0; j < n; j++) {
        result += Arow[j] * *Bptr;
        Bptr += n;
    }
    return result;
}
```

Ниже приводится ассемблерный код цикла из `var_prod_ele`:

Регистры: $n$ в <code>%rdi</code> , $Arow$ в <code>%rsi</code> , $Bptr$ в <code>%rcx</code>	
$4n$ в <code>%r9</code> , $result$ в <code>%eax</code> , $j$ в <code>%rdx</code>	
1 .L24:	loop:
2 movl (%rsi,%rdx,4), %r8d	Прочитать $Arow[j]$
3 imull (%rcx), %r8d	Умножить на $*Bptr$
4 addl %r8d, %eax	Прибавить к $result$
5 addq \$1, %rdx	$j++$
6 addq %r9, %rcx	$Bptr += n$
7 cmpq %rdi, %rdx	Сравнить $j:n$
8 jne .L24	Если $!=$ , перейти к <code>loop</code>

Как видите, программа использует масштабированное значение  $4n$  (регистр `%r9`) для наращивания  $Bptr$  и значение  $n$  (регистр `%rdi`) для проверки условия завершения цикла. Необходимость в двух значениях неочевидна в коде на C из-за масштабирования арифметики указателей.

Из вышесказанного следует, что при включенной оптимизации GCC может распознавать закономерности обхода элементов многомерного массива в программе и генерировать код, не использующий операцию умножения, потребность в которой может возникнуть при прямом применении уравнения 3.1. Независимо от того, генерируется ли код на основе указателей, как в листинге 3.17 (b), или на основе массивов, как в листинге 3.18 (b), эти оптимизации могут значительно повысить производительность программы.

## 3.9. Структуры разнородных данных

В языке C существуют два механизма создания типов данных путем объединения объектов разных типов: *структуры*, которые объявляются с помощью ключевого слова `struct` и объединяют несколько объектов в единую конструкцию; и *объединения*, которые объявляются с помощью ключевого слова `union` и позволяют ссылаться на объекты, используя разные типы.

### 3.9.1. Структуры

Объявление `struct` в языке C создает тип данных, объединяющий объекты, возможно, разных типов, в один объект. К разным компонентам структуры можно обращаться по их именам. Реализация структур подобна реализации массивов в том смысле, что все компоненты структуры располагаются в смежных участках памяти, а указателем на структуру служит адрес ее первого байта. Компилятор сохраняет информацию о каждом типе структуры, в частности смещение в байтах каждого поля. Он генерирует ссылки на элементы структуры, используя эти смещения как сдвиги в инструкциях, ссылающихся на ячейки в памяти.

Например, рассмотрим следующее объявление структуры:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

Эта структура содержит четыре поля: два 4-байтных значения типа `int`, двухэлементный массив типа `int` и 8-байтный целочисленный указатель, что в сумме дает 24 байта:

Смещение	0	4	8	16	24
Содержимое	i	j	a[0]	a[1]	p

Обратите внимание, что массив `a` встроен в структуру. Числа в верхней части диаграммы указывают смещения полей в байтах от начала структуры.

Для доступа к полям структуры компилятор генерирует код, который добавляет соответствующее смещение к адресу структуры. Например, предположим, что переменная `r` типа `struct rec` \* находится в регистре `%rdi`. Тогда следующий код скопирует элемент `r->i` в элемент `r->j`:

```
Регистры: r в %rdi
1  movl    (%rdi), %eax    Получить r->i
2  movl    %eax, 4(%rdi)   Сохранить в r->j
```

Поскольку поле `i` имеет смещение 0, адрес этого поля совпадает со значением `r`. Чтобы сохранить значение в поле `j`, код добавляет смещение 4 к адресу `r`.

**Новичок в C? Представление объекта как структуры типа struct**

Тип данных `struct` в языке C очень близок к объектам в языках C++ и Java. Он позволяет программисту сохранять информацию о том или ином логическом объекте в общей структуре данных и ссылаться на них по именам элементов.

Например, графическая программа может представить прямоугольник как структуру:

```
struct rect {
    long llx;           /* Координата X нижнего левого угла */
    long lly;           /* Координата Y нижнего левого угла */
    unsigned long width; /* Ширина (в пикселях) */
    unsigned long height; /* Высота (в пикселях) */
    unsigned color;      /* Цвет */
};
```

Мы можем объявить переменную `r` типа `struct rect` и присвоить следующие значения полям:

```
struct rect r;
r.llx = r.lly = 0;
r.color = 0xFF00FF;
r.width = 10;
r.height = 20;
```

где выражение `r.llx` выбирает поле `llx` структуры `r`.

Также есть возможность объявить переменную и инициализировать ее поля в одной инструкции:

```
struct rect r = { 0, 0, 10, 20, 0xFF00FF };
```

Обычной практикой считается передача указателя на структуру вместо нее самой. Например, следующая функция вычисляет площадь прямоугольника, при этом в функцию передается указатель на структуру `struct rect`, а не сама структура:

```
long area(struct rect *rp) {
    return (*rp).width * (*rp).height;
}
```

Выражение `(*rp).width` разыменовывает указатель и выбирает поле `width` полученной структуры. Круглые скобки нужны для того, чтобы компилятор не мог интерпретировать выражение `*rp.width` как `*(rp.width)`, поскольку это неправильно. Сочетание разыменования и выбора поля настолько распространено, что в языке для этой цели предусмотрено специальное обозначение `->`. То есть выражение `rp->width` эквивалентно выражению `(*rp).width`. Например, мы можем написать функцию, которая поворачивает прямоугольник влево на 90°:

```
void rotate_left(struct rect *rp) {
    /* Поменять местами ширину и высоту */
    long t = rp->height;
    rp->height = rp->width;
    rp->width = t;
    /* Сдвинуть в новый нижний левый угол */
    rp->llx -= t;
}
```

Объекты в C++ и Java устроены сложнее, чем структуры в языке C, прежде всего из-за того, что они связывают с объектом набор *методов*, которые можно вызвать для выполнения тех или иных вычислений. В языке C мы можем оформить эти вычисления в виде обычных функций, таких как функция `area` или `rotate_left`, показанных выше.

Чтобы получить указатель на объект в структуре, достаточно прибавить смещение соответствующего поля к адресу структуры. Например, мы можем сгенерировать указатель `&(r->a[1])`, прибавив смещение  $8 + 4 \cdot 1 = 12$ . Имея указатель `r` в регистре `%rdi` и целочисленный индекс `i` в регистре `%rsi`, мы можем получить указатель `&(r->a[i])` с помощью всего лишь одной инструкции:

Регистры: `r` в `%rdi`, `i` в `%rsi`

```
1 leaq    8(%rdi,%rsi,4), %rax    Записать в %rax адрес &r->a[i]
```

В заключение приведем пример реализации оператора:

```
r->p = &r->a[r->i + r->j];
```

Первоначально `r` хранится в регистре `%rdi`:

Регистры: <code>r</code> в <code>%rdi</code>		
1	<code>movl    4(%rdi), %eax</code>	Получить <code>r-&gt;j</code>
2	<code>addl    (%rdi), %eax</code>	Прибавить <code>r-&gt;i</code>
3	<code>cltq</code>	Расширить до 8 байт
4	<code>leaq    8(%rdi,%rax,4), %rax</code>	Вычислить <code>&amp;r-&gt;a[r-&gt;i + r-&gt;j]</code>
5	<code>movq    %rax, 16(%rdi)</code>	Сохранить в <code>r-&gt;p</code>

Как показывают эти примеры, выборка различных полей структуры обрабатывается исключительно на этапе компиляции. Машинный код не содержит информации, касающейся объявлений полей или их имен.

#### Упражнение 3.41 (решение в конце главы)

Взгляните на следующее объявление структуры:

```
struct prob {
    int *p;
    struct {
        int x;
        int y;
    } s;
    struct prob *next;
};
```

Это объявление показывает, что одна структура может быть встроена в другую, подобно тому, как массивы могут встраиваться в структуры и в другие массивы.

Следующая процедура (некоторые выражения опущены) выполняет некоторые действия с этой структурой:

```
void sp_init(struct prob *sp) {
    sp->s.x = _____;
    sp->p   = _____;
    sp->next = _____;
}
```

1. Определите смещения в байтах следующих полей:

```
p:    _____
s.x:  _____
s.y:  _____
next: _____
```

2. Сколько байтов занимает эта структура в памяти?
3. Для `sp_init` компилятор сгенерировал следующий ассемблерный код:

```
void sp_init(struct prob *sp)
sp в %rdi
```

```

1 sp_init:
2     movl    12(%rdi), %eax
3     movl    %eax, 8(%rdi)
4     leaq    8(%rdi), %rax
5     movq    %rax, (%rdi)
6     movq    %rdi, 16(%rdi)
7     ret

```

Опираясь на этот код, восстановите отсутствующие выражения в исходном коде `sp_init`.

#### Упражнение 3.42 (решение в конце главы)

Ниже показано объявление типа структуры `ELE` и прототип функции `fun`:

```

struct ELE {
    long    v;
    struct ELE *p;
};

long fun(struct ELE *ptr);

```

При компиляции функции `fun` GCC сгенерировал следующий ассемблерный код:

```

    long fun(struct ELE *ptr)
    ptr в %rdi
1 fun:
2     movl    $0, %eax
3     jmp     .L2
4 .L3:
5     addq    (%rdi), %rax
6     movq    8(%rdi), %rdi
7 .L2:
8     testq   %rdi, %rdi
9     jne     .L3
10    rep; ret

```

1. Используйте свои навыки, воссоздайте исходный код функции `fun` на C.
2. Опишите, какие элементы данных реализует эта структура и какую операцию выполняет `fun`.

### 3.9.2. Объединения

Объединения – это способ обойти систему контроля типов в языке C. Объединения позволяют сослаться на конкретный объект, используя разные типы. Синтаксис объявления объединения идентичен синтаксису объявления структуры, но их семантика существенно различается. Поля объединения ссылаются не на разные, а на один и тот же блок памяти.

Рассмотрим следующие объявления:

```

struct S3 {
    char c;
    int i[2];
    double v;
};

union U3 {

```

```
char c;
int i[2];
double v;
};
```

При компиляции в архитектуре x86-64 получаются смещения полей и общие размеры типов данных S3 и U3, показанные в следующей таблице:

Тип	c	i	v	Размер
S3	0	4	16	24
U3	0	0	0	8

(Вскоре вы узнаете, почему i имеет смещение 4 в S3, а не 1, и почему v имеет смещение 16, а не 9 или 12.) Для указателя p типа union U3\* все ссылки: p->c, p->i[0] и p->v – будут ссылаться на начало структуры данных. Также обратите внимание, что общий размер объединения равен размеру самого большого из его полей.

Объединения могут пригодиться во множестве случаев. В то же время они могут оказаться источниками программных ошибок, потому что позволяют обойти защиту системы типов в языке C. Объединения можно использовать, когда известно заранее, что использование одного из полей исключает использование других. Если объявить два поля как часть объединения, можно сэкономить некоторое пространство памяти.

Например, предположим, что нам нужно построить бинарное дерево, в котором каждый лист содержит два значения типа double, а каждый узел – два указателя на два дочерних узла, но не содержит данных. Если объявить такое дерево как структуру

```
struct node_s {
    struct node_s *left;
    struct node_s *right;
    double data[2];
};
```

то для хранения каждого узла потребуется 32 байта, при этом половина этих байтов будет пустовать в каждом конкретном узле. Но если объявить тип узла как объединение

```
union node_u {
    struct {
        union node_u *left;
        union node_u *right;
    } internal;
    double data[2];
};
```

то каждый узел будет занимать всего 16 байт. Если принять, что n – это указатель на узел типа union node\_u \*, то его можно использовать для ссылки на данные в листе как n->data[0] и n->data[1], а на ветви внутреннего узла – как n->internal.left и n->internal.right.

Однако при таком подходе невозможно определить, является ли данный узел листом или внутренним узлом. В этом случае обычно добавляется поле типа перечисления, определяющее разные варианты объединения, и создается структура, включающая это поле и объединение:

```
typedef enum { N_LEAF, N_INTERNAL } nodetype_t;

struct node_t {
    nodetype_t type;
    union {
        struct {
            struct node_t *left;
```

```

        struct node_t *right;
    } internal;
    double data[2];
} info;
};

```

Такая структура занимает в памяти 24 байта: 4 байта для `type` и либо по 8 байт для `info.internal.left` и `info.internal.right`, либо 16 байт для `info.data`. Как мы расскажем ниже, между полем `type` и элементами объединения добавляются дополнительные 4 байта, в результате чего общий размер структуры составляет  $4 + 4 + 16 = 24$ . В данном случае экономия памяти получается незначительная и не оправдывает увеличившейся громоздкости кода. Однако при использовании структур данных с большим числом полей экономия может оказаться более существенной.

Объединения можно использовать для доступа к битовым комбинациям разных типов данных. Например, представим, что мы используем простое приведение типа для преобразования значения `d` типа `double` в значение `u` типа `unsigned long`:

```
unsigned long u = (unsigned long) d;
```

Значение `u` будет представлять целую часть `d`. За исключением случая, когда `d` равно 0,0, битовое представление `u` будет сильно отличаться от битового представления `d`. Теперь рассмотрим следующий код, получающий значение типа `unsigned long` из числа `double`:

```

unsigned long double2bits(double d) {
    union {
        double d;
        unsigned long u;
    } temp;
    temp.d = d;
    return temp.u;
};

```

В этом примере мы переписываем аргумент в объединение, используя один тип данных, и затем читаем его, используя другой тип. В результате `u` будет иметь то же битовое представление, что и `d`, включая поля знакового бита, показателя степени и мантиссы, как описано в разделе 3.11. Числовое значение `u` не будет иметь ничего общего с числовым значением `d`, за исключением случая, когда `d` равно 0,0.

При использовании объединений с типами данных разного размера проблемы упорядочения байтов становятся еще более актуальными. Например, предположим, что мы написали процедуру, которая создает 8-байтное значение `double`, используя комбинации битов, заданные двумя 4-байтными значениями типа `unsigned`:

```

double uu2double(unsigned word0, unsigned word1)
{
    union {
        double d;
        unsigned u[2];
    } temp;

    temp.u[0] = word0;
    temp.u[1] = word1;
    return temp.d;
}

```

В архитектурах с обратным порядком следования байтов (*little-endian*), таких как x86-64, аргумент `word0` разместится в четырех младших байтах значения `d`, а `word1` – в четырех старших байтах. В архитектурах с прямым порядком следования байтов (*big-endian*) роли аргументов меняются.

**Упражнение 3.43 (решение в конце главы)**

Представьте, что вам поручили проверить, насколько правильный код доступа к структуре и объединению генерирует компилятор C. Для этого вы объявляете следующую структуру:

```
typedef union {
    struct {
        long u;
        short v;
        char w;
    } t1;
    struct {
        int a[2];
        char *p;
    } t2;
} u_type;
```

и пишете серию функций следующего вида:

```
void get(u_type *up, mun *dest) {
    *dest = выражение;
}
```

с разными *выражениями* доступа и *типами* аргумента *dest*, соответствующими типам *выражений* доступа. Затем вы исследуете код, сгенерированный компилятором, и пытаетесь определить, соответствует ли он вашим ожиданиям.

Предположим, что эти функции получают аргументы *up* и *dest* в регистрах *%rdi* и *%rsi* соответственно. Подставьте в пустующие ячейки в следующей таблице *mun* и последовательности из одной-трех инструкций, вычисляющие *выражение* и сохраняющие результат в *dest*.

Выражение	Тип	Код
up->t1.u	long	movq (%rdi), %rax movq %rax, (%rsi)
up->t1.v	-----	-----
&up->t1.w	-----	-----
up->t2.a	-----	-----
up->t2.a[up->t1.u]	-----	-----
*up->t2.p	-----	-----

### 3.9.3. Выравнивание

Многие компьютерные системы накладывают ограничения на допустимые адреса для данных простых типов, требуя, чтобы адреса объектов некоторых типов были кратны некоторому заданному значению *K* (обычно 2, 4 или 8). Соблюдение таких *требований к выравниванию* упрощает конструирование аппаратных компонентов, обеспечивающих интерфейс между процессором и системой памяти. Например, предположим, что



при каждом обращении к памяти процессор извлекает сразу 8 байт по адресу, кратному 8. Если мы сможем гарантировать, что любое значение `double` будет выровнено так, что его адрес всегда будет кратным 8, то процессор сможет прочитать или записать это значение одним обращением к памяти. В противном случае ему придется обратиться к памяти дважды, потому что объект может находиться в двух 8-байтных блоках памяти.

Аппаратные средства с архитектурой x86-64 будут работать правильно независимо от выравнивания данных. Тем не менее компания Intel рекомендует выравнивать данные, чтобы получить более высокую производительность системы памяти. Их требования к выравниванию основаны на принципе, что любой простой объект с размером  $K$  байт должен иметь адрес, кратный  $K$ . Это требование приводит к следующим выравниваниям:

$K$	Типы
1	<code>char</code>
2	<code>short</code>
4	<code>int</code> , <code>float</code>
8	<code>long</code> , <code>double</code> , <code>char *</code>

Выравнивание автоматически выполняется, если гарантируется, что каждый тип организован и размещен в памяти так, что каждый объект конкретного типа удовлетворяет требованиям выравнивания. Компилятор помещает в ассемблерный код соответствующие директивы, указывая желаемое выравнивание для глобальных данных. Например, ассемблерный код с таблицей переходов (раздел 3.6.8) содержит в строке 2 следующую директиву:

```
.align 8
```

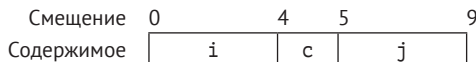
Она гарантирует, что данные, определяемые за ней (в данном случае начало таблицы переходов), будут начинаться с адресов, кратных 8. Поскольку каждый элемент таблицы имеет длину 8 байт, то все последующие элементы подчиняются условию выравнивания по границам 8-байтных блоков.

При компиляции структур компилятору часто приходится делать отступы при размещении полей в памяти, чтобы каждый элемент структуры удовлетворял требованиям к выравниванию. В таких случаях выравнивается также и начальный адрес структуры.

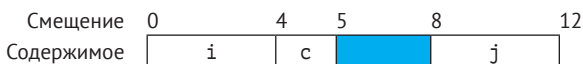
Рассмотрим, например, следующее объявление структуры:

```
struct S1 {
    int i;
    char c;
    int j;
};
```

Предположим, что компилятор использовал минимальное 9-байтное распределение, как показано на следующей диаграмме:



В этом случае невозможно удовлетворить требование к выравниванию по 4-байтной границе для обоих полей `i` (смещение 0) и `j` (смещение 5). Вместо этого компилятор вставляет три байта (в следующей диаграмме показаны в виде заштрихованного прямоугольника) между полями `c` и `j`.



В результате поле  $j$  получает смещение 8, а общий размер увеличивается до 12 байт. Более того, компилятор должен сделать так, чтобы любой указатель  $p$  типа `struct S1 *` удовлетворял требованию выравнивания по границам 4-байтных блоков. Используя прежние обозначения, положим, что указатель  $p$  имеет значение  $x_p$ . Тогда  $x_p$  должно быть кратным 4. Это гарантирует, что  $p \rightarrow i$  (адрес  $x_p$ ) и  $p \rightarrow j$  (адрес  $x_p + 8$ ) будут удовлетворять требованию выравнивания по 4-байтной границе.

В дополнение к этому компилятору может потребоваться добавить пустые байты в конец структуры, чтобы каждый элемент массива структур удовлетворял требованию выравнивания. Например, рассмотрим следующее объявление структуры:

```
struct S2 {
    int i;
    int j;
    char c;
};
```

Если отвести под эту структуру 9 байт, мы все еще сможем выполнить требование к выравниванию полей по 4-байтным границам, если начальный адрес структуры будет кратен 4. Теперь рассмотрим следующее объявление:

```
struct S2 d[4];
```

Если под каждую структуру в массиве выделять 9 байт, то условие выравнивания для каждого элемента в массиве  $d$  будет нарушено, потому что элементы будут иметь адреса  $x_d$ ,  $x_d + 9$ ,  $x_d + 18$  и  $x_d + 27$ . Вместо этого компилятор выделит для каждой структуры 12 байт, добавив в конец по 3 неиспользуемых байта:

Смещение	0	4	8	9
Содержимое	i	j	c	

Как результат элементы массива  $d$  будут иметь адреса  $x_d$ ,  $x_d + 12$ ,  $x_d + 24$ ,  $x_d + 36$ . Если адрес  $x_d$  кратен четырем, то требование к выравниванию будет выполняться для всех элементов.

#### Упражнение 3.44 (решение в конце главы)

Для каждого из следующих объявлений структур определите смещение каждого поля, общий размер структуры и предъявляемые к ним требования выравнивания в архитектуре x86-64.

1. `struct P1 { int i; char c; int j; char d; };`
2. `struct P2 { int i; char c; char d; long j; };`
3. `struct P3 { short w[3]; char c[3] };`
4. `struct P4 { short w[5]; char *c[3] };`
5. `struct P5 { struct P3 a[2]; struct P2 t };`

#### Упражнение 3.45 (решение в конце главы)

Взгляните на следующее объявление структуры:

```
struct {
    char *a;
    short b;
```

```
double c;
char d;
float e;
char f;
long g;
int h;
} rec;
```

и ответьте на вопросы:

1. Какое смещение в байтах имеет каждое поле?
2. Какой размер имеет эта структура?
3. Переупорядочьте поля в структуре, чтобы минимизировать объем напрасно расходимой памяти, а затем определите смещения полей и общий размер переупорядоченной структуры.

### Случай обязательного выравнивания

Поддержка выравнивания повышает эффективность большинства инструкций x86-64, но не влияет на поведение программы. Однако некоторые модели процессоров Intel и AMD будут неправильно работать при применении мультимедийных инструкций SSE к данным, хранящимся в памяти без выравнивания. Эти инструкции работают с 16-байтными блоками данных, а инструкции, передающие данные между блоком SSE и памятью, требуют, чтобы адреса данных в памяти были кратны 16. Любая попытка обратиться к памяти с адресом вразрез с этим требованием приведет к *исключению* (см. раздел 8.1), которое по умолчанию влечет за собой завершение программы.

Соответственно, любой компилятор и система времени выполнения для процессора x86-64 должны гарантировать, что любая память, выделенная для хранения структуры данных, которая может извлекаться или сохраняться в регистре SSE, должна удовлетворять требованию к выравниванию по 16-байтной границе. Это требование имеет следующие два следствия:

- начальный адрес любого блока, сгенерированного функцией выделения памяти (`alloca`, `malloc`, `calloc` или `realloc`), должен быть кратен 16;
- кадр стека для большинства функций должен быть выровнен по 16-байтной границе. (Это требование имеет ряд исключений.)

Более поздние версии процессоров x86-64 реализуют мультимедийные инструкции AVX. Помимо предоставления расширенного набора инструкций SSE, процессоры, поддерживающие AVX, также не имеют обязательного требования к выравниванию.

## 3.10. Комбинирование инструкций управления потоком выполнения и передачи данных в машинном коде

К настоящему моменту мы узнали, как в машинном коде реализуются управление потоком выполнения программы и различные структуры данных. В этом разделе мы рассмотрим способы организации взаимодействий между данными и инструкциями управления. Для начала детально исследуем указатели – одну из самых важных концепций языка программирования C, о которой тем не менее многие программисты имеют весьма поверхностное представление. Затем познакомимся с приемами использования символьного отладчика GDB для изучения особенностей работы машинного кода. Потом мы покажем, как понимание программ в машинном коде позволяет

находить ошибку переполнения буфера – уязвимость, существующую во многих реальных системах. Наконец, исследуем, как программы машинного уровня реализуют случаи, когда объем памяти в стеке, необходимый функции, может меняться между вызовами.

### 3.10.1. Указатели

Указатели являются одной из центральных особенностей языка C. Они обеспечивают универсальный способ ссылки на элементы в разных структурах данных. Указатели часто оказываются источником недопонимания для начинающих программистов, однако идеи, положенные в основу указателей, достаточно просты. Далее представлены некоторые ключевые принципы указателей и показано, как они выражаются в машинном коде:

- *каждый указатель имеет тип.* Этот тип подсказывает тип объекта, на который ссылается указатель. Возьмем для примера следующие объявления указателей:

```
int *ip;
char **cpp;
```

В этом примере переменная `ip` – это указатель на объект типа `int`, а переменная `cpp` – указатель на объект, который сам является указателем на объект типа `char`. В общем случае, если объект имеет тип  $T$ , указатель имеет тип  $T^*$ . Специальный тип `void *` представляет обобщенный указатель. Например, функция `malloc` возвращает обобщенный указатель, который преобразуется в типизированный указатель посредством явного или неявного приведения типа. Типы указателей отсутствуют в машинном коде – эта абстракция предлагается в помощь программистам только на уровне языка C, чтобы помочь им избежать ошибок адресации;

- *каждый указатель имеет значение.* Этим значением является адрес некоторого объекта заданного типа. Специальное значение `NULL` (0) соответствует ситуации, когда указатель ни на что не указывает;
- *указатели создаются с помощью оператора &.* Этот оператор может быть применен к любому выражению на языке C, которое относится к категории *левосторонних* (`lvalue`) выражений, которые могут указываться слева от оператора присваивания. Примерами могут служить элементы структур, объединений и массивов. Мы уже видели, что в машинном коде для вычисления выражения при операторе `&` часто используется инструкция `leaq`, которая изначально предназначена для вычисления адресов в памяти;
- *указатели разыменовываются с помощью оператора \*.* Результатом является значение, имеющее тип, ассоциированный с указателем. Разыменование осуществляется ссылкой на память, куда сохраняется или откуда извлекается значение;
- *массивы и указатели тесно связаны друг с другом.* На имя массива можно ссылаться, как если бы это была переменная-указатель (с той лишь разницей, что значение этой «переменной» нельзя изменить). Ссылка на массив (например, `a[3]`) дает тот же результат, что и арифметическая операция с указателем и последующим разыменованием (например, `*(a+3)`). Ссылки на массивы и арифметические операции с указателями требуют масштабирования смещений с учетом размера объектов. Когда мы пишем выражение `p+i` для указателя `p` со значением `p`, то адрес вычисляется как  $p + L \cdot i$ , где  $L$  – размер типа данных, связанного с `p`;
- *приведение указателя к другому типу изменяет его тип, но не значение.* В результате приведения типа может измениться коэффициент масштабирования, ис-

пользуемый в арифметических операциях с указателем. Например, если  $p$  – указатель типа `char *`, имеющий значение  $p$ , то выражение `(int *)p+7` фактически будет вычислено как  $p + 28$ , а `(int * (p + 7))` – как  $p + 7$  (напомним, эта операция приведения типа имеет более высокий приоритет, чем операция сложения);

- *указатели могут также ссылаться на функции.* Это открывает широкие возможности для сохранения и передачи ссылок на программный код, который можно вызвать в другой части программы. Например, если имеется функция со следующим прототипом:

```
int fun(int x, int *p);
```

то тогда можно объявить указатель и присвоить ему адрес этой функции, как показано ниже:

```
int (*fp)(int, int *);
fp = fun;
```

После этого функцию можно вызвать по указателю:

```
int y = 1;
int result = fp(3, &y);
```

Значением указателя на функцию является адрес первой инструкции в машинном коде, реализующем эту функцию.

#### Новичок в C? Указатели на функции

Синтаксис объявления указателей на функции выглядит сложным для начинающих программистов. Например, такие объявления, как

```
int (*f)(int*);
```

полезно читать, начиная изнутри (с  $f$ ) наружу. При таком подходе ясно видно, что  $f$  – это указатель, о чем свидетельствует часть `(*)`; что это указатель на функцию, о чем говорит пара круглых скобок `(int*)`, следующих за указателем; что эта функция принимает один аргумент типа `int *` и возвращает результат типа `int`.

Скобки, окружающие `*f`, совершенно необходимы, потому что иначе получится объявление

```
int *f(int*);
```

которое можно прочесть так:

```
(int *) f(int*);
```

То есть оно будет интерпретировано компилятором как прототип функции  $f$ , которая принимает аргумент `int *` и возвращает результат `int *`.

Керниган (Kernighan) и Ритчи (Ritchie) [61, разд. 5.12] представляют весьма полезное учебное пособие по чтению объявлений на языке C.

### 3.10.2. Жизнь в реальном мире: использование отладчика GDB

Отладчик GNU GDB предлагает ряд полезных возможностей для оценки и анализа работы программ машинного уровня во время их выполнения. В примерах и упражнениях, приводимых в данной книге, мы не раз пытались оценить поведение программы только на основании анализа программного кода. Отладчик GDB позволяет изучать поведение программы, наблюдая за ней и сохраняя при этом значительный контроль над ее выполнением.

В табл. 3.14 представлены примеры некоторых команд отладчика GDB, которые помогут вам в отладке программ, скомпилированных для архитектуры x86-64. Для начала полезно запустить утилиту OBJDUMP, чтобы получить дизассемблированную версию программы. Все последующие примеры использования GDB основаны на применении программы в файле prog, дизассемблированный код которой приводился в конце раздела 3.2.2. Мы запускаем отладчик GDB следующей командой:

```
linux> gdb prog
```

**Таблица 3.14.** Примеры команд отладчика GDB. Эти примеры иллюстрируют некоторые приемы отладки программ с помощью GDB

Команда	Назначение
<b>Запуск и останов</b>	
quit	Выйти из отладчика GDB
run	Запустить программу (с этой командой можно передать программе аргументы командной строки)
kill	Остановить программу
<b>Контрольные точки</b>	
break multstore	Установить контрольную точку на входе в функцию multstore
break *0x400540	Установить контрольную точку на инструкции с адресом 0x400540
delete 1	Удалить контрольную точку 1
delete	Удалить все контрольные точки
<b>Выполнение</b>	
stepi	Выполнить одну инструкцию
stepi 4	Выполнить четыре инструкции
nexti	Действует подобно stepi, но не входит в функции
continue	Возобновить выполнение
finish	Продолжить выполнение до конца текущей функции
<b>Анализ кода</b>	
disas	Дизассемблировать текущую функцию
disas multstore	Дизассемблировать функцию multstore
disas 0x400544	Дизассемблировать функцию, включающую адрес 0x400544
disas 0x400540, 0x40054d	Дизассемблировать код между указанными адресами
print /x \$rip	Вывести значение счетчика инструкций в шестнадцатеричном виде
<b>Анализ данных</b>	
print \$rax	Вывести содержимое %rax в десятичном виде
print /x \$rax	Вывести содержимое %rax в шестнадцатеричном виде
print /t \$rax	Вывести содержимое %rax в двоичном виде
print 0x100	Вывести 0x100 в десятичном представлении
print /x 555	Вывести 555 в шестнадцатеричном представлении
print /x (\$rsp+8)	Вывести содержимое ячейки памяти с адресом (%rsp+8)
print *(long *) 0x7fffffff818	Вывести длинное целое, хранящееся в памяти с адресом 0x7fffffff818
print *(long *) (\$rsp+8)	Вывести длинное целое, хранящееся в памяти с адресом (%rsp+8)
x/2g 0x7fffffff818	Вывести два (8-байтных) слова, хранящихся в памяти, начиная с адреса 0x7fffffff818
x/20b multstore	Вывести первые 20 байт функции multstore

Команда	Назначение
<b>Полезная информация</b>	
info frame	Вывести информацию о текущем кадре стека
info registers	Вывести значения всех регистров
help	Вывести справочную информацию о GDB

Основная схема отладки заключается в расстановке контрольных точек в местах, представляющих наибольший интерес. Их можно устанавливать в точки входа в функции или по адресам инструкций. Когда во время выполнения программа доходит до одной из контрольных точек, она останавливается и передает управление пользователю. После останова программы в контрольной точке мы можем исследовать состояние регистров процессора и ячеек памяти. Также можно перейти в режим пошагового выполнения и выполнить несколько инструкций по одной или позволить программе продолжить выполнение до достижения следующей контрольной точки.

Как можно заключить из нашего примера, отладчик GDB обладает не совсем четким синтаксисом команд, однако оперативная справка (доступна в GDB как команда `help`) помогает нивелировать этот недостаток. В последние годы многие программисты предпочитают использовать DDD – расширение для GDB, предоставляющее графический интерфейс.

### 3.10.3. Ссылки на ячейки за границами выделенной памяти и переполнение буфера

Выше мы видели, что язык C никак не проверяет выход ссылок за границы массивов и что локальные переменные хранятся в стеке вместе с информацией о состоянии, такой как значения регистров и адреса возврата. Такое сочетание может привести к серьезным программным ошибкам, если состояние, хранимое в стеке, окажется повреждено в результате записи значения в элемент массива, находящийся за его границами. Попытка программы восстановить регистр или выполнить команду `ret` с подобным поврежденным состоянием может привести к серьезным последствиям.

Наиболее частой причиной повреждения состояния является *переполнение буфера*. Довольно часто в стек помещаются символьные массивы, предназначенные для хранения строк, но иногда размер строки может превысить размер выделенного для нее массива. Это можно проиллюстрировать на примере следующей программы:

```
/* Реализация библиотечной функции gets() */
char *gets(char *s)
{
    int c;
    char *dest = s;

    while ((c = getchar()) != '\n' && c != EOF)
        *dest++ = c;
    if (c == EOF && dest == s)
        /* Нет символов для чтения */
        return NULL;
    *dest++ = '\0'; /* Добавить признак конца строки */
    return s;
}

/* Прочитать введенную строку и вывести ее обратно */
void echo()
{

```

```

char buf[8]; /* Буфер слишком маленький! */
gets(buf);
puts(buf);
}

```

Предыдущий код представляет реализацию библиотечной функции `gets` и демонстрирует серьезную проблему в этой функции. Она читает строку из стандартного ввода и останавливается, только если встретит символ перевода строки или возникнет какая-то ошибка. Она копирует строку в память, на которую ссылается аргумент `s`, и завершает прочитанную строку пустым символом (`'\0'`). Мы покажем, как пользоваться функцией `gets` в функции `echo`, которая просто читает строку со стандартного ввода и передает ее в стандартный вывод.

Проблема функции `gets` в том, что она не имеет возможности определить, достаточно ли отведено памяти для сохранения всей строки. В нашем примере с функцией `echo` мы намеренно выбрали буфер очень маленьким: он рассчитан на хранение всего лишь восьми символов. Любая строка, длина которой больше семи символов, вызовет запись за пределы отведенного массива.

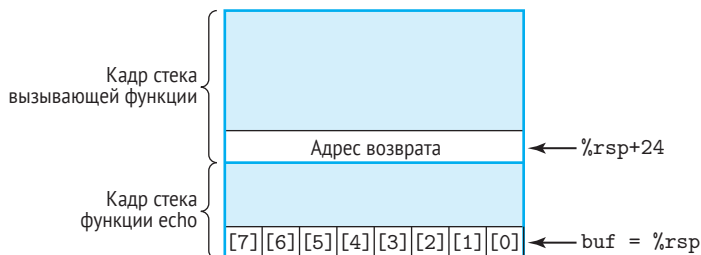
Анализ ассемблерного кода, сгенерированного при компиляции функции `echo`, позволяет понять, как организован стек:

```

void echo()
1 echo:
2   subq    $24, %rsp      Выделить 24 байта на стеке
3   movq    %rsp, %rdi     Адрес buf -- как адрес в %rsp
4   call    gets          Вызвать gets
5   movq    %rsp, %rdi     Адрес buf -- как адрес в %rsp
6   call    puts          Вызвать puts
7   addq    $24, %rsp      Освободить пространство на стеке
8   ret              Возврат

```

Рисунок 3.8 иллюстрирует организацию стека функции `echo`. Программа выделила 24 байта на стеке, уменьшив указатель стека на 24 (строка 2). Массив символов `buf` располагается на вершине стека, как следует из того факта, что содержимое `%rsp` копируется в `%rdi` для передачи в аргументе функциям `gets` и `puts`. 16 байт между `buf` и адресом возврата не используются.



**Рис. 3.8.** Организация стека функции `echo`. Массив символов `buf` находится непосредственно под хранимым состоянием. Запись за пределы массива `buf` может повредить состояние программы

Пока пользователь вводит не больше семи символов, строка, возвращаемая функцией `gets` (включая завершающий пустой символ), уместается в пространстве, выделенном для `buf`. Однако при вводе более длинной строки произойдет перезапись некоторой информации, хранящейся в стеке. С увеличением длины введенной строки повреждается следующая информация:



Количество введенных символов	Повреждаются
0–7	Ничего
8–23	Неиспользуемое пространство в стеке
24–31	Адрес возврата
32+	Состояние вызывающей функции

Ввод строки с длиной до 23 символов не влечет никаких серьезных последствий, но если длина строки превысит это значение, то будет поврежден адрес возврата и, возможно, другое сохраненное состояние. Если будет поврежден адрес возврата, то инструкция `ret` (строка 8) заставит программу перейти в совершенно неожиданное место. Ничего из этого неочевидно из кода на С. Возможность записи в память за пределами допустимых границ такими функциями, как `gets`, можно заметить только при изучении машинного кода программы.

Наша функция `echo` имеет простую, но ущербную реализацию. Более совершенная версия предусматривает использование функции `fgets`, которая принимает дополнительный аргумент с максимальным числом читаемых символов. В упражнении 3.71 вам будет предложено написать функцию `echo`, способную обрабатывать строки произвольной длины. В общем случае использование функции `gets` или любой другой, способной вызвать переполнение буфера, считается дурным тоном в программировании. К сожалению, существует еще несколько часто используемых библиотечных функций, включая `strcpy`, `strcat` и `sprintf`, которые способны сгенерировать последовательность байтов произвольной длины и не имеют аргументов, с помощью которых им можно было бы сообщить максимальный размер буфера [97]. Использование таких функций может привести к появлению уязвимостей переполнения буфера.

**Упражнение 3.46 (решение в конце главы)**

В листинге 3.19 показана (некачественная) реализация функции, которая читает строку из стандартного ввода, копирует ее во вновь выделенное пространство памяти и возвращает указатель на результат.

Рассмотрим следующий сценарий: программа вызывает процедуру `get_line` и помещает на стек адрес возврата `0x400776`, регистр `%rbx` хранит значение

`0x0123456789ABCDEF`.

Вы вводите с клавиатуры строку

`0123456789012345678901234`

Программа останавливается из-за ошибки сегментации. Вы запускаете отладчик GDB и выясняете, что ошибка произошла во время выполнения команды `ret` в `get_line`.

1. Заполните следующую таблицу, поместив в нее всю информацию о состоянии стека после выполнения инструкции в строке 3. Подпишите числовые значения, хранящиеся в стеке (например, «адрес возврата»), и укажите в ячейке справа их шестнадцатеричные значения (если известны). Каждая ячейка соответствует 8 байтам. Укажите позицию регистра `%rsp`. Напомним, что символы 0–9 имеют коды ASCII `0x30–0x39`.

00 00 00 00 00 40 00 76	Адрес возврата

2. Дополните таблицу, добавив в нее информацию о состоянии стека после вызова функции `gets` (строка 5).
3. К какому адресу попытается вернуться `get_line`?
4. Значения каких регистров будут повреждены при попытке `get_line` вернуть управление?
5. Кроме возможности переполнения буфера, какие еще два недостатка имеет функция `get_line`?

### Листинг 3.19. Программный код на С и на ассемблере для упражнения 3.46

(a) Код на С

```
/* Очень некачественный код.
   Он служит иллюстрацией плохой практики программирования.
   См. упражнение 3.46. */
char *get_line()
{
    char buf[4];
    char *result;
    gets(buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return result;
}
```

(b) Дизассемблированный код вплоть до вызова `gets`

```
char *get_line()
1  0000000000400720 <get_line>:
2  400720: 53                push %rbx
3  400721: 48 83 ec 10       sub $0x10,%rsp
   Создайте таблицу с информацией о стеке для этой точки
4  400725: 48 89 e7          mov %rsp,%rdi
5  400728: e8 73 ff ff ff   callq 4006a0 <gets>
   Добавьте в таблицу информацию о стеке для этой точки
```

Последствия могут быть гораздо более разрушительными, чем переполнение буфера, если заставить программу выполнять не свойственную ей функцию. Это один из широко распространенных способов атак на системы в компьютерных сетях. Обычно программе передается строка, содержащая байты выполняемого кода, который еще называют *подставным кодом* (эксплойтом – exploit code), плюс несколько дополнительных байтов, подменяющих адрес возврата адресом подставного кода. В результате при выполнении инструкции `ret` происходит переход на подставной код.

В одной из форм подобных атак подставной код обращается к системному вызову и запускает командную оболочку, которая предоставляет злоумышленнику возможность пользоваться некоторым множеством функций операционной системы. В другой форме подставной код выполняет некоторые другие несанкционированные задачи, восстанавливает стек и выполняет команду `ret` во второй раз, благодаря чему достигается иллюзия нормального возврата управления вызывающей программе.

В качестве примера можно привести знаменитого интернет-червя, появившегося в ноябре 1988 года, который использовал четыре разных способа получения доступа к компьютерам. Одним из способов была атака на основе переполнения буфера в демоне `fingerd`, который обслуживает запросы со стороны команды `FINGER`. Вызывая команду `FINGER` со специально подготовленной строкой, программа-червь могла вызвать пере-

полнение буфера в демоне на удаленном сайте и выполнить программный код, открывающий червь доступ к удаленной системе. Получив доступ к системе, червь начинал тиражировать себя и потреблять практически все вычислительные ресурсы машины. В результате сотни машин были фактически парализованы, пока специалисты по обеспечению безопасности не нашли способа уничтожить червя. Автор этой программы был изобличен и осужден. Его приговорили к трем годам заключения условно, 400 часам общественно полезных работ и к штрафу в 10 500 долларов. Однако и по сей день находятся люди, ищущие слабые места в системах, которые позволили бы им проводить атаки переполнением буфера. Все это лишний раз подчеркивает необходимость более внимательного отношения к программированию. Любые интерфейсы с внешними системами должны быть «пуленепробиваемыми», чтобы никакие действия внешних агентов не могли заставить систему изменить свое поведение.

### Черви и вирусы

Черви и вирусы – это фрагменты программного кода, которые стремятся распространиться на другие компьютеры. Согласно определению, сформулированному Спаффордом (Spafford) [105], *червь* (worm) – это программа, которая может самостоятельно запускаться и распространять работоспособную версию самой себя на другие машины. *Вирус* (virus) – это фрагмент кода, который добавляет себя в другие программы, в том числе и в операционные системы. Вирус не может выполняться самостоятельно. В средствах массовой информации термин «вирус» используется как общее название различных стратегий для распространения злонамеренного кода среди систем, поэтому люди часто называют «вирусом» то, что правильнее было бы назвать «червем».

## 3.10.4. Предотвращение атак методом переполнения буфера

Атаки методом переполнения буфера стали настолько распространенными и вызвали столько проблем с компьютерными системами, что в современные компиляторы и операционные системы были внедрены механизмы противодействия, затрудняющие проведение этих атак и ограничивающие способы, которыми злоумышленник может получить контроль над системой через атаку переполнением буфера. В этом разделе мы представим механизмы, появившиеся в последних версиях GCC для Linux.

### Рандомизация стека

Чтобы заставить систему выполнить подставной код, злоумышленник должен внедрить в систему и сам код, и его адрес. Для создания указателя необходимо знать адрес в стеке, где будет находиться строка. Традиционно адрес стека в программах было легко предсказать. Для всех систем, использующих одну и ту же комбинацию программы и версии операционной системы, местоположение стека было стабильным. Так, например, если злоумышленник сможет определить адреса стека обычного веб-сервера, то он сможет организовать атаку, которая увенчается успехом на многих машинах. Используя аналогию с инфекционными заболеваниями, можно сказать, что многие системы уязвимы для одного и того же штамма вируса, это явление часто называют *монокультурой безопасности* [96].

Идея *рандомизации стека* заключается в выборе разного местоположения стека при каждом следующем запуске программы. Таким образом, один и тот же код, выполняясь на многих машинах, будет использовать разные адреса для стека. Это реализуется путем выделения в стеке пространства случайного размера от 0 до  $n$  байт в начале программы, например с помощью функции `alloca`, которая выделяет пространство указанного размера в стеке. Это пространство не используется программой, но из-за его

наличия все последующие данные располагаются в стеке по разным адресам в разных запусках программы. Диапазон размера выделяемого пространства  $n$  должен быть достаточно большим, чтобы получить достаточно большие вариации в адресах стека, но достаточно маленьким, чтобы не тратить впустую слишком много места.

Следующий код демонстрирует простой способ определения «типичного» адреса стека:

```
int main() {
    long local;
    printf("local at %p\n", &local);
    return 0;
}
```

Этот код просто выводит адрес локальной переменной в функции `main`. Мы провели эксперимент, запустив эту программу 10 000 раз на машине Linux в 32-разрядном режиме, и получили вариации адресов от `0xff7fc59c` до `0xffffd09c`, то есть ширина диапазона составила около  $2^{23}$ . На более современной машине в 64-битном режиме адреса варьировались от `0x7fff0001b698` до `0x7fffffaa4a8`, то есть ширина диапазона составила почти  $2^{32}$ .

Рандомизация стека стала стандартной практикой в системах Linux. Это один из более широкого класса методов, известных как *рандомизация разметки адресного пространства* (Address-Space Layout Randomization, ASLR) [99]. С использованием ASLR различные части программы, включая программный код самой программы, библиотеки, стек, глобальные переменные и динамическая память размещаются в разных областях памяти при каждом следующем запуске программы. Это означает, что одна и та же программа, запущенная на разных компьютерах, будет использовать очень разные адреса в памяти, что может предотвратить или сильно усложнить некоторые виды атак.

Однако очень упорный злоумышленник может преодолеть рандомизацию, используя метод грубого перебора, предпринимая многократные попытки атак с разных адресов. Распространенный трюк – включить длинную последовательность инструкций `por` (произносится как «ноу оп», сокращение от «no operation» – «нет операции») перед фактическим подставным кодом (кодом эксплойта). Эта инструкция не выполняет никаких действий, кроме последовательного увеличения счетчика инструкций. Если злоумышленнику удастся угадать и адрес окажется где-то в середине этой последовательности, то программа выполнит ее и затем попадет в подставной код. Такую последовательность инструкций `por` обычно называют «por sled» (салазки `por`) [97] – этот термин отражает идею «скатывания» программы по последовательности инструкций `por` к подставному коду, как с горы на салазках. Если злоумышленник использует 256-байтные салазки `por`, то рандомизацию по  $n = 2^{23}$  можно взломать путем перебора  $2^{15} = 32\,768$  начальных адресов, что вполне возможно при определенном упорстве. Перечислить  $2^{24} = 16\,777\,216$  адресов для 64-разрядного случая значительно сложнее. Как видите, рандомизация стека и другие приемы ASLR могут потребовать от злоумышленников прикладывать серьезные усилия для успешной атаки на систему и, следовательно, значительно снизить скорость распространения вируса или червя, но не способны обеспечить абсолютную защиту.

#### Упражнение 3.47 (решение в конце главы)

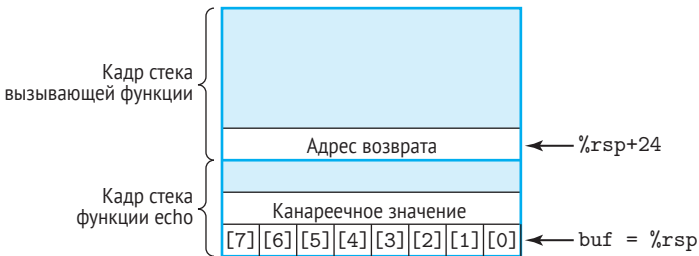
Запустив нашу программу проверки стека 10 000 раз в системе Linux версии 2.6.16, мы получили адреса в диапазоне от минимального `0xffffb754` до максимального `0xffffd754`.

1. Вычислите примерную ширину диапазона адресов.
2. Если попытаться использовать прием переполнения буфера со 128-байтными салазками `por`, сколько попыток потребуется для проверки всех начальных адресов?

## Обнаружение повреждения стека

Вторая линия обороны – возможность определить факт повреждения стека. На примере функции `echo` (рис. 3.8) мы видели, что повреждение обычно происходит, когда программа выходит за пределы отведенного буфера. В языке С нет надежного способа предотвратить запись за пределы массива. Однако программа может попытаться определить факт такой записи до наступления каких-либо вредных последствий.

Последние версии GCC внедряют в сгенерированный код механизм, известный как *защита стека* (*stack protector*), обнаруживающий переполнения буфера. Идея состоит в том, чтобы сохранить специальное *канареечное значение*<sup>4</sup> в кадре стека между локальным буфером и остальной информацией о состоянии в стеке, как показано на рис. 3.9 [26, 97]. Это канареечное значение, также называемое *защитным значением*, генерируется случайным образом при каждом запуске программы, поэтому злоумышленнику будет трудно определить, что это такое. Перед восстановлением состояния регистров и возвратом из функции программа проверяет, было ли изменено канареечное значение какой-либо операций внутри этой функции или той, которую она вызывала. Если да, то программа завершается с сообщением об ошибке.



**Рис. 3.9.** Организация стека функции `echo` с включенной защитой стека.

Специальное «канареечное» значение помещается между массивом `buf` и другой информацией о состоянии. Код проверяет канареечное значение, чтобы определить, имело ли место повреждение стека

Последние версии GCC пытаются определить, уязвима ли функция для атак методом переполнения стека, и автоматически вставляют код обнаружения переполнения. Фактически, чтобы исключить эту проверку из нашей предыдущей программы демонстрации переполнения стека, нам пришлось скомпилировать ее с параметром `-fno-stack-protector`. Компиляция функции `echo` без этого параметра и, следовательно, с включенной защитой стека дает следующий ассемблерный код:

```
void echo()
1  echo:
2      subq    $24, %rsp           Выделить 24 байта в стеке
3      movq    %fs:40, %rax        Взять канареечное значение
4      movq    %rax, 8(%rsp)       Сохранить в стеке
5      xorl    %eax, %eax          Очистить регистр
6      movq    %rsp, %rdi          Получить адрес buf в стеке
7      call    gets               Вызвать gets
8      movq    %rsp, %rdi          Получить адрес buf в стеке
9      call    puts               Вызвать puts
10     movq    8(%rsp), %rax        Извлечь канареечное значение из стека
11     xorq    %fs:40, %rax        Сравнить с истинным значением
12     je      .L9                 Если =, перейти к ok
13     call    __stack_chk_fail    Стек поврежден!
```

<sup>4</sup> Термин «канарейка» (канареечное значение) заимствован из истории использования этих птиц для обнаружения опасных газов в угольных шахтах.

```

14 .L9:                                ok:
15     addq    $24, %rsp              Освободить память, выделенную в стеке
16     ret

```

Как видите, эта версия функции извлекает значение из памяти (строка 3) и сохраняет его в стеке со смещением 8 от %rsp, сразу за область, выделенной для buf. Аргумент %fs:40 указывает, что канареечное значение читается из памяти с использованием *сегментированной адресации*, механизма, появившегося в процессорах 80286 и редко встречающегося в современных программах. Специальный сегмент, в котором хранится «канарейка», можно пометить как «доступный только для чтения», чтобы злоумышленник не мог перезаписать канареечное значение. Перед восстановлением регистров и возвратом функция сравнивает значение в стеке с истинным канареечным значением (с помощью инструкции `xorq` в строке 11). Если они идентичны, то `xorq` вернет 0, и функция завершится обычным образом. Ненулевое значение указывает, что канарейка в стеке изменилась, и поэтому вызывается процедура обработки ошибок.

Защита стека довольно надежно предотвращает атаки методом переполнения буфера, сопряженные с повреждением информации, хранящейся в стеке, и лишь ненамного снижает производительность, отчасти потому, что GCC вставляет защитный код, только если в функции имеется локальный буфер типа `char`. Конечно, есть и другие способы повредить состояние выполняющейся программы, но снижение уязвимости стека препятствует проведению многих распространенных атак.

#### Упражнение 3.48 (решение в конце главы)

Функции `intlen`, `len` и `iptoa` реализуют очень запутанный алгоритм вычисления количества десятичных цифр, необходимых для представления целого числа. Мы используем их для изучения некоторых аспектов механизма защиты стека в GCC.

```

int len(char *s) {
    return strlen(s);
}

void iptoa(char *s, long *p) {
    long val = *p;
    sprintf(s, "%ld", val);
}

int intlen(long x) {
    long v;
    char buf[12];
    v = x;
    iptoa(buf, &v);
    return len(buf);
}

```

Далее демонстрируется ассемблерный код, сгенерированный компилятором для `intlen` с включенной и выключенной защитой стека:

(a) Без защиты

```

int intlen(long x)
x % %rdi
1 intlen:
2     subq    $40, %rsp
3     movq    %rdi, 24(%rsp)
4     leaq    24(%rsp), %rsi
5     movq    %rsp, %rdi
6     call    iptoa

```

(b) С защитой

```

int intlen(long x)
x in %rdi
1 intlen:
2     subq    $56, %rsp
3     movq    %fs:40, %rax
4     movq    %rax, 40(%rsp)
5     xorl    %eax, %eax
6     movq    %rdi, 8(%rsp)
7     leaq    8(%rsp), %rsi
8     leaq    16(%rsp), %rdi
9     call    iptoa

```

1. Для обеих версий определите адрес buf, v и (если имеется) канареечного значения в кадре стека.
2. Как переупорядочение локальных переменных в защищенном коде может повысить защищенность от атак переполнением буфера?

## Ограничение областей выполняемого кода

Последний шаг – лишить злоумышленника возможности внедрять выполняемый код в систему. Один из способов – ограничить области памяти, где может находиться выполняемый код. В типичных программах выполняемый код, сгенерированный компилятором, занимает ограниченную часть памяти. Для остальных областей памяти можно запретить выполнение кода и разрешить только чтение и запись. Как будет показано в главе 9, пространство виртуальной памяти логически разделено на *страницы*, обычно по 2048 или 4096 байт каждая. Аппаратное обеспечение поддерживает разные формы *защиты памяти* с использованием специальных флагов доступа, которые могут использоваться как ядром операционной системы, так и пользовательскими программами. Многие системы позволяют определять три формы доступа: чтение (данных из памяти), запись (данных в память) и выполнение (обработка содержимого памяти как машинного кода). Исторически сложилось так, что архитектура x86 объединяла управление доступом для чтения и выполнения в один 1-битный флаг, поэтому любая страница, отмеченная как доступная для чтения, одновременно считалась доступной для выполнения. Стек должен был быть доступен и для чтения, и для записи, поэтому байты в стеке тоже могли выполняться. Позднее были реализованы различные схемы, ограничивающие чтение некоторых страниц, но не их выполнение, однако они, как правило, весьма отрицательно сказывались на эффективности.

Совсем недавно AMD представила бит NX («no-execute» – запрет выполнения) для организации защиты памяти в своих 64-разрядных процессорах, разделив режимы доступа для чтения и выполнения, и Intel последовала их примеру. Этим флагом можно пометить стек как доступный для чтения и записи, но не для выполнения, причем проверка доступности страницы для выполнения производится аппаратно, то есть без потери эффективности.

Некоторым программам нужна возможность динамически генерировать и выполнять код. Например, системы динамической компиляции (Just-In-Time, JIT) динамически генерируют код для программ, написанных на интерпретируемых языках, таких как Java, чтобы повысить эффективность их выполнения. Возможность ограничения выполнения только того кода, который был сгенерирован компилятором в процессе выполнения оригинальной программы, зависит от языка и операционной системы.

Описанные нами методы – рандомизация, защита стека и ограничение областей памяти для хранения выполняемого кода – являются тремя наиболее распространенными механизмами снижения уязвимости программ для атаки переполнением буфера. Ни один из них не требует особых усилий со стороны программиста и практически не



влияет на производительность. Каждый по отдельности уменьшает степень уязвимости, а все вместе они оказываются еще более эффективными. К сожалению, способы атак на компьютеры тоже развиваются [85, 97], поэтому черви и вирусы продолжают нарушать работу многих машин.

### 3.10.5. Поддержка кадров стека переменного размера

К настоящему моменту мы исследовали машинный код множества функций, но для всех них компилятор может заранее определить размер кадра стека. Однако некоторым функциям может потребоваться локальное хранилище переменного объема. Например, функция может вызывать `alloca`, стандартную библиотечную функцию, для выделения произвольного количества байтов памяти в стеке. Аналогично некоторые функции могут использовать локальные массивы переменного размера, хранящиеся в стеке.

Информацию, представленную в этом разделе, по праву можно рассматривать как один из аспектов реализации процедур, но мы отложили ее обсуждение до настоящего момента, потому что она требует понимания массивов и особенностей выравнивания.

Код в листинге 3.20 (а) демонстрирует пример функции, использующей массив переменного размера. Функция объявляет локальный массив `p` из  $n$  указателей, где  $n$  передается в первом аргументе. Для массива требуется выделить  $8n$  байт в стеке, причем значение  $n$  может меняться в разных вызовах функции. По этой причине компилятор не может определить, сколько места следует выделить для кадра стека функции. Кроме того, программа генерирует ссылку на адрес локальной переменной `i`, которая по этой причине также должна храниться в стеке. Во время выполнения функция должна иметь доступ и к локальной переменной `i`, и к элементам массива `p`. При возврате она должна освободить память и установить указатель стека в позицию хранимого адреса возврата.

**Листинг 3.20.** Функция, требующая использования указателя кадра. Массив переменного размера предполагает невозможность определения размера кадра стека во время компиляции

(а) Код на C

```
long vframe(long n, long idx, long *q) {
    long i;
    long *p[n];
    p[0] = &i;
    for (i = 1; i < n; i++)
        p[i] = q;
    return *p[idx];
}
```

(б) Фрагменты сгенерированного ассемблерного кода

```
long vframe(long n, long idx, long *q)
n в %rdi, idx в %rsi, q в %rdx
Здесь показана только часть кода
1  vframe:
2  pushq    %rbp                Сохранить прежнее значение %rbp
3  movq     %rsp, %rbp          Установить указатель кадра
4  subq     $16, %rsp           Выделить память для i (%rsp = s1)
5  leaq     22(,%rdi,8), %rax
6  andq     $-16, %rax
7  subq     %rax, %rsp          Выделить память для массива p (%rsp = s2)
8  leaq     7(%rsp), %rax
9  shrq     $3, %rax
10 leaq     0(,%rax,8), %r8      Записать &p[0] в %r8
11 movq     %r8, %rcx           Записать &p[0] в %rcx (%rcx = p)
```

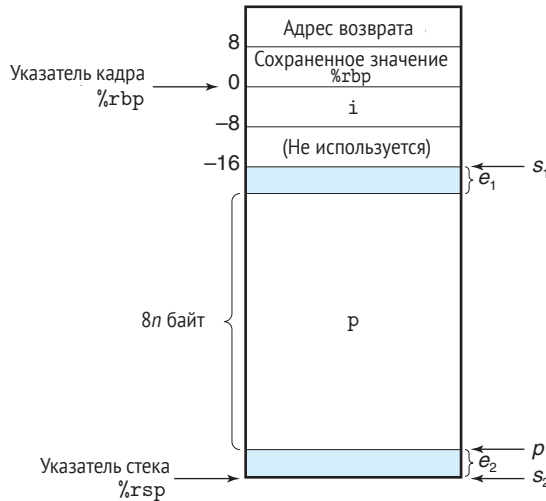


```

    . . .
Код инициализации цикла
i в %rax и в стеке, n в %rdi, p в %rcx, q в %rdx
12 .L3:                                loop:
13     movq    %rdx, (%rcx,%rax,8)      Записать q в p[i]
14     addq    $1, %rax                 Нарастить i
15     movq    %rax, -8(%rbp)           Сохранить в стеке
16 .L2:
17     movq    -8(%rbp), %rax           Извлечь i из стека
18     cmpq    %rdi, %rax              Сравнить i:n
19     jl      .L3                     Если <, перейти к loop
    . . .
Код завершения функции
20     leave   %rsp                    Восстановить %rbp и %rsp
21     ret
    Возврат

```

Для управления кадром стека переменного размера в архитектуре x86-64 используется регистр `%rbp`, который служит *указателем кадра* (иногда его называют *базовым указателем*, в соответствии с буквами bp (base pointer – базовый указатель) в имени регистра `%rbp`). При использовании этого указателя кадр стека организован, как показано на рис. 3.10. Как видите, код должен сохранить в стеке предыдущее значение `%rbp`, потому что этот регистр относится к набору регистров, сохраняемых вызываемым. Далее, на протяжении всего выполнения функции, код хранит в `%rbp` ссылку на эту позицию и обращается к локальным переменным фиксированной длины, таким как `i`, используя смещение относительно `%rbp`.



**Рис. 3.10.** Структура стека функции `vframe`. Функция использует регистр `%rbp` как указатель кадра. Комментарии справа относятся к упражнению 3.49

В листинге 3.20 (b) показаны фрагменты ассемблерного кода, сгенерированного компилятором GCC для функции `vframe`. Код в начале функции настраивает кадр стека и выделяет место для массива `p`. Сначала он перемещает содержимое регистра `%rbp` в стек и записывает в него текущий адрес вершины стека (строки 2–3). Затем выделяет в стеке 16 байт, первые 8 из которых используются для хранения локальной переменной `i`, а вторые 8 остаются неиспользуемыми. Далее он выделяет место для массива `p`

(строки 5–11). Детали, касающиеся размера выделяемого пространства его адреса  $p$  в этом пространстве, обсуждаются в упражнении 3.49, а пока только отметим, что к тому времени, когда программа достигает строки 11, она (1) выделит не менее  $8n$  байт в стеке и (2) разместит массив  $p$  в выделенной области так, чтобы для него было доступно не менее  $8n$  байт.

Код инициализации цикла демонстрирует примеры обращения к локальным переменным  $i$  и  $p$ . Строка 13 показывает, как записывается элемент массива  $p[i]$  в  $q$ . Эта инструкция использует значение регистра  $\%rsc$  как адрес начала  $p$ . Также можно видеть, как реализуются изменение (строка 15) и чтение (строка 17) переменной  $i$ . Адрес  $i$  задается ссылкой  $-8(\%rbp)$ , то есть со смещением  $-8$  относительно указателя кадра.

В конце функции восстанавливается предыдущее значение указателя кадра с помощью инструкции `leave` (строка 20). Эта инструкция не имеет аргументов и эквивалентна двум следующим инструкциям:

<code>movq</code>	<code>%rbp, %rsp</code>	Установить указатель стека в начало кадра
<code>popq</code>	<code>%rbp</code>	Восстановить сохраненное значение <code>%rbp</code> и установить указатель стека в конец кадра стека вызывающей функции

То есть в указатель стека сначала записывается адрес сохраненного значения `%rbp`, а затем это значение выталкивается из стека в регистр `%rbp`. Эта комбинация инструкций приводит к освобождению всего кадра стека.

В более ранних версиях кода x86 указатель кадра использовался в каждом вызове функции. В архитектуре x86-64 он используется, только когда кадр стека может иметь переменный размер, как в случае с функцией `vframe`. Исторически сложилось так, что большинство компиляторов использовали указатели кадров при генерации кода для архитектуры IA32. Последние версии GCC отошли от этого соглашения. Обратите внимание, что код, использующий указатели кадров, можно смешивать с кодом, который этого не делает, при условии что все функции будут следовать соглашению о сохранении вызывающим в отношении регистра `%rbp`.

### Упражнение 3.49 (решение в конце главы)

В этом упражнении мы исследуем логику кода в строках 5–11 в листинге 3.20 (b), выделяющего место для массива переменного размера  $p$ . Как отмечено в комментариях,  $s_1$  обозначает адрес вершины стека после выполнения инструкции `subq` в строке 4. Эта инструкция выделяет пространство для локальной переменной  $i$ . Пусть  $s_2$  обозначает адрес вершины стека после выполнения инструкции `subq` в строке 7. Эта инструкция выделяет память для локального массива  $p$ . Наконец, пусть  $p$  обозначает значение, записываемое в регистры `%r8` и `%rsc` в строках 10–11. Оба этих регистра используются для ссылки на массив  $p$ .

Справа на рис. 3.10 показаны местоположения, обозначенные как  $s_1$ ,  $s_2$  и  $p$ . На рисунке также видно, что между значениями  $s_2$  и  $p$  может иметься смещение в  $e_2$  байт. Это пространство останется неиспользуемым. Также может иметься смещение в  $e_1$  байт между концом массива  $p$  и позицией, обозначенной как  $s_1$ .

1. Объясните математическим языком логику вычисления  $s_2$  в строках 5–7.

*Подсказка:* вспомните, как в двоичном виде выглядит число  $-16$ , и подумайте о том, как оно влияет на инструкцию `andq` в строке 6.

2. Объясните математическим языком логику вычисления  $p$  в строках 8–10.

*Подсказка:* обратитесь к обсуждению деления на степень двойки в разделе 2.3.7.

3. Для следующих значений  $n$  и  $s_1$  проследите выполнение кода и определите получающиеся в результате значения  $s_2$ ,  $p$ ,  $e_1$  и  $e_2$ .

$n$	$s_1$	$s_2$	$p$	$e_1$	$e_2$
5	2 065	_____	_____	_____	_____
6	2 064	_____	_____	_____	_____

4. Какие соглашения о выравнивании гарантирует этот код для значений  $s_2$  и  $p$ ?

### 3.11. Вычисления с плавающей точкой

Инфраструктура поддержки вычислений с плавающей точкой в процессорах характеризуется множеством аспектов, которые определяют, как программы, работающие с данными с плавающей точкой, выражаются в машинном коде, включая:

- хранение и извлечение значений с плавающей точкой. Обычно для этого используются регистры той или иной формы;
- инструкции, выполняющие операции с данными с плавающей точкой;
- соглашения, определяющие порядок передачи значений с плавающей запятой функциям в аргументах и их возврат в виде результатов;
- соглашения о сохранении регистров во время вызовов функций, например какие регистры должны сохраняться вызывающим, а какие – вызываемым.

Чтобы понять, как работает поддержка операций с плавающей точкой в архитектуре x86-64, будет полезно вернуться немного назад. С момента появления Pentium/MMX в 1997 году компаниями Intel и AMD было реализовано несколько поколений мультимедийных инструкций для обработки графики и изображений. Эти инструкции изначально были ориентированы на параллельное выполнение нескольких операций в режиме, известном как *одиночный поток команд, множественный поток данных* (Single-Instruction, Multiple-Data, SIMD). В этом режиме одна и та же операция применяется сразу к нескольким разным значениям данных. На протяжении многих лет эти расширения продолжали развиваться, и после каждого существенного изменения их названия менялись: с MMX на SSE (Streaming SIMD Extensions – потоковые расширения SIMD), а совсем недавно они стали называться AVX (Advanced Vector eXtensions – усовершенствованные векторные расширения). В каждом поколении также были разные версии. Каждое из этих расширений использует для управления данными набор регистров, называемых регистрами «ММ» в расширениях MMX, «ХММ» в SSE и «УММ» в AVX, и с разрядностью от 64 бит для регистров ММ до 128 – для ХММ и 256 – для УММ. Так, например, каждый регистр УММ может хранить восемь 32-разрядных значений или четыре 64-разрядных значения, которые могут быть целыми числами или числами с плавающей точкой.

Начиная с расширения SSE2, реализованного в Pentium 4 в 2000 году, в наборе мультимедийных инструкций появились инструкции для работы со *скалярными* данными с плавающей точкой – одиночными значениями, хранящимися в младших 32 или 64 разрядах регистров ХММ или УММ. Этот скалярный режим предоставляет набор регистров и инструкций, которые ближе к тому, как другие процессоры поддерживают операции с плавающей точкой. Все процессоры, способные выполнять код x86-64, поддерживают SSE2 или выше, и, соответственно, реализация операций с плавающей точкой в x86-64 основана на SSE или AVX, включая соглашения о передаче аргументов и возвращаемых значений [77].

Далее в нашем обсуждении мы будем основываться на AVX2 – второй версии расширений AVX, реализованной в процессорах Core i7 Haswell в 2013 году. GCC генерирует код AVX2, если задан параметр командной строки `-mavx2`. Код, основанный на разных

версиях SSE, а также на первой версии AVX, концептуально похож, хотя и отличается именами инструкций и форматами. Мы будем рассматривать только инструкции, генерируемые компилятором GCC при компиляции программ, выполняющих операции с плавающей точкой. По большей части это скалярные инструкции AVX, но мы также опишем случаи применения инструкций, предназначенных для работы с целыми векторами данных. Более полное описание использования SIMD-расширений SSE и AVX вы найдете в приложении в интернете «OPT:SIMD», представленном в разделе 5.9.2. Желающие могут обратиться к описанию отдельных инструкций в документации AMD и Intel [4, 51]. Так же как в случае с целочисленными операциями, обратите внимание, что формат АТТ, который мы используем в нашем обсуждении, отличается от формата Intel, используемого в этих документах. В частности, в этих документах операнды инструкций следуют в обратном порядке.

255	127	0	
%ymm0	%xmm0		1-й арг. с плав. точкой/ Возвращаемое значение
%ymm1	%xmm1		2-й аргумент с плав. точкой
%ymm2	%xmm2		3-й аргумент с плав. точкой
%ymm3	%xmm3		4-й аргумент с плав. точкой
%ymm4	%xmm4		5-й аргумент с плав. точкой
%ymm5	%xmm5		6-й аргумент с плав. точкой
%ymm6	%xmm6		7-й аргумент с плав. точкой
%ymm7	%xmm7		8-й аргумент с плав. точкой
%ymm8	%xmm8		Сохраняется вызывающим
%ymm9	%xmm9		Сохраняется вызывающим
%ymm10	%xmm10		Сохраняется вызывающим
%ymm11	%xmm11		Сохраняется вызывающим
%ymm12	%xmm12		Сохраняется вызывающим
%ymm13	%xmm13		Сохраняется вызывающим
%ymm14	%xmm14		Сохраняется вызывающим
%ymm15	%xmm15		Сохраняется вызывающим

**Рис. 3.11.** Мультимедийные регистры. Эти регистры используются для хранения данных с плавающей точкой. Каждый регистр УММ хранит 32 байта. К младшим 16 байтам можно обращаться как к регистру ХММ

Как показано на рис. 3.11, архитектура AVX поддержки операций с плавающей точкой позволяет хранить данные в 16 регистрах YMM с именами %ymm0–%ymm15. Каждый регистр YMM имеет размер 256 разрядов (32 байта). При работе со скалярными данными эти регистры хранят только по одному значению с плавающей точкой в младших 32 разрядах (для чисел типа float) или 64 разрядах (для чисел типа double). Ассемблерный код обращается к регистрам по их именам SSE XMM: %xmm0–%xmm15, где каждый регистр XMM – это младшие 128 разрядов (16 байт) соответствующего регистра YMM.

### 3.11.1. Операции перемещения и преобразования данных

В табл. 3.15 показан набор инструкций для перемещения значений с плавающей точкой между памятью и регистрами XMM, а также между парами регистров XMM без преобразования. Инструкции, ссылающиеся на ячейки памяти, являются *скалярными*, то есть они работают с отдельными значениями, а не с упакованными наборами. Данные хранятся либо в памяти (в таблице обозначены как  $M_{32}$  и  $M_{64}$ ), либо в регистрах XMM (обозначены в таблице как X). Эти инструкции будут работать правильно независимо от выравнивания данных, однако, согласно рекомендациям по оптимизации, 32-разрядные данные должны размещаться в памяти по границам 4-байтных блоков, а 64-разрядные – 8-байтных. Ссылки на память указываются так же, как для целочисленных инструкций MOV, со всеми возможными комбинациями смещения, базового регистра, индексного регистра и коэффициента масштабирования.

**Таблица 3.15.** Инструкции перемещения значений с плавающей точкой. Эти инструкции перемещают значения между памятью и регистрами, а также между парами регистров (X: регистр XMM (например, %xmm3);  $M_{32}$ : 32-разрядная ячейка памяти;  $M_{64}$ : 64-разрядная ячейка памяти)

Инструкция	Источник	Приемник	Описание
vmovss	$M_{32}$	X	Перемещение значения одинарной точности
vmovss	X	$M_{32}$	Перемещение значения одинарной точности
vmovsd	$M_{64}$	X	Перемещение значения двойной точности
vmovsd	X	$M_{64}$	Перемещение значения двойной точности
vmovaps	X	X	Перемещение с выравниванием упакованных значений одинарной точности
vmovapd	X	X	Перемещение с выравниванием упакованных значений двойной точности

GCC использует инструкции скалярного перемещения только для передачи данных из памяти в регистр XMM или из регистра XMM в память. Для передачи данных между двумя регистрами XMM используется пара инструкций для копирования всего содержимого одного регистра XMM в другой, а именно vmovaps для данных одинарной точности и vmovapd для данных двойной точности. В этих случаях различия между копированием всего регистра и только младших разрядов с фактическим значением не влияют ни на функциональность программы, ни на скорость выполнения, поэтому выбор между этими инструкциями и инструкциями для работы со скалярными данными не имеет реального значения. Буква «a» в названиях этих инструкций означает «aligned» (с выравниванием). При использовании для чтения и записи памяти они вызывают исключение, если адрес не удовлетворяет требованиям к выравниванию по границам 16-байтных блоков. При передаче между двумя регистрами возможность неправильного выравнивания отсутствует.

В качестве примера различных операций перемещения значений с плавающей точкой рассмотрим функцию на C:

```
float float_mov(float v1, float *src, float *dst) {
    float v2 = *src;
    *dst = v1;
    return v2;
}
```

и соответствующий ей ассемблерный код:

```
float float_mov(float v1, float *src, float *dst)
v1 в %xmm0, src в %rdi, dst в %rsi
1 float_mov:
2 vmovaps %xmm0, %xmm1      Скопировать v1
3 vmovss  (%rdi), %xmm0      Прочитать v2 из src
4 vmovss  %xmm1, (%rsi)      Записать v1 в dst
5 ret                      Вернуть v2 в %xmm0
```

В этом примере можно видеть использование инструкции `vmovaps` для копирования данных из одного регистра в другой и инструкции `vmovss` для копирования данных из памяти в регистр XMM и из регистра XMM в память.

В табл. 3.16 и 3.17 показаны наборы инструкций для преобразования между значениями с плавающей точкой и целыми числами, а также между различными форматами с плавающей точкой. Все эти инструкции являются скалярными – они работают с отдельными значениями.

Инструкции в табл. 3.16 преобразуют значение с плавающей точкой, находящееся в регистре XMM или в памяти, и записывают результат в регистр общего назначения (например, `%eax`, `%ebx` и т. д.). При преобразовании значений с плавающей запятой в целые числа они выполняют усечение, округляя значения в сторону нуля, как того требует стандарт C и большинства других языков программирования.

**Таблица 3.16.** Инструкции с двумя операндами для преобразования значений с плавающей точкой. Эти инструкции преобразуют данные с плавающей точкой в целочисленные значения (X: регистр XMM (например, `%xmm3`);  $R_{32}$ : 32-разрядный регистр общего назначения (например, `%eax`),  $R_{64}$ : 64-разрядный регистр общего назначения (например, `%rax`),  $M_{32}$ : 32-разрядная ячейка памяти;  $M_{64}$ : 64-разрядная ячейка памяти)

Инструкция	Источник	Приемник	Описание
<code>vcvttsd2si</code>	$X/M_{32}$	$R_{32}$	Преобразование с усечением числа с плавающей точкой одинарной точности в целое число
<code>vcvttsd2si</code>	$X/M_{64}$	$R_{32}$	Преобразование с усечением числа с плавающей точкой двойной точности в целое число
<code>vcvttsd2siq</code>	$X/M_{32}$	$R_{64}$	Преобразование с усечением числа с плавающей точкой одинарной точности в целое 64-разрядное число
<code>vcvttsd2siq</code>	$X/M_{64}$	$R_{64}$	Преобразование с усечением числа с плавающей точкой двойной точности в целое 64-разрядное число

Инструкции в табл. 3.17 преобразуют целое число в число с плавающей точкой. Они используют необычный формат с тремя операндами: с двумя источниками и одним приемником. Первый операнд определяет ячейку памяти или регистр общего назначения. В нашем случае мы можем игнорировать второй операнд, потому что его значение влияет только на старшие байты результата. Приемником должен быть регистр XMM.

Обычно операнды, представляющие второй источник и приемник, идентичны, как в следующей инструкции:

```
vcvtssi2sdq    %rax, %xmm1, %xmm1
```

Эта инструкция извлекает длинное целое число из регистра `%rax`, преобразует его в значение типа `double` и сохраняет результат в младших байтах XMM-регистра `%xmm1`.

**Таблица 3.17.** Инструкции с тремя операндами для преобразования значений с плавающей точкой. Эти инструкции преобразуют целочисленные значения из первого источника в значения с плавающей точкой в приемнике. Значение второго источника не влияет на младшие байты результата (X: регистр XMM (например, `%xmm3`);  $M_{32}$ : 32-разрядная ячейка памяти;  $M_{64}$ : 64-разрядная ячейка памяти)

Инструкция	Источник 1	Источник 2	Приемник	Описание
<code>vcvtssi2ss</code>	$M_{32}/R_{32}$	X	X	Преобразование целого числа в число с плавающей точкой одинарной точности
<code>vcvtssi2sd</code>	$M_{32}/R_{32}$	X	X	Преобразование целого числа в число с плавающей точкой двойной точности
<code>vcvtssi2ssq</code>	$M_{64}/R_{64}$	X	X	Преобразование 64-разрядного целого числа в число с плавающей точкой одинарной точности
<code>vcvtssi2sdq</code>	$M_{64}/R_{64}$	X	X	Преобразование 64-разрядного целого числа в число с плавающей точкой двойной точности

Наконец, для преобразования между двумя разными форматами с плавающей точкой текущие версии GCC генерируют код, который требует отдельного описания. Предположим, что младшие 4 байта в `%xmm0` содержат значение с плавающей точкой одинарной точности; тогда, казалось бы, для преобразования этого значения в значение с двойной точностью и с сохранением результата в младших 8 байтах регистра `%xmm0` можно просто использовать инструкцию:

```
vcvtss2sd    %xmm0, %xmm0, %xmm0
```

Однако вместо этого GCC генерирует следующий код:

```
Преобразование числа с плавающей точкой из представления
с одинарной точностью в представление с двойной точностью
1  vunpcklps  %xmm0, %xmm0, %xmm0    Извлечь первый элемент вектора
2  cvtpps2pd  %xmm0, %xmm0           Преобразовать 2-элементный вектор
                                     в число с двойной точностью
```

Инструкция `vunpcklps` обычно используется для чередования значений в двух регистрах XMM и сохранения их в третьем. То есть если один исходный регистр содержит слова  $[s_3, s_2, s_1, s_0]$ , а другой  $[d_3, d_2, d_1, d_0]$ , то значением регистра-приемника будет  $[s_1, d_1, s_0, d_0]$ . В предыдущем коде во всех трех операндах используется один и тот же регистр, поэтому если исходный регистр содержал значения  $[x_3, x_2, x_1, x_0]$ , то инструкция обновит регистр, поместив в него значения  $[x_1, x_1, x_0, x_0]$ . Инструкция `cvtpps2pd` расширяет два младших значения одинарной точности в исходном регистре XMM до двух значений с двойной точностью в регистре-приемнике. При применении к результату предыдущей инструкции `vunpcklps` она даст значения  $[dx_0, dx_0]$ , где  $dx_0$  – результат преобразования  $x$  в значение двойной точности. Таким образом, эти две инструкции преобразуют исходное значение с одинарной точностью в младших 4 байтах в `%xmm0` в значение с двойной точностью и сохранение двух его копий в `%xmm0`. Не совсем понятно, почему GCC генерирует такой код, потому что нет ни выгоды, ни необходимости дублировать значение в регистре XMM.

Аналогичный код GCC генерирует для преобразования значения с двойной точностью в значение с одинарной точностью:

```

    Преобразование значения с двойной точностью
    в значение с одинарной точностью
1  vmovddup    %xmm0, %xmm0      Извлечь первый элемент вектора
2  cvtupd2psx  %xmm0, %xmm0      Преобразовать 2-элементный вектор
                                   в число с одинарной точностью

```

Предположим, что эти инструкции получают регистр `%xmm0`, содержащий два значения двойной точности  $[x_1, x_0]$ . Тогда инструкция `vmovddup` оставит в нем  $[x_0, x_0]$ , а инструкция `cvtupd2psx` преобразует эти значения в значения с одинарной точностью, упаковывает их в младшую половину регистра и сбросит старшую половину в 0, дав результат  $[0.0, 0.0, x_0, x_0]$  (напомним, что нулевое значение с плавающей точкой 0.0 представлено битовой комбинацией из одних нулей). И снова нет никакой ценности в преобразовании значения из одной точности в другую таким способом вместо применения единственной инструкции:

```
vcvtisd2ss  %xmm0, %xmm0, %xmm0
```

Для примера рассмотрим реализацию еще нескольких операций преобразования значений с плавающей точкой в следующей функции на C:

```

double fcvt(int i, float *fp, double *dp, long *lp)
{
    float f = *fp; double d = *dp; long l = *lp;
    *lp = (long) d;
    *fp = (float) i;
    *dp = (double) l;
    return (double) f;
}

```

GCC генерирует следующий ассемблерный код:

```

double fcvt(int i, float *fp, double *dp, long *lp)
i в %edi, fp в %rsi, dp в %rdx, lp в %rcx
1  fcvt:
2  vmovss      (%rsi), %xmm0      Получить f = *fp
3  movq        (%rcx), %rax       Получить l = *lp
4  cvttisd2siq (%rdx), %r8        Получить d = *dp и
                                   преобразовать в long
5  movq        %r8, (%rcx)        Сохранить по адресу в lp
6  cvtsi2ss     %edi, %xmm1, %xmm1 Преобразовать i во float
7  vmovss      %xmm1, (%rsi)      Сохранить по адресу в fp
8  cvtsi2sdq    %rax, %xmm1, %xmm1 Преобразовать l в double
9  vmovsd      %xmm1, (%rdx)      Сохранить по адресу в dp
                                   Следующие две инструкции преобразуют f в double
10 vunpcklps    %xmm0, %xmm0, %xmm0
11 cvtups2pd    %xmm0, %xmm0
12 ret          Вернуть f

```

Все аргументы передаются функции `fcvt` через регистры общего назначения, потому что они представляют либо целые числа, либо указатели. Результат возвращается в регистр `%xmm0`. Как показано на рис. 3.11, этот регистр предназначен для возврата значений типа `float` или `double`. В этом примере мы видим ряд инструкций перемещения и преобразования, представленных в табл. 3.15–3.17, а также метод преобразования из одинарной точности в двойную, выбираемый компилятором GCC.



Упражнение 3.50 (решение в конце главы)

В следующем коде на C выражения val1–val4 отображаются в программные значения i, f, d и l:

```
double fcvt2(int *ip, float *fp, double *dp, long l)
{
    int i = *ip; float f = *fp; double d = *dp;
    *ip = (int)    val1;
    *fp = (float)  val2;
    *dp = (double) val3;
    return (double) val4;
}
```

Определите типы выражений, опираясь на следующий ассемблерный код x86-64, сгенерированный для этой функции:

```
double fcvt2(int *ip, float *fp, double *dp, long l)
ip в %rdi, fp в %rsi, dp в %rdx, l в %rcx
Результат возвращается в %xmm0
1 fcvt2:
2     movl    (%rdi), %eax
3     vmovss  (%rsi), %xmm0
4     vcvttss2si    (%rdx), %r8d
5     movl    %r8d, (%rdi)
6     vcvtsi2ss     %eax, %xmm1, %xmm1
7     vmovss  %xmm1, (%rsi)
8     vcvtsi2sdq    %rcx, %xmm1, %xmm1
9     vmovsd  %xmm1, (%rdx)
10    vunpcklps    %xmm0, %xmm0, %xmm0
11    vcvtps2pd    %xmm0, %xmm0
12    ret
```

Упражнение 3.51 (решение в конце главы)

Следующая функция на C преобразует аргумент типа src\_t в возвращаемое значение типа dest\_t, где эти два типа определены с помощью typedef:

```
dest_t cvt(src_t x)
{
    dest_t y = (dest_t) x;
    return y;
}
```

Предположим, что этот код компилируется для архитектуры x86-64 и аргумент x находится либо в %xmm0, либо в соответствующей части регистра %rdi (то есть в %rdi или в %edi). Для преобразования типа и копирования значения в соответствующую часть регистра %rax (целочисленный результат) или %xmm0 (результат с плавающей точкой) должны использоваться одна или две инструкции. Впишите в следующую таблицу соответствующие инструкции вместе с регистрами – источниками и приемниками.

$T_x$	$T_y$	Инструкции
long	double	vcvtsi2sdq %rdi, %xmm0
double	int	_____
double	float	_____
long	float	_____
float	long	_____

### 3.11.2. Операции с плавающей точкой в процедурах

В архитектуре x86-64 для передачи значений с плавающей точкой в функции и из функций используются регистры ХММ. При этом, как показано на рис. 3.11, действуют следующие соглашения:

- в ХММ-регистрах `%xmm0-%xmm7` может передаваться до восьми аргументов с плавающей точкой. Эти регистры используются в порядке перечисления аргументов. Дополнительные аргументы с плавающей точкой могут передаваться через стек;
- возвращаемое значение с плавающей точкой передается из функции в регистре `%xmm0`;
- все регистры ХММ сохраняются вызывающим. Вызываемый может изменять любые из этих регистров без предварительного сохранения.

Когда функция принимает комбинацию аргументов с указателями, целыми числами и числами с плавающей точкой, то указатели и целые числа передаются в регистрах общего назначения, а значения с плавающей точкой – в регистрах ХММ. Это означает, что соответствие аргументов регистрам зависит не только от порядка следования, но и от их типов. Вот несколько примеров:

```
double f1(int x, double y, long z);
```

Эта функция получит `x` в `%edi`, `y` в `%xmm0` и `z` в `%rsi`.

```
double f2(double y, int x, long z);
```

Эта функция получит свои аргументы в тех же регистрах, что и функция `f1`.

```
double f1(float x, double *y, long *z);
```

Эта функция получит `x` в `%xmm0`, `y` в `%rdi` и `z` в `%rsi`.

#### Упражнение 3.52 (решение в конце главы)

Для каждого из следующих объявлений функций определите, в каких регистрах будут переданы аргументы:

1. `double g1(double a, long b, float c, int d);`
2. `double g2(int a, double *b, float *c, long d);`
3. `double g3(double *a, double b, int c, float d);`
4. `double g4(float a, int *b, float c, double d);`

### 3.11.3. Арифметические операции с плавающей точкой

В табл. 3.18 перечислены скалярные инструкции из набора AVX2 для выполнения арифметических операций с плавающей точкой. Они принимают один ( $S_1$ ) или два ( $S_1, S_2$ ) операнда-источника и операнд-приемник  $D$ . Первый операнд-источник  $S_1$  может быть регистром ХММ или ячейкой памяти. Второму операнд-источнику и операнд-приемнику должны быть регистрами ХММ. Для каждой операции имеются инструкции для вычислений с одинарной и двойной точностью. Результат сохраняется в регистре-приемнике.

**Таблица 3.18.** Скалярные арифметические операции с плавающей точкой. Они принимают один или два операнда-источника и один операнд-приемник

С одинарной точностью	С двойной точностью	Действие	Описание
vaddss	vaddsd	$D \leftarrow S_2 + S_1$	Сложение с плавающей точкой
vsubss	vsubsd	$D \leftarrow S_2 - S_1$	Вычитание с плавающей точкой
vmulss	vmulsd	$D \leftarrow S_2 \times S_1$	Умножение с плавающей точкой
vdivss	vdivsd	$D \leftarrow S_2 / S_1$	Деление с плавающей точкой
vmaxss	vmaxsd	$D \leftarrow \max(S_2, S_1)$	Максимальное из двух чисел с плавающей точкой
vminss	vminsd	$D \leftarrow \min(S_2, S_1)$	Минимальное из двух чисел с плавающей точкой
sqrts	sqrtsd	$D \leftarrow \sqrt{S_1}$	Корень квадратный из числа с плавающей точкой

В качестве примера рассмотрим следующую функцию:

```
double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}
```

Вот ее ассемблерный код для архитектуры x86-64:

```
double funct(double a, float x, double b, int i)
a в %xmm0, x в %xmm1, b в %xmm2, i в %edi
1 funct:
    Следующие две инструкции преобразуют x в тип double
2    vunpcklps    %xmm1, %xmm1, %xmm1
3    vcvtps2pd    %xmm1, %xmm1
4    vmulsd      %xmm0, %xmm1, %xmm0    Умножить a на x
5    vcvtsi2sd    %edi, %xmm1, %xmm1    Преобразовать i в double
6    vdivsd      %xmm1, %xmm2, %xmm2    Вычислить b/i
7    vsubsd      %xmm2, %xmm0, %xmm0    Вычесть из a*x
8    ret
```

Три аргумента с плавающей точкой – a, x и b – передаются в XMM-регистрах %xmm0–%xmm2, а целочисленный аргумент i – в регистре %edi. Для преобразования аргумента x в double используется стандартная последовательность из двух инструкций (строки 2–3). Еще одна инструкция используется для преобразования аргумента i в double (строка 5). Значение функции возвращается в регистре %xmm0.

**Упражнение 3.53 (решение в конце главы)**

Типы четырех аргументов для следующей функции на C определены через typedef:

```
double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
{
    return p/(q+r) - s;
}
```

Для этой функции GCC сгенерировал следующий ассемблерный код:

```
double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
1 funct1:
2    vcvtsi2ssq    %rsi, %xmm2, %xmm2
```

```

3    vaddss    %xmm0, %xmm2, %xmm0
4    vcvtsi2ss    %edi, %xmm2, %xmm2
5    vdivss    %xmm0, %xmm2, %xmm0
6    vunpcklps    %xmm0, %xmm0, %xmm0
7    vcvtps2pd    %xmm0, %xmm0
8    vsubsd     %xmm1, %xmm0, %xmm0
9    ret

```

Определите возможные комбинации типов этих четырех аргументов (верных ответов может быть несколько).

### Упражнение 3.54 (решение в конце главы)

Функция `funct2` имеет следующий прототип:

```
double funct2(double w, int x, float y, long z);
```

Для этой функции GCC сгенерировал следующий код:

```

double funct2(double w, int x, float y, long z)
w в %xmm0, x в %edi, y в %xmm1, z в %rsi
1  funct2:
2    vcvtsi2ss    %edi, %xmm2, %xmm2
3    vmulss    %xmm1, %xmm2, %xmm1
4    vunpcklps    %xmm1, %xmm1, %xmm1
5    vcvtps2pd    %xmm1, %xmm2
6    vcvtsi2sdq    %rsi, %xmm1, %xmm1
7    vdivsd     %xmm1, %xmm0, %xmm0
8    vsubsd     %xmm0, %xmm2, %xmm0
9    ret

```

Напишите версию функции `funct2` на C.

## 3.11.4. Определение и использование констант с плавающей точкой

В отличие от целочисленных арифметических операций, AVX-операции с плавающей запятой не могут принимать непосредственные значения в качестве операндов. Вместо этого компилятор должен сохранить константы в памяти, после чего код сможет извлекать эти значения и использовать по назначению. Все это иллюстрирует следующая функция преобразования значения температуры из шкалы Цельсия в шкалу Фаренгейта:

```

double cel2fahr(double temp)
{
    return 1.8 * temp + 32.0;
}

```

Соответствующий ассемблерный код для архитектуры x86-64:

```

double cel2fahr(double temp)
temp в %xmm0
1  cel2fahr:
2    vmulsd     .LC2(%rip), %xmm0, %xmm0    Умножить на 1.8
3    vaddsd     .LC3(%rip), %xmm0, %xmm0    Прибавить 32.0

```

```
4   ret
5   .LC2:
6   .long   3435973837      Младшие четыре байта числа 1.8
7   .long   1073532108      Старшие четыре байта числа 1.8
8   .LC3:
9   .long   0               Младшие четыре байта числа 32.0
10  .long   1077936128      Старшие четыре байта числа 32.0
```

Как видите, функция извлекает значение 1.8 из ячейки памяти с меткой .LC2 и значение 32.0 из ячейки памяти с меткой .LC3. Судя по коду, значения, связанные с этими метками, определяются парой объявлений .long со значениями в десятичном формате. Но как их интерпретировать в значения с плавающей точкой? Давайте возьмем объявление с меткой .LC2, здесь мы видим два значения: 3435973837 (0хссссссссd) и 1073532108 (0хffсссссс). Поскольку в данной машине используется обратный порядок следования байтов (little-endian), то первое значение дает младшие 4 байта, а второе – старшие 4 байта. Из старших байтов можно извлечь поле показателя степени 0х3ff (1023), из которого, вычтя смещение 1023, получаем показатель степени 0. Объединив разряды из двух значений, составляющие дробную часть, получаем поле дробной части 0хссссссссссссссd, которое является двоичным дробным представлением числа 0.8. К этому числу добавляем подразумеваемую ведущую единицу и получаем 1.8.

**Упражнение 3.55 (решение в конце главы)**

Покажите, как значение в памяти с меткой .LC3 декодируется в число 32.0.

### 3.11.5. Поразрядные логические операции с числами с плавающей точкой

Иногда можно встретить код, сгенерированный компилятором GCC, который выполняет поразрядные операции с регистрами XMM. В табл. 3.19 показаны некоторые инструкции, похожие на свои аналоги, выполняющие операции с регистрами общего назначения. Все эти операции воздействуют на упакованные данные, то есть они обновляют регистр-приемник целиком, применяя поразрядную операцию ко всем данным в двух регистрах-источниках. Однако напомним, что для нас представляют интерес только скалярные данные, то есть влияние этих инструкций на младшие 4 или 8 байт операнда-приемника. Эти операции часто оказываются простым и удобным способом манипулирования значениями с плавающей запятой, как поясняется в следующем упражнении.

**Таблица 3.19.** Поразрядные операции с упакованными данными. Эти инструкции выполняют логические операции со всеми 128 битами в регистре XMM

С одинарной точностью	С двойной точностью	Действие	Описание
vxorps	xorpd	$D \leftarrow S_2 \wedge S_1$	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ
vandps	andpd	$D \leftarrow S_2 \& S_1$	Поразрядное И

**Упражнение 3.56 (решение в конце главы)**

Взгляните на следующую функцию на языке C, где EXPR – это макрос, объявленный с помощью директивы #define:

```
double simplefun(double x) {
    return EXPR(x);
}
```

Нижe приводится ассемблерный код с инструкциями AVX2, сгенерированный для различных определений EXPR, где значение  $x$  хранится в `%xmm0`. Все они соответствуют некоторой полезной операции со значениями с плавающей запятой. Определите, что это за операции. Для ответа вам потребуется понимание битовых комбинаций констант, извлекаемых из памяти.

1.

```
1  vmovsd    .LC1(%rip), %xmm1
2  vandpd    %xmm1, %xmm0, %xmm0
3  .LC1:
4  .long     4294967295
5  .long     2147483647
6  .long     0
7  .long     0
```

2.

```
1  vxorpd    %xmm0, %xmm0, %xmm0
```

3.

```
1  vmovsd    .LC2(%rip), %xmm1
2  vxorpd    %xmm1, %xmm0, %xmm0
3  .LC2:
4  .long     0
5  .long     -2147483648
6  .long     0
7  .long     0
```

### 3.11.6. Операции сравнения значений с плавающей точкой

В наборе AVX2 имеются две инструкции для сравнения значений с плавающей точкой:

Инструкция	Основана на	Описание
<code>vucomiss <math>S_1, S_2</math></code>	$S_2 - S_1$	Сравнение значений с одинарной точностью
<code>vucomisd <math>S_1, S_2</math></code>	$S_2 - S_1$	Сравнение значений с двойной точностью

Эти инструкции аналогичны инструкциям CMP (см. раздел 3.6) в том, как они сравнивают операнды  $S_1$  и  $S_2$  (но в обратном порядке, как можно было бы ожидать) и устанавливают флаги условий, чтобы указать результат сравнения. Так же как `cmprq`, они следуют соглашениям формата ATN о перечислении операндов в обратном порядке. Аргумент  $S_2$  должен находиться в регистре XMM, а  $S_1$  может находиться либо в регистре XMM, либо в памяти.

Команды сравнения значений с плавающей точкой устанавливают три флага: флаг нуля ZF, флаг переноса CF и флаг четности PF. Мы не описывали флаг четности в разделе 3.6.1, потому что он обычно не используется в коде, сгенерированном компилятором GCC для архитектуры x86. При выполнении целочисленных операций этот флаг устанавливается, когда самая последняя арифметическая или логическая операция дала значение, в котором младший байт имеет четное количество единиц. В операциях сравнения значений с плавающей точкой этот флаг устанавливается, когда любой из операндов не является числом (NaN). По соглашению, любое сравнение в языке C счита-

ется неудачным, если один из аргументов не является числом, и этот флаг используется для обнаружения такого состояния. Например, даже сравнение  $x == x$  дает 0, если  $x$  равно  $NaN$ .

Флаги условий устанавливаются следующим образом:

Порядок $S_2:S_1$	CF	ZF	PF
Неупорядочиваемые значения	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

Строка «Неупорядочиваемые значения» соответствует ситуации, когда любой из операндов равен  $NaN$ . Эти ситуации можно обнаружить по состоянию флага четности PF. Обычно инструкция `jp` («jump on parity» – перейти, если четное) используется для условного перехода, когда сравнение значений с плавающей точкой дает неопределенный результат. В остальном флаги переноса и нуля получают такие же значения, как и при сравнении целых без знака: ZF устанавливается, когда операнды равны, CF устанавливается, когда  $S_2 < S_1$ . Для условных переходов по различным комбинациям флагов используются такие инструкции, как `ja` и `jb`.

**Листинг 3.21.** Иллюстрация условного ветвления при выполнении операций с плавающей точкой

(a) Код на C

```
typedef enum {NEG, ZERO, POS, OTHER} range_t;
range_t find_range(float x)
{
    int result;
    if (x < 0)
        result = NEG;
    else if (x == 0)
        result = ZERO;
    else if (x > 0)
        result = POS;
    else
        result = OTHER;
    return result;
}
```

(b) Сгенерированный ассемблерный код

```
range_t find_range(float x)
x в %xmm0
1 find_range:
2 vxorps %xmm1, %xmm1, %xmm1      Установить %xmm1 = 0
3 vucomiss %xmm0, %xmm1           Сравнить 0:x
4 ja .L5                          Если >, перейти к neg
5 vucomiss %xmm1, %xmm0           Сравнить x:0
6 jp .L8                          Если NaN, перейти к posornan
7 movl $1, %eax                  result = ZERO
8 je .L3                          Если =, перейти к done
9 .L8:                            posornan:
10 vucomiss .LC0(%rip), %xmm0     Сравнить x:0
11 setbe %al                      Установить result = NaN ? 1 : 0
12 movzbl %al, %eax              Дополнение нулями
13 addl $2, %eax                  result += 2 (POS для > 0,
```

```

14  ret
15  .L5:
16  movl    $0, %eax
17  .L3:
18  rep; ret

```

OTHER для NaN)

```

Возврат
neg:
result = NEG
done:
Возврат

```

В качестве примера сравнения значений с плавающей точкой в листинге 3.21 (а) приводится функция на С, которая классифицирует аргумент  $x$  в соответствии с его положением на числовой прямой относительно 0.0. В качестве результата функция возвращает значение перечислимого типа. Перечислимые типы в С кодируются как целые числа, поэтому функция может вернуть: 0 (NEG), 1 (ZERO), 2 (POS) и 3 (OTHER). Последний результат возвращается, когда  $x$  имеет значение *NaN*.

При компиляции `find_range` GCC генерирует код, показанный в листинге 3.21 (b). Это не самый эффективный код – он трижды сравнивает  $x$  с 0.0, хотя всю требуемую информацию можно получить с помощью единственного сравнения. Он также дважды генерирует константу с плавающей точкой 0.0 – один раз с использованием `vxorps` и один раз путем чтения значения из памяти. Давайте проследим ход выполнения функции для каждого из четырех возможных результатов:

$x < 0.0$

Выполнится ветвление `ja` в строке 4 с переходом в конец и возвратом значения 0.

$x = 0.0$

Ветвления `ja` (строка 4) и `jp` (строка 6) не будут выполнены, но выполнится ветвление `je` (строка 8) с возвратом 1 в регистре `%eax`.

$x > 0.0$

Не выполнится ни одно из трех ветвлений. Инstrukция `setbe` (строка 11) даст в результате 0, к которому инструкция `addl` (строка 13) прибавит 2, чтобы получить возвращаемое значение 2.

$x = \text{NaN}$

Выполнится ветвление `jp` (строка 6). Третья инструкция `vucomis` (строка 10) установит флаги переноса и нуля, поэтому инструкция `setbe` (строка 11) и следующая за ней инструкция `movzbl` (строка 12) запишут в `%eax` число 1, которое затем будет увеличено инструкцией `addl` (строка 13), чтобы получить возвращаемое значение 3.

В упражнениях 3.73 и 3.74 для самостоятельного решения вам будет предложено вручную написать более эффективную реализацию `find_range`.

### Упражнение 3.57 (решение в конце главы)

Имеется функция `funct3` со следующим прототипом:

```
double funct3(int *ap, double b, long c, float *dp);
```

Для этой функции GCC сгенерировал следующий ассемблерный код:

```

double funct3(int *ap, double b, long c, float *dp)
ap в %rdi, b в %xmm0, c в %rsi, dp в %rdx
1  funct3:
2      vmovss    (%rdx), %xmm1
3      vcvtsi2sd    (%rdi), %xmm2, %xmm2
4      vucomisd    %xmm2, %xmm0
5      jbe      .L8

```



```

6   vcvtsi2ssq    %rsi, %xmm0, %xmm0
7   vmulss       %xmm1, %xmm0, %xmm1
8   vunpcklps    %xmm1, %xmm1, %xmm1
9   vcvtps2pd    %xmm1, %xmm0
10  ret
11  .L8:
12  vaddss       %xmm1, %xmm1, %xmm1
13  vcvtsi2ssq    %rsi, %xmm0, %xmm0
14  vaddss       %xmm1, %xmm0, %xmm0
15  vunpcklps    %xmm0, %xmm0, %xmm0
16  vcvtps2pd    %xmm0, %xmm0
17  ret

```

Напишите версию функции на C.

### 3.11.7. Заключительные замечания об операциях с плавающей точкой

В общем и целом стиль машинного кода, сгенерированного для выполнения операций с данными с плавающей точкой с использованием набора инструкций AVX2, напоминает стиль кода для выполнения операций с целочисленными данными. В обоих случаях для хранения, обработки и передачи значений функциям используется набор регистров.

Конечно, при использовании данных разных типов и правил оценки выражений, содержащих смесь разнотипных данных, возникает много сложностей, поэтому код AVX2 включает гораздо больше разных инструкций и форматов, чем обычно можно встретить в функциях, выполняющих только целочисленную арифметику.

AVX2 также может ускорить вычисления за счет параллельного выполнения операций с упакованными данными. Разработчики компиляторов работают над автоматизацией преобразования скалярного кода в параллельный, но пока наиболее надежным способом достижения высокой производительности за счет параллелизма является использование расширений языка C для управления векторами данных, поддерживаемых GCC. Более подробно об этом рассказывается в приложении в интернете «OPT:SIMD», представленном в разделе 5.9.2.

## 3.12. Итоги

В этой главе мы заглянули за стену абстракции, образованную языком C, чтобы получить представление о программировании на машинном уровне. Имея в своем распоряжении компилятор, который генерирует представление программы на языке ассемблера, мы можем получить более полную информацию о компиляторе и о его возможностях оптимизации, а также о самой машине, ее типах данных и о наборе инструкций. В главе 5 мы покажем, как знание характеристик компилятора может помочь писать эффективные программы. Мы также получили более полное представление о том, как программа хранит данные в разных областях памяти. В главе 12 мы рассмотрим множество примеров, когда прикладному программисту важно знать, находится ли переменная в стеке, в какой-то структуре в динамической памяти или среди глобальных данных. Понимание того, как программы отображаются в машинный код, позволит лучше понять различия между этими видами памяти.

Представление программы на языке ассемблера может сильно отличаться от ее представления на языке C. В языке ассемблера различия между типами данных минимальны. Программа представляет собой последовательность инструкций, каждая из которых

выполняет единственную операцию. Некоторые части состояния программы, такие как регистры и стек, непосредственно доступны программисту. Для операций с данными и управления ходом выполнения программы доступны только низкоуровневые инструкции. Компилятору приходится использовать множество различных инструкций, чтобы сгенерировать код, реализующий различные структуры данных и всевозможные операции с ними, а также управляющие конструкции, такие как условные операторы, циклы и процедуры. Мы рассмотрели множество аспектов языка C и узнали, как компилируются программы, написанные на этом языке. Мы также видели, что отсутствие проверки границ в C часто является причиной ошибки переполнения буфера, и в силу этого обстоятельства многие системы уязвимы для несанкционированного доступа, хотя современные меры безопасности, предоставляемые системой времени выполнения и компилятором, помогают сделать программы более безопасными.

Мы пока исследовали только отображение конструкций языка C в машинный код архитектуры x86-64, однако многое из того, что нам удалось рассмотреть, применимо ко многим другим сочетаниям языков программирования высокого и машинного уровня. Например, компиляция программного кода на C++ во многом подобна компиляции кода на C. По сути дела, ранние реализации языка C++ просто выполняли преобразование на уровне исходного кода программ на C++ в программы на C и генерировали объектный код с помощью компилятора C. Объекты в языке C++ представлены структурами, подобными структурам `struct` в языке C. Методы представлены указателями на программный код, реализующий их. В отличие от C и C++ язык Java реализован совершенно иначе. Объектный код Java – это особое двоичное представление, известное как *байт-код Java*. Этот код можно рассматривать как программу машинного уровня для *виртуальной машины*. Как следует из этого названия, такая машина реализована не аппаратно, а программно. Программный интерпретатор обрабатывает байт-код, имитируя поведение виртуальной машины. Альтернативный подход, известный как *динамическая компиляция* (*just-in-time compilation*), основан на идее преобразования байт-кода в машинные инструкции прямо в процессе выполнения. Это обеспечивает более высокую скорость работы, когда код выполняется несколько раз, например в циклах. Преимущество использования байт-кода заключается в том, что один и тот же код может «выполняться» на самых разных машинах, в то время как машинный код, который мы рассматривали выше, может выполняться только на машинах с архитектурой x86-64.

## Библиографические заметки

Компании Intel и AMD предоставляют обширную документацию по своим процессорам. В их число входят общие описания аппаратного обеспечения с точки зрения программиста на языке ассемблера [2, 50], а также подробные справочники по отдельным инструкциям [3, 51]. Чтение описаний инструкций осложняется тем фактом, что (1) вся документация основана на формате представления ассемблерного кода Intel, (2) для каждой инструкции имеется множество вариантов, использующих разные режимы адресации и выполнения, и (3) отсутствуют наглядные примеры. Тем не менее они остаются авторитетными источниками информации о поведении каждой инструкции.

Организация x86-64.org отвечает за определение *двоичного прикладного интерфейса* (Application Binary Interface, ABI) для кода x86-64, действующего в системах Linux [77]. В этом интерфейсе подробно описываются порядок компоновки процедур, форматы файлов с двоичным кодом и ряд других особенностей, необходимых для правильного выполнения программ в машинном коде.

Как мы уже отмечали, формат АТТ, используемый компилятором GCC, сильно отличается от формата Intel, используемого в документации Intel и другими компиляторами (включая компиляторы Microsoft).

Книга Мучника (Muchnick), посвященная проектированию компиляторов [80], считается наиболее исчерпывающим описанием технологий оптимизации, которые рассматриваются в данной книге, таких как соглашения по использованию регистров.

Во многих книгах рассматривается проблема использования переполнения буферов для несанкционированного доступа через интернет. Подробный анализ программы-червя, появившейся в 1988 году, был опубликован Спаффордом (Spafford [105]) и другими сотрудниками Массачусетского технологического института (Massachusetts Institute of Technology, MIT), которым удалось остановить его распространение [35]. С тех пор в печати появилось множество статей и проектов, направленных на реализацию и предотвращение атак переполнением буфера. В книге Сикорда (Seacord) [97] вы найдете обширную информацию о переполнении буфера и других атаках на код, генерируемый компиляторами C.

## Домашние задания

### Упражнение 3.58 ♦

Для функции с прототипом

```
long decode2(long 314x, long y, long z);
```

компилятор GCC сгенерировал следующий ассемблерный код:

```
1 decode2:
2   subq    %rdx, %rsi
3   imulq   %rsi, %rdi
4   movq    %rsi, %rax
5   salq    $63, %rax
6   sarq    $63, %rax
7   xorq    %rdi, %rax
8   ret
```

Параметры *x*, *y* и *z* передаются в регистрах *%rdi*, *%rsi* и *%rdx*. Процедура сохраняет и возвращает значение в регистре *%rax*.

Напишите на языке C функцию *decode2*, которая возвращает тот же результат, что и программа на ассемблере.

### Упражнение 3.59 ♦♦

Следующий код вычисляет 128-разрядное произведение двух 64-разрядных целых со знаком – *x* и *y* – и сохраняет результат в памяти:

```
1 typedef __int128 int128_t;
2
3 void store_prod(int128_t *dest, int64_t x, int64_t y) {
4     *dest = x * (int128_t) y;
5 }
```

GCC сгенерировал следующий ассемблерный код:

```
1 store_prod:
2   movq    %rdx, %rax
3   cqto
4   movq    %rsi, %rcx
5   sarq    $63, %rcx
6   imulq   %rax, %rcx
7   imulq   %rsi, %rdx
8   addq    %rdx, %rcx
9   mulq    %rsi
10  addq    %rcx, %rdx
```

```

11    movq    %rax, (%rdi)
12    movq    %rdx, 8(%rdi)
13    ret

```

Этот код использует три операции умножения, необходимые для получения 128-разрядного произведения на 64-разрядной машине. Опишите используемый алгоритм и добавьте комментарии в ассемблерный код, чтобы показать, как он реализует ваш алгоритм.

*Подсказка:* операции расширения аргументов  $x$  и  $y$  до 128 разрядных значений можно записать как  $x = 2^{64} \cdot x_h + x_l$  и  $y = 2^{64} \cdot y_h + y_l$ , где  $x_h, x_l, y_h$  и  $y_l$  – 64-разрядные значения. Точно так же 128-разрядное произведение можно записать как  $p = 2^{64} \cdot p_h + p_l$ , где  $p_h$  и  $p_l$  – 64-разрядные значения. Покажите, как код вычисляет значения  $p_h$  и  $p_l$  в терминах  $x_h, x_l, y_h$  и  $y_l$ .

### Упражнение 3.60 ♦♦

Взгляните на следующий ассемблерный код:

```

        long loop(long x, int n)
        x в %rdi, n в %esi
1   loop:
2       movl    %esi, %ecx
3       movl    $1, %edx
4       movl    $0, %eax
5       jmp     .L2
6   .L3:
7       movq    %rdi, %r8
8       andq    %rdx, %r8
9       orq     %r8, %rax
10      salq    %cl, %rdx
11   .L2:
12      testq    %rdx, %rdx
13      jne     .L3
14      rep; ret

```

Он был сгенерирован при компиляции кода на C со следующей общей структурой:

```

1 long loop(long x, int n)
2 {
3     long result = ____;
4     long mask;
5     for (mask = ____; mask ____; mask = ____) {
6         result |= ____;
7     }
8     return result;
9 }

```

Ваша задача – восстановить недостающие фрагменты, чтобы получить программу, эквивалентную сгенерированному ассемблерному коду. Напомним, что функция возвращает результат в регистре `%rax`. Возможно, вам поможет подробный анализ ассемблерного кода до, во время и после выполнения цикла, чтобы уловить связь между регистрами и переменными программы.

1. В каких регистрах хранятся значения  $x$ ,  $n$ , `result` и `mask`?
2. Какие начальные значения получают `result` и `mask`?
3. Какие условия проверяются для значения `mask`?
4. Как изменяется значение `mask`?
5. Как изменяется `result`?
6. Восстановите недостающие фрагменты в коде на C.

### Упражнение 3.61 ♦♦

В разделе 3.6.6 мы исследовали следующий код на роль кандидата для использования условной передачи данных:

```
long cread(long *xp) {
    return (xp ? *xp : 0);
}
```

Там же мы показали пробную реализацию с использованием инструкции условного перемещения, отметив при этом, что она ошибочная, потому что может попытаться прочесть значение из нулевого адреса.

Напишите функцию `cread_alt` на C, которая действует так же, как `cread`, но при этом ее можно скомпилировать с использованием инструкции условной передачи данных. Сгенерированный при компиляции код должен использовать условную инструкцию перемещения, а не одну из инструкций перехода.

### Упражнение 3.62 ♦♦

Следующий код демонстрирует пример ветвления по значению перечислимого типа в операторе выбора `switch`. Напоминаем, что перечислимые типы в C – это просто способ добавления некоторого множества имен, с которыми связаны конкретные целые числа. По умолчанию значения, присваиваемые этим именам, начинаются с нуля и возрастают с каждым значением. В нашем коде действия, связанные с метками вариантов `case`, опущены.

```
1 /* Перечислимый тип определяет множество констант со значениями от 0 и выше */
2 typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;
3
4 long switch3(long *p1, long *p2, mode_t action)
5 {
6     long result = 0;
7     switch(action) {
8     case MODE_A:
9
10    case MODE_B:
11
12    case MODE_C:
13
14    case MODE_D:
15
16    case MODE_E:
17
18    default:
19
20    }
21    return result;
22 }
```

Часть сгенерированного ассемблерного кода, реализующего различные действия, представлена в листинге 3.22. В комментариях указаны местоположения аргументов, значения, хранящиеся в регистрах, и метки различных вариантов для переходов.

**Листинг 3.22.** Ассемблерный код для упражнения 3.62. Этот код реализует различные варианты в операторе `switch`

```
p1 в %rdi, p2 в %rsi, action в %edx
1 .L8:                                MODE_E
```

```

2    movl    $27, %eax
3    ret
4    .L3:                                MODE_A
5    movq    (%rsi), %rax
6    movq    (%rdi), %rdx
7    movq    %rdx, (%rsi)
8    ret
9    .L5:                                MODE_B
10   movq    (%rdi), %rax
11   addq    (%rsi), %rax
12   movq    %rax, (%rdi)
13   ret
14   .L6:                                MODE_C
15   movq    $59, (%rdi)
16   movq    (%rsi), %rax
17   ret
18   .L7:                                MODE_D
19   movq    (%rsi), %rax
20   movq    %rax, (%rdi)
21   movl    $27, %eax
22   ret
23   .L9:                                default
24   movl    $12, %eax
25   ret

```

Восстановите недостающие фрагменты в коде на С. Обратите внимание, что среди вариантов есть один, который «проваливается» в следующий за ним, – попробуйте найти его и восстановить.

### Упражнение 3.63 ♦♦

Это упражнение дает вам возможность проверить свои навыки в восстановлении кода оператора switch по дизассемблированному коду. В следующей процедуре опущено тело оператора switch.

```

1 long switch_prob(long x, long n) {
2     long result = x;
3     switch(n) {
4         /* Восстановите тело оператора */
5
6     }
7     return result;
8 }

```

В листинге 3.23 представлен дизассемблированный машинный код процедуры.

**Листинг 3.23.** Дизассемблированный код для упражнения 3.63.

```

long switch_prob(long x, long n)
x в %rdi, n в %rsi
1 0000000000400590 <switch_prob>:
2 400590: 48 83 ee 3c          sub $0x3c,%rsi
3 400594: 48 83 fe 05          cmp $0x5,%rsi
4 400598: 77 29               ja 4005c3 <switch_prob+0x33>
5 40059a: ff 24 f5 f8 06 40 00 jmpq *0x4006f8(,%rsi,8)
6 4005a1: 48 8d 04 fd 00 00 00 lea 0x0(,%rdi,8),%rax
7 4005a8: 00
8 4005a9: c3                  retq

```

```

 9  4005aa: 48 89 f8          mov %rdi,%rax
10  4005ad: 48 c1 f8 03       sar $0x3,%rax
11  4005b1: c3              retq
12  4005b2: 48 89 f8          mov %rdi,%rax
13  4005b5: 48 c1 e0 04       shl $0x4,%rax
14  4005b9: 48 29 f8          sub %rdi,%rax
15  4005bc: 48 89 c7          mov %rax,%rdi
16  4005bf: 48 0f af ff       imul %rdi,%rdi
17  4005c3: 48 8d 47 4b       lea 0x4b(%rdi),%rax
18  4005c7: c3              retq

```

Таблица переходов находится в другой области памяти. В строке 5, в инструкции косвенного перехода, можно видеть, что начало таблицы переходов находится по адресу 0x4006f8. Используя отладчик GDB, можно проверить шесть 8-байтных слов в таблице переходов командой `x/6gx 0x4006f8`. Отладчик GDB выведет следующий дамп:

```

(gdb) x/6gx 0x4006f8
0x4006f8: 0x00000000004005a1 0x00000000004005c3
0x400708: 0x00000000004005a1 0x00000000004005aa
0x400718: 0x00000000004005b2 0x00000000004005bf

```

Восстановите тело оператора `switch` в процедуре на C, чтобы его поведение совпадало с поведением дизассемблированного кода.

### Упражнение 3.64 ♦♦♦

Взгляните на следующий исходный код, где R, S и T – это константы, объявленные с помощью директивы `#define`:

```

1 long A[R][S][T];
2
3 long store_ele(long i, long j, long k, long *dest)
4 {
5     *dest = A[i][j][k];
6     return sizeof(A);
7 }

```

Компилятор GCC сгенерировал следующий ассемблерный код:

```

long store_ele(long i, long j, long k, long *dest)
i в %rdi, j в %rsi, k в %rdx, dest в %rcx
1 store_ele:
2     leaq    (%rsi,%rsi,2), %rax
3     leaq    (%rsi,%rax,4), %rax
4     movq    %rdi, %rsi
5     salq    $6, %rsi
6     addq    %rsi, %rdi
7     addq    %rax, %rdi
8     addq    %rdi, %rdx
9     movq    A(,%rdx,8), %rax
10    movq    %rax, (%rcx)
11    movl    $3640, %eax
12    ret

```

1. Распространите уравнение 3.1 на случай с тремя измерениями, чтобы получить формулу для вычисления местоположения элемента массива `A[i][j][k]`.
2. Используя приобретенные навыки, определите значения R, S и T по ассемблерному коду.

### Упражнение 3.65 ♦

Следующий код транспонирует (поворачивает) матрицу с размерами  $M \times M$ , где  $M$  – это константа, объявленная с помощью директивы `#define`:

```
1 void transpose(long A[M][M]) {
2     long i, j;
3     for (i = 0; i < M; i++)
4         for (j = 0; j < i; j++) {
5             long t = A[i][j];
6             A[i][j] = A[j][i];
7             A[j][i] = t;
8         }
9 }
```

При компиляции с уровнем оптимизации `-O1` компилятор GCC сгенерировал следующий код для вложенного цикла:

```
1 .L6:
2     movq    (%rdx), %rcx
3     movq    (%rax), %rsi
4     movq    %rsi, (%rdx)
5     movq    %rcx, (%rax)
6     addq    $8, %rdx
7     addq    $120, %rax
8     cmpq    %rdi, %rax
9     jne     .L6
```

Как видите, GCC преобразовал операции индексирования в операции с указателями.

1. В каком регистре находится указатель на элемент массива `A[i][j]`?
2. В каком регистре находится указатель на элемент массива `A[j][i]`?
3. Каково значение  $M$ ?

### Упражнение 3.66 ♦

Взгляните на следующий исходный код, где `NR` и `NC` – это макросы, объявленные с помощью директивы `#define`. Они вычисляют размеры массива `A` на основе параметра `n`. Этот код вычисляет сумму элементов столбца `j` массива.

```
1 long sum_col(long n, long A[NR(n)][NC(n)], long j) {
2     long i;
3     long result = 0;
4     for (i = 0; i < NR(n); i++)
5         result += A[i][j];
6     return result;
7 }
```

Компилятор GCC сгенерировал следующий ассемблерный код:

```
long sum_col(long n, long A[NR(n)][NC(n)], long j)
n в %rdi, A в %rsi, j в %rdx
1 sum_col:
2     leaq    1(,%rdi,4), %r8
3     leaq    (%rdi,%rdi,2), %rax
4     movq    %rax, %rdi
5     testq   %rax, %rax
6     jle     .L4
7     salq    $3, %r8
8     leaq    (%rsi,%rdx,8), %rcx
9     movl    $0, %eax
```



```

10    movl    $0, %edx
11    .L3:
12    addq    (%rcx), %rax
13    addq    $1, %rdx
14    addq    %r8, %rcx
15    cmpq    %rdi, %rdx
16    jne     .L3
17    rep; ret
18    .L4:
19    movl    $0, %eax
20    ret

```

Используя полученные навыки, определите, как объявлены макросы NR и NC.

### Упражнение 3.67 ♦♦

В этом упражнении вам предстоит исследовать ассемблерный код, сгенерированный компилятором GCC для функций, принимающих и возвращающих структуры, и понять, как обычно реализуются такие функции.

В следующем коде на C представлена функция `process`, принимающая и возвращающая структуру, а также функция `eval`, которая вызывает функцию `process`:

```

1  typedef struct {
2      long a[2];
3      long *p;
4  } strA;
5
6  typedef struct {
7      long u[2];
8      long q;
9  } strB;
10
11 strB process(strA s) {
12     strB r;
13     r.u[0] = s.a[1];
14     r.u[1] = s.a[0];
15     r.q = *s.p;
16     return r;
17 }
18
19 long eval(long x, long y, long z) {
20     strA s;
21     s.a[0] = x;
22     s.a[1] = y;
23     s.p = &z;
24     strB r = process(s);
25     return r.u[0] + r.u[1] + r.q;
26 }

```

Для этих двух функций GCC сгенерировал следующий ассемблерный код:

```

    strB process(strA s)
1 process:
2     movq    %rdi, %rax
3     movq    24(%rsp), %rdx
4     movq    (%rdx), %rdx
5     movq    16(%rsp), %rcx
6     movq    %rcx, (%rdi)
7     movq    8(%rsp), %rcx
8     movq    %rcx, 8(%rdi)

```

```

9     movq    %rdx, 16(%rdi)
10    ret

    long eval(long x, long y, long z)
    x в %rdi, y в %rsi, z в %rdx
1   eval:
2     subq    $104, %rsp
3     movq    %rdx, 24(%rsp)
4     leaq    24(%rsp), %rax
5     movq    %rdi, (%rsp)
6     movq    %rsi, 8(%rsp)
7     movq    %rax, 16(%rsp)
8     leaq    64(%rsp), %rdi
9     call    process
10    movq    72(%rsp), %rax
11    addq    64(%rsp), %rax
12    addq    80(%rsp), %rax
13    addq    $104, %rsp
14    ret

```

1. В строке 2 функции eval выделяет 104 байта в стеке. Нарисуйте схему кадра стека для eval, показывающую значения, которые хранятся в стеке до вызова process.
2. Какое значение передает eval при вызове process?
3. Как process обращается к элементам структуры в аргументе s?
4. Как process изменяет поля структуры результата r?
5. Нарисуйте аналогичную схему кадра стека для eval и покажите, как eval обращается к элементам структуры r после возврата из process.
6. Какие основные принципы, связанные с передачей и возвратом структур, вы подметили?

### Упражнение 3.68 ♦♦♦

В следующем коде A и B – это константы, объявленные с помощью директивы #define:

```

1   typedef struct {
2       int x[A][B]; /* Неизвестные константы A и B */
3       long y;
4   } str1;
5
6   typedef struct {
7       char array[B];
8       int t;
9       short s[A];
10      long u;
11  } str2;
12
13  void setVal(str1 *p, str2 *q) {
14      long v1 = q->t;
15      long v2 = q->u;
16      p->y = v1+v2;
17  }

```

Для setVal компилятор GCC сгенерировал следующий ассемблерный код:

```

    void setVal(str1 *p, str2 *q)
    p в %rdi, q в %rsi
1   setVal:
2     movslq  8(%rsi), %rax
3     addq    32(%rsi), %rax

```

```

4   movq    %rax, 184(%rdi)
5   ret

```

Какие значения имеют А и В?

### Упражнение 3.69 ♦♦♦

Вам поручена поддержка большой программы на С, и вы столкнулись со следующим кодом:

```

1  typedef struct {
2      int first;
3      a_struct a[CNT];
4      int last;
5  } b_struct;
6
7  void test(long i, b_struct *bp)
8  {
9      int n = bp->first + bp->last;
10     a_struct *ap = &bp->a[i];
11     ap->x[ap->idx] = n;
12 }

```

Объявления константы CNT и структуры a\_struct находятся в файле, для доступа к которому у вас нет необходимых прав. К счастью, у вас есть копия объектного кода, которую можно дизассемблировать с помощью программы OBJDUMP, что вы и сделали, получив следующую дизассемблированную версию:

```

void test(long i, b_struct *bp)
i в %rdi, bp в %rsi
1  0000000000000000 <test>:
2      0:  8b 8e 20 01 00 00      mov     0x120(%rsi),%ecx
3      6:  03 0e                    add     (%rsi),%ecx
4      8:  48 8d 04 bf              lea     (%rdi,%rdi,4),%rax
5      c:  48 8d 04 c6              lea     (%rsi,%rax,8),%rax
6     10:  48 8b 50 08              mov     0x8(%rax),%rdx
7     14:  48 63 c9                movslq  %ecx,%rcx
8     17:  48 89 4c d0 10          mov     %rcx,0x10(%rax,%rdx,8)
9     1c:  c3                      retq

```

Используя полученные навыки, определите:

1. Значение CNT.
2. Полное определение структуры a\_struct. Предположите, что эта структура содержит только два поля, idx и x, и оба они хранят целые числа со знаком.

### Упражнение 3.70 ♦♦♦

Взгляните на следующее объявление объединения:

```

1  union ele {
2      struct {
3          long *p;
4          long y;
5      } e1;
6      struct {
7          long x;
8          union ele *next;
9      } e2;
10 };

```

Как показывает это объявление, структуры тоже могут быть частью объединения.

Следующая функция (в которой опущены некоторые выражения) работает со связным списком, содержащим элементы с типом этого объединения:

```
1 void proc (union ele *up) {
2     up->_____ = *( _____ ) - _____;
3 }
```

1. Какое смещение в байтах имеют следующие поля:

```
e1.p      _____
e1.y      _____
e2.x      _____
e2.next   _____
```

2. Сколько всего байтов занимает структура?
3. Компилятор сгенерировал для `proc` следующий код:

```
void proc (union ele *up)
up в %rdi
1 proc:
2     movq    8(%rdi), %rax
3     movq    (%rax), %rdx
4     movq    (%rdx), %rdx
5     subq    8(%rax), %rdx
6     movq    %rdx, (%rdi)
7     ret
```

Опираясь на эту информацию, восстановите недостающие выражения в исходном коде `proc`.

*Подсказка:* некоторые ссылки на объединение могут иметь неоднозначную интерпретацию. Однако эти неоднозначности легко разрешаются, если проследить, куда ведут ссылки. В каждом случае есть только один ответ, не требующий никаких преобразований и не нарушающий никаких ограничений типов.

### Упражнение 3.71 ♦

Напишите функцию `good_echo`, которая читает строку из стандартного ввода и выводит ее в стандартный вывод. Ваша реализация должна поддерживать ввод строк произвольной длины. Можете воспользоваться библиотечной функцией `fgets`, но прежде убедитесь, что ваша функция работает правильно, даже когда входная строка требует больше места в памяти, чем вы отвели для своего буфера. Ваш код должен также выявлять условия ошибки и сообщать о них. При необходимости обращайтесь к определениям стандартных функций ввода/вывода в документации [45, 61].

### Упражнение 3.72 ♦♦

В листинге 3.24 (а) показан код функции, действующей подобно функции `vfunct` (листинг 3.20 (а)). Выше мы использовали `vfunct`, чтобы показать, как используется указатель кадра для управления кадрами стека переменного размера. Новая функция `aframe` выделяет место для локального массива `p` вызовом библиотечной функции `alloca`. Эта функция похожа на более часто используемую функцию `malloc`, с той лишь разницей, что выделяет место в стеке. При возврате из процедуры это пространство автоматически освобождается.

**Листинг 3.24.** Код для упражнения 3.72. Эта функция похожа на `vfunct` из листинга 3.20

(а) Код на C

```
1 #include <alloca.h>
```

```

2
3 long aframe(long n, long idx, long *q) {
4     long i;
5     long **p = alloca(n * sizeof(long *));
6     p[0] = &i;
7     for (i = 1; i < n; i++)
8         p[i] = q;
9     return *p[idx];
10 }

```

(b) Фрагмент сгенерированного ассемблерного кода

```

long aframe(long n, long idx, long *q)
n в %rdi, idx в %rsi, q в %rdx
1 aframe:
2     pushq    %rbp
3     movq     %rsp, %rbp
4     subq     $16, %rsp                Выделить место для i (%rsp = s1)
5     leaq     30(, %rdi, 8), %rax
6     andq     $-16, %rax
7     subq     %rax, %rsp                Выделить место для массива p (%rsp = s2)
8     leaq     15(%rsp), %r8
9     andq     $-16, %r8                Записать &p[0] в %r8
.
.
.

```

В листинге 3.24 (b) показан фрагмент ассемблерного кода, устанавливающий указатель кадра и выделяющий место для локальных переменных  $i$  и  $p$ . Он очень похож на соответствующий код в `vframe`. Здесь использованы те же обозначения, что и в упражнении 3.49: в указатель стека записывается значение  $s_1$  в строке 4 и  $s_2$  в строке 7. Начальный адрес массива  $p$  устанавливается в значение  $p$  в строке 9. Между  $s_2$  и  $p$  может возникнуть дополнительное неиспользуемое пространство  $e_2$ . Аналогично между концом массива  $p$  и  $s_1$  может возникнуть дополнительное неиспользуемое пространство  $e_1$ .

1. Объясните математическим языком логику вычисления  $s_2$ .
2. Объясните математическим языком логику вычисления  $p$ .
3. Определите значения  $n$  и  $s_1$ , которые дают минимальное и максимальное значения  $e_1$ .
4. Какие свойства выравнивания гарантирует этот код для значений  $s_2$  и  $p$ ?

### Упражнение 3.73 ♦

Напишите функцию на ассемблере, поведение которой соответствует поведению функции `find_range` в листинге 3.21. Она должна иметь только одну инструкцию сравнения значений с плавающей точкой и использовать условные переходы для создания правильного результата. Протестируйте свой код на всех  $2^{32}$  возможных значениях аргументов. В приложении в интернете ASM:EASM, представленном в разделе 3.3, описывается, как внедрять функции на ассемблере в программы на языке C.

### Упражнение 3.74 ♦♦

Напишите функцию на ассемблере, поведение которой соответствует поведению функции `find_range` в листинге 3.21. Она должна иметь только одну инструкцию сравнения значений с плавающей точкой и использовать условные переходы для создания

правильного результата. Попробуйте использовать инструкцию `stovr` (перемещает данные, если установлен флаг четности). В приложении в интернете ASM:EASM, представленном в разделе 3.3, описывается, как внедрять функции на ассемблере в программы на языке C.

### Упражнение 3.75 ♦

Стандарт ISO C99 включает расширения для поддержки комплексных чисел. Любой тип с плавающей точкой можно изменить с помощью ключевого слова `complex`. Вот несколько примеров функций, которые работают с комплексными данными и вызывают соответствующие библиотечные функции:

```

1 #include <complex.h>
2
3 double c_imag(double complex x) {
4     return cimag(x);
5 }
6
7 double c_real(double complex x) {
8     return creal(x);
9 }
10
11 double complex c_sub(double complex x, double complex y) {
12     return x - y;
13 }
```

Для этих функций GCC сгенерировал следующий ассемблерный код:

```

double c_imag(double complex x)
1 c_imag:
2     movapd    %xmm1, %xmm0
3     ret

double c_real(double complex x)
4 c_real:
5     rep; ret

double complex c_sub(double complex x, double complex y)
6 c_sub:
7     subsd     %xmm2, %xmm0
8     subsd     %xmm3, %xmm1
9     ret
```

Опираясь на эти примеры, определите:

1. Как комплексные значения передаются функции?
2. Как комплексные значения возвращаются из функции?

## Решения упражнений

### Решение упражнения 3.1

Это упражнение дает возможность попрактиковаться в работе с операндами разных видов.

Операнд	Значение	Комментарий
%rax	0x100	Регистр
0x104	0xAB	Абсолютный адрес

Операнд	Значение	Комментарий
\$0x108	0x108	Непосредственное значение
(%rax)	0xFF	Адрес 0x100
4(%rax)	0xAB	Адрес 0x104
9(%rax,%rdx)	0x11	Адрес 0x10C
260(%rcx,%rdx)	0x13	Адрес 0x108
0xFC(%rcx,4)	0xFF	Адрес 0x100
(%rax,%rdx,4)	0x11	Адрес 0x10C

### Решение упражнения 3.2

Как вы не раз видели, ассемблерный код, сгенерированный компилятором GCC, добавляет суффиксы в имена инструкций, а дизассемблер – нет. Умение переключаться между этими двумя формами – важный навык, которому нужно учиться. Одна важная особенность состоит в том, что в архитектуре x86-64 ссылки на память всегда задаются полными 8-байтными регистрами, такими как %rax, даже если операнд является 1-, 2- или 4-байтным.

Вот код с добавленными суффиксами:

```
movl    %eax, (%rsp)
movw    (%rax), %dx
movb    $0xFF, %bl
movb    (%rsp,%rdx,4), %dl
movq    (%rdx), %rax
movw    %dx, (%rax)
```

### Решение упражнения 3.3

В большей части книги мы опираемся на ассемблерный код, сгенерированный компилятором GCC, поэтому умение писать правильный ассемблерный код не является критически важным навыком. Тем не менее это упражнение поможет вам поближе познакомиться с различными типами инструкций и операндов.

Вот код с описанием причин ошибок:

```
movb    $0xF, (%ebx)    %ebx нельзя использовать в роли адресного регистра
movl    %rax, (%rsp)    Несоответствие между суффиксом инструкции
                        и именами регистров
movw    (%rax), 4(%rsp)  Ссылкой на память может быть только один операнд,
                        но не оба сразу
movb    %al, %sl        Нет регистра с именем %sl
movq    %rax, $0x123    Непосредственное значение не может быть приемником
movl    %eax, %rdx       Неверный размер операнда-приемника
movb    %si, 8(%rbp)    Несоответствие между суффиксом инструкции
                        и именем регистра
```

### Решение упражнения 3.4

Это упражнение позволяет получить дополнительный опыт работы с различными инструкциями перемещения данных и понять, как они связаны с типами данных и правилами преобразования в языке C. Нюансы преобразования между значениями со знаком и без знака и значениями разного размера усложняют эту задачу.

src_t	dest_t	Инструкция	Комментарий
long	long	movq (%rdi), %rax movq %rax, (%rsi)	Читает 8 байт Сохраняет 8 байт
char	int	movsbl (%rdi), %eax movl %eax, (%rsi)	Преобразует char в int Сохраняет 4 байта
char	unsigned	movsbl (%rdi), %eax movl %eax, (%rsi)	Преобразует char в int Сохраняет 4 байта
unsigned char	long	movzbl (%rdi), %eax <sup>5</sup> movq %rax, (%rsi)	Читает байт и дополняет нулями Сохраняет 8 байт
int	char	movl (%rdi), %eax movb %al, (%rsi)	Читает 4 байта Сохраняет младший байт
unsigned	unsigned char	movl (%rdi), %eax movb %al, (%rsi)	Читает 4 байта Сохраняет младший байт
char	short	movsbw (%rdi), %ax movw %ax, (%rsi)	Читает байт и расширяет знаковый разряд Сохраняет 2 байта

Решение упражнения 3.5

Обратное проектирование – отличный способ освоить систему. В этом упражнении вы должны были выполнить обратное действие – на основе кода, сгенерированного компилятором, восстановить исходный код на языке C, чтобы узнать, почему был получен именно такой ассемблерный код. «Моделирование» лучше начать со значений x, y и z, на которые ссылаются указатели xp, yp и zp соответственно. В данном случае мы можем описать поведение кода так:

```
void decode1(long *xp, long *yp, long *zp)
xp в %rdi, yp в %rsi, zp в %rdx
decode1:
movq    (%rdi), %r8    Получить x = *xp
movq    (%rsi), %rcx   Получить y = *yp
movq    (%rdx), %rax   Получить z = *zp
movq    %r8, (%rsi)    Сохранить x по адресу в yp
movq    %rcx, (%rdx)   Сохранить y по адресу в zp
movq    %rax, (%rdi)   Сохранить z по адресу в xp
ret
```

Отсюда легко получить исходный код на C:

```
void decode1(long *xp, long *yp, long *zp)
{
    long x = *xp;
    long y = *yp;
    long z = *zp;

    *yp = x;
    *zp = y;
    *xp = z;
}
```

Решение упражнения 3.6

Это упражнение показывает, насколько разносторонней является инструкция leaq, оно позволяет попрактиковаться в декодировании операндов разных форм. Несмотря на то

<sup>5</sup> В этом случае компилятор GCC генерирует инструкцию movzbl, несмотря на то что конечная цель – расширить 1-байтное значение до 8-байтного. См. комментарии в табл. 3.4.



что эта инструкция относится к категории инструкций для работы с памятью в табл. 3.2, она на самом деле не обращается к памяти.

Инструкция	Результат
leaq 6(%rax), %rdx	$6 + x$
leaq (%rax,%rcx), %rdx	$x + y$
leaq (%rax,%rcx,4), %rdx	$x + 4y$
leaq 7(%rax,%rax,8), %rdx	$7 + 9x$
leaq 0xA(%rcx,4), %rdx	$10 + 4y$
leaq 9(%rax,%rcx,2), %rdx	$9 + x + 2y$

Решение упражнения 3.7

И снова обратное проектирование оказывается полезным приемом изучения взаимосвязи между кодом на C и на ассемблере.

Лучший способ решения задач этого типа – добавить в ассемблерный код комментарии, описывающие выполняемые операции. Например:

```
long scale2(long x, long y, long z)
x в %rdi, y в %rsi, z в %rdx
scale2:
leaq (%rdi,%rdi,4), %rax    5 * x
leaq (%rax,%rsi,2), %rax    5 * x + 2 * y
leaq (%rax,%rdx,8), %rax
ret
```

Отсюда легко вывести отсутствующее выражение:

```
long t = 5 * x + 2 * y + 8 * z;
```

Решение упражнения 3.8

Это упражнение призвано помочь вам проверить понимание арифметических инструкций и их операндов. Последовательность инструкций построена так, чтобы результат каждой предыдущей инструкции не влиял на поведение последующих.

Инструкция	Приемник	Значение
addq %rcx,%rax)	0x100	0x100
subq %rdx,8(%rax)	0x108	0xA8
imulq \$16,(%rax,%rdx,8)	0x118	0x110
incq 16(%rax)	0x110	0x14
decq %rcx	%rcx	0x0
subq %rdx,%rax	%rax	0xFD

Решение упражнения 3.9

Это упражнение дает вам возможность написать небольшой фрагмент на языке ассемблера. Код, показанный в решении, сгенерирован компилятором GCC. Загрузив параметр `n` в регистр `%ecx`, он затем использует 1-байтный регистр `%cl`, чтобы задать величину сдвига для инструкции `sarq`. Использование инструкции `movl` может показаться странным, учитывая, что длина `n` составляет 8 байт, но имейте в виду, что величина сдвига всегда определяется по младшему байту.

```

long shift_left4_rightn(long x, long n)
x в %rdi, n в %rsi
shift_left4_rightn:
movq    %rdi, %rax    Получить x
salq    $4, %rax      x <<= 4
movl    %esi, %ecx    Получить n (4 байта)
sarq    %cl, %rax      x >>= n

```

### Решение упражнения 3.10

Это очень простое упражнение, потому что ассемблерный код практически полностью повторяет структуру кода на С.

```

long t1 = x | y;
long t2 = t1 >> 3;
long t3 = ~t2;
long t4 = z-t3;

```

### Решение упражнения 3.11

1. Эта инструкция обнуляет регистр %rdx, используя свойство  $x \wedge x = 0$  для любого  $x$ . Она соответствует выражению на С:  $x = 0$ .
2. Более прямолинейный способ обнуления регистра %rdx – использовать инструкцию `movq $0, %rdx`.
3. Однако инструкция `xorq` занимает всего 3 байта, а инструкция `movq` – 7. Другие способы обнуления %rdx основаны на свойстве всех инструкций, изменяющих младшие 4 байта, обнулять старшие байты. То есть также можно использовать `xorl %edx, %edx` (2 байта) или `movl $0, %edx` (5 байт).

### Решение упражнения 3.12

Можно просто заменить инструкцию `cqto` любой инструкцией, обнуляющей регистр %rdx, и вместо `idivq` использовать `divq`, в результате чего получится следующий код:

```

void uremdiv(unsigned long x, unsigned long y,
             unsigned long *qp, unsigned long *rp)
x в %rdi, y в %rsi, qp в %rdx, rp в %rcx
1 uremdiv:
2 movq    %rdx, %r8    Скопировать qp
3 movq    %rdi, %rax    Переместить x в младшие 8 байт делимого
4 movl    $0, %edx     Обнулить старшие 8 байт делимого
5 divq    %rsi          Разделить на y
6 movq    %rax, (%r8)   Сохранить частное в qp
7 movq    %rdx, (%rcx)  Сохранить остаток в rp
8 ret

```

### Решение упражнения 3.13

Важно понимать, что ассемблерный код не хранит информации о типах значений. Вместо этого используются разные инструкции для манипулирования операндами, в зависимости от размеров значений и наличия знака. При восстановлении кода на С из последовательности инструкций приходится проводить исследования, чтобы вывести типы данных в программе.

1. Суффикс `l` и идентификаторы регистров указывают, что операция выполняется с 32-разрядными операндами, это операция `<` и она сравнивает значения в дополнительном коде. Мы можем сделать вывод, что тип `data_t` – это `int`.
2. Суффикс `w` и идентификаторы регистров указывают, что операция выполняется с 16-разрядными операндами, это операция `>=` и она сравнивает значения в дополнительном коде. Мы можем сделать вывод, что тип `data_t` – это `short`.

3. Суффикс `b` и идентификаторы регистров указывают, что операция выполняется с 8-разрядными операндами, это операция `<=` и она сравнивает значения без знака. Мы можем сделать вывод, что тип `data_t` – это `unsigned char`.
4. Суффикс `q` и идентификаторы регистров указывают, что операция выполняется с 64-разрядными операндами, это операция `!=` и она одинаково выполняется для аргументов со знаком, без знака и указателей. Мы можем сделать вывод, что тип `data_t` – это `long`, `unsigned long` или указатель любого типа.

### Решение упражнения 3.14

Это упражнение подобно упражнению 3.13, за исключением того, что здесь используются инструкции `TEST`, а не `CMPL`.

1. Суффикс `q` и идентификатор регистра указывают, что операция выполняется с 64-разрядным операндом, это операция `>=` и она сравнивает значения со знаком. Мы можем сделать вывод, что тип `data_t` – это `long`.
2. Суффикс `w` и идентификатор регистра указывают, что операция выполняется с 16-разрядным операндом, это операция `==` и она не различает значения со знаком и без знака. Мы можем сделать вывод, что тип `data_t` – это `short` или `unsigned short`.
3. Суффикс `b` и идентификатор регистра указывают, что операция выполняется с 8-разрядным операндом, это операция `>` и она сравнивает значения без знака. Мы можем сделать вывод, что тип `data_t` – это `unsigned char`.
4. Суффикс `l` и идентификатор регистра указывают, что операция выполняется с 32-разрядным операндом, это операция `<=` и она сравнивает значения со знаком. Мы можем сделать вывод, что тип `data_t` – это `int`.

### Решение упражнения 3.15

Это упражнение требует проанализировать дизассемблированный код и поразмыслить над особенностями представления конечной точки перехода. Одна из его целей – дать вам возможность попрактиковаться в шестнадцатеричной арифметике.

1. Инструкция `je` определяет свою цель как `0x4003fc+0x02`, то есть конечной точкой перехода является адрес `0x4003fe`.

```
4003fa: 74 02    je    4003fe
4003fc: ff d0    callq *%rax
```

2. Инструкция `je` определяет свою цель как `0x400431-12` (потому что `0xf4` – это представление числа `-12` в дополнительном коде). Судя по дизассемблированному коду, конечной точкой перехода является адрес `0x400425`:

```
40042f: 74 f4    je    400425
400431: 5d       pop   %rbp
```

3. Согласно приведенному коду, эта инструкция выполняет переход в точку с абсолютным адресом `0x400547`. Из анализа байтов следует, что этот адрес находится на 2 байта дальше инструкции `pop`. Вычитая одно из другого, получаем адрес `0x400543`. Это подтверждает оригинальный код, сгенерированный дизассемблером:

```
400543: 77 02    ja    400547
400545: 5d       pop   %rbp
```

4. Читая байты в обратном порядке, можно заметить, что точка перехода находится на расстоянии `0xfffff73`, то есть в `-141`. Прибавляя это значение к `0x4005ed` (адресу инструкции `pop`), получаем адрес `0x400560`:

```

4005e8: e9 73 ff ff ff    jmpq    400560
4005ed: 90               nop

```

### Решение упражнения 3.16

Комментирование ассемблерного кода и написание кода на C, реализующего те же действия, – полезная привычка, помогающая при анализе программ на ассемблере. Это упражнение дает вам возможность попрактиковаться на исследовании простого примера потока управления, а также поближе познакомиться с реализацией логических операций.

1. Вот код на C:

```

void goto_cond(long a, long *p) {
    if (p == 0)
        goto done;
    if (*p >= a)
        goto done;
    *p = a;
done:
    return;
}

```

2. Первое условное ветвление является частью реализации выражения &&. Если проверка *p* на неравенство нулю терпит неудачу, то код пропускает проверку *a > \*p*.

### Решение упражнения 3.17

Цель этого упражнения – помочь вам усвоить общие правила трансляции и их применения.

1. Чтобы выполнить трансляцию с использованием этого альтернативного правила, достаточно переставить несколько строк кода:

```

long gotodiff_se_alt(long x, long y) {
    long result;
    if (x < y)
        goto x_lt_y;
    ge_cnt++;
    result = x - y;
    return result;
x_lt_y:
    lt_cnt++;
    result = y - x;
    return result;
}

```

2. В большинстве случаев выбор правила – вопрос вкуса. Но стоит отметить, что оригинальное правило лучше подходит для общего случая, когда нет оператора *else*. В этом случае изменить правило трансляции можно следующим образом:

```

t = условное-выражение;
if (!t)
    goto done;
инструкция-then
done:

```

Результат трансляции на основе альтернативного правила в этом случае выглядит сложнее.

### Решение упражнения 3.18

Это упражнение требует проанализировать структуру вложенных ветвлений, чтобы увидеть, как применялось наше правило трансляции операторов `if`. В целом же этот машинный код почти линейно повторяет код на С.

```
long test(long x, long y, long z) {
    long val = x+y+z;
    if (x < -3) {
        if (y < z)
            val = x*y;
        else
            val = y*z;
    } else if (x > 2)
        val = x*z;
    return val;
}
```

### Решение упражнения 3.19

Это упражнение закрепляет усвоение нашего метода вычисления штрафа за неправильное предсказание.

1. Мы можем применить нашу формулу напрямую и получить:

$$T_{\text{MP}} = 2(31 - 16) = 30.$$

2. В случае ошибки прогнозирования для выполнения функции потребуется  $16 + 30 = 46$  тактов.

### Решение упражнения 3.20

Это упражнение дает возможность изучить особенности условной передачи данных.

1. Операция `OR` – это деление (`/`). Здесь мы видим пример деления на  $2^3$  путем сдвига вправо (см. раздел 2.3.7). Перед сдвигом на  $k = 3$  необходимо добавить смещение  $2^k - 1 = 7$ , когда делимое – отрицательное число.
2. Вот версия ассемблерного кода с комментариями:

```
long arith(long x)
x в %rdi
arith:
    leaq    7(%rdi), %rax    temp = x+7
    testq   %rdi, %rdi       Проверить x
    cmovns  %rdi, %rax       Если x >= 0, temp = x
    sarq    $3, %rax         result = temp >> 3 (= x/8)
    ret
```

Программа создает временное значение, равное  $x + 7$ , ожидая, что  $x$  будет отрицательным и потребует смещения. Команда `cmovns` условно заменяет это число на  $x$ , когда  $x \geq 0$ , а затем выполняется сдвиг на 3 позиции, чтобы получить  $x/8$ .

### Решение упражнения 3.21

Это упражнение похоже на упражнение 3.18, за исключением того, что некоторые условные выражения реализованы посредством условной передачи данных. Отображение данного кода в исходный код на С может показаться сложной задачей, но, немного поразмыслив, вы обнаружите, что он довольно точно следует правилам трансляции.

```
long test(long x, long y) {
    long val = 8*x;
    if (y > 0) {
```

```

        if (x < y)
            val = y-x;
        else
            val = x&y;
    } else if (y <= -2)
        val = x+y;
    return val;
}

```

### Решение упражнения 3.22

1. Если построить таблицу факториалов, вычисленных с типом `int`, то получится следующее:

n	n!	Верно?
1	1	Да
2	2	Да
3	6	Да
4	24	Да
5	120	Да
6	720	Да
7	5 040	Да
8	40 320	Да
9	362 880	Да
10	3 628 800	Да
11	39 916 800	Да
12	479 001 600	Да
13	1 932 053 504	Нет

Как видите, при вычислении  $13!$  произошло переполнение. Как мы узнали в упражнении 2.35, проверить факт переполнения по результату  $x$  при попытке вычислить  $n!$  можно, вычислив  $x/n$  и посмотрев, равно ли оно  $(n - 1)!$  (при условии что мы уже убедились, что при вычислении  $(n - 1)!$  не произошло переполнения). В этом примере мы получаем  $1\,932\,053\,504/13 = 161\,004\,458,667$ . Второй способ проверки: как видно по результатам в таблице, любой факториал больше  $10!$  должен быть кратен 100 и иметь нули в последних двух цифрах. Верное значение  $13!$  составляет 6 227 020 800.

2. Выполнение вычислений с типом данных `long` позволяет расширить диапазон до  $20!$ , что дает в результате 2 432 902 008 176 640 000.

### Решение упражнения 3.23

Код, сгенерированный при компиляции циклов, часто сложно анализировать, потому что компилятор может применять множество различных оптимизаций, а также потому, что иногда трудно увидеть связь между программными переменными и регистрами. Этот конкретный пример демонстрирует несколько мест, где ассемблерный код не является результатом прямой трансляции кода на C.

1. Параметр  $x$  передается в функцию в регистре `%rdi`, однако по коду видно, что на этот регистр нет ни одной ссылки внутри цикла. Зато видно, что в строках 2–5

регистры %rax, %rcx и %rdx инициализируются значениями x, x\*x и x+x. Отсюда можно сделать вывод, что эти регистры хранят программные переменные.

2. Компилятор обнаружил, что указатель p всегда ссылается на x и, следовательно, выражение (\*p)++ просто увеличивает x. Он объединил это увеличение на 1 с увеличением на y, сгенерировав инструкцию leaq в строке 7.
3. Код с комментариями выглядит следующим образом:

```

long dw_loop(long x)
x первоначально хранится в %rdi
1  dw_loop:
2      movq    %rdi, %rax           Скопировать x в %rax
3      movq    %rdi, %rcx
4      imulq   %rdi, %rcx           Вычислить y = x*x
5      leaq    (%rdi,%rdi), %rdx     Вычислить n = 2*x
6      .L2:                          loop:
7      leaq    1(%rcx,%rax), %rax     Вычислить x += y + 1
8      subq    $1, %rdx             Уменьшить n
9      testq   %rdx, %rdx           Проверить n
10     jg      .L2                  Если > 0, перейти к loop
11     rep; ret                     Возврат

```

### Решение упражнения 3.24

Этот ассемблерный код являет яркий пример прямолинейной трансляции цикла с использованием метода перехода в середину. Вот как выглядит соответствующий код на C:

```

long loop_while(long a, long b)
{
    long result = 1;
    while (a < b) {
        result = result * (a+b);
        a = a+1;
    }
    return result;
}

```

### Решение упражнения 3.25

Нельзя сказать, что сгенерированный код точно следует шаблону защищенного-do, и все же нетрудно заметить, что ему эквивалентен следующий код на C:

```

long loop_while2(long a, long b)
{
    long result = b;
    while (b > 0) {
        result = result * a;
        b = b-a;
    }
    return result;
}

```

Вы часто будете видеть случаи, особенно при компиляции с более высокими уровнями оптимизации, когда GCC допускает некоторые вольности в генерируемом коде, сохраняя при этом требуемую функциональность.

### Решение упражнения 3.26

Воссоздание исходного кода на С на основе имеющегося ассемблерного кода является ярким примером обратного проектирования.

1. В этом примере кода ясно видно использование приема трансляции с переходом в середину – инструкция `jmp` в строке 3.
2. Вот исходный код на С:

```
long fun_a(unsigned long x) {
    long val = 0;
    while (x) {
        val ^= x;
        x >>= 1;
    }
    return val & 0x1;
}
```

3. Этот код определяет *четность* аргумента `x`. То есть он возвращает 1, если `x` содержит нечетное число единиц, и 0 в противном случае.

### Решение упражнения 3.27

Это упражнение имеет целью помочь вам закрепить понимание особенностей реализации циклов.

```
long fact_for_gd_goto(long n)
{
    long i = 2;
    long result = 1;
    if (n <= 1)
        goto done;
loop:
    result *= i;
    i++;
    if (i <= n)
        goto loop;
done:
    return result;
}
```

### Решение упражнения 3.28

Это упражнение похоже на упражнение 3.26, но немного труднее, потому что код внутри цикла сложнее, а общий принцип его действия менее знаком.

1. Вот исходный код на С:

```
long fun_b(unsigned long x) {
    long val = 0;
    long i;
    for (i = 64; i != 0; i--) {
        val = (val << 1) | (x & 0x1);
        x >>= 1;
    }
    return val;
}
```



2. При трансляции использовался шаблон защищенного-do, но компилятор обнаружил, что  $i$  инициализируется значением 64 и удовлетворяет проверке  $i \neq 0$ , поэтому начальная проверка не требуется.
3. Этот код переставляет биты в  $x$  в обратном порядке, создавая зеркальное отображение. Для этого он сдвигает биты в  $x$  вправо и вталкивает их в  $val$  со сдвигом влево.

### Решение упражнения 3.29

Представленное нами правило преобразования цикла `for` в цикл `while` слишком упрощено – это единственный аспект, требующий особого внимания.

1. Применение нашего правила трансляции даст следующий код:

```
/* Упрощенное преобразование цикла for в цикл while */
/* ВНИМАНИЕ: этот код содержит ошибку! */
long sum = 0;
long i = 0;
while (i < 10) {
    if (i & 1)
        /* Оператор continue превращает цикл в бесконечный */
        continue;
    sum += i;
    i++;
}
```

Этот код будет выполнять цикл до бесконечности, потому что условие выполнения оператора `continue` помешает обновлению переменной цикла  $i$ .

2. Универсальное решение – заменить оператор `continue` оператором `goto`, который пропускает остальную часть тела цикла и переходит непосредственно к обновлению переменной цикла:

```
/* Верное преобразование цикла for в цикл while */
long sum = 0;
long i = 0;
while (i < 10) {
    if (i & 1)
        goto update;
    sum += i;
update:
    i++;
}
```

### Решение упражнения 3.30

Это упражнение дает возможность провести анализ потока управления оператора `switch`. Для ответа на вопросы придется отыскать нужную информацию в ассемблерном коде:

- в строке 2 ассемблерного кода к  $x$  прибавляется 1, чтобы установить нижнюю границу диапазона равной 0. Из этого следует, что минимальная метка равна  $-1$ ;
- строки 3 и 4 производят переход к варианту по умолчанию, если приведенное значение больше 8. Отсюда следует, что максимальная метка случая равна  $-1 + 8 = 9$ ;
- строки 6 (вариант 3) и 9 (вариант 6) в таблице переходов содержат один и тот же адрес (.L2) для инструкции перехода в строке 4, соответствующий варианту по умолчанию. Следовательно, варианты 3 и 6 отсутствуют в теле оператора;

- строки 3 и 10 в таблице переходов, соответствующие вариантам 0 и 7, содержат один и тот же адрес;
- строки 5 и 7 в таблице переходов, соответствующие вариантам 2 и 4, содержат один и тот же адрес.

Из этого вытекают следующие выводы.

1. Метки вариантов в теле оператора switch выбора имеют значения -1, 0, 1, 2, 4, 5 и 7.
2. Варианты с адресами перехода .L5 имеют метки 0 и 7.
3. Варианты с адресами перехода .L7 имеют метки 2 и 4.

### Решение упражнения 3.31

Для восстановления исходного кода на C по ассемблерному коду с реализацией оператора switch необходимо объединить информацию, имеющуюся в самом коде и в таблице переходов. Судя по инструкции ja (строка 3), код для варианта по умолчанию находится по адресу .L2. Единственный другой повторяющийся адрес в таблице переходов – .L5, он соответствует вариантам C и D. Код в строке 8 «проваливается» ниже, поэтому адрес .L7 соответствует варианту A, а адрес .L3 – варианту B. Остается только адрес .L6, который соответствует оставшемуся варианту E.

Таким образом, исходный код на C выглядит следующим образом:

```
void switcher(long a, long b, long c, long *dest)
{
    long val;
    switch(a) {
    case 5:
        c = b ^ 15;
        /* Проваливается дальше */
    case 0:
        val = c + 112;
        break;
    case 2:
    case 7:
        val = (c + b) << 2;
        break;
    case 4:
        val = a;
        break;
    default:
        val = b;
    }
    *dest = val;
}
```

### Решение упражнения 3.32

Трассировка выполнения программы с таким уровнем детализации требует знания многих аспектов вызова процедур и возврата из них. Здесь ясно видно, как вызывающая функция передает управление другой функции и как получает его обратно. Также видно, что аргументы передаются через регистры %rdi и %rsi, а результат возвращается в регистре %rax.

Метка	Инструкция		Состояние (к началу)					Описание
	PC	Инструкция	%rdi	%rsi	%rax	%rsp	*%rsp	
M1	0x400560	callq	10	–	–	0x7fffffff820	–	Вызов first(10)
F1	0x400548	lea	10	–	–	0x7fffffff818	0x400565	Вход в first
F2	0x40054c	sub	10	11	–	0x7fffffff818	0x400565	
F3	0x400550	callq	9	11	–	0x7fffffff818	0x400565	Вызов last(9, 11)
L1	0x400540	mov	9	11	–	0x7fffffff810	0x400555	Вход в last
L2	0x400543	imul	9	11	9	0x7fffffff810	0x400555	
L3	0x400547	retq	9	11	99	0x7fffffff810	0x400555	Возврат 99 из last
F4	0x400555	repz retq	9	11	99	0x7fffffff818	0x400565	Возврат 99 из first
M2	0x400565	mov	9	11	99	0x7fffffff820	–	Возобновление main

### Решение упражнения 3.33

Это упражнение немного сложнее из-за того, что в вычислениях участвуют данные разных размеров.

Опишем сначала одно из возможных решений, а затем объясним второе. Если предположить, что первое сложение (строка 3) реализует `*u += a`, а второе (строка 4) реализует `*v += b`, то получается, что `a` передается в первом аргументе в `%edi` и преобразуется из 4-байтного представления в 8-байтное перед сложением с 8-байтным слагаемым в `%rdx`. То есть переменная `a` должна иметь тип `int`, а переменная `u` – тип `long` \*. Учитывая, что младший байт аргумента `b` прибавляется к байту, на который указывает `%rcx`, переменная `v` должна иметь тип `char` \*, но тип `b` нельзя определить однозначно – в действительности он может иметь размер 1, 2, 4 или 8 байт. Эта неоднозначность разрешается анализом возвращаемого значения 6, вычисленного как сумма размеров `a` и `b`. Поскольку мы знаем, что `a` имеет размер 4 байта, то можно сделать вывод, что `b` должно иметь размер 2 байта.

Версия функции с комментариями объясняет все эти детали:

```
int procpobrl(int a, short b, long *u, char *v)
a в %edi, b в %si, u в %rdx, v в %rcx
1 procpobrl:
2 movslq %edi, %rdi      Преобразовать a в long
3 addq    %rdi, (%rdx)    Прибавить к *u (long)
4 addb    %sil, (%rcx)    Прибавить младший байт b к *v
5 movl    $6, %eax       Вернуть 4 + 2
6 ret
```

А теперь рассмотрим альтернативное решение: один и тот же ассемблерный код был бы верен, если бы две суммы были вычислены в обратном порядке. То есть аргументы `a` и `b` и аргументы `u` и `v` можно поменять местами и получить следующий прототип:

```
int procpobrl(int b, short a, long *v, char *u);
```

### Решение упражнения 3.34

В этом примере демонстрируется использование регистров, сохраняемых вызываемым кодом, и стека для хранения локальных данных.

1. Инструкции в строках 9–14 сохраняют локальные значения `a0`–`a5` в регистры `%rbx`, `%r15`, `%r14`, `%r13`, `%r12` и `%rbp` соответственно, которые должны сохраняться вызываемым кодом.

2. Локальные значения  $a_6$  и  $a_7$  сохраняются в стеке со смещениями 0 и 8 относительно вершины стека (строки 16 и 18).
3. После сохранения шести локальных переменных программа исчерпала запас регистров, которые должны сохраняться вызываемым кодом, поэтому два оставшихся локальных значения сохраняются в стеке.

### Решение упражнения 3.35

Это упражнение помогает понять, как работают рекурсивные функции. Важно усвоить, что рекурсивный код имеет точно такую же структуру, как и другие функции, которые мы видели до сих пор. Соблюдения правил выделения кадра стека и сохранения регистров достаточно, чтобы обеспечить правильную работу рекурсивных функций.

1. Регистр `%rbx` хранит значение параметра  $x$ , поэтому его можно использовать для вычисления результата.
2. Ассемблерный код был сгенерирован из следующего кода на C:

```
long rfun(unsigned long x) {
    if (x == 0)
        return 0;
    unsigned long nx = x>>2;
    long rv = rfun(nx);
    return x + rv;
}
```

### Решение упражнения 3.36

Это упражнение проверяет, насколько правильно вы понимаете такие вещи, как размерность данных и индексирование массивов. Обратите внимание на то обстоятельство, что указатель на любой тип данных имеет размер 8 байт. Тип данных `short` имеет размер 2 байта, а тип `int` – 4 байта.

Массив	Размер элемента	Общий размер	Начальный адрес	Элемент $i$
S	2	14	$x_s$	$x_s + 2i$
T	8	24	$x_t$	$x_t + 8i$
U	8	48	$x_u$	$x_u + 8i$
V	4	32	$x_v$	$x_v + 4i$
W	8	32	$x_w$	$x_w + 8i$

### Решение упражнения 3.37

Это упражнение – вариант упражнения с целочисленным массивом `E`. Важно понимать различия между указателем и объектом, на который он указывает. Поскольку тип данных `short` имеет размер 2 байта, все индексы умножаются на коэффициент 2. А вместо инструкции `movl`, как раньше, мы используем инструкцию `movw`.

Выражение	Тип	Значение	Ассемблерный код
$S+1$	short *	$x_s + 2$	<code>leaq 2(%rdx),%rax</code>
$S[3]$	short	$M[x_s + 6]$	<code>movw 6(%rdx),%ax</code>
$\&S[i]$	short *	$x_s + 2i$	<code>leaq (%rdx,%rcx,2),%rax</code>
$S[4*i+1]$	short	$M[x_s + 8i + 2]$	<code>movw 2(%rdx,%rcx,8),%ax</code>
$S+i-5$	short *	$x_s + 2i - 10$	<code>leaq -10(%rdx,%rcx,2),%rax</code>

### Решение упражнения 3.38

Это упражнение требует правильного применения операций масштабирования для вычисления адресов и уравнения 3.1 для вычисления старшего индекса – индекса строки. Первый шаг – снабдить ассемблерный код комментариями, которые помогут определить, как следует вычислять адресные ссылки:

```

long sum_element(long i, long j)
i в %rdi, j в %rsi
1 sum_element:
2 leaq    0(,%rdi,8), %rdx      Вычислить 8i
3 subq    %rdi, %rdx           Вычислить 7i
4 addq    %rsi, %rdx           Вычислить 7i + j
5 leaq    (%rsi,%rsi,4), %rax   Вычислить 5j
6 addq    %rax, %rdi           Вычислить i + 5j
7 movq    Q(,%rdi,8), %rax     Извлечь M[x0 + 8 (5j + i)]
8 addq    P(,%rdx,8), %rax     Прибавить M[xp + 8 (7i + j)]
9 ret

```

Теперь видно, что ссылка на матрицу P – это смещение  $8 \cdot (7i + j)$  в байтах, а ссылка на матрицу Q – это смещение  $8 \cdot (5j + i)$  в байтах. Отсюда можно определить, что в матрице P 7 столбцов, а в матрице Q 5 столбцов. Следовательно,  $M = 5$  и  $N = 7$ .

### Решение упражнения 3.39

Эти вычисления производятся прямым применением уравнения 3.1:

- для  $L = 4$ ,  $C = 16$  и  $j = 0$  указатель Aptr вычисляется как  $x_A + 4 \cdot (16i + 0) = x_A + 64i$ ;
- для  $L = 4$ ,  $C = 16$ ,  $i = 0$  и  $j = k$  Bptr вычисляется как  $x_B + 4 \cdot (16 \cdot 0 + k) = x_B + 4k$ ;
- для  $L = 4$ ,  $C = 16$ ,  $i = 16$  и  $j = k$  Bend вычисляется как  $x_B + 4 \cdot (16 \cdot 16 + k) = x_B + 1024 + 4k$ .

### Решение упражнения 3.40

Это упражнение требует хорошего знания ассемблерного кода, генерируемого компилятором, чтобы понимать, как он осуществляет оптимизацию. В данном случае компилятор применил особенно хитроумную оптимизацию.

Для начала рассмотрим следующий код на C, а затем мы покажем, как он получается из приведенного в упражнении ассемблерного кода:

```

/* Записывает во все диагональные элементы значение val */
void fix_set_diag_opt(fix_matrix A, int val) {
    int *Abase = &A[0][0];
    long index = 0;
    long iend = N*(N+1);
    do {
        Abase[index] = val;
        index += (N+1);
    } while (index != iend);
}

```

Эта функция вводит переменную Abase типа `int *`, указывающую на начало массива A. Этот указатель ссылается на последовательность 4-байтных целых чисел, состоящую из элементов A, размещенных в памяти по строкам. Далее вводится целочисленная переменная index, которая принимает значения индексов диагональных элементов в A с учетом свойств, согласно которым индексы соседних диагональных элементов отличаются друг от друга на величину  $N + 1$  и достижение индекса  $N$ -го диагонального элемента ( $N(N + 1)$ ) говорит о выходе за пределы массива.

Фактический ассемблерный код следует этой общей форме, но приращения указателя необходимо масштабировать с коэффициентом 4. В комментариях мы отметили регистр %rax как содержащий значение index4, равное index в версии на C, но масштабированное с коэффициентом 4. Для  $N = 16$  наша точка остановки для index4 будет равна  $4 \cdot 16(16 + 1) = 1088$ .

```
void fix_set_diag(fix_matrix A, int val)
A в %rdi, val в %rsi
1 fix_set_diag:
2     movl    $0, %eax           Установить index4 = 0
3     .L13:                      loop:
4     movl    %esi, (%rdi,%rax)   Записать val в Abase[index4/4]
5     addq    $68, %rax          Увеличить index4 += 4(N+1)
6     cmpq    $1088, %rax        Сравнить index4: 4N(N+1)
7     jne     .L13              Если !=, перейти к loop
8     rep; ret                  Возврат
```

### Решение упражнения 3.41

Это упражнение требует обратить особое внимание на размещение структуры в памяти и на код для доступа к ее полям. Объявление структуры – это вариант примера в тексте. Оно показывает, что структуры могут вкладываться друг в друга.

1. Вот как эта структура размещается в памяти:

Смещение	0	8	12	16	24
Содержимое	p	s.x	s.y	next	

2. Она занимает 24 байта.
3. Как обычно, начнем с добавления комментариев в ассемблерный код:

```
void sp_init(struct prob *sp)
sp в %rdi
1 sp_init:
2     movl    12(%rdi), %eax      Получить sp->s.y
3     movl    %eax, 8(%rdi)      Сохранить в sp->s.x
4     leaq    8(%rdi), %rax      Вычислить &(sp->s.x)
5     movq    %rax, (%rdi)       Сохранить в sp->p
6     movq    %rdi, 16(%rdi)     Сохранить sp в sp->next
7     ret
```

Теперь можно воссоздать код на C:

```
void sp_init(struct prob *sp)
{
    sp->s.x = sp->s.y;
    sp->p = &(sp->s.x);
    sp->next = sp;
}
```

### Решение упражнения 3.42

Это упражнение демонстрирует реализацию в машинном коде очень распространенной структуры данных и операций с ней. Для начала добавим комментарии в ассемблерный код, попутно отметив, что два поля структуры находятся со смещениями 0 (поле v) и 8 (поле p).

```
long fun(struct ELE *ptr)
ptr в %rdi
```

```
1 fun:
2   movl    $0, %eax          result = 0
3   jmp     .L2               перейти к middle
4   .L3:                      loop:
5   addq    (%rdi), %rax       result += ptr->v
6   movq    8(%rdi), %rdi      ptr = ptr->p
7   .L2:                      middle:
8   testq   %rdi, %rdi        Проверить ptr
9   jne     .L3               Если != NULL, перейти к loop
10  rep; ret
```

1. Опираясь на код с комментариями, можно воссоздать версию на C:

```
long fun(struct ELE *ptr) {
    long val = 0;
    while (ptr) {
        val += ptr->v;
        ptr = ptr->p;
    }
    return val;
}
```

2. Каждая структура является элементом односвязного списка, где поле *v* является значением элемента, а *p* – указателем на следующий элемент. Функция *fun* вычисляет сумму значений элементов в списке.

Решение упражнения 3.43

Структуры и объединения вводят набор простых понятий, но требуется практика, чтобы привыкнуть к различным приемам ссылок на них и их реализациям.

Выражение	Тип	Код
up->t1.u	long	movq (%rdi), %rax movq %rax, (%rsi)
up->t1.v	short	movw 8(%rdi), %ax movw %ax, (%rsi)
&up->t1.w	char *	addq \$10, %rdi movq %rdi, (%rsi)
up->t2.a	int *	movq %rdi, (%rsi)
up->t2.a[up->t1.u]	int	movq (%rdi), %rax movl (%rdi,%rax,4), %eax movl %eax, (%rsi)
*up->t2.p	char	movq 8(%rdi), %rax movb (%rax), %al movb %al, (%rsi)

Решение упражнения 3.44

Знание правил выравнивания и размещения структур в памяти совершенно необходимо для определения объема памяти, выделяемого для той или иной структуры, а также для понимания кода доступа к таким структурам, генерируемого компилятором. В этом упражнении вам предлагалось правильно оценить все особенности нескольких примеров структур.

1. `struct P1 { int i; char c; int j; char d; };`

i	c	j	d	Всего	Выравнивание
0	4	8	12	16	4

2. `struct P2 { int i; char c; char d; long j; };`

i	c	j	d	Всего	Выравнивание
0	4	5	8	16	8

3. `struct P3 { short w[3]; char c[3] };`

w	c	Всего	Выравнивание
0	6	10	2

4. `struct P4 { short w[5]; char *c[3] };`

w	c	Всего	Выравнивание
0	16	40	8

5. `struct P5 { struct P3 a[2]; struct P2 t };`

a	t	Всего	Выравнивание
0	24	40	8

### Решение упражнения 3.45

Цель этого упражнения – закрепление понимания выравнивания и размещения структур в памяти.

1. Размеры объектов и их смещения в памяти:

Поле	a	b	c	d	e	f	g	h
Размер	8	2	8	1	4	1	8	4
Размер	0	8	16	24	28	32	40	48

2. Общий размер структуры 56 байт. В конце структура должна быть дополнена 4 байтами, чтобы удовлетворить требование к выравниванию по границам 8-байтных блоков.
3. Одна из рабочих стратегий, когда общий размер всех элементов кратен степени 2, – упорядочить элементы структуры в порядке убывания их размеров. Последовав этому правилу, получаем такое объявление:

```
struct {
    char    *a;
    double  c;
    long    g;
    float   e;
    int     h;
    short   b;
    char    d;
    char    f;
} rec;
```

и следующие смещения:



Поле	a	c	g	e	h	b	d	f
Размер	8	8	8	4	4	2	1	1
Размер	0	8	16	24	28	32	34	35

В конце структура должна быть дополнена 4 байтами, чтобы удовлетворить требование к выравниванию по границам 8-байтных блоков.

### Решение упражнения 3.46

Это упражнение охватывает широкий диапазон тем, таких как кадры стека, представление строк, коды ASCII и порядок следования байтов. Оно демонстрирует опасность ссылок на ячейки памяти, выходящие за заданные пределы, и пагубные последствия переполнения буфера.

#### 1. Стек после строки 3

00 00 00 00 00 40 00 76	Адрес возврата
01 23 45 67 89 AB CD EF	Сохраненный регистр %rbx
	← buf = %rsp

#### 2. Стек после строки 5

00 00 00 00 00 40 00 76	Адрес возврата
01 23 45 67 89 AB CD EF	Сохраненный регистр %rbx
35 34 33 32 31 30 39 38	
37 36 35 34 33 32 31 30	← buf = %rsp

- Программа попытается вернуться по адресу `0x400034`. Младшие 2 байта были затерты кодом символа «4» и завершающим пустым символом.
- Сохраненное значение регистра %rbx изменилось на `0x3332313039383736`. Это значение будет загружено в регистр перед возвратом из `get_line`.
- Функция `malloc` должна вызываться со значением `strlen(buf)+1` в аргументе, и вызывающий код должен также проверить на равенство `NULL` возвращаемое значение функции `gets`.

### Решение упражнения 3.47

- Это примерно соответствует диапазону  $2^{15}$  адресов.
- 128-байтные салазки `por` будут охватывать  $2^7$  адресов в каждой проверке, поэтому всего потребуется около  $2^6 = 64$  попыток.

Этот пример ясно показывает, что степень рандомизации в данной версии Linux лишь немного сдержит от атаки переполнением буфера.

### Решение упражнения 3.48

Это упражнение дает еще одну возможность увидеть, как код для архитектуры x86-64 управляет стеком, а также лучше понять, как защищаться от атак переполнением буфера.

1. В незащищенной версии строки 4 и 5 вычисляют местоположение  $v$  и  $buf$ , которые имеют смещения 24 и 0 относительно `%rsp`. В защищенной версии канареечное значение хранится со смещением 40 (строка 4), а  $v$  и  $buf$  – со смещениями 8 и 16 (строки 7 и 8).
2. В защищенной версии локальная переменная  $v$  находится ближе к вершине стека, чем  $buf$ , поэтому переполнение  $buf$  не повредит значение  $v$ .

### Решение упражнения 3.49

Этот код использует несколько трюков арифметики на уровне битов, которые мы уже видели. Но чтобы понять это, код нужно тщательно изучить.

1. Инструкция `leaq` в строке 5 вычисляет значение  $8n + 22$ , которое затем округляется до ближайшего кратного 16 с инструкцией `andq` в строке 6. В результате получается значение  $8n + 8$ , когда  $n$  нечетное, и  $8n + 16$ , когда  $n$  четное. Затем это значение вычитается из  $s_1$ , чтобы получить  $s_2$ .
2. Последовательность из этих трех инструкций округляет  $s_2$  до ближайшего кратного 8. Они используют комбинацию смещений и сдвигов, которую мы видели, когда обсуждали деление на степень 2 в разделе 2.3.7.
3. Эти два примера можно рассматривать как случаи минимизации и максимизации значений  $e_1$  и  $e_2$ .

$n$	$s_1$	$s_2$	$p$	$e_1$	$e_2$
5	2065	2017	2024	1	7
6	2064	2000	2000	16	0

4. Здесь видно, что  $s_2$  вычисляется так, чтобы сохранить любое смещение  $s_1$  ближайшим кратным 16. Также видно, что  $p$  будет выравниваться по 8-байтным границам, как рекомендуется для массивов с 8-байтными элементами.

### Решение упражнения 3.50

Это упражнение требует выполнить код по шагам, обращая особое внимание на то, какие инструкции преобразования и перемещения данных используются. В подобном случае извлекаются и конвертируются следующие данные:

- значение по адресу `fp` извлекается, преобразуется в `int` (строка 4) и затем сохраняется по адресу `ip`. Отсюда можно сделать вывод, что `val1` равно `d`;
- значение по адресу `ip` извлекается, преобразуется в число с плавающей точкой (строка 6) и сохраняется по адресу `fp`. Отсюда можно сделать вывод, что `val2` равно `i`;
- значение `l` преобразуется в `double` (строка 8) и сохраняется по адресу `dp`. Отсюда можно сделать вывод, что `val3` равно `l`;
- строка 3 извлекает значение по адресу `fp`. Две инструкции в строках 10–11 преобразуют его в `double` и оставляют результат как возвращаемое значение в регистре `%xmm0`. Отсюда можно сделать вывод, что `val4` равно `f`.

### Решение упражнения 3.51

Эти инструкции можно определить по соответствующим строкам в табл. 3.16 и 3.17 или использовать одну из последовательностей инструкций для преобразования между форматами с плавающей точкой.

$T_x$	$T_y$	Инструкции
long	double	<code>vcvtsi2sdq %rdi, %xmm0</code>
double	int	<code>vcvttsd2si %xmm0, %eax</code>
double	float	<code>vunpcklpd %xmm0, %xmm0, %xmm0</code> <code>vcvtupd2ps %xmm0, %xmm0</code>
long	float	<code>vcvtsi2ssq %rdi, %xmm0, %xmm0</code>
float	long	<code>vcvtss2siq %xmm0, %rax</code>

### Решение упражнения 3.52

Основные правила выбора регистров для передачи аргументов просты (несмотря на усложнения, связанные с появлением аргументов других типов [77]).

1. `double g1(double a, long b, float c, int d);`  
Регистры: `a` в `%xmm0`, `b` в `%rdi`, `c` в `%xmm1`, `d` в `%esi`.
2. `double g2(int a, double *b, float *c, long d);`  
Регистры: `a` в `%edi`, `b` в `%rsi`, `c` в `%rdx`, `d` в `%rcx`.
3. `double g3(double *a, double b, int c, float d);`  
Регистры: `a` в `%rdi`, `b` в `%xmm0`, `c` в `%esi`, `d` в `%xmm1`.
4. `double g4(float a, int *b, float c, double d);`  
Регистры: `a` в `%xmm0`, `b` в `%rdi`, `c` в `%xmm1`, `d` в `%xmm2`.

### Решение упражнения 3.53

Из ассемблерного кода видно, что в регистрах `%rdi` и `%rsi` передаются два целочисленных аргумента. Назовем их `i1` и `i2`. Также есть два аргумента с плавающей точкой, которые передаются в регистрах `%xmm0` и `%xmm1`. Назовем их `f1` и `f2`.

С учетом этих договоренностей добавим комментарии в ассемблерный код:

```

Ссылки в комментариях на аргументы: i1 (%rdi), i2 (%esi),
                                         f1 (%xmm0) и f2 (%xmm1)

double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
1 funct1:
2   vcvtsi2ssq    %rsi, %xmm2, %xmm2    Получить i2 и преобразовать
                                         из long в float
3   vaddss       %xmm0, %xmm2, %xmm0    Прибавить f1 (float)
4   vcvtsi2ss    %edi, %xmm2, %xmm2    Получить i1 и преобразовать
                                         из int в float
5   vdivss       %xmm0, %xmm2, %xmm0    Вычислить i1 / (i2 + f1)
6   vunpcklps    %xmm0, %xmm0, %xmm0
7   vcvtpps2pd   %xmm0, %xmm0
8   vsubsd       %xmm1, %xmm0, %xmm0    Преобразовать в double
                                         Вычислить i1 / (i2 + f1) - f2 (double)
9   ret

```

Теперь видно, что код вычисляет значение  $i1/(i2+f1)-f2$ . Также видно, что `i1` имеет тип `int`, `i2` – тип `long`, `f1` – тип `float` и `f2` – тип `double`. Единственная неоднозначность в именовании аргументов обусловлена коммутативностью сложения, что дает два возможных результата:

```

double funct1a(int p, float q, long r, double s);
double funct1b(int p, long q, float r, double s);

```

### Решение упражнения 3.54

Это упражнение легко решить путем пошагового выполнения ассемблерного кода и определения результатов, вычисляемых на каждом шаге, как показано ниже:

```
double funct2(double w, int x, float y, long z)
w в %xmm0, x в %edi, y в %xmm1, z в %rsi
1 funct2:
2  vcvtsi2ss      %edi, %xmm2, %xmm2    Преобразовать x в float
3  vmulss        %xmm1, %xmm2, %xmm1    Умножить на y
4  vunpcklps     %xmm1, %xmm1, %xmm1
5  vcvtps2pd     %xmm1, %xmm2
6  vcvtsi2sdq    %rsi, %xmm1, %xmm1    Преобразовать x*y в double
7  vdivsd        %xmm1, %xmm0, %xmm0    Преобразовать z в double
8  vsubsd        %xmm0, %xmm2, %xmm0    Вычислить w/z
9  ret        Вычесть из x*y
```

Отсюда можно заключить, что функция вычисляет  $y * x - w/z$ .

### Решение упражнения 3.55

Эта упражнение решается с использованием тех же рассуждений, которые применялись, чтобы показать, как значение в памяти с меткой .LC2 декодируется в число 1.8, только здесь все немного проще.

Здесь мы видим два значения: 0 и 1077936128 (0x40400000). Из старших байтов извлекаем поле показателя степени 0x404 (1028), из которого вычитаем смещение 1023 и получаем показатель степени 5. Объединяя биты поля дробного значения из двух фрагментов, получаем 0, но с подразумеваемой ведущей 1, что дает значение 1.0. Следовательно, константа имеет значение  $1.0 \times 25 = 32.0$ .

### Решение упражнения 3.56

1. Здесь мы видим, что 16 байт, хранящихся в памяти, начиная с адреса .LC1, образуют маску, в которой младшие 8 байт во всех разрядах содержат единицы, кроме самого старшего бита, который служит знаковым разрядом значения двойной точности. Применение операции И к этой маске и регистру %xmm0 сбрасывает знаковый бит в x, давая абсолютное значение. На самом деле мы сгенерировали этот код, определив `EXPR(x)` как `fabs(x)`, где функция `fabs` определена в `<math.h>`.
2. Инструкция `vxorpd` сбрасывает регистр в ноль, поэтому ее можно рассматривать как способ сгенерировать константу с плавающей точкой со значением 0.0.
3. Здесь мы видим, что 16 байт, хранящихся в памяти, начиная с адреса .LC2, образуют маску с одним единичным битом в позиции знакового разряда в значении в регистре XMM. Применение операции ИСКЛЮЧАЮЩЕЕ ИЛИ к этой маске и регистру %xmm0 меняет знак x, вычисляя выражение  $-x$ .

### Решение упражнения 3.57

И снова начнем с добавления комментариев в ассемблерный код, включая условные ветвления:

```
double funct3(int *ap, double b, long c, float *dp)
ap в %rdi, b в %xmm0, c в %rsi, dp в %rdx
1 funct3:
2  vmovss        (%rdx), %xmm1          Получить d = *dp
3  vcvtsi2sd     (%rdi), %xmm2, %xmm2    Получить a = *ap и
                                         преобразовать в double
4  vucomisd      %xmm2, %xmm0           Сравнить b:a
5  jbe          .L8                     Если <=, перейти к lesseq
```

6	vcvtsi2ssq	%rsi, %xmm0, %xmm0	Преобразовать c в float
7	vmulss	%xmm1, %xmm0, %xmm1	Умножить на d
8	vunpcklps	%xmm1, %xmm1, %xmm1	
9	vcvtps2pd	%xmm1, %xmm0	Преобразовать в double
10	ret		Возврат
11	.L8:		lesseq:
12	vaddss	%xmm1, %xmm1, %xmm1	Вычислить d+d = 2.0 * d
13	vcvtsi2ssq	%rsi, %xmm0, %xmm0	Преобразовать c в float
14	vaddss	%xmm1, %xmm0, %xmm0	Вычислить c + 2*d
15	vunpcklps	%xmm0, %xmm0, %xmm0	
16	vcvtps2pd	%xmm0, %xmm0	Преобразовать в double
17	ret		Возврат

Теперь можно написать версию `funct3` на C:

```
double funct3(int *ap, double b, long c, float *dp) {
    int a = *ap;
    float d = *dp;
    if (a < b)
        return c*d;
    else
        return c+2*d;
}
```

## Архитектура процессора

- 4.1. Архитектура набора команд Y86-64.
- 4.2. Логическое проектирование и язык HCL описания управляющей части аппаратной системы.
- 4.3. Последовательная реализация Y86-64 (SEQ).
- 4.4. Общие принципы конвейерной обработки.
- 4.5. Конвейерная реализация Y86-64.
- 4.6. Итоги.

Библиографические заметки.

Домашние задания.

Решения упражнений.

Современные микропроцессоры входят в число наиболее сложных систем, когда-либо созданных человеком. Один кремниевый кристалл размером с ноготь может содержать несколько высокопроизводительных процессоров, большой объем кеш-памяти, а также логику, необходимую для взаимодействия с внешними устройствами. В том, что касается производительности, на сегодняшний день процессоры, выполненные в виде одной микросхемы, затмевают суперкомпьютеры, занимавшие целые помещения и стоившие больше 10 млн долларов каких-то 20 лет назад. Даже процессоры, встроенные в бытовые приборы, мобильные телефоны, карманные записные книжки, игровые приставки и т. д., оказываются намного более мощными, чем могли предложить первые создатели компьютерной техники.

До сих пор мы рассматривали компьютерные системы на уровне программ, написанных на машинном языке. Из предыдущих глав мы поняли, что процессор призван выполнять некую последовательность инструкций, каждая из которых в свою очередь выполняет некоторую элементарную операцию, например сложение двух чисел. Любая инструкция кодируется в двоичной форме в виде последовательности одного или нескольких байтов. Инструкции, поддерживаемые конкретным процессором, и их кодирование на уровне байтов называются *архитектурой набора команд* (ISA, Instruction Set Architecture). Разные семейства процессоров, таких как Intel IA32 и x86-64, IBM/Freescale Power и ARM, имеют разные ISA. Программа, скомпилированная для машины одного типа, не будет выполняться на машине другого типа. С другой стороны, в рамках каждого отдельного семейства существует много различных моделей процессоров. Каждый производитель постоянно выпускает новые модели своих процессоров, увеличивая их мощность и сложность, однако все модели одного семейства остаются совместимыми на уровне ISA. Процессоры популярных семейств, таких как x86-64, выпускаются многими производителями. А архитектура набора команд обеспечивает концептуальный уровень абстракции, отделяющий создателей компиляторов, которым необходимо знать, какие инструкции имеются и как они кодируются, от проек-

тировщиков процессоров, задача которых – создавать машины, способные выполнять эти инструкции.

В этой главе мы вкратце рассмотрим процесс проектирования процессоров и познакомимся со способом выполнения аппаратной системой конкретного набора команд. Такой подход поможет лучше понять работу компьютеров, а также разобраться в технологических тонкостях, с которыми приходится сталкиваться производителям компьютерной техники. Одна из важных особенностей состоит в том, что фактический процессор может функционировать несколько иначе, чем предполагает модель ISA. Может показаться, что модель ISA требует *последовательного* выполнения инструкций, каждая из которых извлекается из памяти и выполняется до конца, прежде чем будет извлечена следующая инструкция. Однако, выполняя разные этапы нескольких инструкций одновременно, процессор может достичь большей производительности, чем при последовательном выполнении инструкций. Чтобы подтвердить, что процессор получает те же результаты, что и при последовательном выполнении инструкций, используются специальные механизмы. Идея использования хитроумных способов повышения производительности с одновременной поддержкой функциональности простой и более абстрактной модели хорошо известна в области вычислительной техники. В качестве примера можно привести применение кеширования в браузерах и таких структур данных, как сбалансированные двоичные деревья и хеш-таблицы.

### Развитие компьютерных технологий

Чтобы понять, насколько далеко ушли компьютерные технологии за последние четыре десятилетия, рассмотрим пример с двумя процессорами.

Первый суперкомпьютер Cray 1 был доставлен в Лос-Аламосскую национальную лабораторию в 1976 году. Это был самый быстрый компьютер в мире, способный выполнять до 250 млн арифметических операций в секунду. Он имел 8 Мбайт оперативной памяти – максимум, поддерживаемый оборудованием. Машина также была очень большой – она весила 5 т, потребляла 115 кВт и стоила 9 млн долларов. Всего таких машин было изготовлено около 80 штук.

Чип Apple ARM A7, представленный в 2013 году и предназначенный для использования в iPhone 5S, содержит два процессора, каждый из которых способен выполнять несколько миллиардов арифметических операций в секунду, и 1 Гбайт оперативной памяти. Сам телефон с этим микропроцессором весит всего 112 г, потребляет около 1 Вт и стоит менее 800 долларов. За первые выходные дни после его появления было продано более 9 млн единиц. Помимо того что iPhone 5S является мощным компьютером, его можно использовать для фотосъемки, телефонных звонков и прокладки маршрута для поездки, то есть для выполнения функций, которые были недоступны для Cray 1.

Эти две системы разделяет всего 37 лет, что наглядно демонстрирует огромный прогресс в развитии полупроводниковой технологии. Процессор Cray 1 был построен на основе 100 000 полупроводниковых микросхем, каждая из которых содержала менее 20 транзисторов, а Apple A7 имеет более 1 миллиарда транзисторов на одном кристалле. Для 8-мегабайтной памяти Cray 1 требовалось 8192 микросхемы, в то время как гигабайтная память iPhone находится на одном кристалле.

Вероятность, что кому-то удастся создать свой собственный процессор, невероятно мала. Это задача экспертов и специалистов, работающих в сотне компаний по всему миру. Зачем же тогда изучать проектирование процессоров?

- *Это представляет интеллектуальный интерес.* Знание принципов функционирования тех или иных систем и приборов ценно само по себе. При этом особенно интересно изучать «внутренности» систем, с которыми специалистам и

инженерам по вычислительной технике приходится сталкиваться каждый день, но которые при этом остаются для многих из них тайной за семью печатями. Проектирование процессоров предполагает владение многими принципами инженерной практики и требует создания максимально простой структуры для решения сложных задач.

- *Понимание принципов работы процессора помогает получить представление о работе компьютерной системы в целом.* В главе 6 мы рассмотрим систему памяти и методы, применяемые при создании образа памяти очень большого объема с минимальным временем доступа. Изучение интерфейса «процессор–память» делает наше обсуждение еще более полным.
- *Несмотря на небольшое число проектировщиков процессоров, число проектировщиков аппаратных систем, содержащих процессоры, растет с каждым годом.* Эта тенденция стала общим трендом с внедрением процессоров в такие обыденные системы, как автомобили и бытовая техника. Проектировщики встроенных систем должны понимать принципы работы процессора, потому что такие системы, как правило, проектируются и программируются на более низком уровне абстракции, чем настольные компьютеры.
- *Любой может заняться проектированием процессоров.* Несмотря на небольшое число компаний-производителей микропроцессоров, рабочие группы проектировщиков постоянно расширяются. Проектированием разных аспектов одного процессора могут заниматься до 1000 человек.

В этой главе мы сначала определим простую систему команд, которую будем использовать как рабочий пример для реализации процессора. Мы назовем ее системой команд «Y86-64», потому что ее прообразом является система команд x86-64. Система команд Y86-64 имеет меньше типов данных, инструкций и режимов адресации. Она также имеет более простую систему кодирования на уровне байтов, что делает машинный код менее компактным, по сравнению с аналогичным кодом x86-64, но при этом значительно упрощает проектирование логики декодирования. Впрочем, Y86-64 – вполне совершенная система, позволяющая писать программы, обрабатывающие целочисленные данные. При проектировании процессора, реализующего набор команд Y86-64, нам нужно будет решить множество сложных задач, с которыми приходится сталкиваться разработчикам процессоров.

Затем мы познакомим вас с некоторыми основами проектирования цифровой аппаратуры. Опишем базовые компоненты процессора, а также особенности их взаимодействий друг с другом. Наше обсуждение будет базироваться на булевой алгебре и операциях с битами, описанных в главе 2. Потом представим вам язык HCL (Hardware Control Language) для описания блоков управления аппаратных систем и используем его для описания проектируемого нами процессора. Даже если у вас есть некоторый опыт проектирования логики, обязательно прочитайте этот раздел, чтобы потом не возникало сложностей с недопониманием используемых нами обозначений.

В качестве первого шага на пути проектирования процессора мы представим функционально корректный, но несколько неэффективный процессор Y86-64, основанный на *последовательном* выполнении операций (назовем эту версию SEQ). Этот процессор выполняет по одной инструкции Y86-64 за один цикл. Генератор синхронизирующего сигнала должен иметь довольно низкую частоту, чтобы в течение одного цикла успела выполняться вся последовательность действий. Создать такой процессор можно, однако его производительность будет на порядок ниже той, которой можно добиться для подобного устройства.

Затем, получив последовательную архитектуру SEQ, мы применим к ней серию преобразований и создадим процессор с *конвейерной* обработкой инструкций. Он будет разби-



вать выполнение каждой инструкции на 5 шагов, каждый из которых будет выполняться отдельным аппаратным блоком, или *этапом*. Инструкции будут подаваться на вход конвейера по одной в каждом цикле. В результате процессор будет одновременно выполнять разные этапы обработки пяти инструкций. Чтобы обеспечить видимость последовательного поведения Y86-64, необходимо предусмотреть обработку множества разнообразных условий *прерывания*, когда местоположение или операнды одной инструкции зависят от местоположения или операндов других инструкций, все еще находящихся на конвейере.

Мы разработали комплект инструментов для исследования архитектуры процессора и экспериментов с ней. В их число входит ассемблер Y86-64, имитатор для запуска программ Y86-64 на обычном компьютере, а также имитаторы, моделирующие два процессора с последовательной и один процессор с конвейерной обработкой инструкций. Управляющая логика этих процессоров представлена файлами на языке HCL. Редактируя эти файлы и повторно компилируя имитаторы, можно менять и расширять моделируемое поведение. Здесь также представлены упражнения, в которых вам будет предложено реализовать новые инструкции и модифицировать имеющиеся. Сопровождающий код тестовой программы поможет оценить корректность вносимых изменений. Все эти упражнения имеют целью помочь вам получить максимально полное понимание изложенного материала, а также оценить множество вариантов проектирования, с которыми имеют дело разработчики процессоров.

В приложении в интернете «ARCH:VLOG» (раздел 4.5.9) мы представим определение нашего конвейерного процессора Y86-64 на языке описания оборудования Verilog. Оно включает описание основных модулей и всей структуры процессора. Наши описания на языке HCL мы автоматически будем переводить на язык Verilog. Возможность первичной отладки описания HCL с помощью наших симуляторов поможет нам устранить многие сложные ошибки, которые иначе перекочевали бы в аппаратную конструкцию. Согласно описанию Verilog, существуют коммерческие инструменты и инструменты с открытым исходным кодом для поддержки моделирования и *логического синтеза*, генерирующие фактические схемы для микропроцессоров. Поэтому, несмотря на то что большая часть наших усилий будет направлена на создание графических и текстовых описаний системы по аналогии с разработкой программного обеспечения, сам факт возможности воплощения этих описаний в кристалле демонстрирует, что мы действительно создаем систему, которую можно реализовать в виде оборудования.

## 4.1. Архитектура системы команд Y86-64

Определение архитектуры набора команд, такой как Y86-64, предполагает определение различных компонентов состояния, собственно команд и их кодов, а также набора соглашений о программировании и обработке исключительных ситуаций.

### 4.1.1. Состояние, видимое программисту

Как показано на рис. 4.1, каждая инструкция в программе Y86-64 может читать и изменять некоторую часть состояния процессора. Это состояние называется состоянием, *видимым для программиста*, где «программистом» является некто, пишущий программы на ассемблере, или компилятор, генерирующий машинный код. При реализации процессора будет видно, что это состояние необязательно представлять и организовывать именно так, как подразумевает ISA, если есть уверенность, что программы на машинном уровне имеют доступ к состоянию, видимому программистом. Состояние для Y86-64 похоже на состояние x86-64. Имеется 15 *регистров общего назначения*: %rax, %rcx, %rdx, %rbx, %rsp, %rbp, %rsi, %rdi и c %r8 по %r14. (Мы опустили регистр %r15, имеющийся в архитектуре x86-64, чтобы упростить кодирование инструкций.) Каждый регистр может хранить 64-разрядное слово. Регистр %rsp играет роль *указателя стека* и используется инструкциями управления стеком, а также инструкциями вызова процедур и

возврата из них. Остальные регистры не имеют определенного предназначения. Три однокбитных *флага условий* – ZF, SF и OF – хранят информацию о влиянии последней выполненной арифметической или логической инструкции. *Счетчик инструкций* (PC) хранит адрес выполняемой в данный момент инструкции.

*Память* концептуально организована как большой массив байтов, содержащий саму программу и данные. Программы Y86-64 ссылаются на ячейки памяти, используя *виртуальные адреса*. Комбинация аппаратного и программного обеспечения операционной системы переводит их в фактические, или *физические*, адреса, указывающие место, где фактически хранятся значения. Более подробно виртуальная память рассматривается в главе 9. На данный момент виртуальную память можно представлять как сплошной массив байтов.



**Рис. 4.1.** Состояние Y86-64, видимое программисту. Как и в случае с x86-64, программы для Y86-64 обращаются к регистрам, флагам, счетчику инструкций (PC) и памяти и изменяют их. Код состояния программы указывает, нормально ли работает программа или произошло какое-то особое событие

И последний аспект состояния программы – это код состояния Stat, сообщающий общее состояние выполняющейся программы. Он будет служить признаком нормального выполнения программы или какой-то *исключительной ситуации*, например когда инструкция пытается обратиться к недопустимому адресу памяти. Возможные коды состояния и обработка исключений описаны в разделе 4.1.4.

### 4.1.2. Инструкции Y86-64

На рис. 4.2 представлено краткое описание отдельных инструкций для архитектуры Y86-64. Реализация этой системы команд и есть наша цель. По большому счету, система команд Y86-64 является подмножеством системы команд x86-64. Она включает только операции с 8-байтными целыми числами; меньше режимов адресации и меньший набор операций. Поскольку используются только 8-байтные данные, мы будем называть их «словами». На рис. 4.2 слева показаны ассемблерные мнемоники инструкций, а справа – их байтовые коды. Ассемблерный код имеет формат, аналогичный формату АТТ для x86-64.

Далее приводится краткое описание различных инструкций Y86-64.

- Класс инструкций `movq` включает четыре инструкции: `irmovq`, `rmmovq`, `mrmovq` и `rmrmovq`, явно указывая форму источника и приемника. Источник может быть непосредственным значением (i), регистром (r) или ячейкой памяти (m). Он обо-

значается первым символом в мнемонике. Приемником может быть регистр (r) или ячейка памяти (m). Он обозначается вторым символом в мнемонике. Явное обозначение четырех типов передачи данных пригодится нам при принятии решения об их реализации.

Ссылки на память в двух инструкциях состоят из базового адреса и смещения. Мы не будем поддерживать ни второй индексный регистр, ни какое-либо масштабирование при вычислении адреса, так же как в архитектуре x86-64 мы не предусматриваем инструкции для прямого перемещения данных из одной ячейки памяти в другую и для записи непосредственного значения в память.

Байты	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq rA, rB	2	0	rA	rB						
irmovq V, rB	3	0	F	rB						V
rmmovq rA, D(rB)	4	0	rA	rB						D
rrmovq D(rB), rA	5	0	rA	rB						D
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn								Dest
cmovXX rA, rB	2	fn	rA	rB						
call Dest	8	0								Dest
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Рис. 4.2. Набор инструкций Y86-64. Коды инструкций занимают от 1 до 10 байт.

Код состоит из 1-байтного спецификатора инструкции и может дополнительно включать 1-байтный спецификатор регистра и 8-байтное значение константы. Поле *fn* определяет конкретную целочисленную операцию (OPq), условие перемещения данных (cmovXX) или условие перехода (jXX). Все числовые значения показаны в шестнадцатеричном формате

- Имеется четыре инструкции для операций с целыми числами, имеющие общий формат OPq: addq, subq, andq и xorq. Все они работают только с данными в регистрах, тогда как в x86-64 имеются операции с данными в памяти. Эти инструкции устанавливают три флага условий: ZF, SF и OF (ноль, знак и переполнение).
- Имеется семь инструкций перехода (показаны на рис. 4.2 как jXX): jmp, jle, jl, je, jne, jge и jg. Выбор ветви зависит от типа ветвления и состояний флагов. Условное ветвление происходит точно так же, как в архитектуре x86-64 (см. табл. 3.11).

- Имеется шесть инструкций условного перемещения (на рис. 4.2 показаны как `cmovXX`): `cmovle`, `cmovl`, `cmove`, `cmovne`, `cmovge` и `cmovg`. Они имеют тот же формат, что и инструкция перемещения данных между регистрами `rrmovq`, но регистр-приемник изменяется, только если значения флагов удовлетворяют определенным условиям.
- Инструкция вызова процедуры `call` вталкивает адрес возврата в стек и переходит по адресу назначения. Инструкция `ret` осуществляет возврат из процедуры.
- Инструкции `pushq` и `popq` вталкивают (`push`) и выталкивают (`pop`) значение со стека точно так же, как аналогичные инструкции в архитектуре x86-64.
- Инструкция `halt` останавливает выполнение инструкций. В x86-64 имеется аналогичная инструкция `hlt`. Программам для x86-64 запрещено пользоваться этой инструкцией, потому что она вызывает остановку работы всей системы. В архитектуре Y86-64 инструкция `halt` будет использоваться для остановки имитатора с кодом состояния `HLT` (раздел 4.1.4).

### 4.1.3. Кодирование инструкций

На рис. 4.2 также показано, как кодируются инструкции на уровне байтов. Каждая инструкция занимает от 1 до 10 байт, в зависимости от набора полей. Первый байт каждой инструкции определяет ее тип. Он делится на две части по четыре бита в каждой: старшую (*код*) и младшую (*функция*, обозначена на рис. 4.2 как *fn*). Как показано на рис. 4.2, значения кода изменяются от 0 до 0xF. Значения функций используются, только когда один код определяет группу взаимосвязанных инструкций. Они представлены на рис. 4.3, где показаны конкретные коды целочисленных операций переходов и условного перемещения данных.

Операции	Ветвление				Перемещение												
addq <table><tr><td>6</td><td>0</td></tr></table>	6	0	jmp <table><tr><td>7</td><td>0</td></tr></table>	7	0	jne <table><tr><td>7</td><td>4</td></tr></table>	7	4		rrmovq <table><tr><td>2</td><td>0</td></tr></table>	2	0	cmovne <table><tr><td>2</td><td>4</td></tr></table>	2	4		
6	0																
7	0																
7	4																
2	0																
2	4																
subq <table><tr><td>6</td><td>1</td></tr></table>	6	1	jle <table><tr><td>7</td><td>1</td></tr></table>	7	1	jge <table><tr><td>7</td><td>5</td></tr></table>	7	5		cmovle <table><tr><td>2</td><td>1</td></tr></table>	2	1	cmovge <table><tr><td>2</td><td>5</td></tr></table>	2	5		
6	1																
7	1																
7	5																
2	1																
2	5																
andq <table><tr><td>6</td><td>2</td></tr></table>	6	2	jl <table><tr><td>7</td><td>2</td></tr></table>	7	2	jg <table><tr><td>7</td><td>6</td></tr></table>	7	6		cmovl <table><tr><td>2</td><td>2</td></tr></table>	2	2	cmovg <table><tr><td>2</td><td>6</td></tr></table>	2	6		
6	2																
7	2																
7	6																
2	2																
2	6																
xorq <table><tr><td>6</td><td>3</td></tr></table>	6	3	je <table><tr><td>7</td><td>3</td></tr></table>	7	3			cmove <table><tr><td>2</td><td>3</td></tr></table>	2	3							
6	3																
7	3																
2	3																

**Рис. 4.3.** Коды функций для набора команд Y86-64. Код функции определяет конкретную целочисленную операцию, условие перехода или условие передачи данных. Эти инструкции обозначены на рис. 4.2 как `Opq`, `jXX` и `cmovXX`

Обратите внимание, что `rrmovq` имеет то же значение в поле кода, что и инструкции условного перемещения. Ее можно рассматривать как инструкцию «безусловного перемещения», так же как инструкция `jmp` является инструкцией безусловного перехода – обе инструкции имеют значение 0 в поле функции.

Как показано в табл. 4.1, каждый из 15 регистров общего назначения имеет свой *идентификатор* (ID), принимающий значения в диапазоне от 0 до 0xF. Нумерация регистров в Y86-64 совпадает с нумерацией в x86-64. Регистры находятся в *блоке регистров* процессора – небольшой области оперативной памяти, где идентификаторы регистров играют роль адресов. Идентификатор регистра 0xF будет использоваться как признак запрета доступа к регистрам.

**Таблица 4.1.** Идентификаторы регистров в архитектуре Y86-64. Каждый из 15 регистров общего назначения имеет свой идентификатор (ID) в диапазоне от 0 до 0xF. Идентификатор 0xF указывает на отсутствие регистра-операнда

Идентификатор (ID)	Имя регистра	Идентификатор (ID)	Имя регистра
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	Нет регистра

Некоторые инструкции занимают всего 1 байт, однако инструкции, принимающие операнды, занимают больше байтов. Прежде всего может присутствовать дополнительный *байт спецификатора регистра*, определяющий один или два регистра. На рис. 4.2 поля этих регистров обозначены rA и rB. Как показывают мнемоники, инструкции могут использовать регистры в роли источников и приемников, а также в роли базового регистра для вычисления адреса. Инструкции, не имеющие регистров-операндов (переходы и вызовы процедур), не имеют байта спецификатора регистра. Инструкции, принимающие один регистр-операнд (*irmovq*, *pushq* и *popq*), используют спецификатор второго регистра со значением 0xF. Это соглашение пригодится нам при реализации процессора.

Некоторые инструкции принимают 8-байтное *слово-константу*. Это слово может представлять непосредственное значение данных в *irmovq*, смещение в *rmmovq* и *rrmovq* или адрес в инструкциях перехода и вызова процедуры. Обратите внимание, что в инструкциях перехода и вызова процедур адрес может иметь только абсолютное значение – относительная адресация, как в архитектуре x86-64, не поддерживается. Относительная адресация используется в процессорах, чтобы дать возможность более компактного представления инструкций ветвления в двоичном коде и перемещения программы из одной области памяти в другую без корректировки всех целевых адресов. Поскольку простота описания для нас имеет немаловажное значение, мы будем поддерживать только абсолютную адресацию. Так же как в архитектуре x86-64, все целые числа будут размещаться в памяти в формате с обратным порядком следования байтов (*little-endian*). При записи инструкций в дизассемблированной форме байты будут указываться в прямом порядке.

#### Сравнение кодов инструкций x86-64 и Y86-64

По сравнению с x86-64, архитектура Y86-64 имеет более простую, но в то же время менее компактную схему кодирования инструкций. Поля регистров во всех инструкциях Y86-64 задаются в строго фиксированных позициях, тогда как в разных инструкциях x86-64 они могут находиться в разных позициях. Инструкции x86-64 могут включать непосредственные значения с размерами 1, 2, 4 или 8 байт, а в инструкциях Y86-64 непосредственные значения могут быть представлены только 8 байтами.

Для примера сгенерируем шестнадцатеричный код инструкции `rmmovq %rsp, 0x123456789abcd(%rdx)`. Как показано на рис. 4.2, инструкция `rmmovq` начинается с бай-

та 40. Идентификатор регистра-источника `%rsp` должен быть указан в поле `rA`, а идентификатор базового регистра `%rdx` – в поле `rB`. Используя идентификаторы регистров из табл. 4.1, получаем байт спецификатора регистра 42. Наконец, смещение задается 8-байтным словом-константой. Сначала дополним `0x123456789abcd` ведущими нулями до 8 байт, получив в результате последовательность байтов `00 01 23 45 67 89 ab cd`. А затем запишем их в обратном порядке `cd ab 89 67 45 23 01 00`. Объединив все вместе, получаем шестнадцатеричный код инструкции `4042cdab896745230100`.

Одним из важных свойств любой системы команд является уникальная интерпретация последовательности байтов. Произвольная последовательность байтов либо представляет уникальную последовательность инструкций, либо является недопустимой. Это свойство поддерживается и в архитектуре Y86-64, потому что каждая инструкция имеет уникальную комбинацию кода инструкции и функции в первом байте, и по этому байту можно определить количество и значения любых дополнительных байтов. Данное свойство обеспечивает выполнение процессором объектного кода, смысл которого определяется однозначно. Даже если код встроен в другие байты программы, последовательность инструкций легко определяется до тех пор, пока отсчет ведется с первого байта последовательности. С другой стороны, если начальная позиция последовательности байтов выполняемого кода неизвестна, то нельзя определить с уверенностью способ разбиения этой последовательности на отдельные инструкции. Это вызывает проблемы в дизассемблерах и других инструментах, пытающихся выделить программы в машинном коде из последовательностей байтов объектного кода.

#### Упражнение 4.1 (решение в конце главы)

Определите последовательность байтов для следующей последовательности инструкций Y86-64. Строка `.pos 0x100` указывает, что объектный код начинается с адреса `0x100`.

```
.pos 0x100      # Код начинается с адреса 0x100
    irmovq $15,%rbx
    rrmovq %rbx,%rcx
loop:
    rmmovq %rcx,-3(%rbx)
    addq   %rbx,%rcx
    jmp    loop
```

#### Упражнение 4.2 (решение в конце главы)

Для каждой из перечисленных последовательностей байтов определите кодируемую ими последовательность инструкций Y86-64. Если в последовательности присутствует неправильный байт, укажите последовательность инструкций до этого места и местонахождение неправильного значения. Для каждой последовательности указаны начальный адрес, двоеточие и сама последовательность байтов.

1. `0x100: 30f3fcffffffffffffffff4063000800000000000000`
2. `0x200: a06f800c020000000000000030f30a0000000000000090`
3. `0x300: 505407000000000000000010f0b01f`
4. `0x400: 611373000400000000000000`
5. `0x500: 6362a0f0`

### Разногласия между RISC и CISC

В 1980-х годах в кругах специалистов по компьютерным архитектурам не прекращалась жаркая полемика относительно преимуществ наборов команд RISC и CISC. Сторонники RISC заявляли, что упрощенная система команд в комбинации с конвейерной организацией процессора и передовыми технологиями компиляции может значительно повысить вычислительную мощность определенного набора аппаратных средств. Приверженцы CISC возражали, что если для решения отдельно взятой задачи требуется меньше команд, то и совокупная производительность архитектур этого типа выше.

Крупные компании, в число которых входят Sun Microsystems (SPARC), IBM, Motorola (PowerPC) и Digital Equipment Corporation (Alpha), внедрили линии производства процессоров RISC. Британская компания Acorn Computers Ltd. разработала собственную архитектуру ARM (первоначально аббревиатура расшифровывалась как «Acorn RISC Machine»), которая стала широко использоваться во встроенных устройствах, таких как сотовые телефоны.

В начале 1990-х дебаты поутихли, после того как стало понятно, что ни RISC, ни CISC в чистом виде не в состоянии превзойти комбинированные архитектуры, объединяющие лучшие идеи RISC и CISC. Появились машины RISC, имевшие больше команд, на выполнение которых требовалось несколько циклов. Современные RISC-архитектуры имеют сотни команд, что уже едва ли можно связать с названием «сокращенный набор команд». Идея доступности артефактов реализации для программ машинного уровня оказалась недальновидной. По мере разработки новых моделей процессоров с использованием более совершенных аппаратных структур многие из этих артефактов оказались попросту ненужными, что, впрочем, не помешало им остаться частью набора команд. Ядром архитектуры RISC по-прежнему оставалась система команд, хорошо подходящая для конвейерной организации выполнения.

В CISC-архитектурах, появившихся позднее, также используются преимущества высокопроизводительных конвейерных структур. В разделе 5.7 мы рассмотрим технологии выборки команд CISC и их динамическую трансляцию в последовательность упрощенных RISC-подобных операций. Например, инструкция, складывающая значение в памяти со значением в регистре, транслируется в три операции: одна читает начальное значение из памяти, другая выполняет сложение и третья записывает сумму обратно в память. Поскольку динамическая трансляция может выполняться раньше фактического выполнения, процессор способен поддерживать очень высокую скорость выполнения.

Помимо технологических тонкостей немаловажную роль при определении рыночной успешности различных систем команд сыграли также коммерческие вопросы. Поддержка совместимости с существующими моделями обеспечила процессорам Intel и x86 довольно безболезненную смену поколений.

По мере развития технологии интегральных схем Intel и другие производители процессоров x86 смогли преодолеть неэффективность первоначального набора команд 8086 путем применения методик RISC с целью увеличения производительности до уровня, характерного для лучших RISC-архитектур. Как мы видели в разделе 3.1, эволюция IA32 в x86-64 дала возможность включить некоторые функции RISC в семейство x86. В области же разработки настольных компьютеров, ноутбуков и серверов господство x86 можно считать безраздельным.

Процессоры RISC пользуются огромной популярностью на рынке встраиваемых процессоров, применяемых во множестве устройств: в мобильных телефонах, тормозных системах автомобилей, сетевых устройствах и т. д. Во всех этих областях сокращение стоимости и экономия мощности важнее, чем поддержка обратной совместимости. Исходя из количества проданных процессоров, можно сказать, что у этого рынка весьма успешное будущее.



### Наборы команд RISC и CISC

Иногда x86-64 называют архитектурой с «полным набором команд» (Complex Instruction Set Computer, CISC) и считают противоположностью архитектурам с «сокращенным набором команд» (Reduced Instruction Set Computers, RISC). Архитектуры CISC появились первыми, став потомками самых первых компьютеров. К началу 80-х годов наборы команд для мэйнфреймов и мини-ЭВМ стали активно расширяться, по мере того как проектировщики внедряли новые команды для решения задач высокого уровня, такие как манипуляции с кольцевыми буферами, выполнение операций десятичной арифметики и вычисление многочленов. Первые микропроцессоры появились в начале 1970-х годов и имели ограниченные наборы команд, потому что в то время технология производства интегральных схем серьезно ограничивала возможности. Микропроцессоры развивались очень быстро и к началу 1980-х годов пошли по пути увеличения сложности наборов команд. Для семейства x86 тоже был выбран этот путь, и в результате появилась архитектура IA32, а затем и x86-64. Она также продолжает развиваться, и в ней появляются новые классы команд для поддержки новых потребностей приложений.

Философия проектирования RISC тоже появилась в начале 1980-х годов как альтернатива описанным тенденциям. Под влиянием идей исследователя Джона Кокка (John Cocke) группа экспертов по аппаратным средствам и компиляторам в компании IBM обнаружила, что эффективный код можно генерировать, используя гораздо более простой набор команд. Фактически многие высокоуровневые инструкции, добавляемые в наборы команд, трудно сгенерировать с помощью компилятора, поэтому они используются крайне редко. Более простую систему команд можно реализовать с гораздо меньшим количеством аппаратных средств и организовать ее в эффективную конвейерную структуру, похожую на ту, что описывается в настоящей главе далее. Эту идею IBM поставила на коммерческую основу только спустя много лет, когда были созданы архитектуры Power и PowerPC.

В дальнейшем концепция RISC развивалась профессорами Дэвидом Паттерсоном (David Patterson) в университете Беркли и Джоном Хеннесси (John Hennessy) из Стэнфордского университета. Именно Паттерсон дал название RISC новому классу архитектур, а CISC – существующему, потому что раньше не было необходимости в особом обозначении почти универсального набора команд.

Сравнивая CISC с первоначальными наборами команд RISC, можно заметить следующие общие характеристики.

Набор команд Y86-86 включает атрибуты обеих архитектур – RISC и CISC. От CISC унаследованы флаги условий, переменная длина инструкций и тесная связь процедур со стеком. От RISC унаследованы архитектура загрузки/сохранения, единообразное кодирование инструкций и передача аргументов через регистры. Y86-64 можно рассматривать как набор команд CISC (x86), упрощенный за счет применения некоторых принципов RISC.

CISC	Ранние архитектуры RISC
Большое количество инструкций. Документация Intel, описывающая полный набор команд [51], насчитывает свыше 1200 страниц	Набор команд намного меньше. Обычно меньше 100
Некоторые инструкции имеют продолжительное время выполнения. К ним относятся инструкции, копирующие целые блоки данных из одной области памяти в другую, а также инструкции, копирующие содержимое сразу нескольких регистров в память и из памяти	Инструкции с продолжительным временем выполнения отсутствуют. В некоторых ранних архитектурах RISC не было даже инструкций умножения целых чисел, по причине чего компиляторы были вынуждены реализовать умножение в виде последовательности сложений



CISC	Ранние архитектуры RISC
Разные инструкции могут кодироваться в последовательности байтов разной длины, например инструкции x86-64 могут занимать от 1 до 15 байт	Все инструкции кодируются в последовательности байтов фиксированной длины. Обычно 4 байта
Многочисленные форматы представления операндов. В x86-64 определение операнда в памяти может быть представлено множеством способов из разных комбинаций смещения, базового и индексного регистров и масштабного коэффициента	Простые форматы адресации. Обычно используются только базовый адрес и смещение
Арифметические и логические операции могут применяться как к операндам в памяти, так и в регистрах	Арифметические и логические операции применяются только к операндам в регистрах. Ссылки на ячейки памяти допускаются лишь в инструкциях <i>загрузки</i> значений из памяти в регистры и в инструкциях <i>сохранения</i> значений из регистров в памяти. Это соглашение называется <i>архитектурой загрузки/сохранения</i> (load/store architecture)
Артефакты реализации скрыты от программ машинного уровня. Набор команд обеспечивает четкую абстракцию между программами и способами их выполнения	Артефакты реализации видны программам машинного уровня. В некоторых RISC-архитектурах запрещены определенные последовательности инструкций и переходы на них до выполнения следующей инструкции. Компилятор оптимизирует производительность в рамках указанных ограничений
Флаги условий устанавливаются как побочный эффект выполнения команд, после чего используются для проверки условий ветвления	Флаги условий отсутствуют. Вместо этого для оценки условий используются явные инструкции проверки, которые сохраняют результат в обычном регистре
Процедуры тесно связаны с организацией стека. Стек используется для передачи аргументов процедурам и сохранения адресов возврата	Процедуры тесно связаны с регистрами. Регистры используются для передачи аргументов процедурам и сохранения адресов возврата. Поэтому некоторые процедуры могут вообще не обращаться к памяти. RISC-процессор обычно имеет гораздо больше регистров (до 32)

#### 4.1.4. Исключения в архитектуре Y86-64

Состояние, видимое программисту в архитектуре Y86-64 (рис. 4.1), включает код состояния **Stat**, описывающий общее состояние выполняющейся программы. Возможные значения этого кода перечислены в табл. 4.2. Значение 1 – код с именем AOK – сообщает о нормальном выполнении программы, а все остальные свидетельствуют о возникновении *исключения* некоторого типа. Код 2 (HLT) сообщает, что процессор выполнил инструкцию halt. Код 3 (ADR) сообщает, что процессор попытался обратиться по недопустимому адресу памяти во время выборки инструкции или во время чтения или записи данных. Мы ограничиваем максимальный адрес (точный предел зависит от реализации), и любой доступ к адресу за пределами этой границы вызовет исключение ADR. Код 4 (INS) сообщает, что обнаружена инструкция с недопустимым кодом.

**Таблица 4.2.** Коды состояния Y86-64. Проектируемый нами процессор будет останавливаться при появлении любого кода состояния, кроме AOK

Значение	Имя	Описание
1	AOK	Нормальное выполнение
2	HLT	Встречена инструкция halt
3	ADR	Попытка обращения к недопустимому адресу
4	INS	Встречена недопустимая инструкция

В своей архитектуре Y86-64 мы просто заставим процессор прекратить выполнение инструкций при появлении любого из перечисленных исключений. В более полной реализации процессор обычно вызывает *обработчик исключений* – процедуру, предназначенную для обработки определенного типа обнаруженного исключения. Как рассказывается в главе 8, обработчики исключений могут настраиваться для выполнения различных действий, таких как прерывание программы или вызов обработчика сигналов, определенного пользователем.

### 4.1.5. Программы из инструкций Y86-64

В листинге 4.1 показан ассемблерный код для архитектур x86-64 и Y86-64, полученный при компиляции следующей функции на C:

```
1 long sum(long *start, long count)
2 {
3     long sum = 0;
4     while (count) {
5         sum += *start;
6         start++;
7         count--;
8     }
9     return sum;
10 }
```

**Листинг 4.1.** Сравнение программ на ассемблере для архитектур Y86-64 и x86-64. Функция sum вычисляет сумму целочисленных элементов массива. Код для Y86-64 следует тому же общему шаблону, что и код для x86-64

Код для x86-64

```
long sum(long *start, long count)
start в %rdi, count в %rsi
1 sum:
2     movl    $0, %eax           sum = 0
3     jmp     .L2               Перейти к test
4 .L3:                          loop:
5     addq    (%rdi), %rax       Прибавить *start к sum
6     addq    $8, %rdi          start++
7     subq    $1, %rsi          count--
8 .L2:                          test:
9     testq   %rsi, %rsi        Проверить count
10    jne     .L3 If !=0,       Перейти к loop
11    rep; ret                  Возврат
```

Код для Y86-64

```
long sum(long *start, long count)
```

```

start в %rdi, count в %rsi
1  sum:
2  irmovq    $8,%r8          Константа 8
3  irmovq    $1,%r9          Константа 1
4  xorq      %rax,%rax        sum = 0
5  andq      %rsi,%rsi        Установить флаг
6  jmp       test            Перейти к test
7  loop:
8  mrmovq    (%rdi),%r10      Извлечь *start
9  addq      %r10,%rax        Прибавить к sum
10 addq      %r8,%rdi         start++
11 subq      %r9,%rsi         count--. Установить флаг
12 test:
13 jne       loop            Прервать цикл, если 0
14 ret                                Возврат

```

Код для x86-64 был сгенерирован компилятором GCC. Код для Y86-64 в целом похож на него, но имеет следующие отличия:

- код Y86-64 загружает константы в регистры (строки 2–3), потому что не поддерживает использование непосредственных данных в арифметических инструкциях;
- код Y86-64 требует двух инструкций (строки 8–9) для чтения значения из памяти и сложения со значением в регистре, тогда как код x86-64 может сделать то же самое одной инструкцией `addq` (строка 5);
- написанная вручную реализация для Y86-64 использует свойство инструкции `subq` (строка 11) устанавливать флаги условий, поэтому инструкция `testq`, сгенерированного компилятором GCC (строка 9), не требуется. Однако для правильной работы код Y86-64 должен установить флаги до входа в цикл, что достигается с помощью инструкции `andq` (строка 5).

В листинге 4.2 показано полное содержимое файла с кодом программы на языке ассемблера для Y86-64. Программа содержит и данные, и инструкции. Директивы сообщают адреса размещения кода и данных и требования к выравниванию. Программа определяет такие аспекты, как размещение стека, инициализация данных, инициализация и завершение программы.

Слова в этой программе, начинающиеся с точки, являются *директивами ассемблера*, определяющими адреса для размещения кода или данных. Директива `.pos 0` (строка 2) указывает, что код должен генерироваться для размещения, начиная с адреса 0. Это начальная точка всех программ Y86-64. Следующая инструкция (строка 3) инициализирует указатель стека. Можно заметить, что метка `stack` объявлена в конце программы (строка 40), после директивы `.pos` (строка 39), определяющей адрес `0x200`. То есть в этой программе стек будет начинаться с данного адреса и расти в сторону меньших адресов. Мы должны гарантировать, что стек не увеличится сверх меры и не затрет код или данные программы.

Строки с 8 по 13 программы объявляют массив из четырех слов со значениями

```

0x000d000d000d, 0x00c000c000c0,
0x0b000b000b00, 0xa000a000a000

```

Метка `array` обозначает начало этого массива и выровнена по 8-байтной границе (директивой `.align`). В строках с 16 по 19 находится процедура «main», которая вызывает функцию `sum` и передает ей массив из четырех слов, после чего останавливается.

Как показывает этот пример, из-за того, что единственным доступным инструментом для Y86-64 является ассемблер, чтобы создать программу, программист должен взять на себя задачи, которые обычно поручаются компилятору, компоновщику и системе времени выполнения. К счастью, нам не придется писать большие и сложные программы, и мы сможем обойтись простыми механизмами.

В листинге 4.3 показан результат сборки кода из листинга 4.2 ассемблером YAS. Для удобства чтения этот результат представлен в формате ASCII. В строках, содержащих инструкции или данные, в объектном коде указывается адрес, за которым следует значение, занимающее от 1 до 10 байт.

**Листинг 4.2.** Пример программы на ассемблере Y86-64. Функция `sum` вызывается для вычисления суммы элементов массива

```

1  # Выполнение начинается с адреса 0
2      .pos 0
3      irmovq stack, %rsp      # Установить указатель стека
4      call main               # Запустить основную программу
5      halt                   # Завершить программу
6
7  # Массив из 4 элементов
8      .align 8
9  array:
10     .quad 0x000d000d000d
11     .quad 0x00c000c000c0
12     .quad 0xb000b000b00
13     .quad 0xa000a000a00
14
15  main:
16     irmovq array,%rdi
17     irmovq $4,%rsi
18     call sum                 # sum(array, 4)
19     ret
20
21  # long sum(long *start, long count)
22  # start в %rdi, count в %rsi
23  sum:
24     irmovq $8,%r8            # Константа 8
25     irmovq $1,%r9            # Константа 1
26     xorq %rax,%rax           # sum = 0
27     andq %rsi,%rsi           # Установить флаги
28     jmp test                 # Перейти к test
29  loop:
30     mrmovq (%rdi),%r10        # Извлечь *start
31     addq %r10,%rax            # Прибавить к sum
32     addq %r8,%rdi             # start++
33     subq %r9,%rsi            # count-- . Установить флаги
34  test:
35     jne loop                 # Прервать цикл, если 0
36     ret                     # Возврат
37
38  # Здесь начинается стек и растет в сторону уменьшения адресов
39     .pos 0x200
40  stack:

```

**Листинг 4.3.** Результат работы ассемблера YAS. Каждая строка содержит шестнадцатеричный адрес и от 1 до 10 байт объектного кода

```

0x000: | # Выполнение начинается с адреса 0
0x000: 30f40002000000000000 | .pos 0
0x000: | irmovq stack, %rsp # Установить указатель
0x00a: 80380000000000000000 | стека
0x013: 00 | call main # Запустить основную программу
0x013: | halt # Завершить программу
0x018: | # Массив из 4 элементов
0x018: | .align 8
0x018: | array:
0x018: 0d000d000d000000 | .quad 0x000d000d000d
0x020: c000c000c0000000 | .quad 0x00c000c000c0
0x028: 00b000b000b00000 | .quad 0x0b000b000b00
0x030: 00a000a000a00000 | .quad 0xa000a000a000
0x038: | main:
0x038: 30f71800000000000000 | irmovq array,%rdi
0x042: 30f60400000000000000 | irmovq $4,%rsi
0x04c: 80560000000000000000 | call sum # sum(array, 4)
0x055: 90 | ret
0x056: | # long sum(long *start, long count)
0x056: | # start в %rdi, count в %rsi
0x056: | sum:
0x056: 30f80800000000000000 | irmovq $8,%r8 # Константа 8
0x060: 30f90100000000000000 | irmovq $1,%r9 # Константа 1
0x06a: 6300 | xorq %rax,%rax # sum = 0
0x06c: 6266 | andq %rsi,%rsi # Установить флаги
0x06e: 70870000000000000000 | jmp test # Перейти к test
0x077: | loop:
0x077: 50a70000000000000000 | mrmovq (%rdi),%r10 # Извлечь *start
0x081: 60a0 | addq %r10,%rax # Прибавить к sum
0x083: 6087 | addq %r8,%rdi # start++
0x085: 6196 | subq %r9,%rsi # count--
0x087: | Установить флаги
0x087: 74770000000000000000 | jne loop # Прервать цикл, если 0
0x090: 90 | ret # Возврат
0x200: | # Здесь начинается стек и растет в сторону
0x200: | # уменьшения адресов
0x200: | .pos 0x200
0x200: | stack:

```

Мы уже реализовали *имитатор набора команд*, который назвали YIS. Его цель – моделировать работу программы в машинном коде Y86-64 без имитации поведения какого-либо конкретного процессора. Такую форму моделирования удобно использовать для отладки программ до того, как будет доступно реальное оборудование, а также для проверки результатов моделирования оборудования или запуска программ на самом оборудовании. В процессе выполнения объектного кода нашей программы YIS сгенерировал следующий вывод:

Остановка через 34 шага на PC = 0x13. Состояние: 'HLT'. Флаги: Z=1 S=0 O=0  
Изменения в регистрах:

```
%rax: 0x0000000000000000 0x0000abcdabcdabcd
%rsp: 0x0000000000000000 0x0000000000000200
%rdi: 0x0000000000000000 0x0000000000000038
%r8:  0x0000000000000000 0x0000000000000008
%r9:  0x0000000000000000 0x0000000000000001
%r10: 0x0000000000000000 0x0000a000a000a000
```

Изменения в памяти:

```
0x01f0: 0x0000000000000000 0x0000000000000055
0x01f8: 0x0000000000000000 0x0000000000000013
```

В первой строке имитатор вывел сводную информацию о результатах выполнения: значение счетчика инструкций PC и состояние программы. В списках регистров и ячеек памяти выводятся только слова, изменившиеся в ходе выполнения. Исходные значения (здесь они все равны нулю) показаны слева, а конечные – справа. Как видите, регистр `%rax` содержит `0xabcdabcdabcd` – сумму элементов массива, переданного в процедуру `sum`. Кроме того, здесь видно, что в процессе выполнения был использован стек, начинающийся с адреса `0x200` и растущий в направлении уменьшения адресов, что вызвало изменение слов в памяти по адресам `0x1f0`–`0x1f8`. Адрес последней инструкции в выполняемом коде – `0x090`, а стек опускался вниз до адреса `0x01f0`, то есть при его использовании выполняемый код не повреждался.

#### Упражнение 4.3 (решение в конце главы)

В программах машинного уровня часто встречается операция прибавления константы к содержимому регистра. При использовании инструкций Y86-64, определенных к настоящему моменту, для выполнения этой операции сначала нужно выполнить инструкцию `imovq`, чтобы записать константу в регистр, а затем инструкцию `addq`, чтобы прибавить константу к содержимому целевого регистра. Предположим, что мы решили добавить новую инструкцию `iaddq`, имеющую следующий формат:

Байт	0	1	2	3	4	5	6	7	8	9
iaddq V, rB	C	O	F	rB	V					

Эта инструкция прибавляет постоянное значение **V** к значению в регистре **rB**.

Перепишите функцию `sum` на языке ассемблера для Y86-64 (листинг 4.1) так, чтобы она использовала инструкцию `iaddq`. В исходной версии мы выделили для хранения констант регистры `%r8` и `%r9`. Теперь вы сможете вообще отказаться от использования этих регистров.

#### Упражнение 4.4 (решение в конце главы)

Напишите код для Y86-64, реализующий рекурсивную версию функции `sum` – функцию `rsum`, основываясь на следующем коде на C:

```
long rsum(long *start, long count)
{
    if (count <= 0)
        return 0;
    return *start + rsum(start+1, count-1);
}
```

Используйте соглашения о передаче аргументов в регистрах, принятые в x86-64. Возможно, вам будет полезно скомпилировать код на машине x86-64, а затем перевести инструкции на язык Y86-64.

**Упражнение 4.5 (решение в конце главы)**

Измените код Y86-64 для функции `sum` (листинг 4.1) и реализуйте функцию `absSum`, вычисляющую сумму абсолютных значений элементов массива. Используйте инструкцию *условного перехода* во внутреннем цикле.

**Упражнение 4.6 (решение в конце главы)**

Измените код Y86-64 для функции `sum` (листинг 4.1) и реализуйте функцию `absSum`, вычисляющую сумму абсолютных значений элементов массива. Используйте инструкцию *условного перемещения данных* во внутреннем цикле.

### 4.1.6. Дополнительные сведения об инструкциях Y86-64

Большинство инструкций Y86-64 напрямую изменяют состояние программы, поэтому определить предполагаемый эффект каждой инструкции несложно. Но давайте обратим внимание на две необычные комбинации инструкций.

Инструкция `pushq` уменьшает указатель стека на 8 и записывает значение регистра в память. Однако не совсем понятно, что должен делать процессор при выполнении инструкции `pushq %rsp`, потому что сохраняемый регистр изменяется той же инструкцией. Возможны два варианта действий: (1) втолкнуть в стек исходное значение `%rsp` или (2) втолкнуть в стек уменьшенное значение `%rsp`.

Для нашего процессора Y86-64 мы примем то же соглашение, что действует для x86-64, как определено в следующем упражнении.

**Упражнение 4.7 (решение в конце главы)**

Давайте определим, как ведет себя инструкция `pushq %rsp` в процессоре x86-64. Мы могли бы попробовать прочитать описание этой инструкции в документации Intel, однако проще провести эксперимент на реальной машине. Компилятор C обычно не генерирует эту инструкцию, поэтому придется вручную написать код на ассемблере. Вот наша тестовая функция (в приложении в интернете «ASM:EASM» (раздел 3.3) мы рассказываем, как писать программы, сочетающие код на C и на ассемблере):

```

1  .text
2  .globl pushtest
3  pushtest:
4  movq    %rsp, %rax      Скопировать указатель стека
5  pushq   %rsp            Втолкнуть указатель в стек
6  popq    %rdx            Вытолкнуть обратно
7  subq    %rdx, %rax      Вернуть 0 или 8
8  ret
```

Позэкспериментировав, мы обнаружили, что функция `pushtest` всегда возвращает 0. Какое поведение инструкции `pushq %rsp` в x86-64 это подразумевает?

Подобная неоднозначность имеет место для инструкции `popq %rsp`. Какое значение получит `%rsp`? Прочитанное из памяти или увеличенное предыдущее значение, хранившееся в `%rsp`? Следуя за упражнением 4.7, проведем эксперимент и определим, как действует эта инструкция в процессоре x86-64, после чего будем использовать в нашей архитектуре Y86-64 точно такое же поведение.

**Упражнение 4.8 (решение в конце главы)**

Следующая функция поможет нам определить поведение инструкции `popq %rsp` в x86-64:

1	<code>.text</code>	
2	<code>.globl poptest</code>	
3	<code>poptest:</code>	
4	<code>movq %rsp, %rdi</code>	Сохранить указатель стека
5	<code>pushq \$0xabcd</code>	Втолкнуть тестовое значение
6	<code>popq %rsp</code>	Вытолкнуть в указатель стека
7	<code>movq %rsp, %rax</code>	Вернуть вытолкнутое значение
8	<code>movq %rdi, %rsp</code>	Восстановить указатель стека
9	<code>ret</code>	

Как показали эксперименты, эта функция всегда возвращает `0xabcd`. Какое поведение инструкции `popq %rsp` в x86-64 это подразумевает? Какая другая инструкция в Y86-64 будет иметь такое же поведение?

**Правильное понимание деталей: несоответствия между моделями x86**

Цель упражнений 4.7 и 4.8 – помочь разработать сбалансированный набор соглашений в отношении инструкций, вталкивающих в стек и выталкивающих со стека регистр указателя стека. Кажется, нет никаких причин, почему кто-то пожелал бы использовать эти операции, поэтому возникает естественный вопрос: зачем беспокоиться о таких тонкостях?

Чтобы ответить на этот вопрос, приведем выдержку из документации Intel с описанием инструкции `push [51]`, из которой можно извлечь несколько важных уроков, касающихся важности согласованности:

Для процессоров IA-32 от Intel 286 и выше инструкция `PUSH ESP` вталкивает в стек значение регистра `ESP`, имевшееся до выполнения инструкции. (Это также верно для архитектуры Intel 64 и режимов реальных адресов и виртуализации 8086 в архитектуре IA-32.) В процессоре Intel® 8086 инструкция `PUSH SP` вталкивает в стек обновленное значение регистра `SP` (то есть значение после уменьшения на 2).

Понять точно все детали этого примечания может быть трудно, но совершенно очевидно, что в нем говорится о разном поведении процессора x86 при работе в разных режимах, когда ему приказывают втолкнуть в стек регистр указателя стека. В одних режимах в стек вталкивается исходное значение, в других – уменьшенное. (Интересно отметить, что подобная неоднозначность отсутствует в отношении операции выталкивания значения со стека в регистр указателя стека.) У этого несоответствия есть два недостатка:

- снижает переносимость кода. Программы могут действовать по-разному, в зависимости от режима процессора. Да, эта конкретная инструкция не является общепотребимой, но даже потенциальная несовместимость может иметь серьезные последствия;
- усложняет документацию. В данном случае потребовалось добавить специальное примечание, чтобы попытаться прояснить различия. Документация для x86 уже достаточно сложна и без подобных примечаний.

Из вышесказанного следует, что предварительная проработка деталей и стремление к полной согласованности могут предотвратить появление множества проблем в будущем.



## 4.2. Логическое проектирование и язык HCL

В аппаратных архитектурах для вычисления функций с битами и сохранения битов в памяти разных типов применяются электронные схемы. Современная элементная база представляет различные значения битов в виде напряжения высокого и низкого уровней в сигнальных шинах. По существующей технологии логическое значение 1 представлено высоким напряжением около 1 В, а логическое значение 0 – низким напряжением около 0.0 В. Для реализации любой цифровой системы необходимы три основных компонента: *комбинационная логика* для вычисления функций с битами, *элементы памяти* для хранения битов и *сигналы синхронизации* для управления периодами обновления элементов памяти.

В данном разделе представлено краткое описание этих компонентов. Также мы познакомим вас с языком HCL (Hardware Control Language) для описания блоков управления аппаратных систем, с помощью которого раскрывается управляющая логика процессоров. Здесь HCL представлен неформально. Дополнительные сведения вы найдете в приложении в интернете «ARCH:HCL» (раздел 4.6).

### Современное логическое проектирование

Когда-то проектировщики аппаратных систем проектировали схемы, рисуя всевозможные логические диаграммы (сначала карандашом на бумаге, а позже на графических терминалах компьютеров). В настоящее время большинство проектов оформляются на языке описания аппаратного обеспечения (Hardware Description Language, HDL), похожем на язык программирования, который вместо различных шагов программ описывает аппаратные структуры. В число наиболее широко используемых языков входят: Verilog с синтаксисом, похожим на C, и VHDL с синтаксисом, похожим на Ada. Изначально эти языки разрабатывались для выражения моделей цифровых цепей. В середине 1980-х исследователи разработали программы логического синтеза, способные генерировать логические схемы по описаниям на языке HDL. Ныне существует много коммерческих версий таких программ, и их использование для генерирования цифровых цепей стало основополагающей методикой. Подобный переход от разработки схем вручную к синтезу можно сравнить с переходом с языка ассемблера на язык высокого уровня и генерированием машинного кода с помощью компилятора.

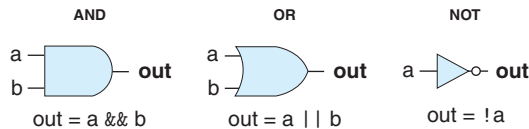
Наш язык HCL позволяет выражать только управляющие части аппаратных систем, поддерживая ограниченный набор операций без модульности. Однако, как мы покажем далее, логика управления – самая сложная часть разработки микропроцессора. Мы создали инструменты, способные напрямую транслировать описания с языка HCL на язык Verilog. Благодаря этому можно объединить этот код с кодом Verilog, описывающим основные аппаратные модули, и сгенерировать описания на языке HDL, из которых можно синтезировать реально работающие микропроцессоры. Тщательно отделив, спроектировав и протестировав логику управления, можно за разумное время создать вполне работающий микропроцессор. В приложении в интернете «ARCH:VLOG» (раздел 4.5.9) описывается, как можно сгенерировать Verilog-версию описания процессора Y86-64.

### 4.2.1. Логические вентили

Логическими вентилями называются базовые вычислительные элементы цифровых цепей. Они генерируют выходной сигнал, эквивалентный некоторой булевой функции, применяемой к битам на входе. На рис. 4.4 показаны стандартные обозначения булевых функций AND (И), OR (ИЛИ) и NOT (НЕ). Под значками вентилях представлены соответствующие выражения на HCL в форме операторов C (раздел 2.1.8): && для AND, || для OR и ! для NOT. Причина использования этих операторов вместо поразрядных

операторов  $\&$ ,  $|$  и  $\sim$  состоит в том, что логические вентили обрабатывают одноразрядные величины, а не целые слова. На рис. 4.4 показаны только версии вентилях AND и OR с двумя входами, однако нередко можно встретить вентили с  $n$  входами для  $n > 2$ . Но и в этом случае мы будем представлять их на языке HCL с использованием бинарных (двухместных) операторов, например представляя трехместную операцию AND с входами  $a$ ,  $b$  и  $c$  как  $a \&\& b \&\& c$ .

Логические вентили всегда активны. При изменении сигналов на входе через какое-то минимальное время изменится сигнал на выходе.



**Рис. 4.4.** Типы логических вентилях. Каждый вентиль генерирует выходной сигнал в соответствии с некоторой булевой функцией, применяемой к его входам

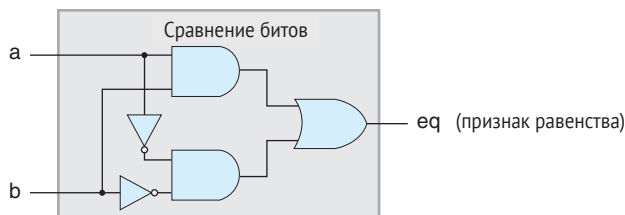
### 4.2.2. Комбинационные цепи и булевы выражения в HCL

Объединяя разные логические вентили в цепь, можно конструировать вычислительные блоки, известные как *комбинационные цепи*. При создании комбинационных цепей следует учитывать следующие ограничения:

- каждый вход логического вентиля должен быть подключен ровно к одному системному входу (известному как *первичный вход*), или выходу некоторого элемента памяти, или выходу некоторого логического вентиля;
- выходы двух и более логических вентилях нельзя соединять между собой, иначе каждый из них может попытаться установить в линии другой уровень напряжения, что может вызвать установку некорректного уровня или сбой в цепи;
- цепь должна быть *ациклической*. То есть в цепи не должно быть путей через последовательность вентилях, образующих замкнутый цикл. Такие циклы могут вызвать непредсказуемый результат, вычисляемый цепью.

На рис. 4.5 показан пример простой комбинационной цепи, которая может оказаться довольно полезной. Она имеет два входа  $a$  и  $b$  и единственный выход  $eq$ , на котором устанавливается 1, если на оба входа,  $a$  и  $b$ , подаются единицы (определяется верхним вентилем AND) или нули (определяется нижним вентилем AND). На языке HCL функцию этой цепи можно записать так:

```
bool eq = (a && b) || (!a && !b);
```



**Рис. 4.5.** Комбинационная цепь для проверки равенства битов.

На выходе появится 1, если на оба входа поданы одинаковые сигналы, 0 или 1

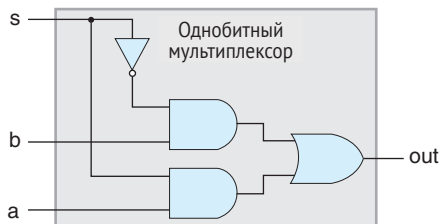
Этот код просто определяет битовый сигнал **eq** (что обозначено типом данных `bool`) как функцию от входных значений **a** и **b**. Как показывает этот пример, в языке HCL используется синтаксис, подобный синтаксису C, где символ равенства (=) ассоциирует название сигнала с выражением. Однако, в отличие от C, это выражение не рассматривается как расчет и запись результата в некоторую ячейку памяти, – это всего лишь способ дать имя выражению.

#### Упражнение 4.9 (решение в конце главы)

Напишите выражение HCL для сигнала **xor**, эквивалентного операции EXCLUSIVE-OR (ИСКЛЮЧАЮЩЕЕ ИЛИ) с входными значениями **a** и **b**. Есть ли какая-то связь между **xor** и сигналом **eq**, определенным выше?

На рис. 4.6 показан еще один пример простой, но полезной комбинационной цепи под названием *мультиплексор* (на схемах мультиплексоры часто обозначают как «MUX»). Он выбирает тот или иной входной сигнал для передачи на выход, в зависимости от значения управляющего входного сигнала. В этом однобитном мультиплексоре входными сигналами являются биты **a** и **b**, а управляющим сигналом – бит **s**. На выходе **out** будет установлено значение **a**, если на **s** подается единица, и **b**, если на **s** подается ноль. В этой цепи имеется два вентиля AND (И), которые определяют, передавать ли соответствующие им входные сигналы на вход вентиля OR (ИЛИ). Верхний вентиль AND (И) передает сигнал **b**, когда на **s** подается ноль (потому что на второй вход вентиль подается **!s**), а нижний вентиль AND (И) передает сигнал **a**, когда на **s** подается единица. И снова формирование выходного сигнала можно описать выражением на HCL, используя те же операции, которые выполняет комбинационная цепь:

```
bool out = (s && a) || (!s && b);
```



**Рис. 4.6.** Однобитный мультиплексор. На выходе будет установлено входное значение **a**, если на управляющий вход **s** подается единица, и **b**, если на **s** подается ноль

Написанные здесь выражения HCL демонстрируют четкую параллель между цепями комбинаторной логики и логическими выражениями в C. Для вычисления функций от входов и те, и другие применяют булевы операции. Стоит отметить несколько различий между этими двумя способами выражения вычислений.

- Поскольку комбинационная цепь состоит из серии логических вентиль, она обладает свойством непрерывной реакции на изменения входных сигналов. При изменении определенных входных сигналов по истечении минимального количества времени выходы изменяются соответствующим образом. Выражения в C, напротив, вычисляются только тогда, когда они встречаются на пути выполнения программы.

- Логические выражения в C допускают передачу в аргументах произвольных целых чисел, из которых 0 интерпретируется как ЛОЖЬ, а все остальные – как ИСТИНА. Логические вентили, напротив, оперируют лишь битовыми значениями 0 и 1.
- Логические выражения в C могут вычисляться частично. Если результат операции AND (И) или OR (ИЛИ) можно определить уже по первому аргументу, то второй аргумент не участвует в вычислениях. Например, в выражении на C

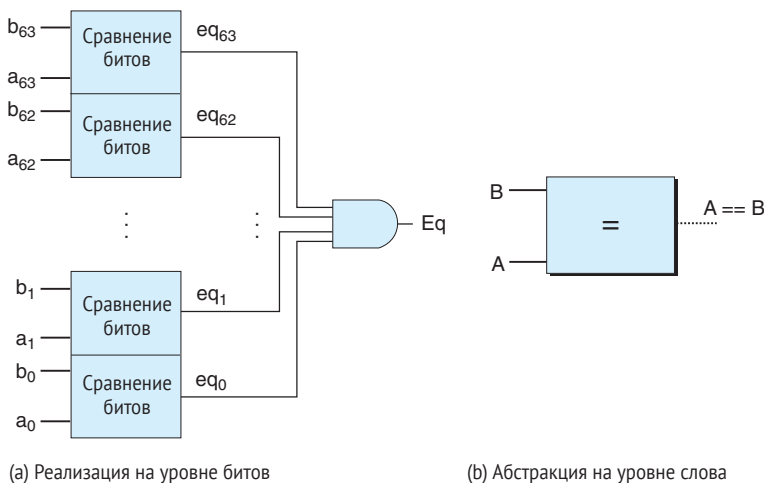
$$(a \ \&\& \ !a) \ \&\& \ \text{func}(b,c)$$

функция `func` не будет вызвана, потому что выражение  $(a \ \&\& \ !a)$  всегда дает 0. Комбинаторная логика, наоборот, не поддерживает никаких правил частичного вычисления. Выходы просто реагируют на изменения входов.

### 4.2.3. Комбинационные цепи для слов и целочисленные выражения в HCL

Из логических вентилях можно создавать комбинационные цепи, способные вычислять намного более сложные функции. Обычно такие цепи оперируют *словами* – группами однобитных сигналов, представляющими целые числа или определенные управляющие комбинации. Например, рассматриваемые здесь проекты процессора будут содержать большое количество слов с длинами в диапазоне от 4 до 64 бит, представляющих целые числа, адреса, коды инструкций и идентификаторы регистров.

Комбинационные цепи для вычислений со словами строятся с использованием однобитных логических вентилях, определяющих значения битов в выходном слове по входным битам. Например, на рис. 4.7 показана комбинационная цепь проверки равенства двух 64-разрядных слов, **A** и **B**. То есть на выход будет выдаваться 1, если, и только если каждый бит в **A** равен соответствующему биту в **B**. Эта цепь реализована с использованием 64 однобитных цепей сравнения, представленных на рис. 4.5. Выходы этих однобитных цепей подаются на вход вентиля AND (**B**), который формирует общий выход цепи.



**Рис. 4.7.** Цепь для проверки равенства слов. На выход будет выводиться 1, только если все биты в слове **A** равны соответствующим битам в слове **B**. Сравнение слов является одной из операций в HCL

В HCL любое слово объявляется как `int` без ограничения длины этого слова. Это делается исключительно для простоты. В полнофункциональном языке описания аппаратного обеспечения можно задать размер каждого слова. HCL позволяет сравнивать слова на предмет их равенства, поэтому функциональность цепи на рис. 4.7 можно описать на уровне слов как

```
bool Eq = (A == B);
```

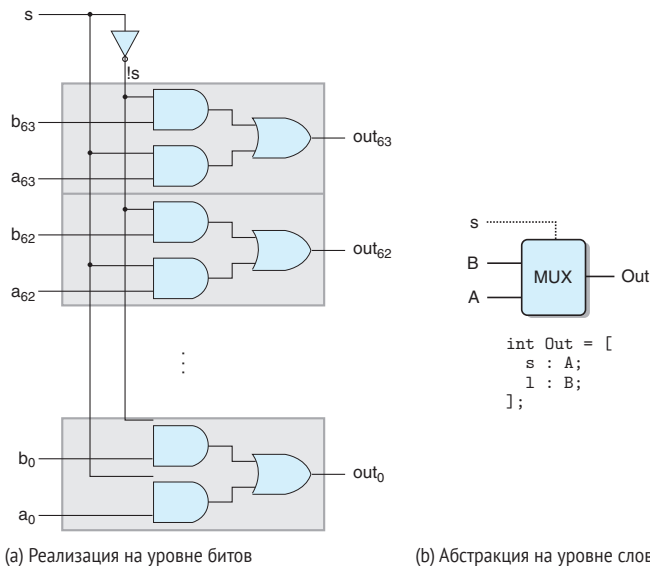
где аргументы **A** и **B** имеют тип `int`. Обратите внимание, что здесь используются те же соглашения о синтаксисе, что и в C, где знак равенства (=) обозначает присваивание, а двойной знак равенства (==) – оператор сравнения.

Как показано справа на рис. 4.7, цепи для слов используют полужирные линии для представления наборов проводов (шин), передающих отдельные биты, и пунктирные линии – для однобитных выходных сигналов.

#### Упражнение 4.10 (решение в конце главы)

Предположим, что вам нужно реализовать схему определения равенства слов, используя цепь EXCLUSIVE-OR (ИСКЛЮЧАЮЩЕЕ ИЛИ) из упражнения 4.9. Спроектируйте такую цепь для 64-разрядного слова, состоящую из 64 однобитных цепей EXCLUSIVE-OR (ИСКЛЮЧАЮЩЕЕ ИЛИ) и двух дополнительных логических вентилей.

На рис. 4.8 представлена цепь мультиплексора на уровне слова. Она выдает 64-разрядное слово **Out**, равное одному из двух входных слов, **A** или **B**, в зависимости от управляющего сигнала **s**. Данная цепь состоит из 64 идентичных однобитных цепей, каждая из которых повторяет структуру однобитного мультиплексора, изображенного на рис. 4.6. Вместо точного воспроизведения однобитного мультиплексора 64 раза в версии для слова число инверторов (обратных преобразователей) уменьшено до одного и его результат **!s** используется всеми однобитными цепями.



**Рис. 4.8.** Цепь мультиплексора слов. На выход *Out* выдается входное слово *A*, если на управляющий вход *s* подается 1, и *B* в противном случае. Мультиплексоры описываются в HCL с использованием *case*-выражений (выражений выбора из нескольких вариантов)

При проектировании процессора в этой книге будут использоваться разные формы мультиплексоров. Они позволяют выбрать слово из некоторого множества входных слов, в зависимости от некоторого количества управляющих условий. В языке HCL мультиплексоры описываются с использованием *case-выражений* (выражений выбора из нескольких вариантов), которые в общем виде имеют такой синтаксис:

```
[
    select1 : expr1;
    select2 : expr2;
    .
    .
    .
    selectk : exprk;
]
```

Выражение содержит набор вариантов, каждый из которых состоит из булева условного выражения *select<sub>i</sub>*, определяющего условия выбора этого варианта, и целочисленного выражения *expr<sub>i</sub>*, определяющего значение результата.

В отличие от оператора выбора *switch* в языке C, здесь не требуется, чтобы условные выражения были взаимоисключающими, потому что они оцениваются последовательно, друг за другом, и всегда выбирается первый вариант, удовлетворяющий условиям, заданным в его условном выражении. Например, мультиплексор слов, показанный на рис. 4.8, можно описать на HCL так:

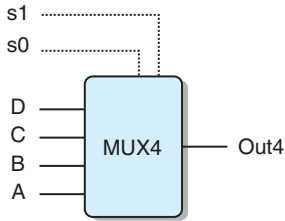
```
word Out = [
    s: A;
    1: B;
];
```

В этом примере второе условное выражение – просто единица, указывающая, что этот вариант должен выбираться всегда, когда не выбирается предыдущий. Так в HCL задается выбор по умолчанию, и фактически все *case-выражения* заканчиваются подобным образом.

Отсутствие требования к однозначности условных выражений упрощает чтение кода HCL. Но реальный аппаратный мультиплексор должен иметь взаимоисключающие условия, управляющие выбором слова на входе, подобно сигналам *s* и *!s* на рис. 4.8. Для преобразования *case-выражения* на языке HCL в аппаратную реализацию программе логического синтеза потребуется проанализировать набор условных выражений и решить все возможные противоречия (конфликты), чтобы гарантировать выбор первого варианта, подпадающего под заданные условия.

Условные выражения могут быть любыми булевыми выражениями, и число вариантов не ограничивается. Это позволяет описывать блоки с множеством входных сигналов и сложными критериями выбора. Например, рассмотрим следующую схему мультиплексора с четырьмя входами (рис. 4.9). Эта цепь делает выбор из четырех входных слов: *A*, *B*, *C* и *D*, проверяя управляющие сигналы *s1* и *s0* и рассматривая их как 2-разрядное двоичное число. Этот мультиплексор можно выразить на языке HCL, используя булевы выражения для описания разных комбинаций управляющих сигналов:

```
word Out4 = [
    !s1 && !s0 : A; # 00
    !s1       : B; # 01
    !s0       : C; # 10
    1         : D; # 11
];
```



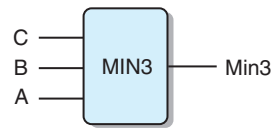
**Рис. 4.9.** Мультиплексор с четырьмя входами. Различные комбинации управляющих сигналов *s1* и *s0* определяют, какое входное слово передается на выход

Комментарии справа (текст от символа # и до конца строки называется комментарием) указывают, какая комбинация *s1* и *s0* определяет выбор данного варианта. Обратите внимание, что условные выражения иногда могут упрощаться, потому что всегда выбирается первый вариант, соответствующий заданному условию. Например, второе условное выражение можно записать как `!s1`, вместо более сложной полной формы `!s1 && s0`, потому что единственная альтернатива, когда *s1* равно 0, представлена в первом условном выражении. Аналогично третье выражение можно записать как `!s0`, а четвертое – просто как 1.

И наконец, последний пример. Предположим, что нужно спроектировать логическую цепь, выбирающую минимальное из трех слов *A*, *B* и *C*, диаграмма которой показана на рис. 4.10.

На языке HCL ее можно выразить так:

```
word Min3 = [
    A <= B && A <= C : A;
    B <= A && B <= C : B;
    1                  : C;
];
```



**Рис. 4.10.** Цепь выбора минимального слова

#### Упражнение 4.11 (решение в конце главы)

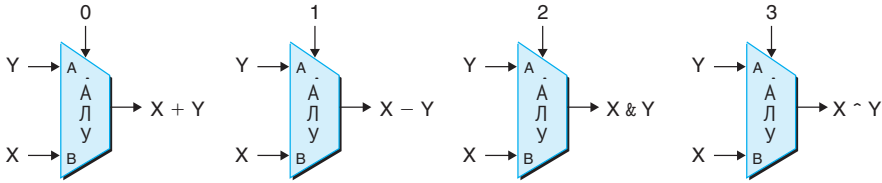
Код HCL для определения минимального из трех слов содержит четыре выражения сравнения вида `X <= Y`. Перепишите код так, чтобы он вычислял тот же результат с использованием трех сравнений.

#### Упражнение 4.12 (решение в конце главы)

Напишите код HCL, описывающий цепь, которая принимает на входе три слова *A*, *B* и *C* и выбирает медиану. То есть на выход должно выдаваться слово со значением между минимальным и максимальным значениями из трех входных слов.

Цепи комбинаторной логики могут выполнять множество самых разных операций со словами. Но подробное описание проектирования таких цепей не является целью этой книги. Одна из наиболее важных комбинационных цепей известна как *арифметико-логическое устройство* (АЛУ). В общем виде она изображена на рис. 4.11. Эта версия имеет три входа: два слова данных, *A* и *B*, и управляющее слово. В зависимости от значения управляющего слова цепь будет выполнять разные арифметические или логические операции с входными данными. Обратите внимание, что четыре операции, изображенные на рис. 4.11, соответствуют четырем целочисленным операциям, входящим в набор команд Y86-64, а управляющие значения – кодам функций этих инструкций (рис. 4.3). Также обратите внимание на порядок операндов в операции вычитания:

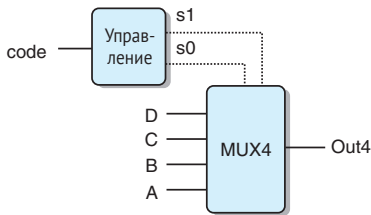
входное слово **A** вычитается из входного слова **B**. Этот порядок соответствует порядку аргументов в будущей инструкции `subl`.



**Рис. 4.11.** Арифметико-логическое устройство (АЛУ). В зависимости от выбора функции на входе цепь будет выполнять одну из четырех арифметических и логических операций

#### 4.2.4. Принадлежность множеству

При проектировании процессора вы неизбежно столкнетесь с множеством ситуаций, когда потребуется сравнить один сигнал с некоторым количеством возможных вариантов сигналов, например чтобы определить категорию инструкции по ее коду. В качестве простого примера предположим, что нужно сгенерировать сигналы **s1** и **s0** для мультиплексора с четырьмя входами, изображенного на рис. 4.9, выбирая старший и младший биты их 2-битного сигнала, как показано на рис. 4.12.



**Рис. 4.12.** Управление мультиплексором с помощью 2-битного сигнала

В этой цепи 2-битный сигнал **code** управляет выбором одного из четырех слов данных **A**, **B**, **C** и **D**. Разложение 2-битного сигнала на два 1-битных сигнала **s1** и **s0** можно выразить с помощью проверок на равенство с возможными значениями **code**:

```
bool s1 = code == 2 || code == 3;
bool s0 = code == 1 || code == 3;
```

То же самое можно выразить более кратко: сигнал **s1** получает значение 1, когда **code** принадлежит множеству {2, 3}, а сигнал **s0** получает значение 1, когда **code** принадлежит множеству {1, 3}:

```
bool s1 = code in { 2, 3 };
bool s0 = code in { 1, 3 };
```

В более общей форме проверка принадлежности множеству выглядит так:

```
iexpr in {iexpr1, iexpr2, ..., iexprk}
```

где проверяемое значение (*iexpr*) и кандидаты на совпадение (от *iexpr*<sub>1</sub> до *iexpr*<sub>k</sub>) являются целочисленными выражениями.

#### 4.2.5. Память и синхронизация

По своей природе комбинационные цепи не предназначены для хранения информации. Они просто реагируют на входные сигналы и генерируют выходные сигналы, применяя некоторую функцию к входам. Для создания цепей *последовательного действия*, то есть системы, имеющей состояние и выполняющей вычисления с этим состоянием,

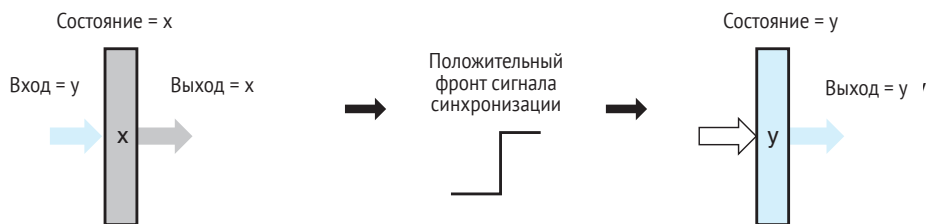


необходимо добавить устройства, хранящие информацию в виде битов. Запоминающие устройства управляются единственным *тактовым генератором*, по сигналам которого новые данные загружаются в устройство. Мы рассмотрим два класса запоминающих устройств:

- *синхронизированные регистры* (или просто *регистры*) хранят отдельные биты или слова. Синхронизирующий сигнал управляет загрузкой входного значения в регистр;
- *память с произвольным доступом* (*оперативная память*, или просто *память*) хранит слова; ячейка памяти для чтения либо записи слова определяется адресом. В числе примеров оперативной памяти можно назвать (1) виртуальную память процессора, которая, благодаря комбинации аппаратных и программных средств, с точки зрения процессора выглядит как большое адресное пространство, позволяет процессору обращаться к любому слову в пределах определенного адресного пространства; и (2) блок регистров, в котором роль адресов играют идентификаторы регистров. В процессоре Y86-64 блок регистров содержит 15 регистров (от `%rax` до `%r14`).

При ближайшем рассмотрении оказывается, что с аппаратной и программной точек зрения слово «регистр» обозначает разные вещи. С аппаратной точки зрения входы и выходы регистра напрямую связаны с остальной цепью. С программной точки зрения регистры представляют небольшой набор адресуемых слов в центральном процессоре, где адресами служат идентификаторы регистров. Обычно эти слова хранятся в блоке регистров, хотя далее в книге мы покажем, что иногда аппаратные компоненты могут передавать слова между инструкциями напрямую, чтобы исключить затраты времени сначала на запись, а потом на чтение регистров. Дабы избежать двусмысленности, эти два класса регистров мы будем называть «аппаратными регистрами» и «программными регистрами» соответственно.

На рис. 4.13 изображен аппаратный регистр и демонстрируется, как он работает. Большую часть времени регистр остается в фиксированном состоянии (изображено как **x**), генерируя выходные данные, равные текущему состоянию. Сигналы, что передаются через предшествующую регистру комбинаторную логику, создают новое значение на входе регистра (**y**), однако выходное значение регистра остается фиксированным, пока сохраняется низкий уровень тактового сигнала. С положительным перепадом тактового сигнала данные на входе загружаются в регистр, и тот переходит в новое состояние (**y**), фиксируя его на своих выходах до следующего положительного перепада тактового сигнала. Важно отметить, что регистр играет роль барьера между комбинаторной логикой в разных частях цепи. Значения передаются с входа на выход регистра только по положительному перепаду тактового сигнала. Наш процессор Y86-64 будет использовать такие синхронизированные регистры для хранения счетчика инструкций (**PC**), флагов условий (**CC**) и состояния программы (**Stat**).



**Рис. 4.13.** Работа регистра. Выходы регистров продолжают хранить текущее состояние до появления высокого уровня тактового сигнала. С положительным перепадом тактового сигнала значения на входах регистров фиксируются и становятся новым состоянием регистра

Диаграмма на рис. 4.14 демонстрирует типичный блок регистров.

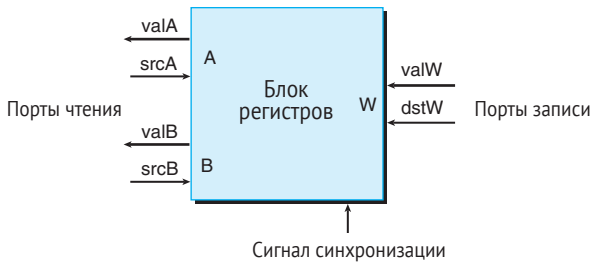


Рис. 4.14. Типичный блок регистров

Этот блок регистров имеет два *порта для чтения* с именами **A** и **B** и один *порт для записи* с именем **W**. Такая оперативная память с *несколькими портами* обеспечивает возможность одновременного выполнения нескольких операций чтения и записи. При наличии блока регистров, изображенного на рис. 4.14, цепь может читать значения из двух программных регистров и изменять состояние третьего. Каждый порт имеет адресный вход, определяющий выбранный программный регистр, а также выход или вход данных, задающий значение этого программного регистра. Адресами служат идентификаторы регистров, которые кодируются в соответствии с табл. 4.1. Два порта чтения имеют адресные входы **srcA** и **srcB** («источник A» и «источник B») и выходные данные **valA** и **valB** («значение A» и «значение B»). Порт записи имеет адресный вход **dstW** («приемник W») и входные данные **valW** («значение W»).

Блок регистров не является логической цепью, потому что имеет внутреннее запоминающее устройство, однако чтение слов из него происходит так, как если бы он был блоком комбинаторной логики, принимающим адреса на входе и выдающим данные на выходе. Когда на вход **srcA** или **srcB** подается идентификатор регистра, по истечении некоторого времени значение, хранящееся в соответствующем программном регистре, появится на выходе **valA** или **valB**. Например, если установить 3 на входе **srcA**, то произойдет чтение программного регистра %ebx, и его значение появится на выходе **valA**.

Запись слов в блок регистров управляется сигналом синхронизации. С каждым положительным перепадом тактового сигнала значение на входе **valW** записывается в программный регистр, определяемый адресом на входе **dstW**. Если на входе **dstW** установлено особое значение 0xF, операция записи не выполняется. Поскольку блок регистров доступен для чтения и для записи, возникает естественный вопрос: «Что произойдет, если цепь попытается одновременно прочитать и записать один и тот же регистр?» Ответ прост: если адрес одного и того же регистра будет установлен на входах порта чтения и порта записи, то с положительным перепадом тактового сигнала на выходе порта чтения появится новое значение. При включении блока регистров в нашу конструкцию процессора мы обязательно учтем это свойство.

Наш процессор имеет оперативную память для хранения программных данных, как показано на рис. 4.15.

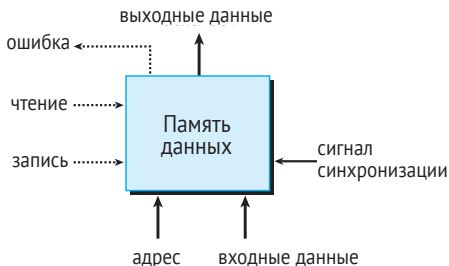


Рис. 4.15. Оперативная память для хранения программных данных

Эта память имеет один адресный вход, вход данных для записи и выход для чтения данных. По аналогии с блоком регистров, чтение из памяти происходит с использованием комбинаторной логики: если подать адрес на адресный вход и установить сигнал управления записью **write** в 0, то после некоторой задержки значение, хранящееся по этому адресу, появится на выходе. Сигнал ошибки **error** будет установлен в 1, если значение адреса окажется вне допустимого диапазона, и в 0 в противном случае. Запись в память управляется сигналом синхронизации: мы устанавливаем желаемый адрес на адресном входе, подаем входные данные и подаем 1 на вход управления записью **write**. Затем, с положительным перепадом тактового сигнала, указанная ячейка в памяти обновится, если установлен действительный адрес. Как и в случае с операцией чтения, сигнал ошибки **error** будет установлен в 1, если значение адреса окажется вне допустимого диапазона. Этот сигнал генерируется комбинаторной логикой, потому что проверка границ является функцией ввода адреса и не требует хранения какого-либо состояния.

В нашем процессоре есть дополнительная память для чтения инструкций. В большинстве реальных систем эта память объединена в единую память с двумя портами: один для чтения инструкций и один для чтения или записи данных.

#### Устройство реальной памяти

Система памяти в полноценном микропроцессоре намного сложнее той, что предполагается нашим проектом. Она состоит из нескольких форм аппаратной памяти, в том числе нескольких запоминающих устройств с произвольным доступом, а также энергонезависимой памяти или магнитного диска, а кроме того, множества аппаратных и программных механизмов для управления этими устройствами. Устройство и характеристики системы памяти описаны в главе 6.

Тем не менее наш простой проект памяти можно использовать для реализации небольших систем, и он может служить абстракцией интерфейса между процессором и памятью для более сложных систем.

## 4.3. Последовательные реализации Y86-64 (SEQ)

Теперь у нас есть все необходимое для реализации процессора Y86-64. В качестве первого шага мы опишем процессор SEQ (от «sequential» – последовательный). В каждом цикле SEQ выполняет все шаги, необходимые для полной обработки одной инструкции. Однако для этого цикл должен длиться довольно долго, поэтому частота тактового генератора будет непозволительно маленькой. Однако цель разработки процессора SEQ состоит лишь в том, чтобы сделать первый шаг к конечной цели реализации эффективного конвейерного процессора.

### 4.3.1. Организация обработки в несколько этапов

Вообще говоря, обработка инструкции предполагает выполнение нескольких элементарных операций. Мы организуем их в определенном порядке, чтобы сформировать универсальную последовательность обработки любых инструкций, даже притом, что инструкции очень разнятся выполняемыми операциями. Детали операций, выполняемых на каждом этапе, зависят от конкретной инструкции. Создание такой схемы позволит спроектировать процессор, наиболее оптимально и эффективно использующий возможности аппаратных средств. Далее приводится неформальное описание этих этапов и выполняемых в них операций.

*Выборка*

На этапе выборки происходит чтение байтов инструкции из памяти с использованием счетчика инструкций (PC), хранящего адрес памяти. Затем из первого байта инструкции извлекаются два 4-битных фрагмента, которые мы будем называть **icode** (код инструкции) и **ifun** (функция инструкции). Далее может извлекаться байт-спецификатор регистра, сообщающий о необходимости использовать один или два регистра-операнда **rA** и **rB**. Затем может извлекаться 8-байтное слово константы **valC**. После этого вычисляется **valP** – адрес инструкции, следующей за текущей. То есть **valP** равно значению PC плюс длина выбранной инструкции.

*Декодирование*

На этапе декодирования извлекаются данные из регистров, **valA** и/или **valB**. Обычно значения извлекаются из регистров, закодированных в полях **rA** и **rB**, однако некоторые инструкции читают регистр **%rsp**.

*Выполнение*

На этом этапе арифметико-логическое устройство выполняет операцию, обозначаемую инструкцией (в соответствии со значением **ifun**), вычисляет эффективный адрес ссылки на ячейку памяти или увеличивает/уменьшает указатель стека. Полученное в результате значение мы назовем **valE**. Возможно, потребуются проверить флаги условий. Для инструкции условного перемещения на этапе выполнения осуществляется проверка флагов и условий ветвления (задаваемых **ifun**), чтобы выяснить необходимость ветвления.

*Обращение к памяти*

На этом этапе может производиться запись или чтение памяти. Записываемое или читаемое значение мы обозначим как **valM**.

*Обратная запись*

На этом этапе производится запись до двух значений в блок регистров.

*Изменение PC*

Вычисляется и записывается в PC адрес следующей инструкции.

Процессор снова и снова выполняет эти этапы в бесконечном цикле и останавливается, только встретив инструкцию **halt** или недопустимую инструкцию либо при попытке обратиться к недопустимому адресу памяти. Процессор с более полной реализацией должен войти в режим обработки исключений и начать выполнение специального кода, определяемого типом исключения.

По предыдущему описанию видно, что для обработки единственной инструкции требуется выполнить на редкость большой объем работы. Процессор должен выполнить не только основную операцию, предполагаемую инструкцией, но также вычислить адреса, обновить указатели стека и определить адрес следующей инструкции. К счастью, в целом поток выполнения одинаков для любых инструкций. При проектировании аппаратных систем важно использовать максимально простые и универсальные схемы обработки, чтобы свести к минимуму общее количество аппаратных компонентов и уместить их на плоской поверхности кристалла. Один из способов уменьшить сложность – обеспечить возможность совместного использования инструкциями как можно большего количества аппаратных компонентов. Например, все рассматриваемые проекты процессоров содержат одно арифметико-логическое устройство, используемое всеми инструкциями по-разному, в зависимости от типа выполняемой операции. Затраты на дублирование аппаратных логических блоков намного выше затрат на дублирование программного кода. Кроме того, аппаратно обрабатывать особые случаи и отличительные особенности намного сложнее, чем программно.

Нашей задачей на данном этапе является такая организация вычислительного процесса, чтобы каждая инструкция соответствовала общей схеме. Для иллюстрации обработки различных инструкций Y86-64 будет использоваться код, показанный в листинге 4.4, а в табл. 4.3–4.6 описывается, как выполняются разные этапы обработки инструкций Y86-64. Эти таблицы заслуживают тщательного изучения. Они прямо отражают аппаратное устройство процессора. Каждая строка в этих таблицах описывает присваивание значения тому или иному сигналу или хранимому состоянию (обозначается операцией присваивания  $\leftarrow$ ). Читать их желательно последовательно, сверху вниз. Потом, когда придет время отобразить вычисления в аппаратную реализацию, станет очевидно, что вычисления необязательно выполнять строго последовательно.

**Листинг 4.4.** Пример последовательности инструкций Y86-64. Мы проследим обработку этих инструкций на различных этапах

```
1 0x000: 30f20900000000000000 | irmovq $9, %rdx
2 0x00a: 30f31500000000000000 | irmovq $21, %rbx
3 0x014: 6123 | subq %rdx, %rbx # Вычесть
4 0x016: 30f48000000000000000 | irmovq $128,%rsp # Упражнение 4.13
5 0x020: 40436400000000000000 | rmmovq %rsp, 100(%rbx) # Сохранить
6 0x02a: a02f | pushq %rdx # Втолкнуть в стек
7 0x02c: b00f | popq %rax # Упражнение 4.14
8 0x02e: 734000000000000000 | je done # Не выполняется
9 0x037: 804100000000000000 | call proc # Упражнение 4.18
10 0x040: | done:
11 0x040: 00 | halt
12 0x041: | proc:
13 0x041: 90 | ret # Возврат
14
```

**Таблица 4.3.** Обработка инструкций OPq, rmmovq и irmovq в последовательной реализации процессора Y86-64 (SEQ). Эти инструкции вычисляют значение и сохраняют результат в регистре. Обозначение **icode:ifun** указывает на наличие двух компонентов в первом байте инструкции, а **rA:rB** – двух компонентов в байте-спецификаторе регистров. Обозначение  $M_1[x]$  указывает на необходимость доступа (чтения или записи) к 1 байту в памяти с адресом x, а  $M_8[x]$  указывает на необходимость доступа к 8 байтам

Этап	OPq rA, rB	rmmovq rA, rB	irmovq V, rB
Выборка	icode : ifun $\leftarrow M1[PC]$ rA :rB $\leftarrow M_1[PC + 1]$  valP $\leftarrow PC + 2$	icode : ifun $\leftarrow M1[PC]$ rA :rB $\leftarrow M_1[PC + 1]$  valP $\leftarrow PC + 2$	icode : ifun $\leftarrow M1[PC]$ rA :rB $\leftarrow M_1[PC + 1]$ valC $\leftarrow M_8[PC + 2]$ valP $\leftarrow PC + 10$
Декодирование	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	valA $\leftarrow R[rA]$	
Выполнение	valE $\leftarrow$ valB OP valA Установка флагов условий	valE $\leftarrow 0 +$ valA	valE $\leftarrow 0 +$ valC
<b>Обращение к памяти</b>			
Обратная запись	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE
Изменение PC	PC $\leftarrow$ valP	PC $\leftarrow$ valP	PC $\leftarrow$ valP

В табл. 4.3 показана обработка инструкций типа OPq (целочисленные и логические операции), rmmovq (обмен данными между регистрами) и irmovq (запись непосредствен-

ного значения в регистр). Для начала рассмотрим целочисленные операции. На рис. 4.2 показана тщательно проработанная схема кодирования инструкций, согласно которой четыре целочисленные операции (addq, subq, andq и xorq) имеют одинаковое значение **icode**. Мы можем обработать их, применив одну и ту же последовательность шагов, за исключением вычислений в АЛУ, которые должны производиться в соответствии с конкретной операцией, закодированной в **ifun**.

Обработка инструкций, выполняющих операции с целыми числами, следует общей схеме, представленной в начале раздела. На этапе выборки не требуется извлекать слово константы, поэтому **valP** вычисляется как PC + 2. На этапе декодирования определяются оба операнда. Они передаются в АЛУ на этапе выполнения вместе со спецификатором функции **ifun**, соответственно, **valE** превращается в результат инструкции. Вычисления в АЛУ показаны в виде выражения **valB OP valA**, где **OP** определяет операцию, обозначенную спецификатором **ifun**. Обратите внимание на порядок следования двух аргументов: этот порядок соответствует соглашениям Y86-64 (и x86-64). Например, предполагается, что инструкция `subq %rax, %rdx` вычисляет значение **R[%rdx] – R[%rax]**. На этапе обращения к памяти в этих инструкциях ничего не происходит, но на этапе обратной записи **valE** записывается в регистр **rB**, а в PC записывается значение **valP**.

Трассировка выполнения инструкции subq

В качестве примера проследим обработку инструкции `subq` в строке 3 объектного кода, показанного в листинге 4.4. Две предшествующие инструкции инициализировали регистры `%rdx` и `%rbx` значениями 9 и 21 соответственно. Кроме того, из листинга видно, что эта инструкция находится в памяти по адресу `0x014` и состоит из двух байт со значениями `0x61` и `0x23`. Этапы выполняются, как показано в следующей таблице, где приводятся обобщенный порядок обработки инструкции **OPq** (табл. 4.3) слева и вычисления, выполняемые этой конкретной инструкцией, справа.

Этап	В общем случае OPq rA, rB	Конкретно для subq %rdx, %rbx
Выборка	icode : ifun ← M <sub>1</sub> [PC] rA :rB ← M <sub>1</sub> [PC + 1]  valP ← PC + 2	icode : ifun ← M <sub>1</sub> [0x014]= 6:1 rA :rB ← M <sub>1</sub> [0x015]= 2:3  valP ← 0x014 + 2 = 0x016
Декодирование	valA ← R[rA] valB ← R[rB]	valA ← R[%rdx]= 9 valB ← R[%rbx]= 21
Выполнение	valE ← valB OP valA Установка флагов условий	valE ← 21 – 9 = 12 ZF ← 0, SF ← 0, OF ← 0
Обращение к памяти		
Обратная запись	R[rB] ← valE	R[%rbx] ← valE = 12
Изменение PC	PC ← valP	PC ← valP = 0x016

Как показывает эта трассировка, мы достигли желаемого эффекта: записали в `%rbx` число 12, установили все три флага условий в ноль и увеличили PC на 2.

Упражнение 4.13 (решение в конце главы)

Заполните правый столбец в следующей таблице, описав порядок обработки инструкции `imovq` в строке 4 в листинге 4.4:

Этап	В общем случае <code>irmovq V, rB</code>	Конкретно для <code>irmovq \$128, %rsp</code>
Выборка	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$	
Декодирование		
Выполнение	$\text{valE} \leftarrow 0 + \text{valC}$	
Обращение к памяти		
Обратная запись	$R[\text{rB}] \leftarrow \text{valE}$	
Изменение PC	$\text{PC} \leftarrow \text{valP}$	

Как выполнение этой инструкции изменяет регистры и PC?

Обработка инструкции `rrmovq` начинается так же, как арифметических инструкций. Однако выборку второго регистра-операнда производить не требуется. Вместо этого на второй вход АЛУ подается 0 и складывается с первым, что дает **valE = valA**, после чего это значение записывается в регистр. Аналогично обрабатывается `irmovq`, за исключением того, что на первый вход АЛУ подается непосредственное значение `valC`. Кроме этого, для `irmovq` счетчик инструкций необходимо увеличить на 10 из-за ее большого размера. Ни одна из этих инструкций не меняет флаги условий.

В табл. 4.4 перечислены операции с памятью, выполняемые инструкциями `rrmovq` и `rrmovq`. Основная последовательность та же, что и раньше, но здесь для прибавления `valC` к `valB` используется АЛУ (арифметико-логическое устройство), с помощью которого вычисляется эффективный адрес (сумма смещения и значения базового регистра). На этапе обращения к памяти происходит либо запись значения регистра `valA`, либо чтение значения `valM`.

В табл. 4.5 показаны этапы обработки инструкций `pushq` и `popq`. Это наиболее сложные инструкции для реализации в архитектуре Y86-64, потому что они включают и обращение к памяти, и изменение указателя стека. Несмотря на то что эти две инструкции имеют сходные последовательности обработки, у них есть важные различия.

**Таблица 4.4.** Вычисления, выполняемые последовательной реализацией Y86-64 (SEQ) при обработке инструкций `rrmovq` и `rrmovq`. Эти инструкции читают или записывают данные в память

Этап	<code>rrmovq rA, D(rB)</code>	<code>rrmovq D(rB), rA</code>
Выборка	$\text{icode} : \text{ifun} \leftarrow M1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$
Декодирование	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	$\text{valB} \leftarrow R[\text{rB}]$
Выполнение	$\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valE} \leftarrow \text{valB} + \text{valC}$
Обращение к памяти	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valE}]$
Обратная запись		$R[\text{rA}] \leftarrow \text{valM}$
Изменение PC	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

**Таблица 4.5.** Вычисления, выполняемые последовательной реализацией Y86-64 (SEQ) при обработке инструкций `pushq` и `popq`. Эти инструкции вталкивают и выталкивают значение со стека

Этап	<code>pushq rA</code>	<code>popq rA</code>
Выборка	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$
Декодирование	$\text{valP} \leftarrow \text{PC} + 2$ $\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\%rsp]$	$\text{valP} \leftarrow \text{PC} + 2$ $\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$
Выполнение	$\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valE} \leftarrow \text{valB} + 8$
Обращение к памяти	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valA}]$
Обратная запись	$R[\%rsp] \leftarrow \text{valE}$	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$
Изменение PC	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

#### Трассировка выполнения инструкции `rmovq`

Давайте проследим обработку инструкции `rmovq` в строке 5 объектного кода, показанного в листинге 4.4. Предшествующая ей инструкция инициализировала регистр `%rsp` значением 128, при этом `%rbx` по-прежнему хранит число 12, вычисленное инструкцией `subq` (строка 3). Также видно, что данная инструкция начинается с адреса `0x020` и занимает 10 байт. Первые два байта имеют значения `0x40` и `0x43`, а оставшиеся четыре представляют число `0x0000000000000064` (десятичное 100) в форме записи с обратным порядком байтов. Этапы выполняются, как показано в следующей таблице:

Этап	В общем случае <code>rmovq rA, D(rB)</code>	Конкретно для <code>rmovq %rsp, 100(%rbx)</code>
Выборка	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$	$\text{icode} : \text{ifun} \leftarrow M_1[0x020] = 4:0$ $\text{rA} : \text{rB} \leftarrow M_1[0x021] = 4:3$ $\text{valC} \leftarrow M_8[0x022] = 100$ $\text{valP} \leftarrow 0x020 + 10 = 0x02a$
Декодирование	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	$\text{valA} \leftarrow R[\%rsp] = 128$ $\text{valB} \leftarrow R[\%rbx] = 12$
Выполнение	$\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valE} \leftarrow 12 + 100 = 112$
Обращение к памяти	$M_8[\text{valE}] \leftarrow \text{valA}$	$M_8[112] \leftarrow 128$
Обратная запись		
Изменение PC	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow 0x02a$

Как показывает эта трассировка, инструкция записала 128 в ячейку памяти с адресом 112 и увеличила PC на 10.

Обработка инструкции `pushq` начинается практически так же, как и предыдущих инструкций, однако на этапе декодирования в качестве идентификатора второго регистра-операнда используется `%rsp`, что вызывает присваивание указателю стека значения **valB**. На этапе выполнения для уменьшения значения указателя стека на 8 используется АЛУ. Это уменьшенное значение используется для записи адреса ячеек-



ки памяти и сохраняется в `%rsp` на этапе обратной записи. Пользуясь значением `valE` в роли адреса для операции записи, мы выполняем соглашение, принятое для Y86-64 (и x86-64), согласно которому инструкция `pushq` должна уменьшать значение указателя стека до записи в память, несмотря на то что фактического изменения указателя стека не происходит до завершения операции с памятью.

**Трассировка выполнения инструкции `pushq`**

Давайте проследим обработку инструкции `pushq` в строке 6 в листинге 4.4. В этой точке в регистре `%rdx` хранится число 9, а в регистре `%rsp` – число 128. Также видно, что данная инструкция начинается с адреса `0x02a` и занимает два байта со значениями `0xa0` и `0x2f`. Этапы выполняются, как показано в следующей таблице:

Этап	В общем случае <code>pushq rA</code>	Конкретно для <code>pushq %rdx</code>
Выборка	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$	$icode : ifun \leftarrow M_1[0x02a] = a:0$ $rA : rB \leftarrow M_1[0x02b] = 2:f$
Декодирование	$valP \leftarrow PC + 2$ $valA \leftarrow R[rA]$ $valB \leftarrow R[\%rsp]$	$valP \leftarrow 0x02a + 2 = 0x02c$ $valA \leftarrow R[\%rdx] = 9$ $valB \leftarrow R[\%rsp] = 128$
Выполнение	$valE \leftarrow valB + (-8)$	$valE \leftarrow 128 + (-8) = 120$
Обращение к памяти	$M_8[valE] \leftarrow valA$	$M_8[120] \leftarrow 9$
Обратная запись	$R[\%rsp] \leftarrow valE$	$R[\%rsp] \leftarrow 120$
Изменение PC	$PC \leftarrow valP$	$PC \leftarrow 0x02c$

Как показывает эта трассировка, инструкция записала 120 в `%rsp` и число 9 в ячейку памяти с адресом 120 и увеличила значение PC на 2.

**Упражнение 4.14 (решение в конце главы)**

Заполните правый столбец в следующей таблице, описав порядок обработки инструкции `rorq` в строке 7 в листинге 4.4:

Этап	В общем случае <code>irmovq V, rB</code>	Конкретно для <code>irmovq \$128, %rsp</code>
Выборка	$icode : ifun \leftarrow M1[PC]$ $rA : rB \leftarrow M1[PC + 1]$	
Декодирование	$valP \leftarrow PC + 2$ $valA \leftarrow R[\%rsp]$ $valB \leftarrow R[\%rsp]$	
Выполнение	$valE \leftarrow 0 + valC$	
Обращение к памяти	$valM \leftarrow M8[valA]$	
Обратная запись	$R[\%rsp] \leftarrow valE$ $R[rA] \leftarrow valM$	
Изменение PC	$PC \leftarrow valP$	

Как выполнение этой инструкции повлияет на регистры и PC?

Обработка инструкции `popq` начинается так же, как `pushq`, за исключением того, что на этапе декодирования создаются две копии указателя стека. Это абсолютно излишне, но, как будет показано далее, при наличии двух значений указателя стека, **valA** и **valB**, последующий процесс обработки становится более похожим на процесс обработки других инструкций, что обеспечивает общее единообразие архитектуры. На этапе выполнения для увеличения указателя стека на 8 используется АЛУ, при этом неувеличенное значение используется как адрес для выполнения операции с памятью. На этапе обратной записи обновляются и регистр указателя стека, и регистр **ra**: в первый записывается увеличенное значение указателя стека, а во второй – значение, прочитанное из памяти. Использование неувеличенного указателя стека в роли адреса для чтения значения из памяти обеспечивает выполнение соглашения, принятого для Y86-64 (и x86-64), согласно которому инструкция `popq` сначала должна прочитать значение из памяти и только потом увеличить указатель стека.

#### Упражнение 4.15 (решение в конце главы)

Какой эффект окажет инструкция `pushq %rsp` в соответствии с этапами, перечисленными в табл. 4.5? Соответствует ли этот эффект желаемому поведению для Y86-64, как определено в упражнении 4.7?

#### Упражнение 4.16 (решение в конце главы)

Предположим, что на этапе обратной записи при выполнении инструкции `popq` выполняется запись в два регистра в порядке, как указано в табл. 4.5. Какой эффект произведет выполнение инструкции `popq %rsp`? Соответствует ли этот эффект желаемому поведению для Y86-64, как определено в упражнении 4.8?

В табл. 4.6 показан порядок обработки трех инструкций передачи управления: разных инструкций перехода, `call` и `ret`. Как видите, обработку этих инструкций можно организовать, следуя общей схеме.

**Таблица 4.6.** Вычисления, выполняемые последовательной реализацией Y86-64 (SEQ) при обработке инструкций `jXX`, `call` и `ret`. Эти инструкции реализуют передачу управления

Этап	<code>jXX Dest</code>	<code>call Dest</code>	<code>ret</code>
Выборка	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$  $\text{valC} \leftarrow M_8[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 9$	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$  $\text{valC} \leftarrow M_8[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 9$	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$  $\text{valP} \leftarrow \text{PC} + 1$
Декодирование		$\text{valB} \leftarrow R[\%rsp]$	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$
Выполнение	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	$\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valE} \leftarrow \text{valB} + 8$
Обращение к памяти		$M_8[\text{valE}] \leftarrow \text{valP}$	$\text{valM} \leftarrow M_8[\text{valA}]$
Обратная запись		$R[\%rsp] \leftarrow \text{valE}$	$R[\%rsp] \leftarrow \text{valE}$
Изменение PC	$\text{PC} \leftarrow \text{Cnd} ? \text{valC} : \text{valP}$	$\text{PC} \leftarrow \text{valC}$	$\text{PC} \leftarrow \text{valM}$

Так же, как в случае с целочисленными операциями, все переходы различаются только способом определения необходимости выполнить переход. Обработка инструкции

перехода начинается с этапов выборки и декодирования, так же как обработка предыдущих инструкций, но не требует наличия байта-спецификатора регистра. На этапе выполнения проверяются флаги и условие перехода, чтобы определить, должен осуществляться переход или нет, с выдачей 1-битного сигнала **Cnd**. На этапе изменения счетчика инструкций РС сигнал проверяется и РС устанавливается равным **valC** (адрес перехода), если сигнал равен единице, и **valP** (адрес следующей инструкции), если флаг равен нулю. Запись  $x ? a : b$  подобна тернарному условному выражению в языке C: оно дает  $a$ , когда  $x$  не равно нулю, и  $b$ , когда  $x$  равно нулю.

**Трассировка выполнения инструкции je**

Давайте проследим обработку инструкции `je` в строке 8 в листинге 4.4. В этой точке все флаги условий имеют нулевые значения, получившиеся в результате выполнения инструкции `subq` (строка 3), поэтому переход не выполняется. Инструкция начинается с адреса `0x02e` и занимает девять байт, первый из которых имеет значение `0x73`, а остальные восемь – версия адреса перехода `0x0000000000000040` в формате с обратным порядком следования байтов. Этапы выполняются, как показано в следующей таблице:

Этап	В общем случае <b>jXX Dest</b>	Конкретно для <b>je 0x040</b>
Выборка	$icode : ifun \leftarrow M_1[PC]$  $valC \leftarrow M_8[PC + 1]$ $valP \leftarrow PC + 9$	$icode : ifun \leftarrow M_1[0x02e] = 7:3$  $valC \leftarrow M_8[0x02f] = 0x040$ $valP \leftarrow 0x02e + 9 = 0x037$
Декодирование		
Выполнение	$Cnd \leftarrow Cond(CC, ifun)$	$Cnd \leftarrow Cond(0, 0, 0, 3) = 0$
Обращение к памяти		
Обратная запись		
Изменение PC	$PC \leftarrow Cnd ? valC : valP$	$PC \leftarrow 0 ? 0x040 : 0x037 = 0x037$

Как показывает эта трассировка, инструкция просто увеличила PC на 9.

**Упражнение 4.17 (решение в конце главы)**

По кодам инструкций (рис. 4.2 и 4.3) видно, что инструкция `ttmovq` является безусловной версией более общего класса инструкций, включающего инструкции условного перемещения. Покажите, как бы вы изменили этапы приведенной ниже инструкции `ttmovq`, чтобы обеспечить единообразную обработку всех шести инструкций условного перемещения. Возможно, вам поможет описание реализации условного поведения в инструкциях **jXX** (табл. 4.6).

Этап	<b>cmovXX rA, rB</b>
Выборка	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 2$
Декодирование	$valA \leftarrow R[rA]$
Выполнение	$valE \leftarrow 0 + valA$
Обращение к памяти	
Обратная запись	$R[rB] \leftarrow valE$
Изменение PC	$PC \leftarrow valP$

Инструкции `call` и `ret` имеют существенное сходство с инструкциями `pushq` и `popq`, отличаясь лишь тем, что вталкивают в стек и выталкивают со стека значение счетчика инструкций. Инструкция `call` вталкивает значение `valP` – адрес инструкции, следующей за инструкцией `call`. На этапе изменения PC инструкция `call` записывает в счетчик инструкций `valC` – адрес начала вызываемой процедуры, а инструкция `ret` – значение `valM`, вытолкнутое со стека.

**Упражнение 4.18 (решение в конце главы)**

Заполните правый столбец в следующей таблице, описав порядок обработки инструкции `call` в строке 9 в листинге 4.4:

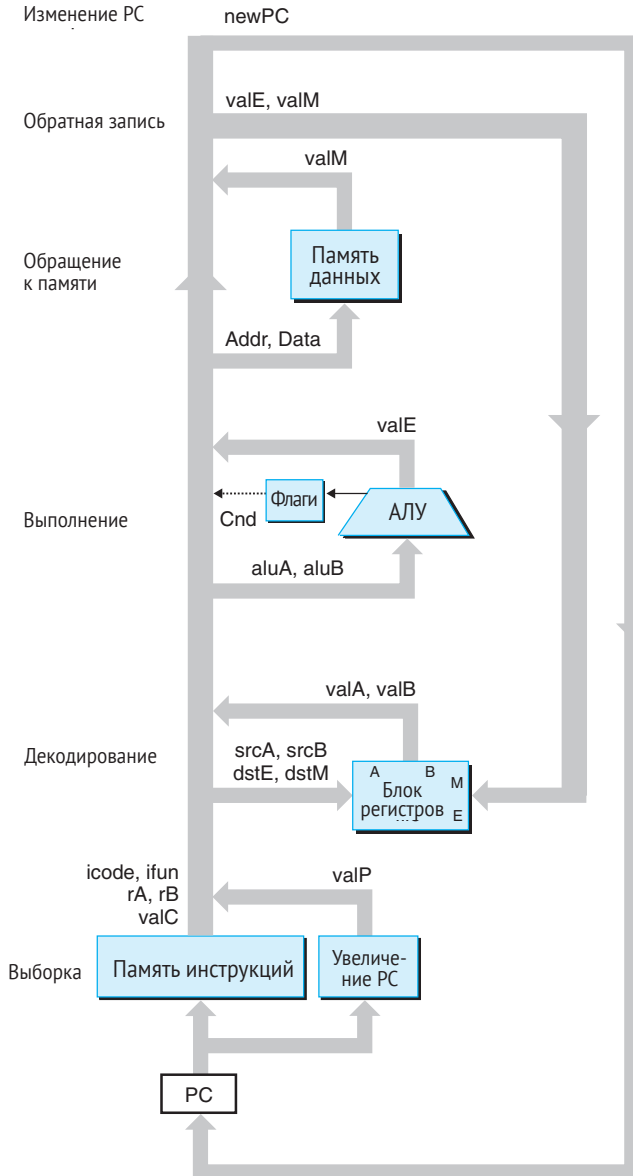
Этап	В общем случае call Dest	Конкретно для call 0x041
Выборка	$icode : ifun \leftarrow M_1[PC]$ $valC \leftarrow M_8[PC + 1]$  $valP \leftarrow PC + 9$	
Декодирование	$valB \leftarrow R[\%rsp]$	
Выполнение	$valE \leftarrow valB + (-8)$	
Обращение к памяти	$M_8[valE] \leftarrow valP$	
Обратная запись	$R[\%rsp] \leftarrow valE$	
Изменение PC	$PC \leftarrow valC$	

Как выполнение этой инструкции повлияет на регистры, PC и память?

Итак, мы создали универсальную схему обработки разных типов инструкций Y86-64. Несмотря на то что все инструкции существенно разнятся своим поведением, мы можем организовать обработку в шесть этапов. Теперь наша задача состоит в том, чтобы спроектировать аппаратную часть, объединяющую и реализующую эти этапы.

**4.3.2. Аппаратная реализация последовательной архитектуры SEQ**

Вычисления, необходимые для реализации всех инструкций Y86-64, можно организовать в виде последовательности из шести основных этапов: выборка, декодирование, выполнение, обращение к памяти, обратная запись и изменение PC. На рис. 4.16 показана абстрактная структура аппаратной архитектуры, выполняющей эти вычисления. Счетчик инструкций сохраняется в регистре, изображенном в нижнем левом углу (с меткой «PC»). Затем информация передается по проводникам (группы которых обозначены широкой полосой серого цвета) сначала вверх, а потом по часовой стрелке. Обработка осуществляется *аппаратными модулями*, обслуживающими разные этапы. Пути обратной связи, ведущие вправо и вниз, несут обновленные значения для записи в регистры и счетчик инструкций. Все операции выполняются аппаратными модулями за один такт, как обсуждалось в разделе 4.3.3. На этой диаграмме отсутствуют некоторые мелкие блоки комбинаторной логики, а также вся управляющая логика, необходимые для функционирования различных аппаратных модулей и передачи соответствующих значений. Мы добавим их позже. Такой способ изображения работы процессоров в виде потока, направленного снизу вверх, не совсем обычен. Причину подобного решения мы объясним позже, когда приступим к проектированию конвейерных процессоров.



**Рис. 4.16.** Абстрактное представление последовательной реализации SEQ.

Информация, обрабатываемая в ходе выполнения инструкции, передается по часовой стрелке, начиная с этапа выборки инструкции, находящегося в левом нижнем углу на схеме

Аппаратные модули связаны с разными этапами обработки:

#### Выборка

Байты инструкции извлекаются с использованием регистра счетчика инструкций (PC), который хранит адрес следующей выполняемой инструкции. На этом же этапе вычисляется **valP** – новое значение счетчика инструкций.

*Декодирование*

Блок регистров имеет два порта для чтения – **A** и **B**, через которые одновременно читаются значения регистров **valA** и **valB**.

*Выполнение*

На этапе выполнения для решения разных задач, в зависимости от типа инструкции, используется арифметико-логическое устройство (АЛУ). Для целочисленных операций выполняется заданная операция. Для других инструкций АЛУ служит сумматором, вычисляющим приращение (положительное или отрицательное) для указателя стека, для вычисления эффективного адреса или просто для передачи данных с входа на выход путем добавления 0.

Регистр флагов содержит три бита, определяющих разные условия. Новые значения флагов вычисляются в АЛУ. При выполнении инструкции условного перемещения решение об изменении регистра-получателя принимается на основе флагов и типа инструкции. Аналогично выполняются инструкции условного перехода – сигнал ветвления **Cnd** вычисляется на основе флагов и типа инструкции.

*Обращение к памяти*

Инструкции, выполняющие операции с памятью, производят чтение или запись слова данных в память. Разделы памяти для инструкций и данных находятся в общем адресном пространстве, но используются с разными целями.

*Обратная запись*

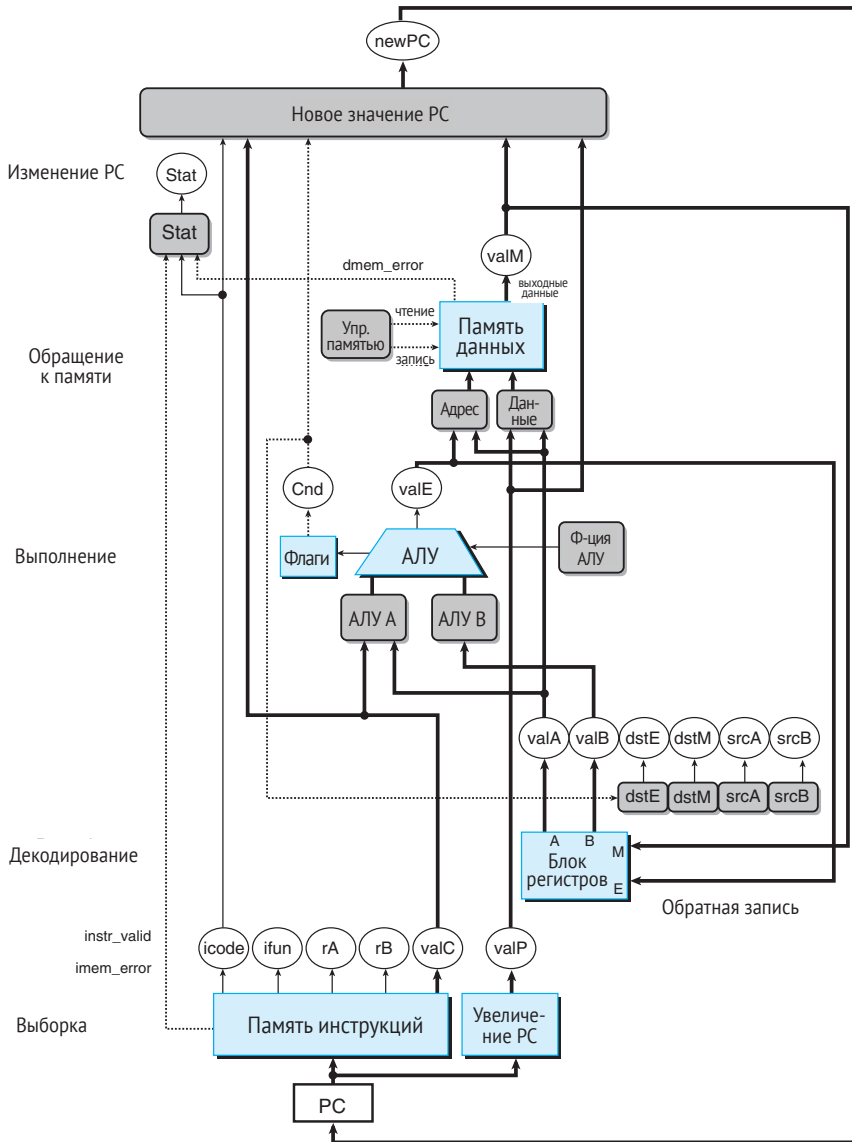
Блок регистров имеет два порта для записи. Порт **E** используется для записи значений, вычисляемых АЛУ, а порт **M** – для записи значений, извлекаемых из памяти для данных.

*Изменение PC*

Новое значение счетчика инструкций выбирается из **valP** (адрес следующей инструкции), **valC** (адрес перехода в инструкциях вызова процедуры или перехода) или **valM** (адрес возврата из процедуры, прочитанный из памяти).

На рис. 4.17 представлена более подробная схема последовательной реализации (хотя и здесь нет некоторых подробностей, которые мы будем обсуждать при изучении отдельных этапов). Здесь показан тот же набор аппаратных модулей, однако теперь связи между ними показаны явно. В этой диаграмме, как и в последующих, используются следующие соглашения:

- *синхронизированные регистры изображаются как белые прямоугольники.* Счетчик инструкций **PC** в последовательной реализации может быть только синхронизированным регистром;
- *аппаратные модули изображаются светло-синими прямоугольниками.* К ним относятся разные типы памяти, АЛУ и т. д. Во всех наших реализациях процессоров будет использоваться лишь самый необходимый набор модулей. Мы будем считать эти модули «черными ящиками» и не будем углубляться в обсуждение их устройства;
- *блоки управляющей логики изображаются в серых прямоугольниках с закругленными углами.* Эти блоки делают выбор из множества источников сигналов или вычисляют некоторую булеву функцию. Мы исследуем эти блоки очень подробно, включая разработку описаний HCL;
- *названия связей отображаются в разрывах линий.* Это всего лишь метки – они не являются аппаратными устройствами;



**Рис. 4.17.** Структура аппаратных модулей последовательной реализации SEQ. Некоторые управляющие сигналы, а также регистры и полные шины данных не показаны

- каналы передачи слов данных изображены линиями средней толщины. Каждая из этих линий фактически является шиной из 64 проводников, служащих для передачи слова из одной части аппаратной системы в другую;
- каналы передачи отдельных байтов изображены тонкими линиями. Каждая из этих линий фактически содержит 4 или 8 проводников, в зависимости от типов значений, передаваемых по этим каналам;
- каналы передачи отдельных битов изображены пунктирными линиями. Они представляют передачу управляющих значений между блоками и модулями.

Все вычисления, показанные в табл. 4.3–4.6, обладают одним общим свойством: каждая строка представляет либо вычисление некоторого значения, такого как, например, **valP**, либо активизацию некоторого аппаратного модуля, например памяти. Эти вычисления и действия перечислены во втором столбце в табл. 4.7. Помимо уже описанных сигналов, в этот список вошли четыре сигнала идентификации регистров: **srcA** – источник **valA**, **srcB** – источник **valB**, **dstE** – регистр, в который записывается **valE**, и **dstM** – регистр, в который записывается **valM**.

**Таблица 4.7.** Идентификация разных этапов в последовательной реализации SEQ. Второй столбец определяет значение или операцию, вычисляемое или выполняемую на том или ином этапе. В качестве примеров приводятся вычисления для инструкций **OPq** и **mrmovq**

Этап	Вычисления	OPq rA, rB	mrmovq D(rB), rA
Выборка	icode, ifun rA, rB valC valP	icode : ifun $\leftarrow M_1[PC]$ rA :rB $\leftarrow M_1[PC + 1]$  valP $\leftarrow PC + 2$	icode : ifun $\leftarrow M_1[PC]$ rA :rB $\leftarrow M_1[PC + 1]$ valC $\leftarrow M_8[PC + 2]$ valP $\leftarrow PC + 10$
Декодирование	valA, srcA valB, srcB	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	valB $\leftarrow R[rB]$
Выполнение	valE Флаги условий	valE $\leftarrow$ valB OP valA Установка флагов	valE $\leftarrow$ valB + valC
Обращение к памяти	Чтение/запись		valM $\leftarrow M_8[valE]$
Обратная запись	E port, dstE M port, dstM	R[rB] $\leftarrow$ valE	R[rA] $\leftarrow$ valM
Изменение PC	PC	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Для иллюстрации в двух правых столбцах в табл. 4.7 показаны вычисления для инструкций **OPq** и **mrmovq**. Для организации вычислений в аппаратных модулях необходимо реализовать управляющую логику, которая будет переносить данные между модулями, чтобы те могли произвести вычисления, предполагаемые той или иной инструкцией. Такова цель блоков управляющей логики, показанных в виде серых прямоугольников с закругленными углами на рис. 4.17. Теперь наша задача состоит в том, чтобы исследовать еще раз отдельные этапы и подробно описать эти блоки.

4.3.3. Синхронизация в последовательной реализации SEQ

В описании табл. 4.3–4.6 отмечалось, что их следует читать так, словно они написаны на языке программирования и все операции присваивания выполняются по порядку, сверху вниз. Однако совокупность аппаратных модулей на рис. 4.17 функционирует совершенно иначе, используя один такт для выполнения всей комбинаторной логики инструкции. Давайте посмотрим, как аппаратные модули могут реализовать поведение, описанное в этих таблицах.

Наша последовательная реализация состоит из комбинаторной логики и двух видов памяти: синхронизированных регистров (счетчика инструкций и регистра флагов) и оперативной памяти (блок регистров, память для инструкций и память для данных). Комбинаторная логика не требует, чтобы действия или управление осуществлялись последовательно: значения передаются через сеть логических вентилей при каждой смене входных данных. Как уже отмечалось, чтение из оперативной памяти подобно комбинаторной логике, когда слово на выходе генерируется при подаче адреса на вход. Это вполне разумное допущение для небольших устройств памяти (таких как блок регистров), но его можно имитировать для более крупных цепей, используя специальные



цепи синхронизации. Поскольку память для инструкций используется только для чтения, этот модуль можно рассматривать как комбинаторную логику.

У нас имеется всего четыре аппаратных модуля, требующих явного управления последовательностью их использования: счетчик инструкций, регистр флагов, память для данных и блок регистров. Управление ими осуществляется через один синхронизирующий сигнал, запускающий процесс загрузки новых значений в регистры и запись значений в оперативную память. В каждом цикле синхронизации в счетчик инструкций загружается новый адрес инструкции. Регистр флагов загружается лишь при выполнении целочисленной операции. Запись в память для данных происходит только при выполнении инструкций `rmovq`, `pushq` и `call`. Два порта записи блока регистров позволяют изменять два регистра в каждом цикле, но мы можем подать в любой порт особый идентификатор регистра `0xF`, чтобы показать, что через данный порт запись не должна производиться.

Такая синхронизация регистров и памяти – все, что требуется для управления последовательностью действий процессора. Наше оборудование дает тот же эффект, что и последовательное выполнение присваиваний, показанных в табл. 4.3–4.6, несмотря на то что все изменения состояний происходят фактически одновременно, и только с положительным перепадом сигнала синхронизации, обозначающего начало следующего цикла. Подобная эквивалентность поддерживается благодаря природе системы команд Y86-64, а также потому, что мы организовали вычисления в соответствии со следующим принципом:

**ПРИНЦИП:** никакого обратного чтения.

Процессору не требуется выполнять обратное чтение состояния, обновленного инструкцией, чтобы завершить ее обработку.

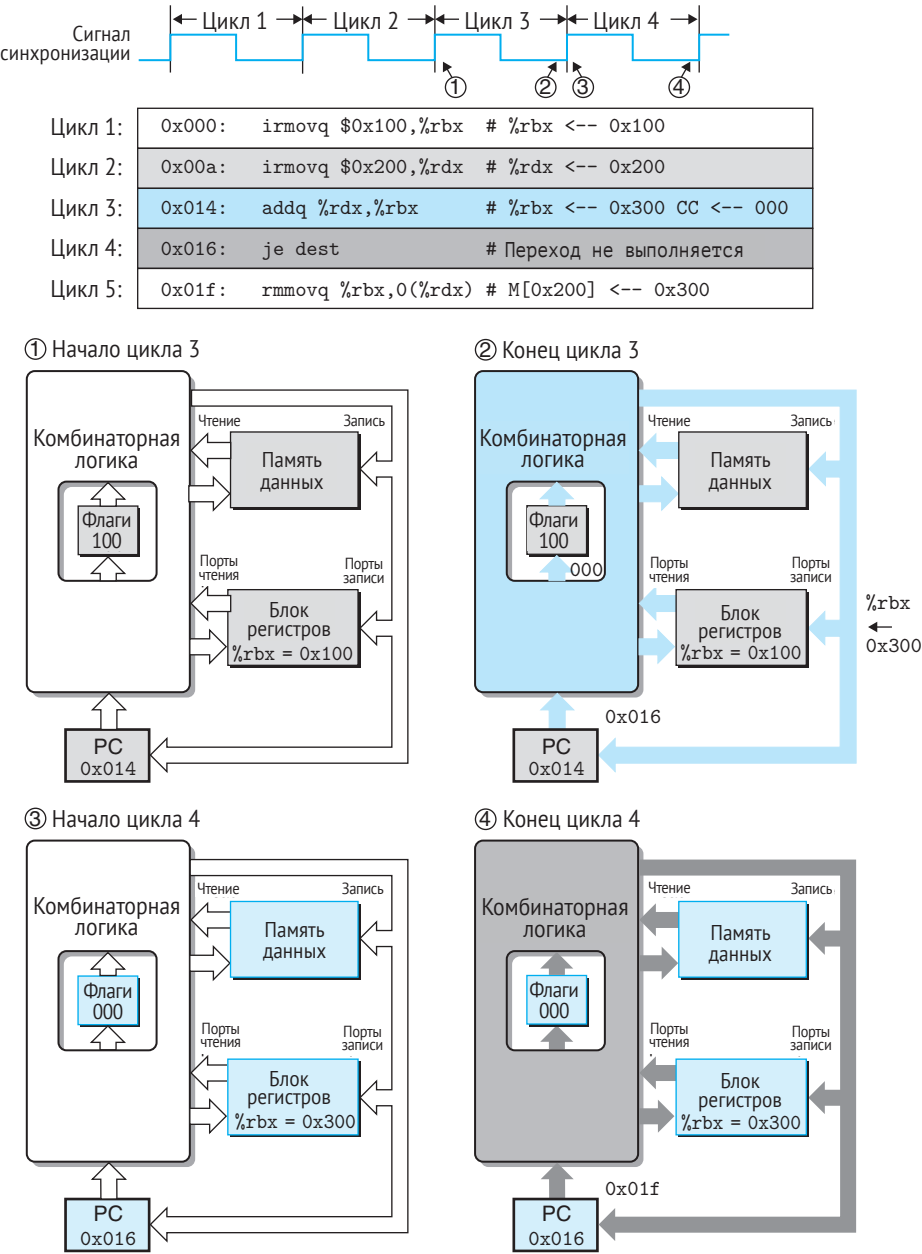
Соблюдение данного принципа является основополагающим условием для успеха реализации. В качестве иллюстрации допустим, что реализация инструкции `pushq` сначала уменьшает `%rsp` на 8 и затем использует обновленное значение `%rsp` как адрес в операции записи. Такой подход нарушает принцип, сформулированный ранее. Для выполнения операции с памятью потребуются прочесть обновленное значение указателя стека из блока регистров. Вместо этого наша реализация (табл. 4.5) генерирует уменьшенное значение указателя стека как сигнал **valE**, после чего использует этот сигнал в роли данных для записи в регистр и в роли адреса для записи в память. В результате записи в регистр и в память могут осуществляться одновременно по положительному перепаду сигнала синхронизации в начале очередного цикла.

Вот еще одна иллюстрация данного принципа: некоторые инструкции (целочисленные операции) устанавливают флаги, а другие (инструкции условного перемещения или перехода) читают их, однако ни одна инструкция не должна явно записывать или читать флаги. Даже если флаги не установлены до начала очередного цикла, они обновятся до того, как понадобятся какой-либо инструкции.

На рис. 4.18 показано, как последовательная аппаратная реализация обрабатывает инструкции в строках 3 и 4 в следующем листинге, где слева показаны адреса инструкций:

```

1  0x000:  irmovq $0x100,%rbx    # %rbx <-- 0x100
2  0x00a:  irmovq $0x200,%rdx    # %rdx <-- 0x200
3  0x014:  addq %rdx,%rbx        # %rbx <-- 0x300 CC <-- 000
4  0x016:  je dest               # переход не выполняется
5  0x01f:  rmmovq %rbx,0(%rdx)    # M[0x200] <-- 0x300
6  0x029:  dest: halt
```



**Рис. 4.18.** Трассировка выполнения двух циклов обработки инструкций в последовательной реализации SEQ. Каждый цикл начинается с обновления элементов состояния (счетчик инструкций, регистр флагов (CC), блок регистров и память для данных) в соответствии с результатами выполнения предыдущей инструкции. Сигналы распространяются через комбинаторную логику, создавая новые значения для элементов состояния. Эти значения загружаются в элементы состояния с началом следующего цикла

Каждая из диаграмм рис. 4.18, подписанных цифрами от 1 до 4, содержит четыре элемента состояния, а также комбинаторную логику и связи между этими элементами. Комбинаторная логика окружает регистр флагов (**CC**) как обертка, потому что некоторая ее часть (например, АЛУ) генерирует входные данные для регистра флагов, а другие части (например, вычисление адреса перехода и логика выбора РС) используют флаги как входные данные. Соединения для чтения и записи в блоке регистров и памяти показаны раздельно, потому что операции чтения пересекают эти модули так, будто они являются комбинаторной логикой, а операции записи управляются тактовым генератором синхронизирующих импульсов.

Цветом на рис. 4.18 показано, как сигналы соотносятся с разными выполняемыми инструкциями. Предполагается, что процесс обработки начинается с флагов (**CC**), перечисленных в порядке ZF, SF и OF, со значениями 100. В начале цикла синхронизации 3 (точка 1 на рис. 4.18) элементы состояния получают значение, обновленное второй инструкцией `irmovq` (строка 2 в листинге). Комбинаторная логика обозначена белым цветом, показывающим, что она еще не успела отреагировать на изменение состояния. Цикл синхронизации начинается с адреса `0x014`, загруженного в счетчик инструкций. Положительный перепад тактового импульса активизирует обработку инструкции `addq` (строка 3 в листинге). Значения пересекают комбинаторную логику, включая чтение данных из оперативной памяти. К концу цикла (точка 2 на рис. 4.18) комбинаторная логика создает новые значения (000) для флагов условий, обновляет программный регистр `%rbx` и записывает новое значение (`0x016`) в счетчик инструкций. В этой точке комбинаторная логика обновляется в соответствии с инструкцией `addq` (показана синим), однако элементы состояния все еще хранят прежние значения, получившиеся при выполнении предшествовавшей инструкции `irmovq` (показана светло-серым).

С положительным перепадом сигнала синхронизации, запускающего цикл 4 (точка 3 на рис. 4.18), происходит обновление счетчика инструкций, блока регистров и регистра флагов (**CC**), поэтому они окрашены в синий цвет, однако комбинаторная логика еще не отреагировала на эти изменения, поэтому она изображена белым цветом. В этом цикле извлекается и выполняется инструкция `je` (строка 4 в листинге), показанная темно-серым цветом. Поскольку флаг ZF равен нулю, переход не происходит. К концу цикла (точка 4 на рис. 4.18) для счетчика инструкций генерируется новое значение `0x01f`. Комбинаторная логика обновляется в соответствии с результатом выполнения инструкции `je` (обозначена темно-серым цветом), однако элементы состояния по-прежнему хранят значения, установленные инструкцией `addq`, пока не начнется следующий цикл.

Как показывает этот пример, использование тактового генератора для управления обновлением элементов состояния и передачи значений через комбинаторную логику вполне достаточно для управления вычислениями, производимыми при обработке каждой инструкции, в данной реализации. С каждым положительным перепадом сигнала синхронизации процессор начинает выполнять новую инструкцию.

#### 4.3.4. Реализация этапов в последовательной версии SEQ

В этом разделе мы опишем на языке HCL работу блоков управляющей логики, необходимых для последовательной реализации. Полная последовательность реализации на HCL приводится в приложении в интернете «ARCH:HCL» (раздел 4.6). Далее мы рассмотрим примеры некоторых блоков; другие же вам будет предложено описать самим в виде упражнений. Обязательно выполните эти упражнения, чтобы закрепить понимание связей между этими блоками и вычислительными требованиями различных инструкций.

Здесь мы не будем рассматривать описание на HCL, определяющее различные целочисленные и булевы сигналы, которые можно использовать как аргументы в операциях HCL. К ним относятся имена различных аппаратных сигналов, а также постоянные зна-

чения кодов разных инструкций, имена регистров, операций АЛУ и кодов состояния. Показаны только те из них, которые должны явно указываться в логике управления. Используемые константы перечислены в табл. 4.8. По общепринятым соглашениям константам присвоены имена из букв верхнего регистра.

**Таблица 4.8.** Константы, используемые в описаниях HCL. К константам относятся постоянные коды инструкций и функций, идентификаторы регистров, операции АЛУ и коды состояния

Имя	Значение (шестнадцатеричное)	Описание
HALT	0	Код инструкции halt
INOP	1	Код инструкции nop
IRRMOVQ	2	Код инструкции irmovq
IIRMOVQ	3	Код инструкции irmovq
IRMMOVQ	4	Код инструкции rmmovq
IMRMOVQ	5	Код инструкции rmmovq
IOPQ	6	Код инструкции, выполняющей целочисленную операцию
IJXX	7	Код инструкции перехода
ICALL	8	Код инструкции call
IRET	9	Код инструкции ret
IPUSHQ	A	Код инструкции pushq
IPOPQ	B	Код инструкции popq
FNONE	0	Код функции по умолчанию
RRSP	4	Идентификатор регистра %rsp
RNONE	A	Признак отсутствия доступа к блоку регистров
ALUADD	0	Функция для операции сложения
SAOK	1	Код состояния нормального выполнения операции
SADR	2	Код состояния для исключения при попытке обращения к недопустимой памяти
SINS	3	Код состояния для исключения при попытке выполнить недопустимую инструкцию
SHLT	4	Код состояния после выполнения инструкции halt

В дополнение к инструкциям, перечисленным в табл. 4.3–4.6, мы добавили обработку инструкций nop и halt. Инструкция nop просто пересекает все этапы без всякой обработки, за исключением увеличения PC на 1. Инструкция halt устанавливает код HLT состояния процессора, фактически останавливая его работу.

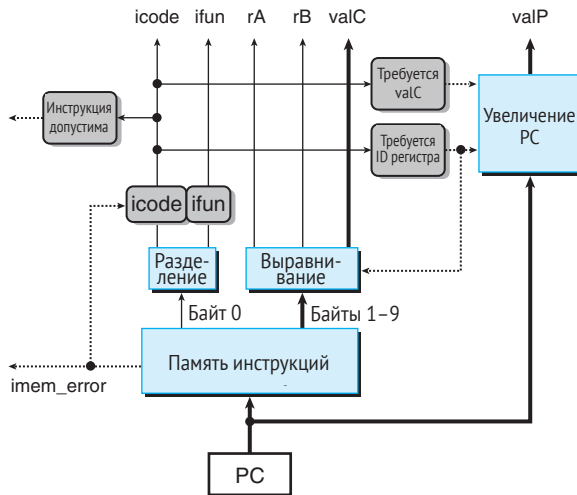
### Этап выборки

Как показано на рис. 4.19, этап выборки вовлекает в работу аппаратный модуль памяти для инструкций. За один раз это устройство читает 10 байт из памяти, используя значение PC как адрес первого байта (байт 0). Этот байт интерпретируется как байт-спецификатор инструкции и разбивается устройством разделения на два 4-битных поля. Блоки управляющей логики, подписанные как «icode» и «ifun», вычисляют коды инструкции и функции, либо равные значениям, прочитанным из памяти, либо,

в случае недействительного адреса инструкции (как отмечено сигналом **imem\_error**), равные значениям, соответствующим инструкции **por**. На основе значения **icode** затем вычисляются три 1-битных сигнала (обозначены пунктирными линиями):

- **instr\_valid** – сигнал соответствия байта действительной инструкции Y86-64. Этот сигнал используется для выявления недействительных инструкций;
- **need\_regids** – сигнал наличия в инструкции байта-спецификатора регистра;
- **need\_valC** – сигнал наличия в инструкции слова с непосредственным значением.

Сигналы **instr\_valid** и **imem\_error** (генерируется, когда адрес инструкции выходит за допустимые границы) используются для вычисления кода состояния на этапе обращения к памяти.



**Рис. 4.19.** Этап выборки в последовательной реализации SEQ. Из памяти инструкций, начиная с адреса в PC, извлекаются 10 байт инструкции. Из этих байтов формируются различные поля, описывающие инструкцию. Блок приращения ПК вычисляет сигнал **valP**

Для примера, HCL-описание **need\_regids** просто определяет, является ли значение **icode** одной из инструкций, имеющих байт-спецификатор регистра:

```
bool need_regids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
               IIRMOVQ, IRMMOVQ, IMRMOVQ };
```

Как показано на рис. 4.19, оставшиеся 9 байт, прочитанных из памяти инструкций, содержат байт-спецификатор регистров и слово непосредственного значения. Эти байты обрабатываются аппаратным модулем выравнивания и преобразуются в поля регистров и слово непосредственного значения. Когда вычисленный сигнал **need\_regids** равен единице, байт 1 делится на спецификаторы регистров **rA** и **rB**. Если **need\_regids** равен нулю, то оба поля получают значение F (RNONE), указывающее, что данная инструкция не использует регистры. Напомню также (рис. 4.2), что для любой инструкции, принимающей только один регистр-операнд, другое поле в байте-спецификаторе регистров получает значение F (RNONE). То есть можно предположить, что сигналы **rA** и **rB** либо кодируют регистры, к которым должна обращаться инструкция, либо указывают на то, что регистр-источник и/или регистр-приемник не используется. Модуль выравнивания также генерирует слово

непосредственного значения **valC**. Оно определяется байтами 1–8 или 2–9, в зависимости от значения сигнала **need\_regids**.

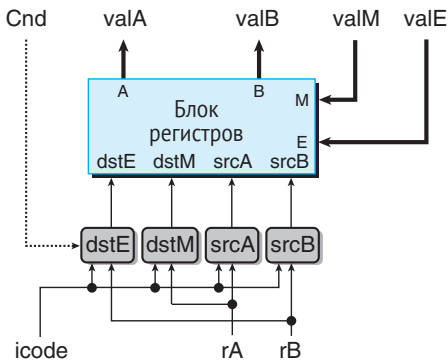
Аппаратный модуль увеличения PC (счетчик инструкций) генерирует сигнал **valP**, исходя из текущего значения PC, и два сигнала: **need\_regids** и **need\_valC**. Используя значения  $p$  (PC),  $r$  (**need\_regids**) и  $i$  (**need\_valC**), этот модуль генерирует значение  $p + 1 + r + 8i$ .

#### Упражнение 4.19 (решение в конце главы)

Напишите код HCL для сигнала **need\_valC** в последовательной реализации.

### Этапы декодирования и обратной записи

На рис. 4.20 подробно описана логика этапов декодирования и обратной записи в последовательной реализации. Эти два этапа объединены вместе, потому что оба обращаются к блоку регистров.



**Рис. 4.20.** Этапы декодирования и обратной записи в последовательной реализации SEQ. На основе полей инструкций генерируются идентификаторы регистров для четырех адресных входов (два для чтения и два для записи) в блок регистров. Значения, прочитанные из блока регистров, становятся сигналами **valA** и **valB**. Два значения обратной записи **valE** и **valM** служат данными для записи

Блок регистров имеет четыре порта. Он поддерживает возможность одновременно выполнения до двух операций чтения (через порты A и B) и двух операций записи (через порты E и M). Каждый порт включает шину адреса и шину данных. Шина адреса служит идентификатором регистра, а шина данных используется для передачи 64-разрядных слов на выход (в порт чтения) или на вход (в порт записи) блока регистров. Два порта чтения имеют адресные входы **srcA** и **srcB**, а два порта записи – адресные входы **dstE** и **dstM**. Специальный идентификатор  $\theta \times F$  (RNONE) на адресном входе порта указывает на отсутствие обращения к регистру.

Четыре блока внизу на рис. 4.20 генерируют четыре разных идентификатора регистров на основе кода инструкции **icode**, спецификаторов регистров **rA** и **rB** и, возможно, сигнала условия **Cnd**, вычисленного на этапе выполнения. Идентификатор регистра **srcA** указывает, какой регистр следует прочитать, чтобы получить **valA**. Желаемое значение зависит от типа инструкции, как показано в описании этапа декодирования в табл. 4.3–4.6. Объединив все эти описания, получаем следующее HCL-описание **srcA** (напомню, что RRSP – это идентификатор регистра %rsp):

```
word srcA = [
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;
    icode in { IPOPOQ, IRET } : RRSP;
    1 : RNONE; # Регистры не используются
];
```

Идентификатор регистра **dstE** указывает, в какой регистр выполняется запись значения **valE** через порт E. В табл. 4.3–4.6 получению желаемого значения соответствует первый шаг этапа обратной записи. Если забыть на время об инструкции условного перемещения, то мы можем объединить регистры-приемники для всех различных инструкций в следующее HCL-описание **dstE**:

```
# ВНИМАНИЕ: это описание не учитывает инструкции условного перемещения
word dstE = [
    icode in { IRRMOVQ } : rB;
    icode in { IIRMOVQ, IOPQ } : rB;
    icode in { IPUSHQ, IPOQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Запись в регистры не выполняется
];
```

Мы еще вернемся к этому сигналу и обсудим реализацию условного перемещения, когда будем рассматривать этап выполнения.

#### Упражнение 4.20 (решение в конце главы)

Сигнал **srcB** определяет регистр, значение которого должно использоваться в качестве сигнала **valB**. Получение желаемого значения показано в виде второго шага этапа декодирования в табл. 4.3–4.6. Напишите HCL-код, генерирующий **srcB**.

#### Упражнение 4.21 (решение в конце главы)

Идентификатор регистра **dstM** задает регистр-приемник для операции записи через порт M, где **valM** хранит значение, прочитанное из памяти. Получение желаемого значения показано в табл. 4.3–4.6 в виде второго шага этапа обратной записи. Напишите HCL-код, генерирующий **dstM**.

#### Упражнение 4.22 (решение в конце главы)

Одновременно оба порта записи в блок регистров используются только инструкцией **porq**. Для инструкции **porq %rsp** в порты записи E и M будет передан один и тот же адрес, но разные данные. Для устранения этого конфликта между портами записи необходимо задать приоритет, чтобы при попытке записи в один и тот же регистр в одном цикле запись производилась бы только из порта с более высоким приоритетом.

Какому из двух портов следует отдать приоритет, чтобы получить поведение, которое мы определили в упражнении 4.5?

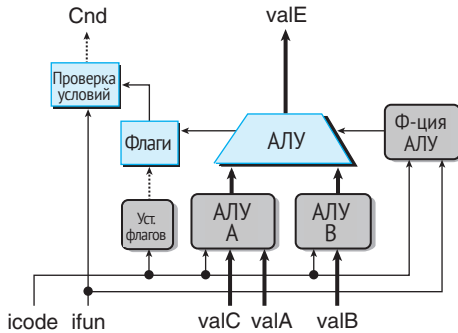
### Этап выполнения

Этап выполнения включает арифметико-логическое устройство (АЛУ), которое реализует операции сложения, вычитания, И (AND) и ИСКЛЮЧАЮЩЕЕ ИЛИ (EXCLUSIVE-OR) с входами **aluA** и **aluB**, исходя из настройки сигнала **alufun**. Эти данные и управляющие сигналы генерируются тремя управляющими блоками, изображенными на рис. 4.21. Выход АЛУ становится сигналом **valE**.

В табл. 4.3–4.6 вычисления в АЛУ показаны как первый шаг этапа выполнения во всех инструкциях. Операнды перечислены в таком порядке: первый операнд – **aluB**, а второй – **aluA**, то есть инструкция **subq** вычитает **valA** из **valB**. Как было показано в таблицах, **aluA** может принимать значения **valA**, **valC**, **-8** или **+8** в зависимости от типа инструк-

ции. Соответственно, поведение управляющего блока, генерирующего **aluA**, можно выразить так:

```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # Другие инструкции не используют АЛУ
];
```



**Рис. 4.21.** Этап выполнения в последовательной реализации SEQ. АЛУ может выполнять целочисленную операцию, предусмотренную инструкцией, или действовать как сумматор. Флаги устанавливаются в соответствии со значением, вычисленным АЛУ. Флаги проверяются, чтобы определить необходимость ветвления, если следующей окажется инструкция условного перехода или условного перемещения

Рассматривая операции АЛУ на этапе выполнения, можно заметить, что в большинстве случаев это устройство используется в качестве сумматора. Однако для инструкций **OPq** необходимо, чтобы оно выполняло операцию, закодированную в поле **ifun** инструкции. Следовательно, HCL-описание для управления АЛУ можно записать так:

```
word alufun = [
    icode == IOPQ : ifun;
    1 : ALUADD;
];
```

Этап выполнения также включает установку флагов. АЛУ генерирует три сигнала, соответствующих флагам условий – ноль (ZF), знак (SF) и переполнение (OF) – при каждом срабатывании. Однако сами флаги должны устанавливаться только при выполнении инструкции **OPq**. По этой причине генерируется дополнительный сигнал установки флагов **set\_cc**, управляющий обновлением регистра флагов:

```
bool set_cc = icode in {IOPQ};
```

Аппаратный модуль, подписанный как «Проверка условий», определяет, должна или не должна условная инструкция выполнить переход или перемещение (рис. 4.3). Он генерирует сигнал **Cnd**, который используется инструкциями условного перемещения для записи значения в **dstE** или адреса перехода в PC.

Для других инструкций сигнал **Cnd** может быть установлен в 1 или в 0, в зависимости от кода функции и кодов состояния, но в этих случаях он будет игнорироваться управляющей логикой. Мы не будем обсуждать устройство этого модуля.

#### Упражнение 4.23 (решение в конце главы)

Опираясь на первый операнд первого шага в этапе выполнения, показанного в табл. 4.3–4.6, опишите сигнал **aluB** на языке HCL.

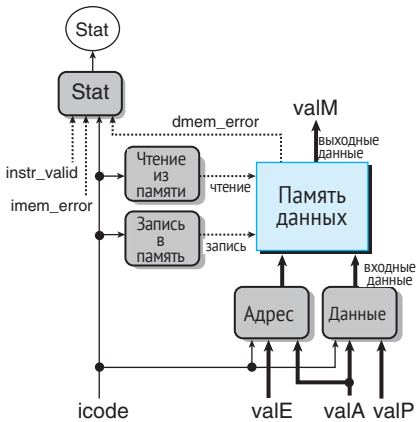


**Упражнение 4.24 (решение в конце главы)**

Инструкции условного перемещения, сокращенно `stovXX`, имеют код инструкции `IRRMVQ`. Эти инструкции, как показано на рис. 4.28, можно реализовать с использованием сигнала **Cnd**, сгенерированного на этапе выполнения. Измените код HCL для **dstE**, чтобы реализовать эти инструкции.

**Этап обращения к памяти**

Задача этапа обращения к памяти – прочитать или записать данные программы. Как показано на рис. 4.22, два управляющих блока генерируют значения адреса и данных для записи в память (в операциях записи). Два других блока генерируют управляющие сигналы, помогающие отличить операции чтения и записи. При выполнении операции чтения устройство памяти данных генерирует значение **valM**.



**Рис. 4.22.** Этап обращения к памяти в последовательной реализации SEQ. На этом этапе может выполняться чтение из памяти данных или запись в нее. Значение, прочитанное из памяти, образует сигнал **valM**

Конкретные операции с памятью в каждой инструкции показаны на этапе обращения к памяти в табл. 4.3–4.6. Обратите внимание, что адрес для операций записи и чтения – всегда **valE** или **valA**. Этот блок можно описать на HCL так:

```
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # Другие инструкции не используют адреса
];
```

Управляющий сигнал `mem_read` должен генерироваться только для инструкций, читающих данные из памяти. Это отражает следующее определение на HCL:

```
bool mem_read = icode in { IMRMVQ, IPOPQ, IRET };
```

**Упражнение 4.25 (решение в конце главы)**

Внимательно исследовав этап обращения к памяти в разных инструкциях (табл. 4.3–4.6), можно заметить, что в память всегда записывается **valA** или **valP**. Напишите HCL-определение для сигнала **mem\_data**.

**Упражнение 4.26 (решение в конце главы)**

Управляющий сигнал **mem\_write** должен устанавливаться только для инструкций, выполняющих запись в память. Напишите определение на HCL для сигнала **mem\_write**.

Последней функцией этапа обращения к памяти является вычисление кода состояния **Stat** по результатам выполнения инструкции, в соответствии со значениями **icode**, **imem\_error** и **instr\_valid**, сгенерированными на этапе выборки, и сигналом **dmem\_error**, сгенерированным памятью данных.

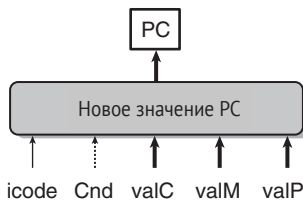
**Упражнение 4.27 (решение в конце главы)**

Напишите HCL-определение кода состояния **Stat**, генерирующее четыре кода состояния: SAOK, SADR, SINS и SHLT (табл. 4.8).

**Этап изменения PC**

Этот этап в последовательной реализации SEQ генерирует новое значение счетчика инструкций (рис. 4.23). Как показывают последние шаги в табл. 4.3–4.6, новое значение для счетчика инструкций извлекается из **valC**, **valM** или **valP**, в зависимости от типа инструкции и от выбора ветвления. Выбор значения на HCL можно выразить следующим образом:

```
word new_pc = [
  # Вызов процедуры (CALL). Использовать постоянное значение valC
  icode == ICALL : valC;
  # Выбрано ветвление. Использовать постоянное значение valC
  icode == IJXX && Cnd : valC;
  # Инструкция RET. Использовать значение со стека
  icode == IRET : valM;
  # По умолчанию: использовать увеличенное значение PC
  1 : valP;
];
```



**Рис. 4.23.** Этап изменения PC. Следующее значение для PC выбирается из сигналов **valC**, **valM** и **valP**, в зависимости от кода инструкции и флага ветвления

**Анализ последовательной реализации**

Мы закончили знакомство с реализацией процессора Y86-64. Как вы могли убедиться, путем организации обработки инструкций в этапы можно реализовать полноценный процессор с использованием небольшого количества аппаратных модулей с общим генератором синхронизирующих сигналов для управления последовательностью вычислений. Управляющая логика должна передавать сигналы между этими модулями и генерировать необходимые управляющие сигналы, исходя из типов инструкций и условий ветвления.

Единственная проблема последовательной реализации – низкая скорость работы. Генератор должен генерировать сигналы синхронизации с достаточно низкой частотой, чтобы сигналы успевали передаваться через все этапы в рамках одного цикла. Для примера рассмотрим обработку инструкции `ret`. После обновления счетчика инструкций в начале цикла синхронизации процессор должен прочитать инструкцию из памяти инструкций, прочитать указатель стека из блока регистров, с помощью АЛУ уменьшать значение указателя стека и прочитать из памяти адрес возврата, чтобы определить следующее значение счетчика инструкций. Все эти операции должны завершиться к концу цикла синхронизации.

При такой реализации возможности аппаратных модулей используются не в полной мере, потому что каждый из них активен только в течение небольшой части цикла синхронизации. Далее мы покажем, что большей производительности можно достичь путем внедрения конвейерной обработки.

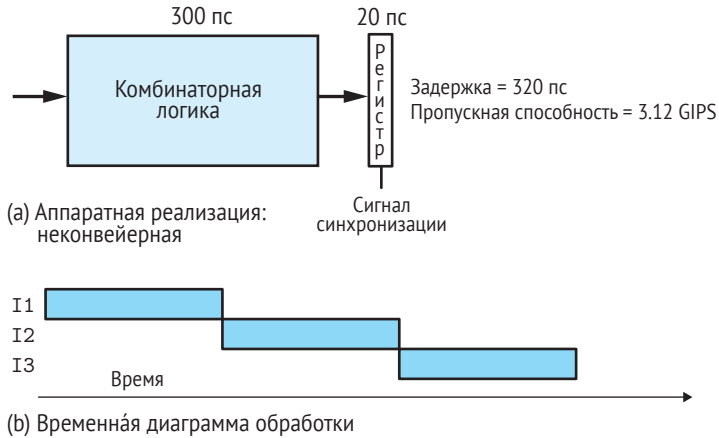
## 4.4. Общие принципы конвейерной обработки

Перед началом проектирования процессора Y86-64 с конвейерной организацией рассмотрим некоторые общие свойства и принципы конвейерных систем. Такие системы знакомы каждому, кто когда-либо посещал столовые с линией раздачи или пользовался услугами автоматической мойки автомобилей. В конвейерной системе выполняемая задача делится на последовательность дискретных этапов. Например, в столовой подобными этапами являются подача салата, основного блюда, десерта и напитков. В автомойке в их число входит распыление воды и моющего средства, ополаскивание, нанесение защитного покрытия и сушка. Вместо ожидания, пока один клиент пройдет всю последовательность этапов от начала до конца, эту последовательность проходят сразу несколько клиентов. На обычной линии раздачи в столовой посетители поддерживают одинаковый конвейерный порядок и проходят все этапы, даже если не покупают какие-то из блюд. На автомойке каждая следующая машина заходит на этап распыления воды, как только предыдущая перемещается на этап ополаскивания. В общем случае автомобили должны перемещаться по линии мойки с одинаковой скоростью, чтобы избежать столкновений.

Основное преимущество конвейерной обработки – увеличение *пропускной способности* системы, т. е. числа клиентов, обслуживаемых в единицу времени, но она же может немного увеличить *время ожидания* до завершения обслуживания предыдущего клиента. Например, посетитель столовой, которому нужен только салат, мог бы пройти через линию раздачи очень быстро, задержавшись только у стойки с салатами. Если же он попытается сделать то же самое в конвейерной системе, то другим посетителям это может не понравиться.

### 4.4.1. Вычислительные конвейеры

Если говорить о вычислительных конвейерах, то «клиентами» в них являются инструкции, а этапы выполняют некоторую часть операций, составляющих инструкцию. На рис. 4.24 (а) показан пример простой аппаратной системы. Она состоит из логики, выполняющей вычисления, за которой следует регистр, предназначенный для хранения результата вычислений. Сигнал синхронизации управляет загрузкой этого регистра через определенные интервалы времени. Примером такой системы может служить декодер в проигрывателе компакт-дисков. Входящими сигналами для него служат биты, прочитанные с диска, а управляющая логика декодирует их в звуковые сигналы. Вычислительные блоки на рис. 4.24 реализованы в виде комбинаторной логики, то есть сигналы проходят через последовательность логических вентилей, после чего, спустя некоторое время, на выходе появляется некоторая функция входного сигнала.



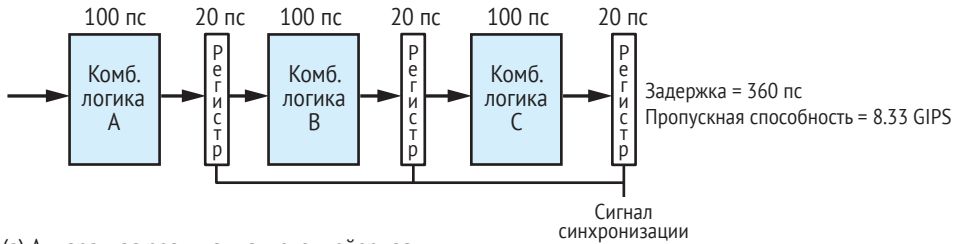
**Рис. 4.24.** Вычислительное оборудование без конвейера. В каждом цикле длительностью 320 пс система тратит 300 пс на вычисление функции комбинаторной логики и 20 пс на сохранение результатов в выходном регистре

В современном логическом проектировании задержки в цепи измеряются в *пикосекундах* (пс). Пикосекунда – это одна триллионная доля секунды ( $10^{-12}$  с). В этом примере мы предположили, что на выполнение комбинаторной логики требуется 300 пс и еще 20 пс – на загрузку результата в регистр. На рис. 4.24 также показана диаграмма, называемая *временной*. На этой диаграмме время течет слева направо. Последовательность инструкций (с именами I1, I2 и I3) выполняется в порядке сверху вниз. Прямоугольники обозначают время, в течение которого обрабатываются инструкции. В этой системе обработка каждой следующей инструкции может начаться не раньше, чем закончится обработка предыдущей. Поэтому прямоугольники не пересекаются во времени. Следующая формула позволяет рассчитать максимальную скорость работы такой системы:

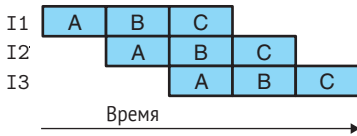
$$\text{Пропускная способность} = \frac{1 \text{ инструкция}}{(20 + 300) \text{ пс}} \cdot \frac{1000 \text{ пс}}{1 \text{ нс}} \approx 3.12 \text{ GIPS}.$$

Пропускная способность выражена в миллиардах инструкций в секунду (Giga-Instructions Per Second, GIPS). Общее время, необходимое на обработку одной инструкции от начала и до конца, называется *задержкой*. В данной системе время задержки составляет 320 пс и является обратной величиной пропускной способности.

Предположим, что вычисления, выполняемые системой, можно разделить на три этапа – А, В и С, – на каждый из которых требуется 100 пс (рис. 4.25). Между этими этапами можно разместить *конвейерные регистры*, и каждая инструкция будет проходить через систему от начала до конца за три шага – три полных цикла синхронизации. Как показано на временной диаграмме на рис. 4.26, этап А обработки инструкции I2 может начаться только после перемещения I1 из этапа А в этап В. В устойчивом состоянии все три этапа будут активны одновременно. С каждым новым циклом одна инструкция будет покидать систему, а другая – входить в нее. Это видно на третьем цикле временной диаграммы, где I1 находится на этапе С, I2 – на этапе В и I3 – на этапе А. В этой системе циклы следуют друг за другом через каждые  $100 + 20 = 120$  пс, обеспечивая пропускную способность порядка 8.33 GIPS. Поскольку обработка одной инструкции занимает 3 цикла, время задержки данного конвейера составляет  $3 \times 120 = 360$  пс. Пропускная способность системы повышается на коэффициент  $8.33/3.12 = 2.67$  за счет добавления дополнительных аппаратных модулей и небольшого увеличения времени задержки ( $360/320 = 1.12$ ). Увеличение времени задержки происходит из-за дополнительного времени, расходуемого на запись в дополнительные конвейерные регистры.



(а) Аппаратная реализация: неконвейерная

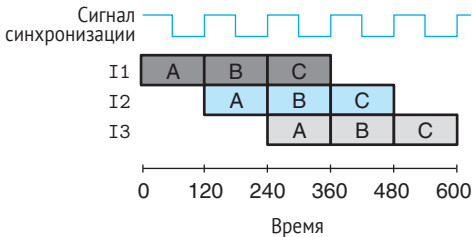


(b) Временная диаграмма обработки

**Рис. 4.25.** Аппаратный вычислительный конвейер с тремя этапами. Вычисление делится на этапы А, В и С. В каждом цикле длительностью 120 пс каждая инструкция проходит один этап

## 4.4.2. Подробное описание работы конвейера

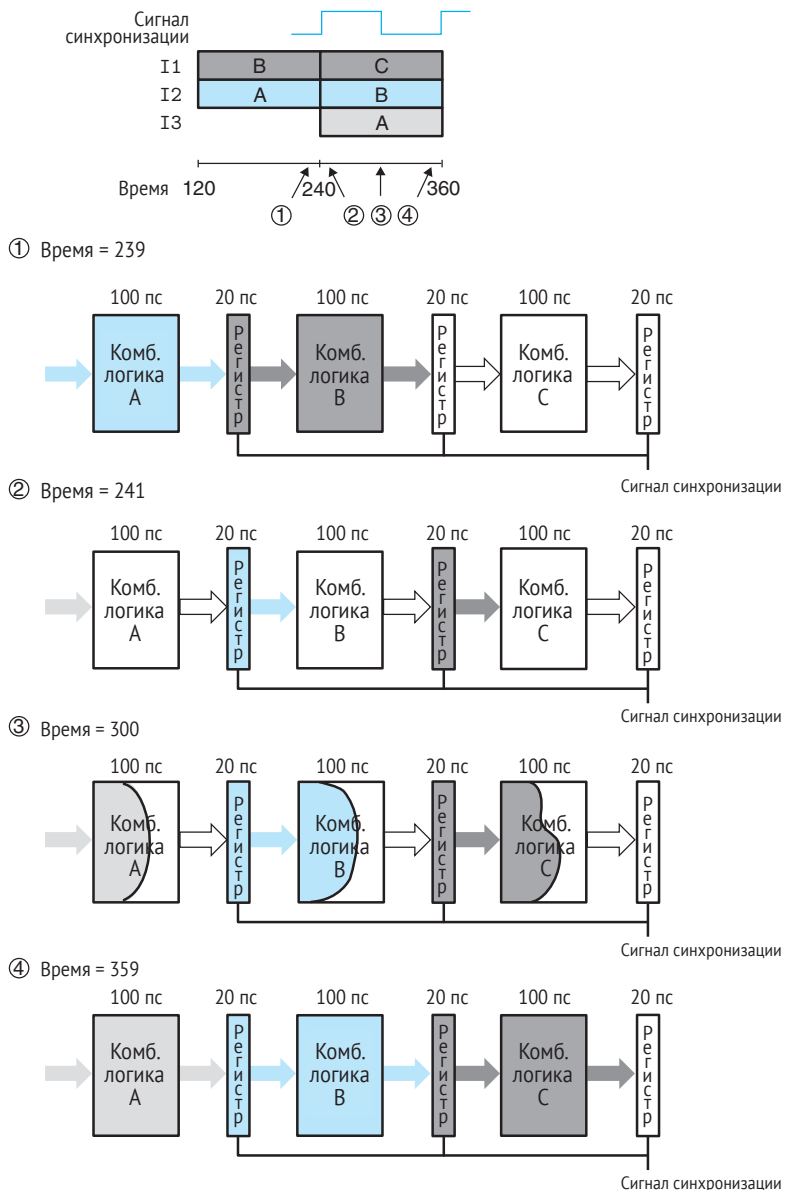
Чтобы лучше разобраться в особенностях конвейерной обработки, рассмотрим синхронизацию и работу вычислительного конвейера немного подробнее. На рис. 4.26 показана временная диаграмма для конвейера с тремя этапами (рис. 4.25). Переход инструкций с одного этапа на другой происходит по сигналу синхронизации, как показано над временной диаграммой. Каждые 120 пс этот сигнал возрастает с 0 до 1, инициируя очередной пакет вычислений в этапах конвейера.



**Рис. 4.26.** Временная диаграмма аппаратного вычислительного конвейера с тремя этапами. Перемещение инструкций из одного этапа в следующий происходит по положительному перепаду тактового сигнала

На рис. 4.27 показано, как действует цепь в интервале между отметками времени 240 и 360, когда инструкция I1 (изображена темно-серым) проходит через этап С, инструкция I2 (изображена синим) – через этап В и инструкция I3 (изображена светло-серым) – через этап А. Непосредственно перед положительным фронтом сигнала синхронизации в момент времени 240 (точка 1) значения, вычисленные на этапе А в ходе обработки инструкции I2, достигли входа первого конвейерного регистра, однако его состояние и выход все еще соответствуют результату, вычисленному на этапе А для инструкции I1. Значения, вычисленные на этапе В в ходе обработки инструкции I1, достигли входа второго конвейерного регистра. С прохождением положительного фронта сигнала синхронизации эти входы загружаются в конвейерные регистры и попадают на их выходы (точка 2). Помимо этого, на этапе А инициализируется обработка инструкции I3. Затем сигналы распространяются через комбинаторную логику разных этапов (точка 3). Волнистые границы на диаграмме в точке 3 показывают, что сигналы могут распространяться через разные секции с разной скоростью. Перед отметкой времени 360 результирующие значения достигают

входов конвейерных регистров (точка 4). С прохождением положительного фронта сигнала синхронизации в момент времени 360 каждая из инструкций завершит свой этап обработки.



**Рис. 4.27.** Один цикл работы конвейера. Непосредственно перед положительным фронтом синхронизирующего сигнала в момент времени 240 (точка 1) инструкции I1 (показана темно-серым) и I2 (показана синим) завершили этапы В и А. С прохождением положительного фронта эти инструкции начинают обрабатываться этапами С и В, а инструкция I3 (показана светло-серым) начинает обрабатываться этапом А (точки 2 и 3). Непосредственно перед следующим положительным фронтом синхронизирующего сигнала результаты обработки инструкций достигают входов конвейерных регистров (точка 4)

На этой подробной диаграмме, иллюстрирующей работу конвейера, видно, что уменьшение частоты синхронизации не меняет поведения конвейера. Результаты подаются на входы конвейерных регистров, однако их состояние остается прежним до появления положительного фронта сигнала. Увеличение частоты синхронизации, напротив, может иметь разрушительный эффект: значениям может не хватить времени, чтобы пройти через комбинаторную логику и достичь входов регистров к следующему положительному фронту синхронизирующего сигнала.

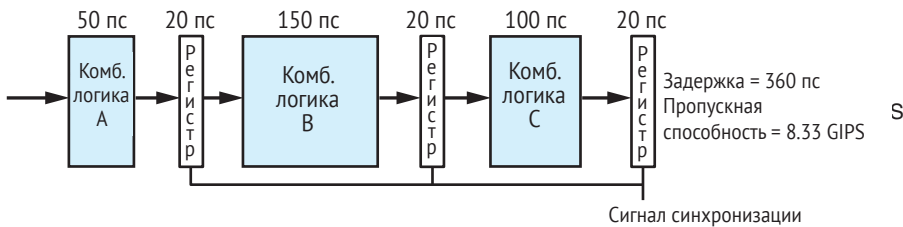
Как уже обсуждалось, когда мы говорили о синхронизации процессора с последовательной архитектурой (раздел 4.3.3), простого размещения синхронизированных регистров между блоками комбинаторной логики достаточно для управления потоком инструкций в конвейере. По мере колебаний сигнала синхронизации различные инструкции пересекают этапы конвейера, не накладываясь друг на друга.

### 4.4.3. Ограничения конвейерной обработки

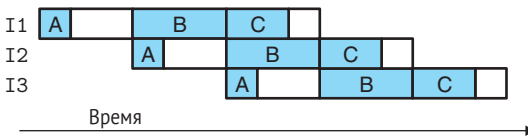
На рис. 4.25 показана идеальная конвейерная система, в которой вычисления можно разделить на три независимых этапа, для выполнения каждого из которых требуется одна треть от общего времени. Но, к сожалению, имеют место другие факторы, часто снижающие эффективность конвейерной обработки.

#### Неравномерное разбиение

На рис. 4.28 показана система, в которой вычисления делятся на три этапа, как в предыдущем примере, однако задержки перехода от одного этапа к другому меняются в диапазоне от 50 до 150 пс. Сумма всех задержек по-прежнему составляет 300 пс. Однако частота синхронизации ограничена задержкой самого медленного этапа. Как показывает временная диаграмма на этой схеме, в каждом цикле синхронизации этап А находится в режиме ожидания (изображен белым прямоугольником) в течение 100 пс, а этап С – в течение 50 пс. Постоянно активным остается только этап В. Цикл синхронизации в такой системе должен длиться  $150 + 20 = 170$  пс, что дает пропускную способность в 5,88 GIPS. Из-за подобного снижения частоты синхронизации общая задержка возрастает до 510 пс.



(а) Аппаратная реализация: конвейерная с тремя этапами и неравномерными задержками на этапах



(b) Временная диаграмма обработки

**Рис. 4.28.** Ограничения конвейерной обработки из-за неравномерных задержек на этапах. Пропускная способность системы ограничена скоростью самого медленного этапа

Деление вычислительной системы на серию этапов с одинаковыми задержками может стать основной трудностью проектировщиков аппаратных средств. Часто некоторые аппаратные компоненты процессора (например, АЛУ и память) нельзя разделить на несколько модулей с более коротким периодом задержки. Это затрудняет создание набора сбалансированных этапов. При проектировании процессора Y86-64 с конвейерной организацией мы не будем вдаваться в такого рода детали, однако не следует забывать о важности оптимизации синхронизации при проектировании фактической системы.

#### Упражнение 4.28 (решение в конце главы)

Предположим, что мы проанализировали комбинаторную логику, изображенную на рис. 4.24, и выяснили, что ее можно разделить на шесть блоков (A–F) с временем задержки 80, 30, 60, 50, 70 и 10 пс соответственно, как показано на рис. 4.29.

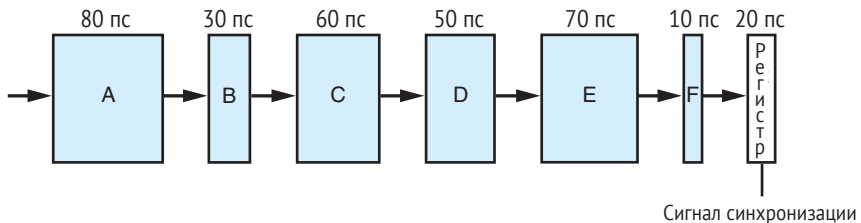


Рис. 4.29. Последовательность из шести блоков

Мы можем создать конвейерную версию данного проекта, добавив между блоками конвейерные регистры. Однако в зависимости от места расположения и количества конвейерных регистров возникают различные варианты организации конвейера с разным количеством этапов и пропускной способностью. Предположим, что конвейерный регистр вносит задержку 20 пс.

1. Добавив один конвейерный регистр, мы получим конвейер с двумя этапами. Куда необходимо вставить регистр, чтобы получить максимальную пропускную способность? Определите эту пропускную способность и время задержки.
2. Куда необходимо вставить два регистра, чтобы получить конвейер с тремя этапами, имеющий максимальную пропускную способность? Определите эту пропускную способность и время задержки.
3. Куда необходимо вставить три регистра, чтобы получить конвейер с четырьмя этапами, имеющий максимальную пропускную способность? Определите эту пропускную способность и время задержки.
4. Сколько этапов должен содержать этот проект, чтобы получить максимально возможную пропускную способность? Опишите этот проект, его пропускную способность и время задержки.

### Уменьшение отдачи от избыточно глубокой конвейеризации

На рис. 4.30 показано действие еще одного ограничивающего фактора конвейерной обработки. В этом примере вычисления разделены на шесть этапов, каждый из которых длится 50 пс. Если в эту последовательность добавить конвейерные регистры между каждой парой этапов, то получится конвейер с шестью этапами. Минимальная продолжительность цикла в такой системе составит  $50 + 20 = 70$  пс, что обеспечит пропускную способность 14,29 GIPS. То есть за счет удвоения числа этапов мы увеличим произво-

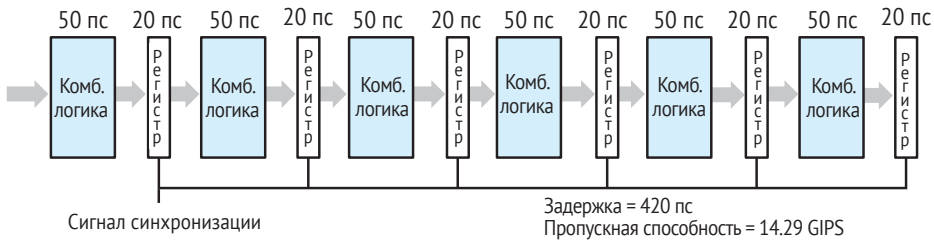


длительность конвейера в  $14,29/8,33 = 1,71$  раза. Даже если сократить вдвое время работы каждого вычислительного блока, мы не получим удвоения пропускной способности из-за задержки в конвейерных регистрах. Эта задержка становится фактором, ограничивающим пропускную способность конвейера. В этом новом проекте на задержку в конвейерных регистрах расходуется 28,6 % общего времени цикла.

В современных процессорах используются очень глубокие конвейеры (от 15 и более этапов), чтобы максимизировать тактовую частоту процессора. Создатели процессоров делят процесс выполнения инструкций на большое количество очень простых шагов, чтобы задержка на каждом этапе была предельно малой. Проектировщики цепей тщательно продумывают конструкцию конвейерных регистров. Разработчикам чипов приходится быть предельно внимательными при построении сети распространения тактовых сигналов, чтобы гарантировать его одновременное изменение во всех частях микросхемы. Все эти факторы являются определяющими при создании высокоскоростных микропроцессоров.

#### Упражнение 4.29 (решение в конце главы)

Предположим, что систему на рис. 4.24 можно разделить на произвольное количество  $k$  этапов с одинаковым временем задержки  $300/k$ , при этом задержка в каждом конвейерном регистре составляет 20 пс.



**Рис. 4.30.** Накладные расходы ограничивают пропускную способность конвейера. Когда комбинаторная логика разбивается на более короткие блоки, задержка, вносимая конвейерными регистрами, становится ограничивающим фактором

1. Опишите функцию зависимости задержки и пропускной способности системы от  $k$ .
2. Какой максимальной пропускной способности можно добиться от этой системы?

#### 4.4.4. Конвейерная обработка с обратной связью

До сих пор мы рассматривали системы, в которых объекты, пересекающие конвейер, будь то автомобили, люди или инструкции, абсолютно независимы друг от друга. Однако в машинных программах, например для архитектуры x86-64 или Y86-64, инструкции могут зависеть друг от друга. Рассмотрим, к примеру, следующую последовательность инструкций Y86-64:

```

1  irmovq  $50, %rax
2  addq    %rax, %rbx
3  mrmovq  100(%rbx), %rdx
  
```

Инструкции в этой последовательности зависят друг от друга *по данным*, как показывают имена регистров, обведенные окружностями, и стрелки между ними. Инструкция

`irmovq` (строка 1) сохраняет свой результат в `%rax`, который затем используется инструкцией `addq` (строка 2). Эта инструкция, в свою очередь, сохраняет результат в `%rbx`, который затем используется инструкцией `mrmovq` (строка 3).

Другой пример зависимостей в последовательности инструкций – наличие инструкций управляющей логики. Рассмотрим следующую последовательность инструкций Y86-64:

```

1  loop:
2      subq %rdx,%rbx
3      jne targ
4      irmovq $10,%rdx
5      jmp loop
6  targ:
7      halt

```

Инструкция `jne` (строка 3) создает *зависимость по управлению*, поскольку выбор следующей выполняемой инструкции зависит от результата проверки: `irmovq` (строка 4) или `halt` (строка 7). В последовательной реализации эти зависимости обрабатывались цепями обратной связи, показанными справа на рис. 4.16. Эта обратная связь передает обновленные значения регистров в блок регистров, а новое значение счетчика инструкций в регистр PC.

На рис. 4.31 показаны опасности внедрения конвейерной обработки в систему, содержащую цепи обратной связи. В первоначальной системе (рис. 4.31 (a)) результат каждой передается в следующую операцию. Это иллюстрирует временная диаграмма (рис. 4.31 (b)), где результат I1 подается на вход I2 и т. д. Просто преобразовав систему в конвейер с тремя этапами без учета зависимостей (рис. 4.31 (c)), мы изменим ее поведение. Как показано на временной диаграмме (рис. 4.31 (d)), результат I1 будет передан на вход I4. То есть, попытавшись ускорить систему за счет внедрения конвейерной обработки, мы изменили ее поведение.

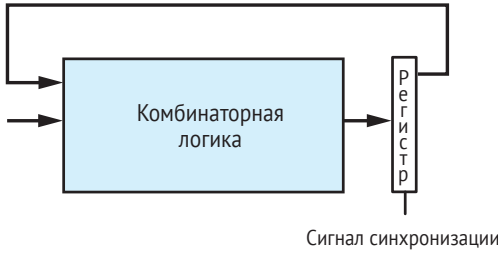
Внедряя конвейерную обработку в процессор Y86-64, мы должны учесть влияние обратных связей. Понятно, что измененное поведение системы, как показано на рис. 4.31, совершенно неприемлемо. Необходимо каким-то образом учесть зависимости между инструкциями, чтобы поведение системы соответствовало модели, определяемой ISA.

## 4.5. Конвейерные реализации Y86-64

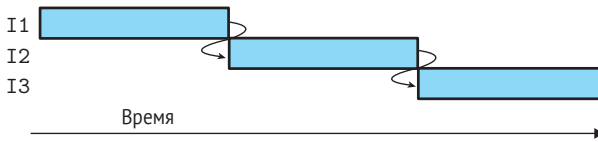
Наконец, все готово для решения основной задачи данной главы: проектирование процессора Y86-64 с конвейерной обработкой. За основу возьмем немного адаптированную последовательную реализацию процессора (SEQ+), вычисляющую значение для PC на этапе выборки, и добавим между этапами конвейерные регистры. В своей первой попытке мы не будем стараться правильно обработать все зависимости по управлению и по данным. Однако после некоторых модификаций мы получим вполне эффективный конвейерный процессор, реализующий Y86-64 ISA.

### 4.5.1. SEQ+: переупорядочение этапов обработки

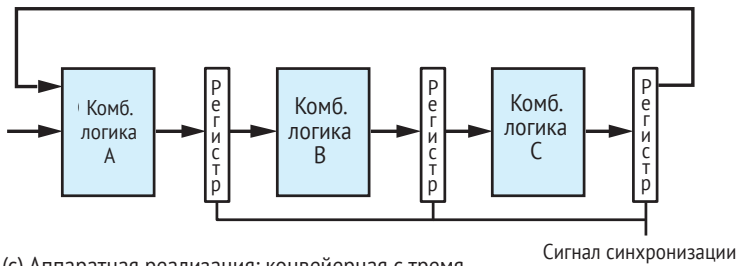
В качестве переходного шага на пути к конвейерной архитектуре мы должны немного изменить порядок выполнения пяти этапов в последовательной версии SEQ, чтобы этап изменения PC находился в начале цикла синхронизации, а не в конце. Для этого нужно лишь немного изменить общую аппаратную структуру, чтобы она лучше справлялась с последовательным выполнением операций на этапах конвейера. Назовем эту модифицированную архитектуру SEQ+.



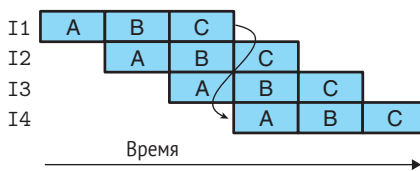
(a) Аппаратная реализация: неконвейерная с обратной связью



(b) Временная диаграмма обработки



(c) Аппаратная реализация: конвейерная с тремя этапами и обратной связью



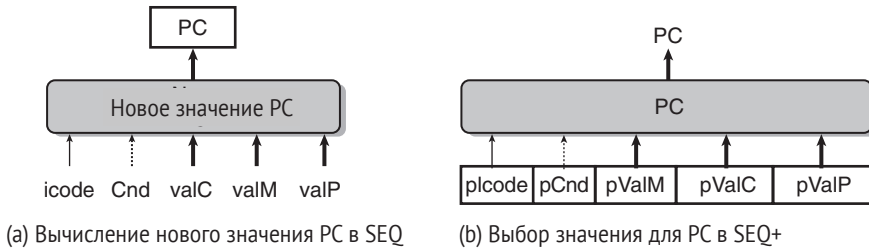
(d) Временная диаграмма обработки

**Рис. 4.31.** Ограничения конвейерной обработки из-за логических зависимостей.

При переходе от системы без конвейера с обратной связью (a) к конвейерной системе (c) мы меняем ее поведение, как это видно на двух временных диаграммах (b и d)

Мы можем переместить этап изменения PC в начало цикла синхронизации и вычислять значение PC для *текущей* инструкции. На рис. 4.32 показаны различия между архитектурами SEQ и SEQ+ в плане вычисления нового значения для PC. В архитектуре SEQ (рис. 4.32 (a)) вычисление происходит в конце цикла синхронизации на основе значений сигналов, вычисленных в течение текущего цикла. В архитектуре SEQ+ (рис. 4.32 (b)) мы добавили регистры состояния для хранения сигналов, вычисленных во время обработки инструкции. Когда начинается новый цикл, значения проходят через ту же логику вычисления PC для текущей в данный момент инструкции. Мы обозначили

регистры как «pIcode», «pCnd» и т. д., чтобы показать, что в любом данном цикле они хранят управляющие сигналы, сгенерированные во время предыдущего цикла.



**Рис. 4.32.** Перенос вычисления нового значения PC. В архитектуре SEQ+ значение счетчика инструкций для текущего состояния вычисляется в самом начале обработки инструкции

#### Где находится PC в архитектуре SEQ+?

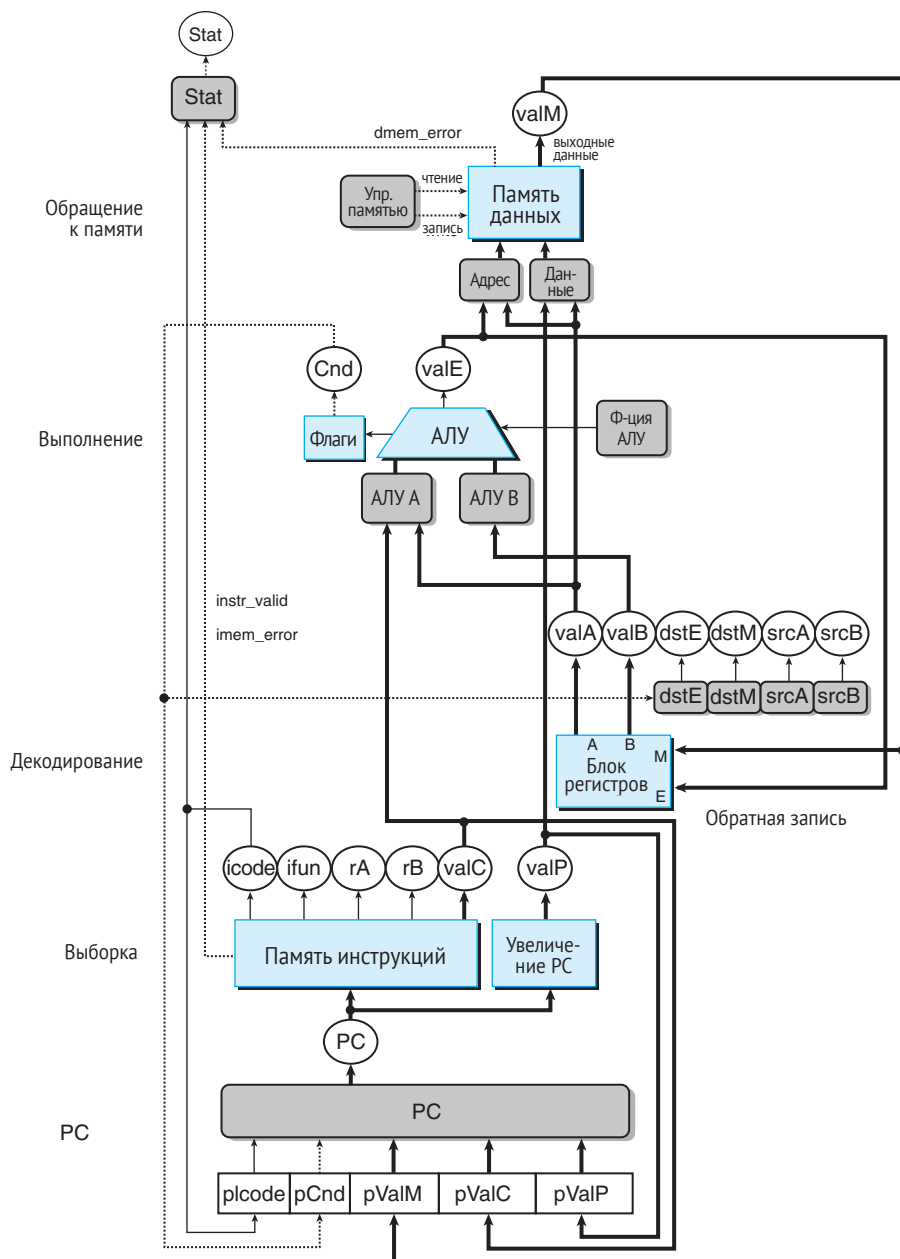
Интересной особенностью модифицированной последовательной архитектуры SEQ+ является отсутствие аппаратного регистра для хранения счетчика инструкций. Вместо этого значение PC вычисляется динамически на основе некоторой информации о состоянии, сохраненной в ходе обработки предыдущей инструкции. Этот пример наглядно демонстрирует, что процессор можно реализовать, отступив от концептуальной модели, подразумеваемой ISA, при условии что он правильно выполняет произвольные программы на машинном языке. Нет необходимости кодировать состояние в форме состояния, видимого программисту, если процессор сможет генерировать правильные значения для любой части состояния, видимого программисту (например, счетчик инструкций). Мы вновь используем данный принцип при создании конвейерной архитектуры. Методы обработки инструкций не по порядку, описанные в разделе 5.7, доводят эту идею до крайности, выполняя инструкции в совершенно ином порядке, чем в программе.

На рис. 4.33 показана подробная схема архитектуры SEQ+. Как видите, она состоит из тех же аппаратных модулей и блоков управления, что и SEQ (рис. 4.17), но с логикой изменения PC, сдвинутой сверху, где она выполнялась в конце цикла, вниз, где она выполняется в начале цикла.

Сдвиг элементов состояния в SEQ+ является примером типичного преобразования, известного как *перенастройка синхронизации цепи* [68]. Перенастройка синхронизации изменяет представление состояния системы без изменения ее логического поведения. Этот подход часто используется для балансировки задержек между различными этапами в конвейерных системах.

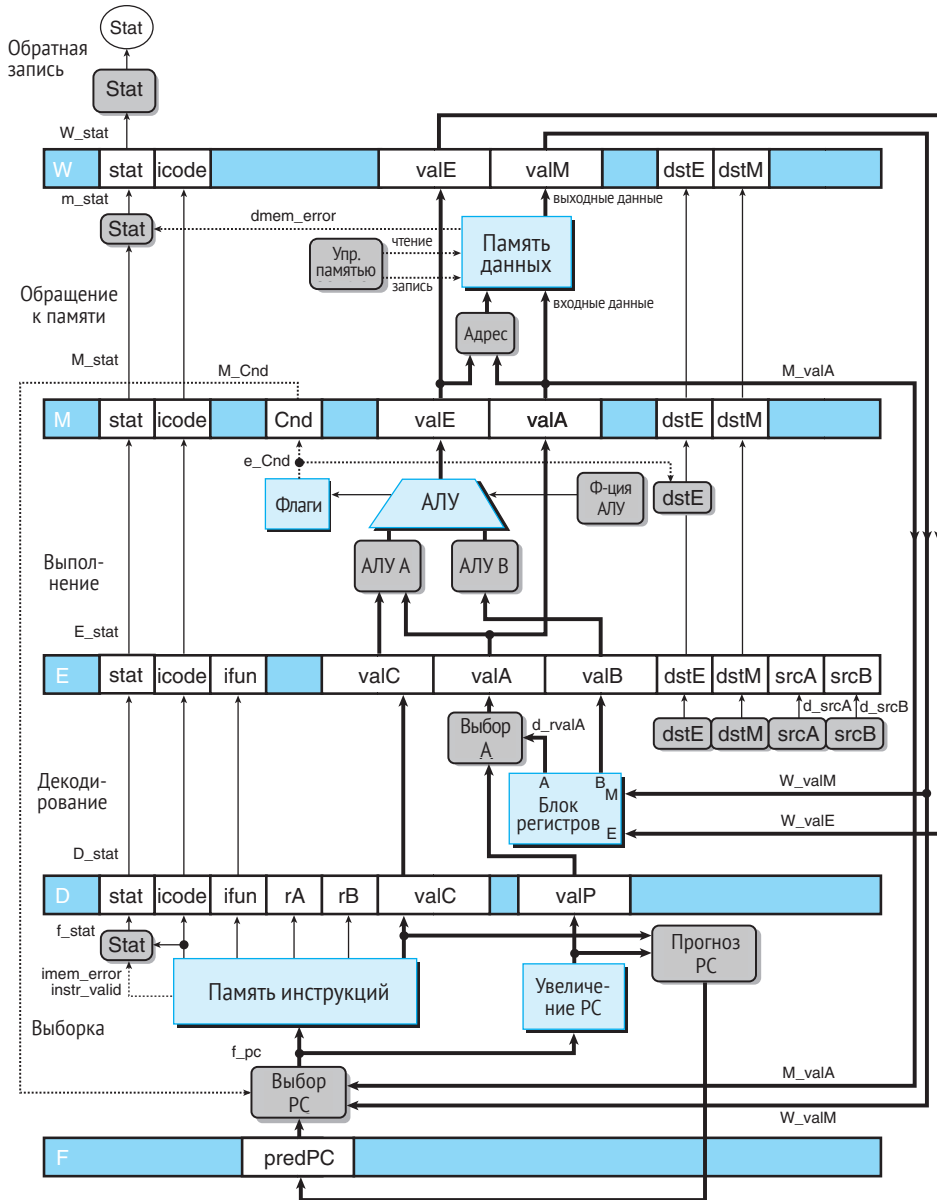
### 4.5.2. Добавление конвейерных регистров

В нашей первой попытке создания конвейерного процессора Y86-64 мы добавим конвейерные регистры между этапами в SEQ+ и определенным образом переупорядочим сигналы, в результате чего получится процессор PIPE–, где минус (–) в названии означает, что этот процессор имеет меньшую производительность, чем процессор, являющийся нашей конечной целью. Абстрактная структура PIPE– показана на рис. 4.34. Конвейерные регистры обозначены на схеме в виде синих прямоугольников. Каждый из них содержит несколько полей, изображенных в виде белых прямоугольников. В этих полях каждый конвейерный регистр хранит несколько байтов и слов. В отличие от меток в прямоугольниках с закругленными углами в схемах, описывающих две предыдущие последовательные версии процессоров (рис. 4.17 и 4.33), эти белые прямоугольники представляют фактические аппаратные компоненты.



**Рис. 4.33.** Аппаратная архитектура SEQ+. Перенос вычисления PC в начало цикла синхронизации делает эту архитектуру более подходящей для конвейерной обработки

Обратите внимание, что в PIPE– используется абсолютно тот же набор аппаратных модулей, что и в последовательной архитектуре SEQ+ (рис. 4.33), только в ней этапы разделены конвейерными регистрами. Различия в сигналах между двумя реализациями обсуждаются в разделе 4.5.3.



**Рис. 4.34.** Аппаратная архитектура PIPE–, первая конвейерная реализация. Добавив конвейерные регистры в архитектуру SEQ+ (рис. 4.33), мы создали конвейер с пятью этапами. Эта версия имеет несколько недостатков, которые мы вскоре рассмотрим

Конвейерные регистры обозначены следующим образом:

- F – хранит *прогнозируемое* значение счетчика инструкций (описывается чуть ниже);
- D – расположен между этапами выборки и декодирования. Хранит информацию о самых последних выбранных инструкциях для обработки на этапе декодирования;

Е – расположен между этапами декодирования и выполнения. Хранит информацию о самых последних декодированных инструкциях для обработки на этапе выполнения;

М – расположен между этапами выполнения и обращения к памяти. Хранит информацию о самых последних выполненных инструкциях для обработки на этапе обращения к памяти. Здесь же хранится информация об условиях ветвления и адреса для инструкций условного перехода;

W – расположен между этапами обращения к памяти и цепями обратной связи, передающими вычисленные результаты в блок регистров и адрес возврата в логику выборки РС при завершении инструкции `ret`.

На рис. 4.35 показано, как следующая последовательность инструкций будет обработана нашим конвейером с пятью этапами. Комментарии обозначают инструкции I1–I5 для справки:

```
1  irmovq  $1,%rax  # I1
2  irmovq  $2,%rbx  # I2
3  irmovq  $3,%rcx  # I3
4  irmovq  $4,%rdx  # I4
5  halt                    # I5
```

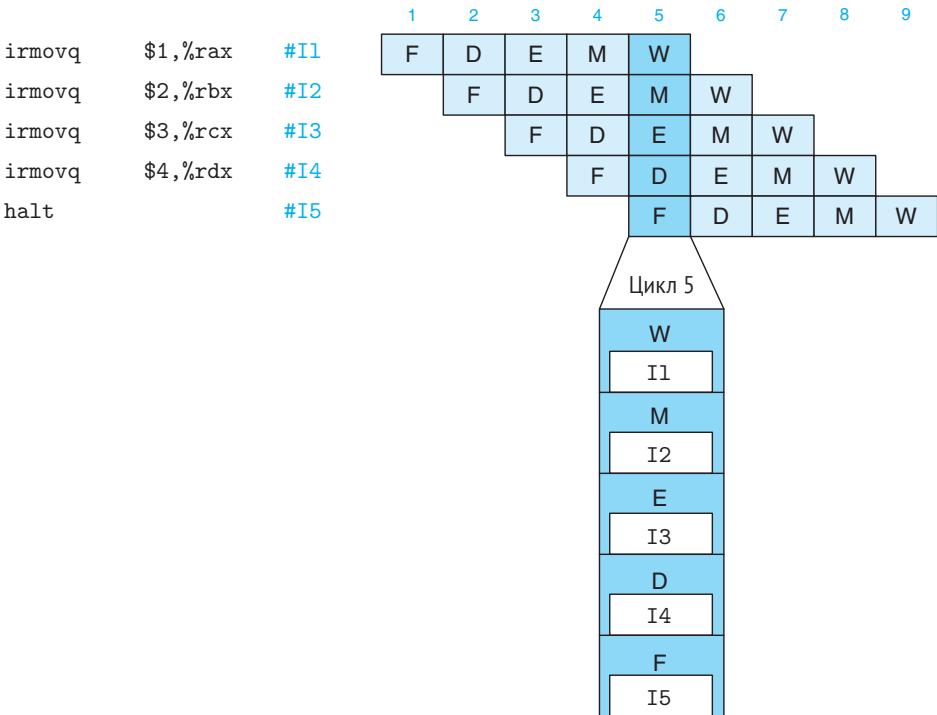


Рис. 4.35. Пример прохождения потока инструкций через конвейер

Справа на рис. 4.35 показана временная диаграмма обработки этой последовательности инструкций. Так же как на временных диаграммах простых конвейерных вычис-

лительных устройств, описанных в разделе 4.4, здесь показано прохождение каждой инструкции через этапы (время течет слева направо). Цифры сверху обозначают циклы синхронизации, в которых выполняются разные этапы. Например, в цикле 1 инструкция I1 обрабатывается разными этапами конвейера, а результаты записываются в блок регистров по окончании этапа 5. Инструкция I2 выбирается в цикле 2, а ее результат записывается по окончании цикла 6 и т. д. В нижней части показано подробное представление конвейера в цикле 5. В этой точке на каждом этапе конвейера обрабатывается своя инструкция.

Схема на рис. 4.35 следует принятым правилам изображения процессоров, согласно которым инструкции перемещаются снизу вверх. В подробном представлении цикла 5 показаны этапы конвейера с этапом выборки внизу и этапом обратной записи сверху, так же как на диаграмме конвейерной реализации (рис. 4.34). Если обратить внимание на порядок обработки инструкций, то можно увидеть, что они следуют в том же порядке, что и в программе. Поскольку программы обычно выполняются в порядке сверху вниз, этот порядок сохранен перемещением конвейера снизу вверх. Это соглашение особенно удобно при работе с имитаторами, сопутствующими данной книге.

### 4.5.3. Переупорядочение сигналов и изменение их маркировки

Наши последовательные реализации SEQ и SEQ+ обрабатывают инструкции по одной, поэтому сигналы, такие как **valC**, **srcA** и **valE**, имеют уникальные значения. В конвейерной архитектуре будет существовать несколько версий этих значений, принадлежащих разным инструкциям, обрабатываемым системой. Например, на подробной диаграмме PIPE– имеется четыре белых прямоугольника, подписанных как **Stat**, представляющих коды состояния для четырех разных инструкций (см. рис. 4.34). Мы должны тщательно следить за использованием правильной версии сигнала, чтобы избежать серьезных ошибок, например сохранения результата, вычисленного одной инструкцией, в регистре-приемнике, заданном другой инструкцией. С этой целью мы решили изменить схему именования сигналов, хранимых в конвейерном регистре, и добавить префикс с именем конвейерного регистра. Например, для ссылки на четыре кода состояния будут использоваться имена **D\_stat**, **E\_stat**, **M\_stat** и **W\_stat**. Нам также нужна возможность ссылаться на сигналы, вычисленные внутри этапа. К именам этих сигналов мы будем добавлять префикс – первую букву из названия этапа в нижнем регистре. Например, на этапах выборки и обращения к памяти используются свои блоки управляющей логики, подписанные как «Stat», вычисляющие коды состояния. В соответствии с нашим решением выходные сигналы этих блоков будут называться **f\_stat** и **m\_stat**. На диаграмме также видно, что фактическое состояние всего процессора **Stat** вычисляется этапом обратной записи на основе значения состояния в конвейерном регистре W.

Этапы декодирования в обеих архитектурах, SEQ+ и PIPE–, генерируют сигналы **dstE** и **dstM**, определяющие регистр-приемник для значений **valE** и **valM**. В SEQ+ эти сигналы можно было подключить непосредственно к адресным входам блока регистров. В PIPE– эти сигналы передаются по конвейеру через этапы выполнения и обращения к памяти и подаются на вход блока регистров только после достижения этапа обратной записи (как показано на подробных схемах этапов). Это делается для того, чтобы обеспечить принадлежность адреса и данных для порта записи одной инструкции. В противном случае на этапе обратной записи сохранялись бы значения, полученные на этапе обратной записи, но в регистр, идентификатор которого задан в инструкции, находящейся на этапе декодирования. Наша главная цель – сохранить всю информацию о каждой инструкции, обрабатываемой конвейером на любом из его этапов.

В PIPE– на этапе декодирования имеется один блок, отсутствующий в SEQ+, – блок «Выбор А». Он генерирует значение **valA** для конвейерного регистра E, выбирая **valP** из



конвейерного регистра D или значение, прочитанное из порта A блока регистров. Цель этого блока – уменьшить объем информации о состоянии, которая должна переноситься в конвейерные регистры E и M. Из всего набора инструкций только `call` использует значение **valP** на этапе обращения к памяти. И только инструкции перехода применяют значение **valP** на этапе выполнения (в случае если переход не выполняется). Ни одна из этих инструкций не использует значение, прочитанное из блока регистров. То есть мы можем уменьшить объем информации о состоянии в конвейерном регистре, объединив эти два сигнала и передавая объединенное значение через конвейер в виде одного сигнала **valA**. Этот шаг избавил нас от блока, подписанного как «Данные» в схемах SEQ (рис. 4.17) и SEQ+ (рис. 4.33), игравших аналогичную роль.

Как показано на рис. 4.34, наши конвейерные регистры содержат поле с кодом состояния **Stat**, который первоначально вычисляется на этапе выборки и может измениться на этапе обращения к памяти. Подробнее о реализации обработки исключительных ситуаций мы поговорим в разделе 4.5.6, когда завершим знакомство с нормальным ходом обработки инструкций. А пока лишь отметим, что наиболее систематический подход состоит в том, чтобы связать код состояния с каждой инструкцией, проходящей через конвейер, как мы показали это на схеме.

#### 4.5.4. Прогнозирование следующего значения PC

Мы предприняли определенные меры в архитектуре PIPE–, помогающие тщательно обрабатывать зависимости управления. Целью проектирования конвейерной версии является *выпуск* новой инструкции в каждом цикле синхронизации, то есть в каждом цикле на этап выполнения должна поступать и в конечном итоге выполняться новая инструкция. Достижение этой цели позволит пропускать по одной инструкции в каждом цикле. Для этого мы должны определить местонахождение следующей инструкции сразу после выборки текущей. К сожалению, если выбранная инструкция является условным ветвлением, то правильный выбор ветви станет известен лишь через несколько циклов – после того как инструкция пройдет этап выполнения. Аналогично, если выбрана инструкция `ret`, то адрес возврата не получится определить, пока инструкция не пройдет этап обращения к памяти.

За исключением инструкций условного перехода и `ret`, мы можем определить адрес следующей инструкции, опираясь на информацию, полученную на этапе выборки. Для `call` и `jmp` (безусловный переход) это будет **valC** – непосредственное значение в инструкции, а для других **valP** – адрес следующей инструкции. Следовательно, в большинстве случаев можно достичь поставленной цели – выпуска новой инструкции в каждом цикле синхронизации – путем *прогнозирования* следующего значения счетчика инструкций (PC). Для большинства типов инструкций такое прогнозирование будет абсолютно надежным. Для условных переходов можно спрогнозировать либо выбор перехода (т. е. новым значением PC станет **valC**), либо продолжение выполнения без перехода (тогда новым значением PC станет **valP**). В любом из этих случаев есть вероятность ошибки прогноза, т. е. выборки и частичного выполнения неверных инструкций. Этот вопрос еще будет рассматриваться в разделе 4.5.8.

Данная методика угадывания направления ветвления с последующей инициализацией выборки инструкций в соответствии с результатами угадывания называется *прогнозированием ветвления*. Она используется практически во всех процессорах в той или иной форме. В свое время проводилось множество экспериментов по выбору эффективных стратегий предсказания ветвления [46, раздел 2.3]. В некоторых системах решению этой задачи отводится достаточно большой объем аппаратных средств. В рассматриваемом же проекте будет применяться простая стратегия прогнозирования: условное ветвление выполняется всегда, поэтому результатом прогноза (новым значением PC) всегда будет **valC**.

### Другие стратегии прогнозирования выбора ветвей

В данном проекте используется стратегия прогнозирования ветвления *«выбирается всегда»*. Исследования показывают, что доля успешных предсказаний этой стратегии составляет порядка 60 % [44, 122]. Доля успешных предсказаний противоположной стратегии *«не выполняется никогда»* (never taken, NT) составляет порядка 40 %. Немного более сложная стратегия под названием *«переход назад выбирается всегда, вперед – никогда»* (backward taken, forward not taken, BTFNT) прогнозирует выбор следующей инструкции с младшим адресом и никогда – со старшим. Доля успешных предсказаний этой стратегии составляет порядка 65 %. Подобный прогресс обусловлен тем фактом, что обычно циклы завершаются инструкцией условного перехода в начало и выполняются многократно. Переходы вперед генерируются условными операторами, вероятность выбора которых меньше. В упражнениях 4.55 и 4.56 вам будет предложено модифицировать конвейерный процессор Y86-64 и реализовать в нем стратегии прогнозирования ветвления NT и BTFNT.

Как рассказывалось в разделе 3.6.6, ошибка прогнозирования ветвления может ухудшить производительность программы, и это является основным мотивом выбора инструкций условной передачи данных вместо условной передачи управления.

Итак, относительно условных переходов решение принято, но нам еще нужно научиться предсказывать новое значение РС при обработке инструкции *ret*. В этом случае, в отличие от условных переходов, мы имеем почти неограниченный набор возможных результатов, потому что адресом возврата может быть любое слово, лежащее на вершине стека. В нашем проекте мы не будем пытаться предсказать адрес возврата, а просто отложим обработку любых других инструкций до тех пор, пока инструкция *ret* не пройдет этап обратной записи. Мы вернемся к этой части реализации в разделе 4.5.8.

### Прогнозирование адреса возврата с использованием стека

В большинстве программ спрогнозировать адрес возврата очень просто, потому что вызовы процедур и возврат из них образуют согласованные пары. В подавляющем большинстве случаев вызываемая процедура возвращает к инструкции, следующей за инструкцией вызова. Это свойство используется в высокопроизводительных процессорах путем включения аппаратного стека в модуль выборки инструкций, в котором сохраняются адреса возврата, сгенерированные инструкциями вызова процедур. Всякий раз, когда выполняется инструкция вызова, соответствующий ей адрес возврата вталкивается в стек. Когда происходит выборка инструкции возврата, верхнее значение выталкивается из этого стека и используется в качестве спрогнозированного адреса возврата. Как и при прогнозировании ветвления, должен иметься механизм восстановления, на случай если прогноз окажется ошибочным, потому что иногда вызовы и возвраты не совпадают. Вообще говоря, такой прогноз очень надежен, а аппаратный стек не является частью состояния, видимого для программиста.

Этап выборки PIPE–, показанный внизу на рис. 4.34, отвечает и за прогнозирование следующего значения РС, и за выбор фактического значения РС для выборки инструкции. Как показано на диаграмме, блок, подписанный как «Прогноз РС», может выбрать либо **valP** (вычисленное путем увеличения РС), либо **valC** (из выбранной инструкции). Выбранное значение хранится в конвейерном регистре F как *спрогнозированное* значение счетчика инструкций. Блок с подписью «Выбор РС» похож на блок «РС» на этапе

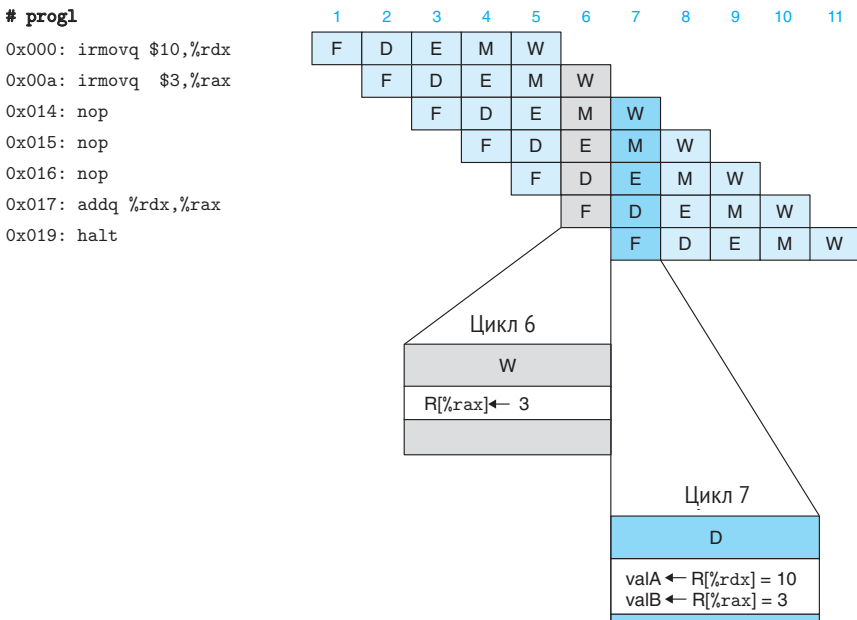
выбора PC в SEQ+ (см. рис. 4.33). Он выбирает одно из трех значений, которые играют роль адресов в памяти инструкций: спрогнозированное значение PC, значение **valP** для инструкции ветвления, не выполнившей переход, который достиг конвейерного регистра M (в поле **M\_valA**), или значение адреса возврата, когда инструкция **ret** достигла конвейерного регистра W (хранится в **W\_valM**).

### 4.5.5. Риски конвейерной обработки

Наша архитектура PIPE— является хорошей отправной точкой для создания конвейерного процессора Y86-64. Однако, как рассказывалось в разделе 4.4.4, внедрение конвейерной обработки в систему с обратной связью может привести к проблемам при наличии зависимостей в последовательности инструкций. Мы должны решить эту проблему, чтобы закончить данный проект. Подобного рода зависимости принимают две формы:

- *зависимость по данным*, когда результаты, вычисленные в одной инструкции, используются в качестве исходных данных в следующей;
- *зависимость по управлению*, когда одна инструкция определяет местоположение следующей инструкции, например выполняет переход, вызывает процедуру или производит возврат из процедуры.

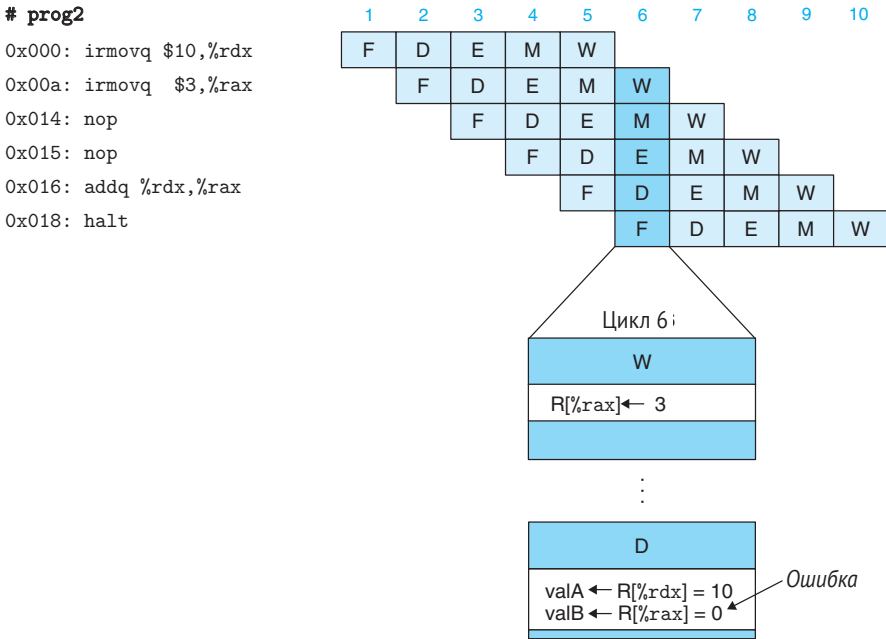
Такие зависимости, которые могут спровоцировать ошибочные вычисления конвейером, называются *рисками*. Подобно зависимостям, риски можно классифицировать как *риски по данным* или как *риски по управлению*. Сначала мы рассмотрим риски по данным, а затем риски по управлению.



**Рис. 4.36.** Выполнение программы prog1 в конвейере без использования специальных механизмов управления конвейером. В цикле 6 вторая инструкция `irmovq` записывает свой результат в программный регистр `%rax`. Инструкция `addq` читает значения операндов в цикле 7 и поэтому получает правильные значения для обоих регистров, `%rdx` и `%rax`

На рис. 4.36 показана обработка последовательности инструкций из программы prog1 процессором PIPE-. Предположим, что в этом и последующих примерах изначально все программные регистры имеют значение 0. Этот код загружает в программные регистры %rdx и %rax значения 10 и 3 соответственно, выполняет три инструкции nop, после чего прибавляет значение в регистре %rdx к значению в регистре %rax. Сосредоточим свое внимание на потенциальных рисках по данным, обусловленных зависимостью по данным между инструкциями irmovq и addq. Справа на рис. 4.36 показана временная диаграмма обработки последовательности инструкций. Этапы конвейера, выполняемые в циклах 6 и 7, выделены особо. Ниже показано более детальное представление этапа обратной записи в цикле 6 и декодирования в цикле 7. После начала цикла 7 обе инструкции irmovq проходят через этап обратной записи, и в результате в блоке регистров оказываются обновленные значения %rdx и %rax. По мере прохождения этапа декодирования инструкцией addq в цикле 7 она прочитает верные значения своих операндов. Зависимость по данным между двумя инструкциями irmovq и инструкцией addq в этом примере не создала риска по данным.

Мы видим, что инструкции из prog1 будут обрабатываться конвейером и получать корректные результаты, потому что три инструкции nop создают задержку между инструкциями с зависимостями по данным. Посмотрим теперь, что произойдет, если убрать эти инструкции nop.



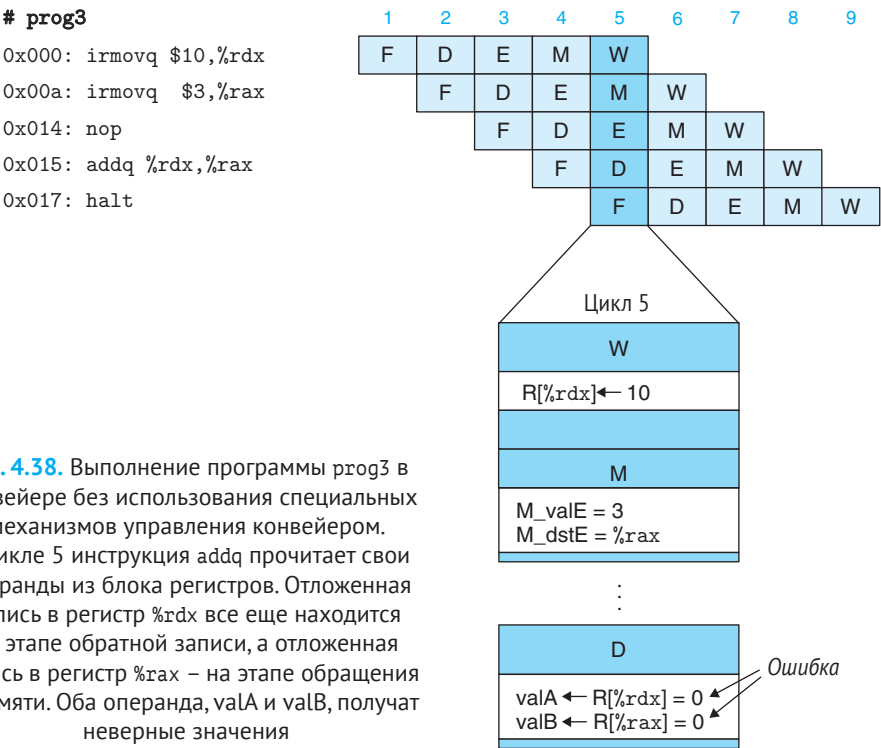
**Рис. 4.37.** Выполнение программы prog2 в конвейере без использования специальных механизмов управления конвейером. Запись в программный регистр %rax откладывается до начала цикла 7, поэтому инструкция addq получает неверное значение для этого регистра на этапе декодирования

На рис. 4.37 показан порядок обработки конвейером инструкций из программы prog2, содержащей две инструкции nop между парой инструкций irmovq, записывающих значения в регистры %rdx и %rax, и инструкцией addq, использующей эти два регистра в качестве операндов. В этом случае критической точкой является цикл 6, когда инструкция addq читает операнды из блока регистров. Детальное представление

операций, выполняемых конвейером во время этого цикла, показано внизу схемы. Первая инструкция `irmovq` прошла этап обратной записи, поэтому программный регистр `%rdx` получает обновленное значение.

Вторая инструкция `irmovq` находится в этом цикле на этапе обратной записи, поэтому запись в программный регистр `%rax` произойдет только с началом цикла 7, по переднему фронту сигнала синхронизации. В результате из регистра `%rax` будет прочитано некорректное нулевое значение (напомню, что согласно нашей предварительной договоренности все регистры имеют начальное значение 0), потому что запись в этот регистр еще не произошла. Понятно, что нам придется адаптировать наш конвейер, чтобы он верно обрабатывал данный риск.

На рис. 4.38 показано, что получится, если между инструкциями `irmovq` и `addq` оставить только одну инструкцию `nop`, как показано в программе `prog3`. На этот раз исследуем поведение конвейера в цикле 5, когда инструкция `addq` проходит через этап декодирования. К сожалению, к этому моменту отложенная запись в регистр `%rdx` все еще находится на этапе обратной записи, а отложенная запись в регистр `%rax` – на этапе обращения к памяти. Как результат инструкция `addq` получит некорректные значения для обоих операндов.



На рис. 4.39 показано, что получится, если убрать все инструкции `nop` из промежутка между инструкциями `irmovq` и `addq`, как показано в программе `prog4`. Теперь исследуем поведение конвейера в цикле 4, когда инструкция `addq` проходит через этап декодирования. К сожалению, отложенная запись в регистр `%rdx` все еще находится на этапе обращения к памяти, а новое значение для регистра `%rax` только вычисляется на этапе выполнения. Как результат инструкция `addq` получит некорректные значения для обоих операндов.



**Рис. 4.39.** Выполнение программы prog4 в конвейере без использования специальных механизмов управления конвейером. В цикле 4 инструкция addq читает свои операнды из блока регистров. Отложенная запись в регистр %rdx все еще находится на этапе обращения к памяти, а новое значение для регистра %rax только вычисляется на этапе выполнения. Оба операнда valA и valB получают неверные значения

Эти примеры иллюстрируют то, что риски по данным могут возникнуть, когда хотя бы один из операндов инструкции обновляется любой из трех предшествующих инструкций. Эти риски возникают от того, что разрабатываемый нами конвейерный процессор читает операнды инструкции из блока регистров на этапе декодирования, но не записывает в него результаты предшествующих инструкций, пока не пройдет еще три цикла, т. е. до того, как инструкция пройдет этап обратной записи.

Как избежать рисков по данным с помощью останова

Один из наиболее распространенных способов избежать рисков заключается в использовании *останова*, когда процессор удерживает в конвейере одну или несколько инструкций до тех пор, пока риск не исчезнет. Наш процессор может избежать рисков по данным, удерживая инструкции на этапе декодирования до тех пор, пока инструкции, генерирующие ее операнды, не пройдут этап обратной записи. Детали этого механизма будут обсуждаться в разделе 4.5.8. Он основан на простом усовершенствовании логики управления конвейером. Эффект останова показан на рис. 4.40 (prog2) и 4.41 (prog4). (Мы исключили prog3 из дальнейшего обсуждения, потому что эффект работает для нее точно так же, как в двух других примерах.) Когда инструкция addq находится на этапе декодирования, управляющая логика конвейера выявляет, что как минимум одна из инструкций на этапе выполнения, обращения к памяти или обратной записи будет обновлять регистр %rdx или регистр %rax. Вместо того чтобы позволить инструкции addq пройти этап с некорректными результатами, конвейер приостанавливает и удерживает эту инструкцию на этапе декодирования в течение одного (prog2) или трех (prog4) дополнительных циклов. Для всех трех программ инструкция addq получит корректные значения ее операндов в цикле 7, после чего продолжит движение по конвейеру.

Удерживая инструкцию addq на этапе декодирования, процессор должен также удерживать на этапе выборки следующую за ней инструкцию halt. Этого можно добиться, зафиксировав значение счетчика инструкций, чтобы инструкция halt выбиралась снова и снова, пока не завершится останов.

# prog2

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

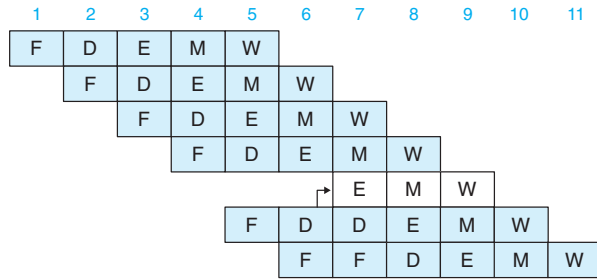
0x014: nop

0x015: nop

*пузырек*

0x016: addq %rdx,%rax

0x018: halt



**Рис. 4.40.** Выполнение программы prog2 конвейером с остановами. После декодирования инструкции addq в цикле 6 логика управления остановами обнаруживает риск по данным из-за незавершенной записи в регистр %rax на этапе обратной записи. Она вставляет «пузырек» в этап выполнения и повторяет декодирование инструкции addq в цикле 7. Фактически процессор динамически вставил инструкцию nop, создав поток выполнения, аналогичный показанному для prog1 (рис. 4.36)

Остановы предполагают удержание группы одних инструкций на своих этапах, в то время как другие инструкции должны продолжать движение по конвейеру. Но что должны обрабатывать этапы, которые при нормальном выполнении обрабатывали бы инструкцию addq? Всякий раз, когда требуется задержать инструкцию на этапе декодирования, в этап выполнения вставляется «пузырек», напоминающий динамически сгенерированную инструкцию nop, не изменяющую регистров, содержимого памяти и флагов условий. На рис. 4.40 и 4.41 эти «пузырьки» изображены белыми прямоугольниками. Стрелка между прямоугольником «D» (decode – этап декодирования) с инструкцией addq и прямоугольником «E» (execute – этап выполнения) с «пузырьком» указывает, что на этап выполнения вместо инструкции addq, которая при обычных условиях перешла бы с этапа декодирования, вставляется «пузырек». Более подробно о механизмах остановки конвейера и вставки «пузырьков» мы поговорим в разделе 4.5.8.

# prog4

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

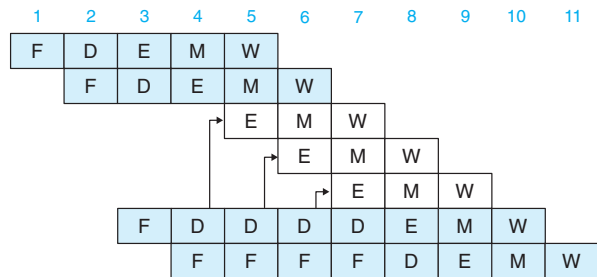
*пузырек*

*пузырек*

*пузырек*

0x014: addq %rdx,%rax

0x016: halt



**Рис. 4.41.** Выполнение программы prog4 конвейером с остановами. После декодирования инструкции addq в цикле 4 логика управления остановами обнаруживает риск по данным для обоих регистров-операндов. Она вставляет «пузырек» в этап выполнения и повторяет декодирование инструкции addq в цикле 5. После этого она снова обнаруживает риски для обоих регистров-операндов, вставляет «пузырек» в этап выполнения и повторяет декодирование инструкции addq в цикле 6. И снова обнаруживает риск по данным для регистра %rax, снова вставляет «пузырек» в этап выполнения и снова повторяет декодирование инструкции addq в цикле 7. Фактически процессор динамически вставил три инструкции nop, создав поток выполнения, аналогичный показанному для prog1 (рис. 4.36)

Использование остановки для устранения рисков по данным приводит к тому, что программы prog2 и prog4 приводятся к потоку выполнения программы prog1 (рис. 4.36).



Вставка одного «пузырька» в prog2 и трех в prog4 оказывает тот же эффект, что и вставка трех инструкций пор между второй инструкцией `irmovq` и инструкцией `addq`. Реализация этого механизма достаточно проста (упражнение 4.53), однако страдает общая производительность. Существует много случаев, когда одна инструкция обновляет значение регистра, а инструкция, следующая непосредственно за ней, использует этот регистр. В таких случаях конвейер будет останавливаться на период до трех циклов, что заметно снизит общую пропускную способность.

Как избежать рисков по данным методом продвижения

В нашем проекте PIPE– исходные операнды извлекаются из блока регистров на этапе декодирования, однако на этапе обратной записи может находиться инструкция, выполняющая запись в один из этих регистров. Вместо остановки до завершения операции записи можно просто передать записываемое значение в конвейерный регистр E в виде исходного операнда. Эта стратегия показана на рис. 4.42, где также можно видеть подробную временную диаграмму цикла 6 при выполнении программы prog2. Логика этапа декодирования определяет, что регистр `%rax` является источником значения для операнда **valB**, а также наличие ожидающей записи в `%rax` через порт E. В этом случае остановка можно избежать, просто используя слово данных, переданное в порт E (сигнал **W\_valE**), в виде значения для операнда **valB**. Такая методика передачи результата с одного этапа в конвейере на другой, более ранний, называется *продвижением данных* (data forwarding) или просто *продвижением*, а иногда *передачей назад*. Она позволяет обрабатывать инструкции в prog2 без задержек. Продвижение данных требует добавления в базовую аппаратную архитектуру дополнительной управляющей логики и соединений для передачи данных.

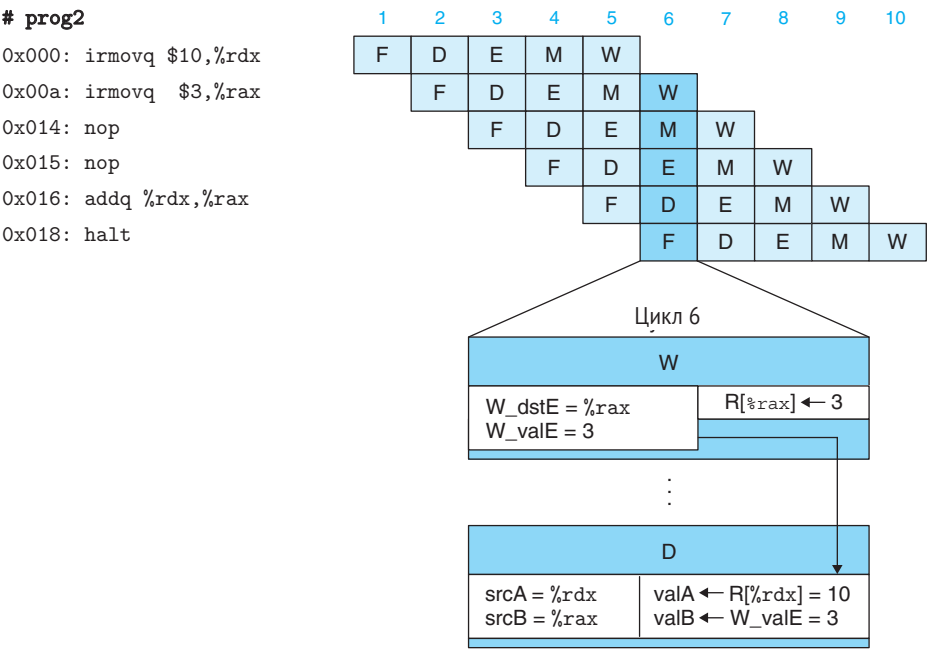


Рис. 4.42. Выполнение программы prog2 конвейером с поддержкой продвижения данных. В цикле 6 логика этапа декодирования обнаруживает наличие отложенной записи в регистр `%rax` на этапе обратной записи и передает (продвигает) это значение в операнд **valB**



### Классы рисков по данным

Потенциальные риски возникают, когда одна из инструкций обновляет некоторую часть состояния программы, которая используется более поздней инструкцией. В Y86-64 состояние программы включает программные регистры, счетчик инструкций, память, регистр флагов и регистр состояния. Рассмотрим возможные риски для каждого из перечисленных элементов состояния.

*Программные регистры.* Эти риски мы уже определили. Они возникают из-за того, что чтение блока регистров происходит на одном этапе, а запись на другом, из-за чего разные инструкции могут ненамеренно влиять друг на друга.

*Счетчик инструкций.* Эти риски обусловлены расхождениями между обновленным и прочитанным значениями счетчика инструкций. Риски отсутствуют, когда логика на этапе выборки правильно прогнозирует новое значение счетчика инструкций до выборки следующей инструкции. Неправильно спрогнозированные ветвления и инструкции *get* требуют особой обработки, которая рассматривается в разделе 4.5.5.

*Память.* Обе операции, запись и чтение, выполняются на этапе обращения к памяти. К тому времени, когда инструкция, выполняющая чтение из памяти, достигнет этого этапа, любая предшествующая ей инструкция, выполняющая запись в память, уже сделает это. С другой стороны, может возникнуть несоответствие между инструкциями, выполняющими запись на этапе обращения к памяти, и инструкциями, извлекаемыми из памяти на этапе выборки, потому что память инструкций и данных находится в общем адресном пространстве. Такая проблема может происходить только с программами, содержащими *самомодифицирующийся код*, когда инструкции выполняют запись в область памяти, откуда извлекаются очередные выполняемые инструкции. Некоторые системы включают сложные механизмы выявления подобных рисков, в других же просто принимается за аксиому невозможность самомодификации кода. Чтобы не усложнять проект, мы будем считать, что программы не могут содержать самомодифицирующийся код и поэтому нет необходимости предпринимать специальные меры для обновления памяти инструкций на основе изменений в памяти данных во время выполнения программы.

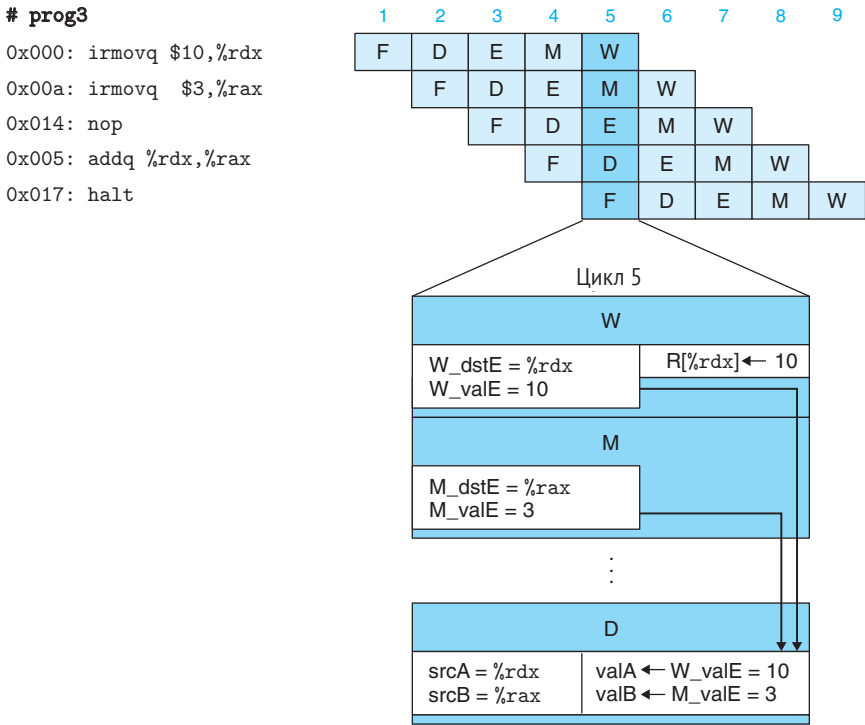
*Регистр флагов.* Флаги устанавливаются целочисленными операциями на этапе выполнения, а анализируются на этапе выполнения (инструкциями условного перемещения) или на этапе обращения к памяти (инструкциями условного перехода). К тому времени, когда инструкция условного перехода попадает на этот этап, любые предшествующие целочисленные операции уже завершили свой этап выполнения. В данном случае не возникает никаких рисков.

*Регистр состояния.* На состояние программы могут повлиять инструкции, обрабатываемые конвейером. Наш механизм связывания кода состояния с каждой инструкцией в конвейере позволяет процессору упорядоченно останавливаться при возникновении любого исключения, как будет обсуждаться в разделе 4.5.6.

Как показывает этот анализ, внимания заслуживают только риски по данным в регистрах, риски по управлению и риски исключений. Подобный систематический анализ важен при проектировании любой сложной системы. Он помогает определить потенциальные сложности в реализации системы и как должны выглядеть тестовые программы для проверки правильности системы.

Как показано на рис. 4.43, продвижение данных также можно использовать, когда имеет место ожидающая запись в регистр на этапе обращения к памяти, как, например, в программе `prog3`, и тем самым избежать необходимости останова. В цикле 5 логика этапа декодирования обнаруживает наличие отложенной записи в регистр `%rdx` на этапе обратной записи, а также отложенной записи в регистр `%rax` на этапе обращения к памяти. В этой ситуации вместо останова до завершения операций записи можно

переписать в операнды **valA** и **valB** значения с этапов обратной записи (сигнал **W\_valE**) и обращения к памяти (сигнал **M\_valE**).

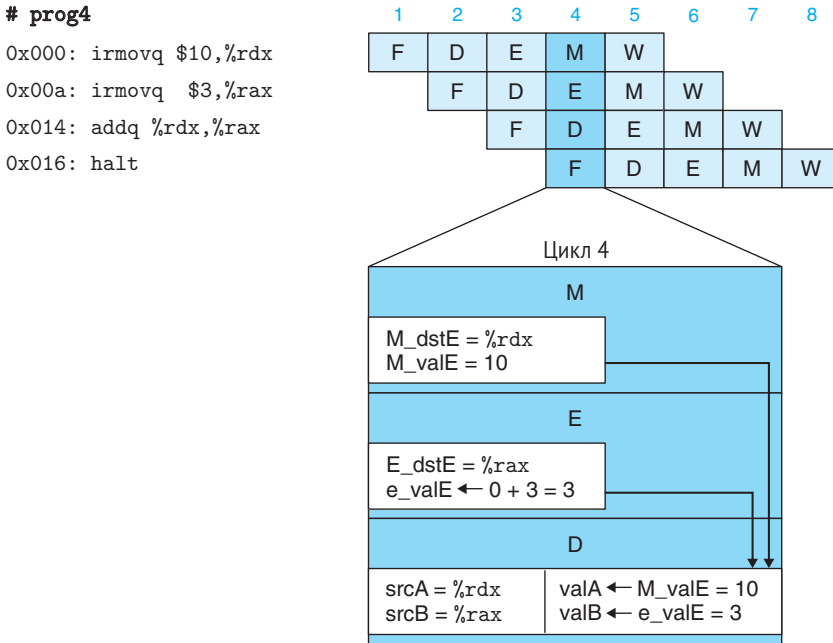


**Рис. 4.43.** Выполнение программы prog3 конвейером с поддержкой продвижения данных. В цикле 5 логика этапа декодирования обнаруживает наличие отложенной записи в регистр %rdx на этапе обратной записи и в регистр %rax на этапе обращения к памяти. Она передает (продвигает) эти значения в операнды valA и valB

Для полноценного использования продвижения данных можно также предусмотреть продвижение значений, вычисленных на этапе выполнения, в этап декодирования, чтобы избежать необходимости останова программы prog4, в ситуации, показанной на рис. 4.44. Здесь в цикле 4 логика декодирования обнаруживает наличие отложенной записи в регистр %rdx на этапе обращения к памяти, а также что позднее значение, вычисленное АЛУ на этапе выполнения, будет записано в регистр %rax. В этой ситуации она могла бы переписать в операнды **valA** и **valB** значение с этапа обращения к памяти (сигнал **M\_valE**) и результат на выходе АЛУ (сигнал **e\_valE**) соответственно. Обратите внимание, что использование результата на выходе АЛУ не вызывает никаких проблем с синхронизацией. Этап декодирования должен только сгенерировать сигналы **valA** и **valB** к концу цикла синхронизации, чтобы в конвейерный регистр E можно было загрузить результаты с этапа декодирования по положительному фронту сигнала синхронизации в начале следующего цикла. До этой точки выход АЛУ будет действителен.

Варианты использования продвижения, показанные на примерах программ от prog2 до prog4, включают продвижение значений, сгенерированных в АЛУ и предназначенных для записи в порт E. Продвижение также можно использовать со значениями, прочитанными из памяти и предназначенными для записи в порт M. На этапе обращения к памяти можно организовать продвижение только что прочитанного значения (сигнал **m\_valM**). На этапе обратной записи можно организовать продвижение значения,

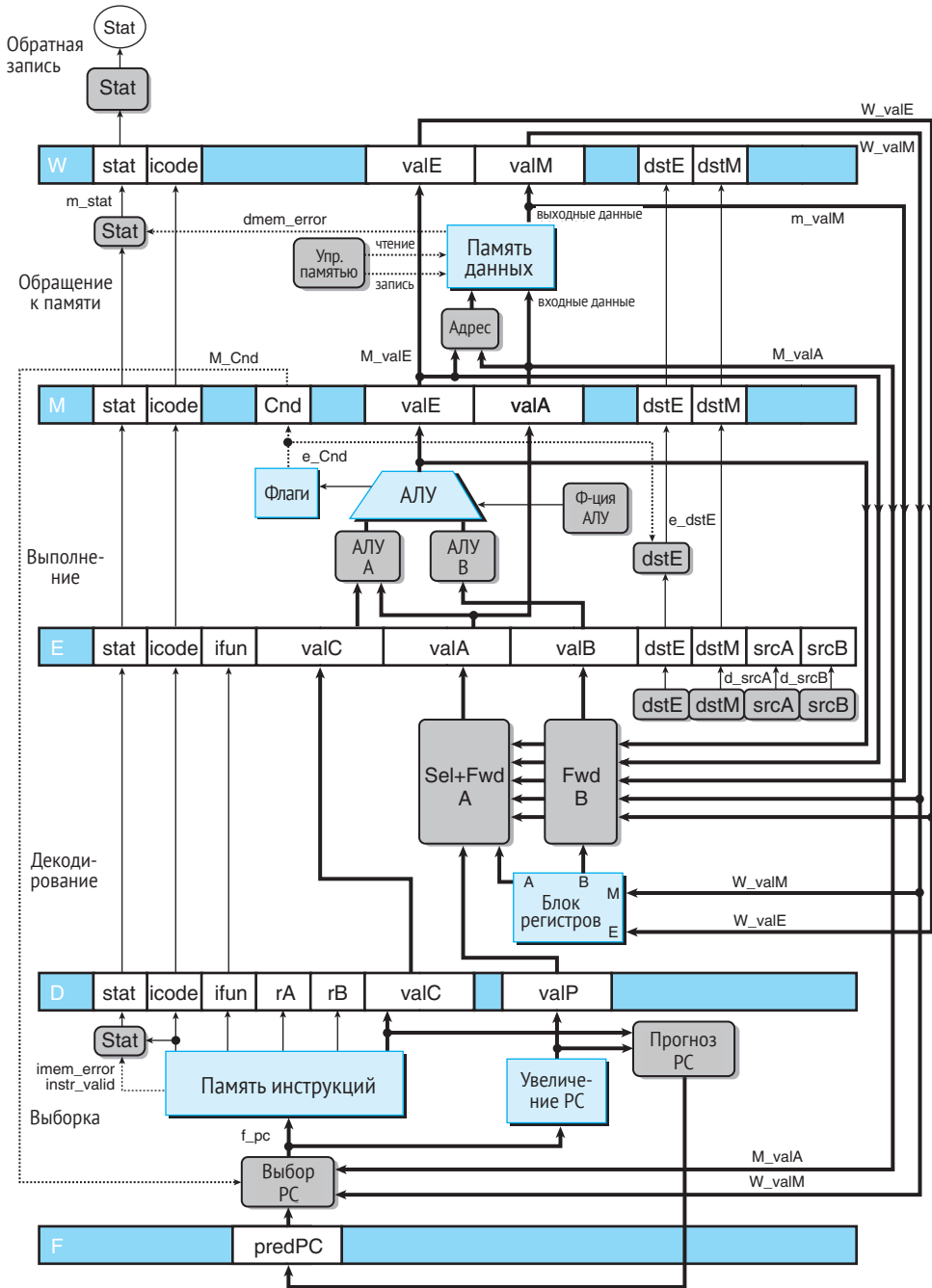
ожидающего записи на входе в порт M (сигнал **W\_valM**). Таким образом, мы имеем пять различных источников значений для продвижения (**e\_valE**, **m\_valM**, **M\_valE**, **W\_valM** и **W\_valE**) и два различных приемника (**valA** и **valB**).



**Рис. 4.44.** Выполнение программы prog4 конвейером с поддержкой продвижения данных. В цикле 4 логика декодирования обнаруживает наличие отложенной записи в регистр %rdx на этапе обращения к памяти и в регистр %rax на этапе выполнения. Она продвигает в valA и valB эти записываемые значения вместо полученных из блока регистров

На расширенных диаграммах (рис. 4.42–4.44) показано, как логика этапа декодирования может определить, какое значение следует использовать: из блока регистров или продвинутое. Каждое значение, которое будет записано обратно в блок регистров, связывается идентификатор регистра. Управляющая логика может сравнить эти идентификаторы с идентификаторами регистров-источников **scrA** и **scrB** для выявления случая продвижения. Может получиться так, что одному идентификатору регистра-источника будут соответствовать сразу несколько идентификаторов регистров-приемников. Для таких случаев необходимо установить приоритет между различными источниками продвижения. Мы обсудим этот вопрос, когда будем более детально рассматривать логику продвижения.

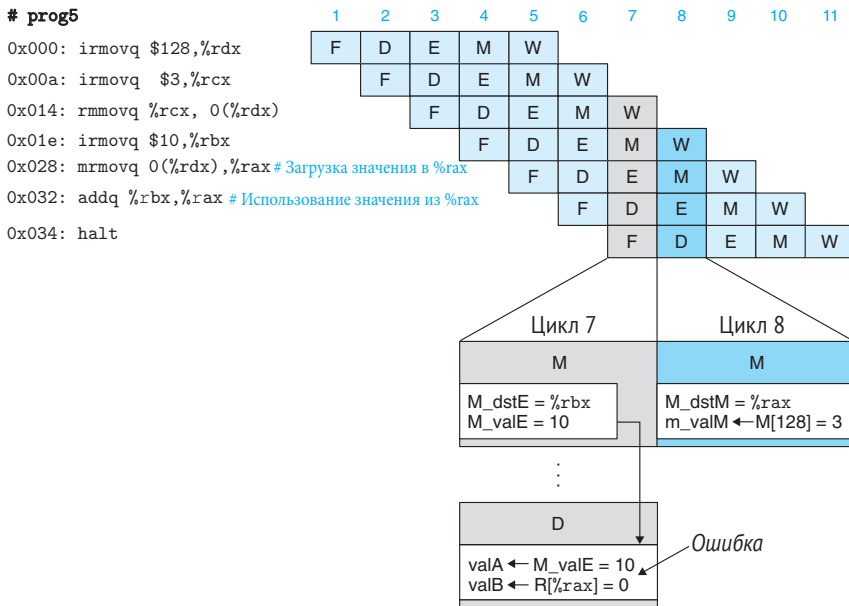
На рис. 4.45 показана абстрактная структура архитектуры PIPE, улучшенной версии PIPE–, которая способна управлять рисками по данным методам продвижения. Сравнивая ее с архитектурой PIPE– (рис. 4.34), можно заметить, что значения из пяти источников продвижения передаются в два блока на этапе декодирования, подписанных как «Sel+Fwd A» (Выбор + Продвижение A) и «Fwd B» (Продвижение B). Блок с подписью «Sel+Fwd A» сочетает роль блока с меткой «Выбор A» в PIPE– с логикой продвижения. Он позволяет значению **valA** в конвейерном регистре E служить увеличенным значением счетчика инструкций **valP**, или значением, полученным из порта A блока регистров, или одним из продвинутых значений. Блок с подписью «Fwd B» реализует логику продвижения для исходного операнда **valB**.



**Рис. 4.45.** Аппаратная архитектура PIPE, наша последняя конвейерная реализация. Дополнительные обходные пути позволяют продвигать результаты трех предыдущих инструкций, избавляя от большинства рисков по данным, и обрабатывать инструкции без остановки конвейера

## Риски по загрузке/использованию данных

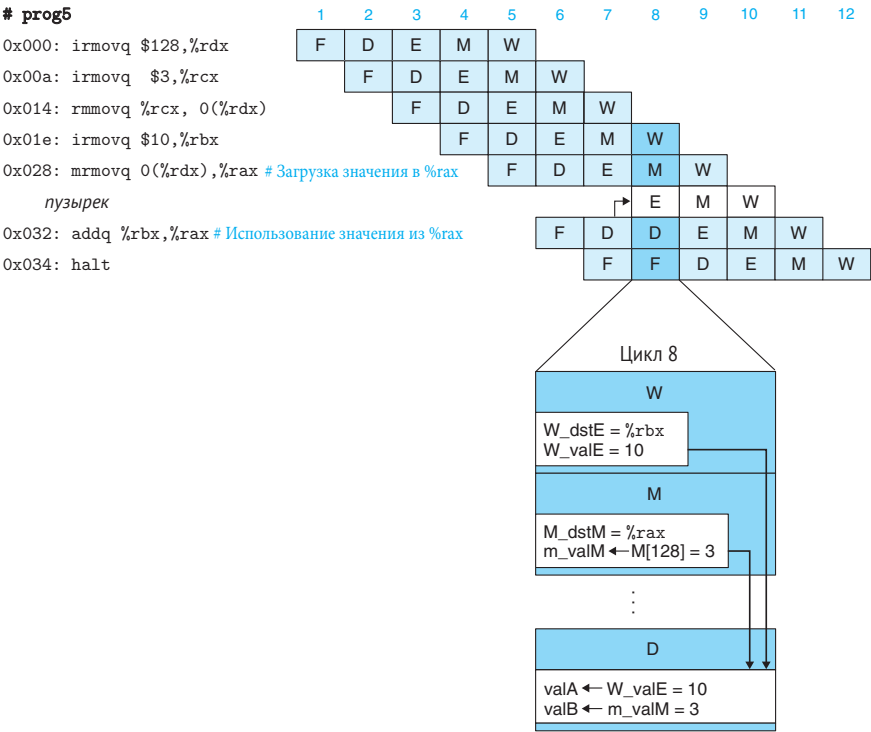
Один из классов рисков по данным нельзя устранить одним только продвижением, потому что чтение из памяти происходит ближе к концу конвейера. На рис. 4.46 показан пример *риска по загрузке/использованию данных*, когда одна инструкция (`movq` с адресом `0x028`) читает значение из памяти в регистр `%rax`, а следующая инструкция (`addq` с адресом `0x032`) использует это значение как операнд-источник. Внизу на рис. 4.46 показаны расширенные диаграммы циклов 7 и 8. Здесь предполагается, что все программные регистры изначально имеют нулевые значения. Инструкция `addq` использует значение из регистра в цикле 7, но оно генерируется инструкцией `movq` только в цикле 8. Чтобы осуществить продвижение операнда из `movq` в `addq`, логике продвижения пришлось бы вернуть значение назад во времени! Поскольку это невозможно, необходимо найти какой-то другой механизм обработки таких рисков по данным. (Риск по данным для регистра `%rbx`, значение которого генерируется инструкцией `irmovq` по адресу `0x01e` и используется инструкцией `addq` по адресу `0x032`, можно устранить путем продвижения.)



**Рис. 4.46.** Пример рисков по загрузке/использованию данных. В цикле 7 инструкция `addq` использует значение регистра `%rax` на этапе декодирования. Предыдущая инструкция `mrmovq` читает новое значение в этот регистр на этапе обращения к памяти в цикле 8, что слишком поздно для инструкции `addq`

Как показано на рис. 4.47, рисков по загрузке/использованию данных можно избежать, объединив приемы остановки и продвижения. Это требует изменения управляющей логики, но позволяет использовать существующие обходные пути. Когда инструкция `movq` достигает этапа выполнения, управляющая логика конвейера обнаруживает, что инструкция на этапе декодирования (`addq`) использует результат, прочитанный из памяти. Тогда она приостанавливает выполнение инструкции на этапе декодирования на один цикл, вставляя «пузырек» в этап выполнения. Как показано на расширенной диаграмме цикла 8, значение, прочитанное из памяти, может быть впоследствии передано с этапа обращения к памяти в инструкцию `addq` на этапе декодирования. Аналогично на этап декодирования с этапа обратной записи передается значение для регистра `%rbx`. Как по-

казано на временной диаграмме стрелкой от прямоугольника с буквой «D» в цикле 7 к прямоугольнику с буквой «E» в цикле 8, вставленный «пузырек» замещает инструкцию `addq`, которая иначе продолжила бы движение по конвейеру.



**Рис. 4.47.** Обработка рисков по загрузке/использованию данных путем останова. При останове инструкции `addq` на один цикл на этапе декодирования появляется возможность передать значение `valB` из инструкции `mrmovq` на этапе обращения к памяти в инструкцию `addq` на этапе декодирования

Подобное использование останова для управления рисками по загрузке/использованию данных называется *блокировкой загрузки*. Блокировки загрузки в комбинации с продвижением достаточно для обработки всех возможных рисков по данным. Поскольку теперь пропускную способность конвейера могут снижать только блокировки загрузки, можно сказать, что мы почти достигли желаемой цели – выполнять по одной инструкции в каждом цикле синхронизации.

Как избежать рисков по управлению

Риски по управлению возникают, когда процессор не может надежно определить адрес следующей инструкции на основе текущей на этапе выборки. Как обсуждалось в разделе 4.5.4, риски по управлению имеют место только в нашем конвейерном процессоре и только для инструкций перехода или возврата. Более того, инструкции перехода вызывают затруднения, только когда конечная точка условного ветвления предсказывается неверно. В этом разделе мы рассмотрим в общих чертах, как избежать этих рисков. Подробная реализация будет представлена в разделе 4.5.8 как часть более общего обсуждения управления конвейером.

Первым рассмотрим случай с инструкцией `ret`. Ниже приводится пример ассемблерного кода программы с адресами инструкций слева для справки:

```

0x000:    irmovq stack,%rsp # Инициализировать указатель стека
0x00a:    call proc        # Вызвать процедуру
0x013:    irmovq $10,%rdx  # Точка возврата
0x01d:    halt
0x020:    .pos 0x20
0x020:    proc:           # proc:
0x020:    ret              # Немедленно вернуться
0x021:    rrmovq %rdx,%rbx # Не выполняется
0x030:    .pos 0x30
0x030:    stack:         # stack: Указатель стека

```

На рис. 4.48 показано, как по нашему мнению конвейер должен обрабатывать инструкцию `ret`. Так же как на предыдущих временных диаграммах, здесь показаны действия, выполняемые конвейером с течением времени (слева вправо). Но, в отличие от предыдущих примеров, здесь инструкции выполняются не в том порядке, в каком они встречаются в программе, потому что эта программа предусматривает выполнение инструкций не в линейном порядке. Для правильной идентификации инструкций следите за их адресами в программе.

# prog6

0x000: `irmovq Stack,%rsp`

0x00a: `call proc`

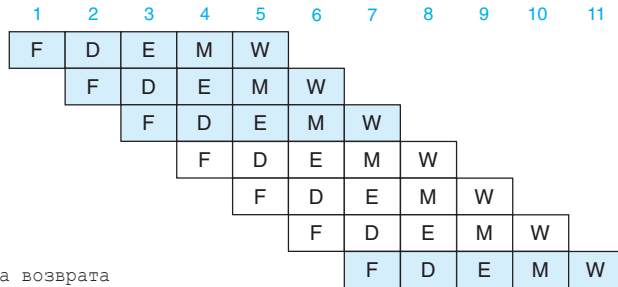
0x020: `ret`

*пузырек*

*пузырек*

*пузырек*

0x013: `irmovq $10,%rdx` # Точка возврата



**Рис. 4.48.** Упрощенная схема обработки инструкции `ret`. Конвейер должен быть приостановлен, пока `ret` пройдет стадии декодирования, выполнения и обращения к памяти, за счет добавления трех «пузырьков». Логика выбора РС получит адрес возврата для выборки следующей инструкции, как только `ret` достигнет стадии обратной записи (цикл 7)

Как показано на этой диаграмме, инструкция `ret` выбирается в цикле 3 и продолжает движение по конвейеру до этапа обратной записи в цикле 7. Пока она пересекает этапы декодирования, выполнения и обращения к памяти, конвейер не может выполнять каких-либо полезных действий. Вместо этого в него вводятся три «пузырька». Как только инструкция `ret` достигнет этапа обратной записи, логика выбора РС запишет в счетчик инструкций адрес возврата и этап выборки извлечет инструкцию `irmovq` в точке возврата (адрес 0x013).

Теперь рассмотрим пример неверного предсказания ветвления. Ниже приводится пример ассемблерного кода программы с адресами инструкций слева для справки:

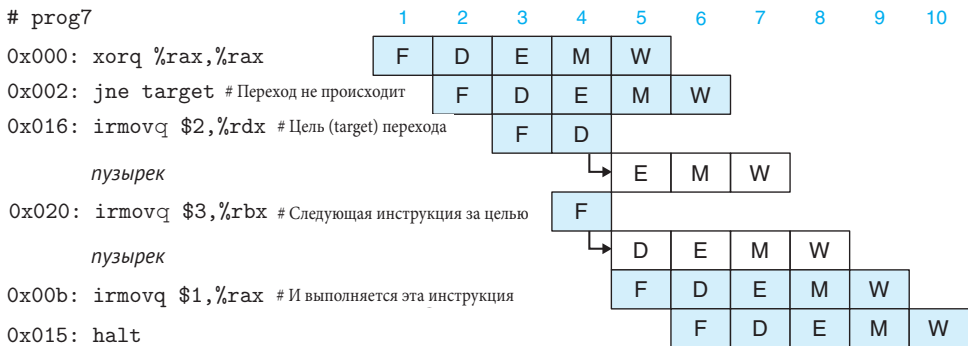
```

0x000:    xorq %rax,%rax
0x002:    jne target        # Переход не происходит
0x00b:    irmovq $1, %rax      # И выполняется эта инструкция
0x015:    halt
0x016:    target:
0x016:    irmovq $2, %rdx      # Цель (target) перехода
0x020:    irmovq $3, %rbx      # Следующая инструкция за целью
0x02a:    halt

```

На рис. 4.49 показано, как обрабатываются эти инструкции. Так же как раньше, инструкции перечисляются в порядке поступления в конвейер, а не в порядке появления в

программе. Поскольку для инструкций перехода используется стратегия прогнозирования «выполняется всегда», в цикле 3 будет выбрана инструкция, находящаяся по адресу перехода, а в цикле 4 – следующая за ней. К тому времени, когда логика этапа выполнения обнаружит, что переход не должен выполняться (это произойдет в цикле 4), в конвейере уже будут находиться две инструкции, выполнение которых не должно продолжиться. К счастью, ни одна из этих инструкций не привела к изменению состояния, видимого программисту. Это может произойти, только когда инструкция достигнет стадии выполнения, где некоторые инструкции способны вызвать изменение флагов. На этом этапе конвейер может просто *отменить* (то есть просто выбросить) две неправильно выбранные инструкции, вставив «пузырьки» в следующем цикле в этапы декодирования и выполнения, одновременно выбирая инструкцию, следующую за инструкцией перехода. Две ошибочно выбранные инструкции просто исчезнут из конвейера и не окажут никакого влияния на видимое программистом состояние. Единственный недостаток такого подхода – два потраченных впустую цикла.



**Рис. 4.49.** Обработка неверно предсказанной инструкции ветвления. Конвейер предсказывает, что ветвление произойдет, поэтому выбирает инструкцию в целевой точке перехода. До того, как в цикле 4, когда инструкция перехода оказывается на этапе выполнения, обнаружится ошибочность предсказания, конвейер успевает выбрать две инструкции. В цикле 5 конвейер «отменяет» обработку двух инструкций, вставляя «пузырьки» в этапы декодирования и выполнения, и выбирает инструкцию, следующую за инструкцией перехода

Это обсуждение рисков по управлению показывает, что с ними можно справиться, внимательно изучив логику управления конвейером. Такие методы, как останов и внедрение «пузырьков», позволяют динамически регулировать поток инструкций в конвейере при возникновении особых условий. Как мы обсудим в разделе 4.5.8, простое расширение базовой конструкции синхронизированных регистров позволит нам останавливать этапы и вводить «пузырьки» в конвейерные регистры в ходе выполнения логики управления конвейером.

### 4.5.6. Обработка исключений

Как мы обсудим в главе 8, какие-то операции, выполняемые процессором, могут привести к *исключительной ситуации*, когда нормальный ход выполнения программы нарушается. Причина исключения может быть *внутренней*, скрытой в выполняющейся программе, или *внешней*, обусловленной каким-либо внешним сигналом. Наш архитектурный набор команд допускает три различных внутренних исключения, которые могут быть вызваны (1) инструкцией halt, (2) инструкцией с недопустимой комбинацией кода инструкции и кода функции и (3) попыткой обращения к недопустимому адресу,



например при выборке очередной инструкции или при попытке прочитать либо записать данные. Более совершенная архитектура процессора также должна предусматривать обработку внешних исключений, например когда процессор получает сигнал о том, что сетевой интерфейс получил новый пакет или пользователь щелкнул кнопкой мыши. Правильная обработка исключений – сложный аспект конструкции любого микропроцессора. Исключения могут возникать в непредсказуемое время и требуют полного прерывания потока инструкций, движущихся через конвейер процессора. Наш пример обработки трех внутренних исключений дает лишь весьма общее представление об истинной сложности правильного обнаружения и обработки исключений.

Далее мы будем называть инструкцию, вызывающую исключение, *исключительной инструкцией*. В случае обращения по недопустимому адресу в памяти фактической исключительной инструкции не существует, и в такой ситуации полезно представить, что по недопустимому адресу находится своего рода «виртуальная исключительная инструкция». Начиная обсуждение, мы решили, что в нашей упрощенной модели ISA процессор должен останавливаться при появлении исключения и устанавливать соответствующий код состояния, как показано в табл. 4.2. То есть код, предшествующий исключительной инструкции, должен быть выполнен полностью, но ни одна из последующих инструкций не должна оказывать никакого влияния на состояние, видимое программисту. В более совершенной архитектуре процессор продолжит работу, вызвав обработчик исключений – процедуру, которая является частью операционной системы, – но реализация такой обработки исключений выходит за рамки нашего обсуждения.

В конвейерной системе обработка исключений имеет ряд тонкостей. Во-первых, исключения могут быть вызваны несколькими инструкциями одновременно. Например, в течение одного цикла на этапе выборки может быть получена инструкция `halt`, а на этапе обращения к памяти может обрабатываться инструкция, обращающаяся к недопустимому адресу. Мы должны определить, о каком из этих исключений процессор должен сообщить операционной системе. Основное правило – больший приоритет должно иметь исключение, инициированное инструкцией, которая продвинулась дальше всего по конвейеру. В приведенном выше примере это будет исключение обращения к недопустимому адресу, предпринятого инструкцией на этапе обращения к памяти. С точки зрения программы на машинном языке, инструкция на этапе обращения к памяти должна выглядеть как выполненная раньше инструкции на этапе выборки, поэтому операционная система должна быть уведомлена только об этом исключении.

Вторая тонкость возникает, когда выбранная и начавшая выполнение инструкция вызывает исключение, а затем отменяется из-за неверно предсказанного перехода. Вот пример такой программы в форме объектного кода:

```
0x000: 6300          | xorq %rax,%rax
0x002: 7416000000000000 | jne target      # Переход не происходит
0x00b: 30f001000000000000 | irmovq $1, %rax # И выполняется эта инструкция
0x015: 00            | halt
0x016:              | target:
0x016: ff          | .byte 0xFF      # Недопустимый код инструкции
```

В этой программе выбранная нами стратегия предсказания прогнозирует, что переход будет выполнен, поэтому процессор попытается выбрать и использовать байт со значением `0xFF` как инструкцию (в ассемблерном коде сгенерирован с использованием директивы `.byte`). В результате этап декодирования обнаружит недопустимую инструкцию. Позже конвейер обнаружит, что переход не должен выполняться, и поэтому инструкция по адресу `0x016` никогда не должна была выбираться. Логика управления конвейером отменит эту инструкцию, поэтому желательно, чтобы исключение не было вызвано.

Третья тонкость возникает из-за того, что различные части состояния системы обновляются конвейерным процессором на разных этапах. Возможно, что инструкция, следующая за инструкцией, вызывающей исключение, изменит какую-то часть состояния до завершения обработки исключительной инструкции. Например, рассмотрим следующую последовательность инструкций, в которой предполагается, что пользовательским программам запрещен доступ к верхним адресам 64-разрядного диапазона:

```

1  irmovq $1,%rax
2  xorq   %rsp,%rsp # Записать 0 в %rsp и биты 100 в регистре флагов
3  pushq  %rax      # Попытаться выполнить запись в адрес 0xfffffffffffff8
4  addq   %rax,%rax  # (Не должна выполняться) Устанавливает биты 000
                        # в регистре флагов

```

Инструкция `pushq` вызывает исключение обращения к недопустимому адресу, потому что уменьшение указателя стека приведет к тому, что он получит значение `0xfffffffffffff8`. Это исключение обнаружится на этапе обращения к памяти. В том же цикле на этапе выполнения будет находиться инструкция `addq`, которая изменяет регистр флагов. Эта последовательность действий нарушит наше требование, согласно которому ни одна инструкция, следующая за исключительной инструкцией, не должна оказывать никакого влияния на состояние системы.

Как было показано выше, сделать правильный выбор из нескольких различных исключений и избежать создания исключений для инструкций, которые выбираются из-за неверно предсказанного ветвления, нам помогло включение логики обработки исключений в структуру конвейера. Это побуждает нас включить также код состояния **stat** в каждый из имеющихся конвейерных регистров (рис. 4.34 и 4.45). Если инструкция генерирует исключение на каком-то этапе, то поле состояния устанавливается так, чтобы указать природу исключения. Состояние исключения передается по конвейеру с остальной информацией для этой инструкции до этапа обратной записи. На этом этапе логика управления конвейером обнаруживает исключение и останавливает выполнение.

Чтобы избежать изменения состояния, видимого программисту, последующими инструкциями, логика управления конвейером должна запрещать любое обновление регистра флагов или памяти данных, когда инструкция вызвала исключение на этапе обращения к памяти или обратной записи. В примере выше управляющая логика обнаружит, что `pushq` вызвала исключение на этапе обращения к памяти, поэтому изменение регистра флагов на этапе выполнения инструкции `addq` будет заблокировано.

Давайте посмотрим, как этот метод обработки исключений справляется с упомянутыми тонкостями. Когда исключение возникает на одном или нескольких этапах конвейера, информация просто сохраняется в полях **stat** конвейерных регистров. Событие не влияет на поток инструкций в конвейере, пока исключительная инструкция не достигнет конечного этапа, кроме запрета на изменение видимого программисту состояния (регистра флагов и памяти) последующими инструкциями в конвейере. Поскольку инструкции достигают этапа обратной записи в том же порядке, в каком они выполнялись бы в процессоре без конвейера, мы гарантируем, что первая инструкция, обнаружившая исключение, прибудет на этап обратной записи первой, после чего выполнение программы может остановиться, и код состояния в конвейерном регистре *W* можно зафиксировать как код состояния программы. Если какая-то инструкция была выбрана и потом отменена, любая информация о состоянии исключения, сгенерированного этой инструкцией, также будет отменена. Никакая инструкция, следующая за исключительной инструкцией, не сможет изменить видимое программисту состояние. Простое правило передачи по конвейеру состояния исключения вместе с остальной информацией об инструкции обеспечивает простой и надежный механизм обработки исключений.

### 4.5.7. Реализация этапов в PIPE

Мы закончили обсуждение деталей версии PIPE конвейерного процессора Y86-64 с продвижением. В ней используется тот же набор аппаратных модулей, что и в последовательных версиях SEQ, но дополнительно были добавлены конвейерные регистры, а также реорганизованы некоторые логические блоки и добавлены новые блоки управляющей логики. В данном разделе мы пройдем через процесс проектирования различных логических блоков и отложим проектирование управляющей логики до следующего раздела. Многие логические блоки идентичны аналогичным блокам в версиях SEQ и SEQ+, отличаясь только выбором соответствующих версий сигналов из конвейерных регистров (их названия начинаются с заглавной буквы с именем конвейерного регистра) или этапов (их названия начинаются с первой буквы в нижнем регистре в названии этапа).

Для примера сравним код HCL, описывающий логику, которая генерирует сигнал **srcA** в версии SEQ, с соответствующим кодом в версии PIPE:

```
# Код для версии SEQ
word srcA = [
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;
    icode in { IPOPQ, IRET } : RRSP;
    1 : RNONE; # Регистры не используются
];

# Код для версии PIPE
word d_srcA = [
    D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : D_rA;
    D_icode in { IPOPQ, IRET } : RRSP;
    1 : RNONE; # Регистры не используются
];
```

Они отличаются только префиксом в именах сигналов в версии PIPE: **D\_** указывает, что источником значения является конвейерный регистр **D**, и **d\_** указывает, что результат сгенерирован на этапе декодирования (**decode**). Чтобы не повторяться, мы не будем показывать код HCL для блоков, отличающихся от блоков в версии SEQ только префиксами в именах. Для справки полный код HCL вы найдете в приложении в интернете «ARCH:HCL» (раздел 4.6).

### Выбор PC и этап выборки

На рис. 4.50 показана подробная схема этапа выборки в версии PIPE. Как уже обсуждалось выше, этот этап также должен выбрать текущее значение для счетчика инструкций и спрогнозировать следующее значение PC. Аппаратные модули для чтения инструкции из памяти и извлечения различных ее полей остались теми же, что и в версии SEQ (описание этапа выборки приводится в разделе 4.3.4).

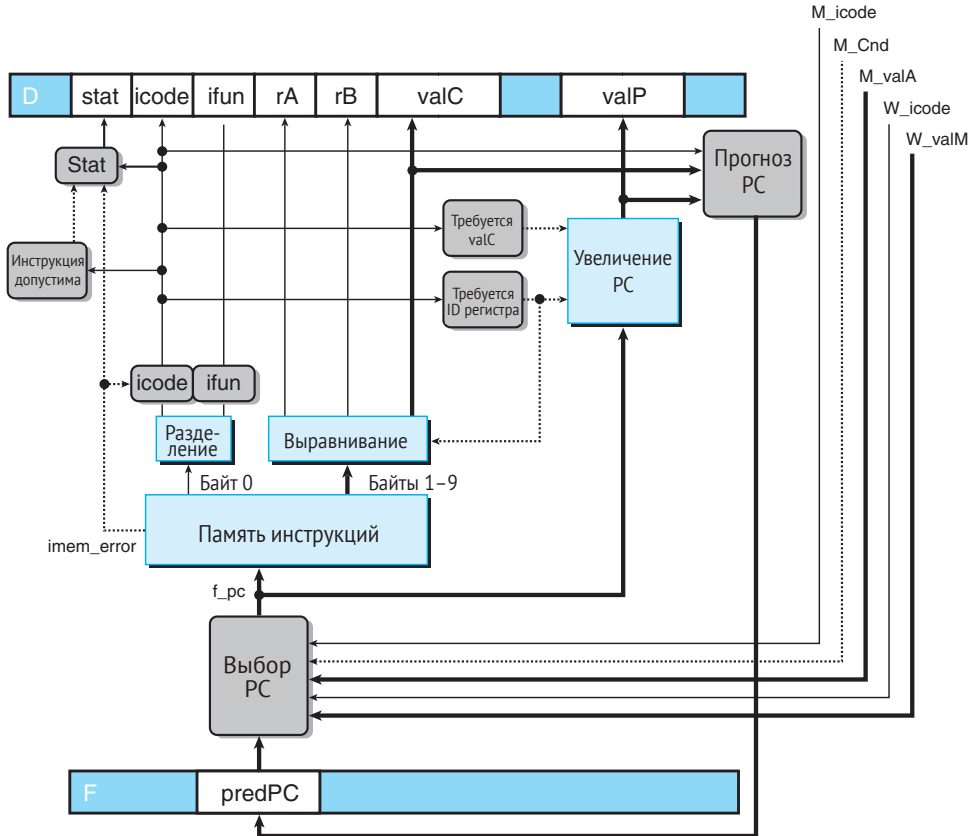
Логика выбора значения для PC выбирает из трех источников. Когда неверно спрогнозированная инструкция ветвления достигнет этапа обращения к памяти, значение **valP** для нее (с адресом следующей инструкции) будет прочитано из конвейерного регистра **M** (сигнал **M\_valA**). Когда инструкция **ret** достигнет этапа обратной записи, адрес возврата будет прочитан из конвейерного регистра **W** (сигнал **W\_valM**). Во всех других случаях спрогнозированное значение счетчика инструкций будет прочитано из конвейерного регистра **F** (сигнал **F\_predPC**):

```
word f_pc = [
    # Неверно спрогнозированное ветвление.
    # Выбрать увеличенное значение PC
    M_icode == IJXX && !M_Cnd : M_valA;
```

```

# Завершение обработки инструкции RET
W_icode == IRET : W_valM;
# По умолчанию: использовать спрогнозированное значение PC
1 : F_predPC;
];

```



**Рис. 4.50.** Логика выборки инструкции и выбора PC в версии PIPE. В пределах времени одного цикла процессор может только предсказать адрес следующей инструкции

Логика прогнозирования PC выбирает **valC** для текущей выбранной инструкции, если это инструкция вызова процедуры или перехода, и **valP** в противном случае:

```

word f_predPC = [
    f_icode in { IJXX, ICALL } : f_valC;
    1 : f_valP;
];

```

Логические блоки, подписанные как «Инструкция допустима», «Требуется ID регистра» и «Требуется valC», подобны аналогичным блокам в версии SEQ с соответствующими именами сигналов.

В этой версии, в отличие от SEQ, необходимо разделить вычисление состояния инструкции на две части. На этапе выборки можно проверить ошибку доступа к памяти, если адрес инструкции выйдет за пределы допустимого диапазона, допустимость инструкции или инструкцию halt. Проверку обращения к недопустимому адресу в памяти данных следует отложить до этапа обращения к памяти.

**Упражнение 4.30 (решение в конце главы)**

Напишите код HCL для сигнала **f\_stat**, сообщающий предварительное состояние для выбранной инструкции.

**Упражнение 4.31 (решение в конце главы)**

Блок **dstE** на этапе декодирования генерирует идентификатор регистра для порта E блока регистров, исходя из полей в конвейерном регистре D, соответствующих выбранной инструкции. В HCL-описании для версии PIPE результирующий сигнал называется **d\_dstE**. Напишите код HCL для этого сигнала, опираясь на HCL-описание сигнала **dstE** в версии SEQ (описание этапа декодирования в разделе 4.3.4). Пока оставьте в стороне логику условного перемещения.

**Этапы декодирования и обратной записи**

На рис. 4.51 приводится подробная схема логики этапов декодирования и обратной записи в версии PIPE. Блоки, обозначенные как **dstE**, **dstM**, **scrA** и **scrB**, очень похожи на аналогичные блоки в последовательной реализации SEQ. Обратите внимание, что идентификаторы регистров, переданные на входы портов записи, поступают с этапа обратной записи (сигналы **W\_dstE** и **W\_dstM**), а не с этапа декодирования. Это объясняется тем, что запись должна осуществляться в регистры-приемники, указанные в инструкции, находящейся на этапе обратной записи.

Основная сложность данного этапа сосредоточена в логике продвижения. Как уже упоминалось, блок «Sel+Fwd A» преследует две цели: объединяет сигнал **valP** с сигналом **valA** для последующих этапов, чтобы уменьшить объем информации о состоянии для хранения в конвейерном регистре, а также реализует логику продвижения операнда-источника **valA**.

Объединение сигналов **valA** и **valP** основано на том факте, что только инструкциям вызова процедур и переходов требуется значение **valP** на последующих этапах, и им нужно значение, прочитанное из порта A блока регистров. Таким выбором управляет сигнал **icode** данного этапа. Когда сигнал **D\_icode** совпадает с кодом инструкции **call** или **jXX**, этот блок выбирает **D\_valP**.

Как упоминалось в разделе 4.5.5, существуют пять разных источников для продвижения, каждый со словом данных и идентификатором регистра-приемника (см. табл. 4.15):

Слово данных	Идентификатор регистра	Описание источника
e_valE	e_dstE	Выход АЛУ
m_valM	M_dstM	Значение в ячейке памяти
M_valE	M_dstE	Ожидающая запись в порт E на этапе обращения к памяти
W_valM	W_dstM	Ожидающая запись в порт M на этапе обратной записи
W_valE	W_dstE	Ожидающая запись в порт E на этапе обратной записи

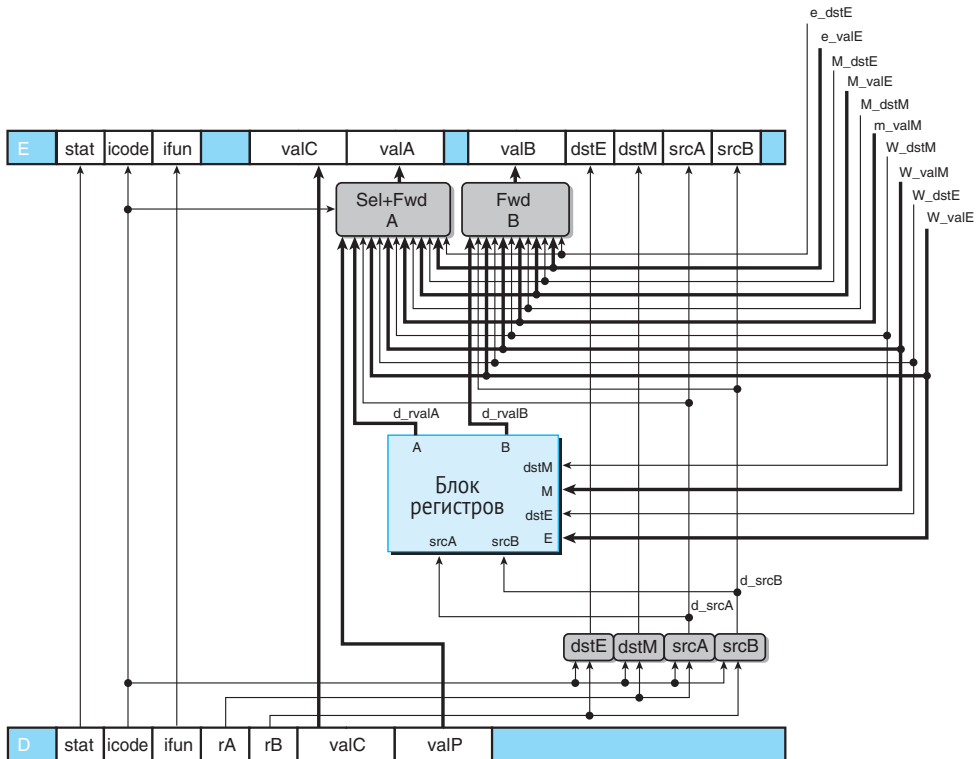
Если ни одно из условий продвижения не выполняется, тогда блок должен выбрать **d\_rvalA** – значение, прочитанное из порта A блока регистров.

Объединив все вместе, получаем HCL-описание нового значения **valA** для конвейерного регистра E:

```

word d_valA = [
  D_icode in { ICALL, IJXX } : D_valP; # Использовать увеличенное значение PC
  d_srcA == e_dstE : e_valE; # Продвинуть valE с этапа выполнения
  d_srcA == M_dstM : m_valM; # Продвинуть valM с этапа обращения к памяти
  d_srcA == M_dstE : M_valE; # Продвинуть valE с этапа обращения к памяти
  d_srcA == W_dstM : W_valM; # Продвинуть valM с этапа обратной записи
  d_srcA == W_dstE : W_valE; # Продвинуть valE с этапа обратной записи
  1 : d_rvalA; # Использовать значение, прочитанное из блока регистров
];

```

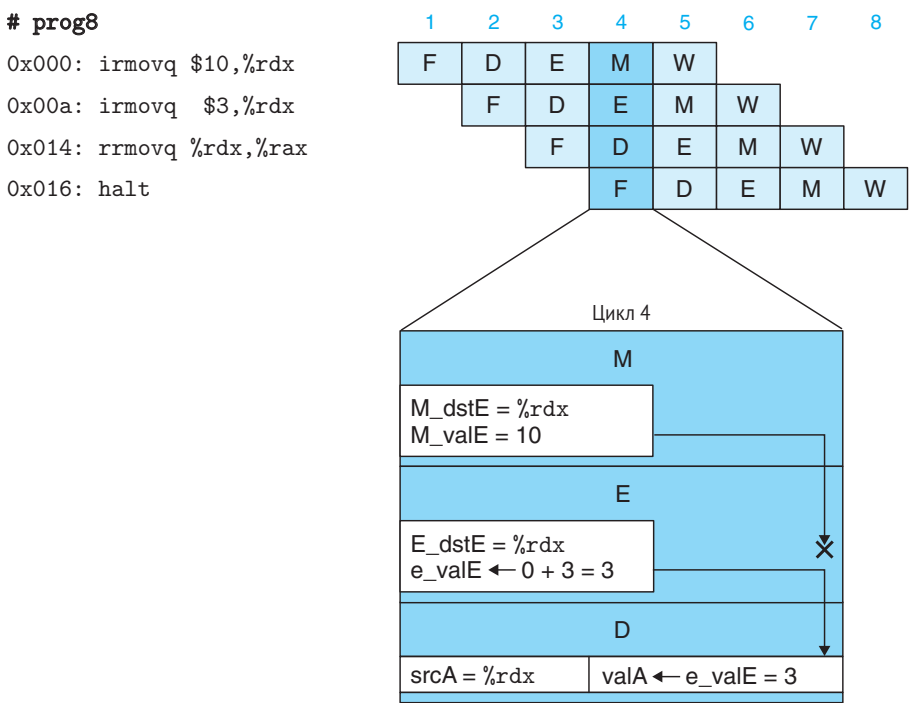


**Рис. 4.51.** Логика этапов декодирования и обратной записи в версии PIPE.

Никакая инструкция не требует одновременно двух значений, `valP` и прочитанного из порта A блока регистров, поэтому эти два значения можно объединить в один сигнал `valA` для последующих этапов. Эту задачу решает блок «Sel+Fwd A», он также реализует логику продвижения операнда-источника `valA`. Блок «Fwd B» реализует логику продвижения операнда-источника `valB`. Регистры-приемники определяются сигналами `dstE` и `dstM` из этапа обратной записи, а не из этапа декодирования, потому что результаты записываются в операнды, указанные в инструкции, которая в настоящий момент находится на этапе обратной записи

Порядок следования вариантов продвижения пяти источников в этом HCL-коде очень важен. Он соответствует порядку, в каком проверяются пять идентификаторов регистра-приемника. При выборе иного порядка, отличного от показанного, для некоторых программ конвейер может работать некорректно. На рис. 4.52 показан пример программы, для которой важен порядок выбора источника для продвижения на этапах выполнения и обращения к памяти. В этой программе первые две инструкции

выполняют запись в регистр `%rdx`, а третья использует этот регистр в качестве операнда-источника. Когда инструкция `rrmovq` достигает этапа декодирования в цикле 4, логика продвижения должна выбрать одно из двух значений для регистра-источника. Какое же значение выбрать? Чтобы разобраться в этом, нужно рассмотреть поведение программы, когда она выполняется по одной инструкции за раз. Первая инструкция `irmovq` запишет в регистр `%rdx` число 10, вторая запишет в этот же регистр число 3, а затем инструкция `rrmovq` прочтает из `%rdx` значение 3. Чтобы повторить это поведение, конвейерная реализация должна всегда продвигать источник с самого раннего этапа конвейера, потому что он обрабатывает самую последнюю инструкцию, изменяющую значение регистра. Поэтому HCL-код, показанный выше, сначала проверит продвижение источника на этапе выполнения, затем на этапе обращения к памяти и, наконец, на этапе обратной записи. Порядок выбора между двумя источниками для продвижения на этапах обращения к памяти и обратной записи имеет значение только для инструкции `rrmovq %rsp`, потому что только она может одновременно осуществлять две операции записи в один регистр.



**Рис. 4.52.** Демонстрация важности порядка выбора источника для продвижения. В цикле 4 значения для `%rdx` доступны на двух этапах, выполнения и обращения к памяти. Логика продвижения должна выбрать источник с этапа выполнения, потому что это последнее сгенерированное значение для данного регистра

**Упражнение 4.32 (решение в конце главы)**

Предположим, что мы поменяли местами третий и четвертый варианты (оба продвигают разные значения с этапа обращения к памяти) в HCL-коде с определением `d_valA`. Опишите поведение инструкции `rrmovq` (строка 5) в следующей программе:

```

1 irmovq $5, %rdx
2 irmovq $0x100, %rsp
3 rmmovq %rdx, 0(%rsp)
4 popq %rsp
5 rrmovq %rsp, %rax

```

#### Упражнение 4.33 (решение в конце главы)

Предположим, что мы поменяли местами пятый и шестой варианты (оба продвигают разные значения с этапа обратной записи) в HCL-коде с определением **d\_valA**. Напишите программу для Y86-64, которая бы выполнялась некорректно. Опишите, какая ошибка произойдет и когда, как она повлияет на поведение программы.

#### Упражнение 4.34 (решение в конце главы)

Напишите HCL-код для сигнала **d\_valB**, выбирающий значение для операнда-источника **valB**, переданного в конвейерный регистр E.

#### Упражнение 4.35 (решение в конце главы)

Наш второй случай в HCL-определении **d\_valA**, где используется сигнал **e\_dstE**, чтобы определить, следует ли выбирать выход **e\_valE** арифметико-логического устройства (АЛУ) для продвижения в качестве источника. Предположим, что вместо него в выборе используется сигнал **E\_dstE** – идентификатор регистра-приемника в конвейерном регистре E. Напишите программу для Y86-64, которая дала бы неверный результат с этой измененной логикой продвижения.

Нам осталось совсем чуть-чуть, чтобы завершить проектирование этапа обратной записи. Как показано на рис. 4.45, общее состояние процессора **Stat** вычисляется блоком на основе значения состояния в конвейерном регистре W. Как рассказывалось в разделе 4.1.1, код состояния должен содержать признак нормальной работы (AOK) или одно из трех исключений. Поскольку конвейерный регистр W хранит состояние последней обработанной инструкции, естественно использовать это значение в качестве признака общего состояния процессора. Единственный особый случай, который следует учесть, – «пузырек» на этапе обратной записи. Наличие «пузырька» соответствует нормальной работе, поэтому код состояния AOK должен устанавливаться и для этого случая:

```

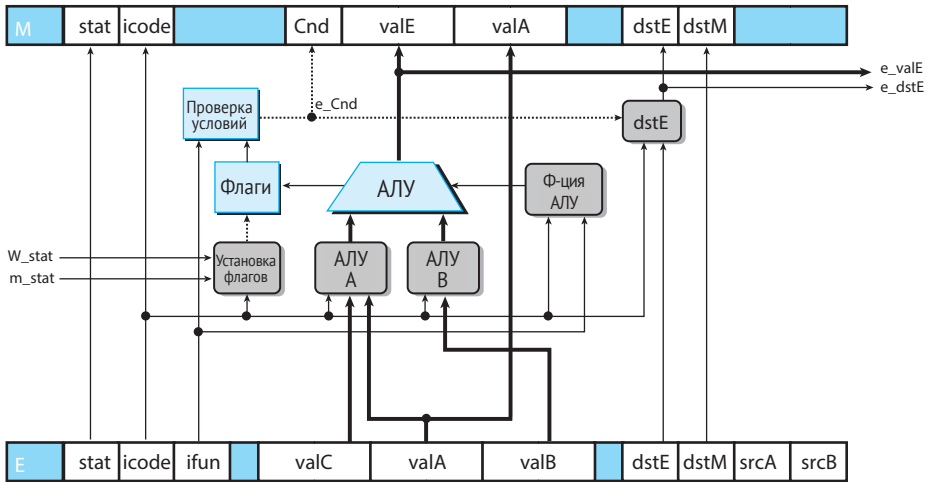
word Stat = [
    W_stat == SBUB : SAOK;
    1 : W_stat;
];

```

### Этап выполнения

На рис. 4.53 показана схема работы этапа выполнения в версии PIPE. Аппаратные модули и логические блоки остались теми же, что и в версии SEQ, с соответствующим переименованием сигналов. На схеме видно, что сигналы **e\_valE** и **e\_dstE** продвигаются в этап декодирования, как один из источников. Единственное отличие заключается в наличии блока, подписанного как «Установка флагов», который принимает на входе сигналы **m\_stat** и **W\_stat** и определяет необходимость изменения флагов. Эти сигналы помогают обнаружить случаи, когда инструкция, вызывающая исключение, проходит через более поздние этапы конвейера и любое изменение флагов должно быть подавлено. Этот аспект проекта обсуждается в разделе 4.5.8.

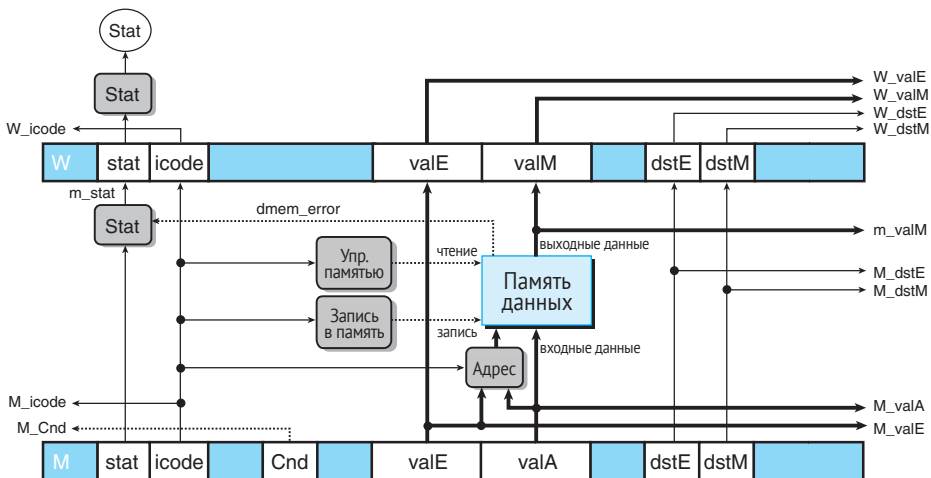




**Рис. 4.53.** Логика этапа выполнения в версии PIPE.  
Она очень похожа на логику реализации SEQ

### Этап обращения к памяти

На рис. 4.54 показана логика этапа обращения к памяти для версии PIPE. Если сравнить ее с этапом обращения к памяти в версии SEQ (рис. 4.22), то можно заметить, что в версии PIPE отсутствует блок «Данные», имеющийся в версии SEQ. Этот блок использовался для выбора между источниками данных **valP** (в инструкциях **call**) и **valA**. Теперь этот выбор осуществляется на этапе декодирования блоком «Sel+Fwd A». Все остальные блоки на этом этапе идентичны аналогичным блокам в версии SEQ, с соответствующим переименованием сигналов. На этой схеме также видно, что многие из значений в конвейерных регистрах M и W передаются в другие части цепи как часть передающей логики продвижения и управления конвейером.



**Рис. 4.54.** Логика этапа обращения к памяти в версии PIPE. Многие сигналы из конвейерных регистров M и W продвигаются на более ранние этапы для передачи результатов обратной записи и адресов команд

**Упражнение 4.36 (решение в конце главы)**

На этом этапе мы можем завершить вычисление кода состояния **Stat** и определить случай использования недопустимого адреса в памяти данных. Напишите код HCL для сигнала **m\_stat**.

## 4.5.8. Управляющая логика конвейера

Теперь мы готовы завершить проектирование PIPE созданием управляющей логики конвейера. Эта логика должна обрабатывать следующие четыре случая, для которых недостаточно других механизмов, таких как продвижение данных и прогнозирование ветвления:

- *риски загрузки и использования.* Конвейер должен приостанавливаться на один цикл между инструкцией, читающей значение из памяти, и инструкцией, использующей это значение;
- *обработка ret.* Конвейер должен приостанавливаться по достижении инструкцией **ret** этапа обратной записи;
- *неверно спрогнозированное ветвление.* К тому времени, как логика ветвления обнаружит, что переход выполняться не должен, несколько инструкций, находящихся по адресу перехода, начнут прохождение конвейера. Эти инструкции необходимо удалить из конвейера и начать выборку инструкций, следующих за инструкцией перехода;
- *исключения.* Когда инструкция вызывает исключение, необходимо предотвратить обновление состояния, видимое программисту, более поздними инструкциями и остановить выполнение, как только исключительная инструкция достигнет этапа обратной записи.

Сначала мы рассмотрим, какие действия необходимо выполнить в каждом отдельном случае, а затем спроектируем управляющую логику для каждого из них.

### Обработка особых случаев

В разделе 4.5.5 мы описали желаемый порядок обработки рисков загрузки/использования (рис. 4.47). Данные из памяти читают только инструкции **lrmovq** и **rrmq**. Когда (1) любая из них находится на этапе выполнения и (2) инструкция, использующая регистр-приемник, находится на этапе декодирования, вторую инструкцию следует задержать на этапе декодирования и вместо нее в следующем цикле передать «пузырек» на этап выполнения. После этого логика продвижения устранил риск по данным. Конвейер может удерживать команду на этапе декодирования, сохраняя конвейерный регистр **D** в фиксированном состоянии. При этом он должен также зафиксировать конвейерный регистр **F**, чтобы следующая инструкция была выбрана повторно. Таким образом, для реализации подобного поведения требуется, обнаружив состояние риска, зафиксировать конвейерные регистры **F** и **D** и передать «пузырек» на этап выполнения.

Желаемый порядок обработки инструкции **ret** мы описали в разделе 4.5.5. Конвейер должен остановиться на три цикла, пока не будет прочитан адрес возврата после перехода инструкции **ret** на этап обращения к памяти. Это было проиллюстрировано упрощенной схемой конвейера на рис. 4.48 на примере обработки следующей программы:

```
0x000:    irmovq stack,%rsp    # Инициализировать указатель стека
0x00a:    call proc           # Вызвать процедуру
```

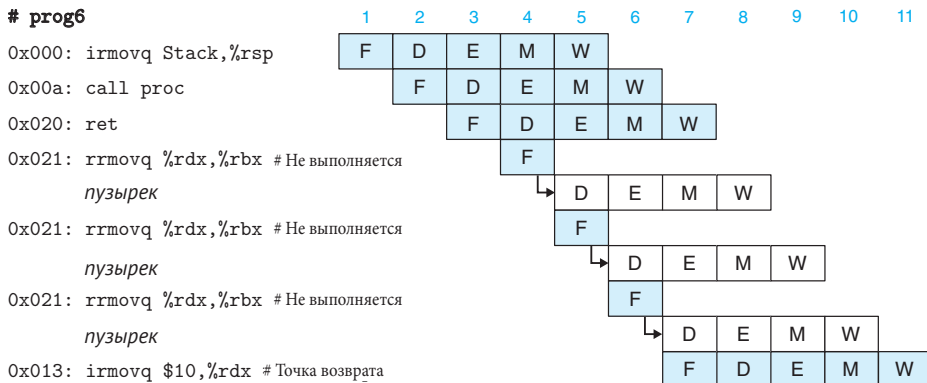
```

0x013:    irmovq $10,%rdx    # Точка возврата
0x01d:    halt
0x020:    .pos 0x20
0x020:    proc:              # proc:
0x020:    ret                # Немедленно вернуться
0x021:    rrmovq %rdx,%rbx   # Не выполняется
0x030:    .pos 0x30
0x030:    stack:            # stack: Указатель стека

```

На рис. 4.55 показан требуемый порядок обработки инструкции `ret` на примере выполнения программы. Обратите внимание, что в этом случае нет возможности передать «пузырек» на этап выборки конвейера. В каждом цикле этап выборки читает *некоторую* инструкцию из памяти команд. Глядя на код HCL с реализацией логики предсказания PC в разделе 4.5.7, можно заметить, что для инструкции `ret` новое значение PC прогнозируется как **valP** – адрес следующей инструкции. В нашем примере программы это будет адрес `0x021` инструкции `rrmovq`, следующей за `ret`. В данном примере это предсказание неверно и не будет верным в большинстве случаев, но мы и не пытаемся предсказать верный адреса возврата. В течение трех циклов синхронизации этап выборки будет снова и снова выбирать инструкции `rrmovq`, заменяя ее «пузырьком» на этапе декодирования. Этот процесс показан на рис. 4.55 в виде трех выборок со стрелками, ведущими вниз к «пузырькам», проходящим через остальные этапы конвейера. Наконец, в цикле 7 будет выбрана инструкция `irmovq`. Сравнивая рис. 4.55 с рис. 4.48, можно заметить, что наша реализация достигает желаемого эффекта, но за счет выборки неверной инструкции в течение трех циклов.

Порядок обработки неверно предсказанного ветвления был описан в разделе 4.5.5 и проиллюстрирован на рис. 4.49. Ошибочное предсказание будет обнаружено, когда инструкция перехода достигнет стадии выполнения. После этого в следующем цикле управляющая логика внедрит «пузырьки» в этапы декодирования и выполнения, отменяя две ошибочно выбранные инструкции. В том же цикле конвейер прочитает правильную инструкцию на этапе выборки.



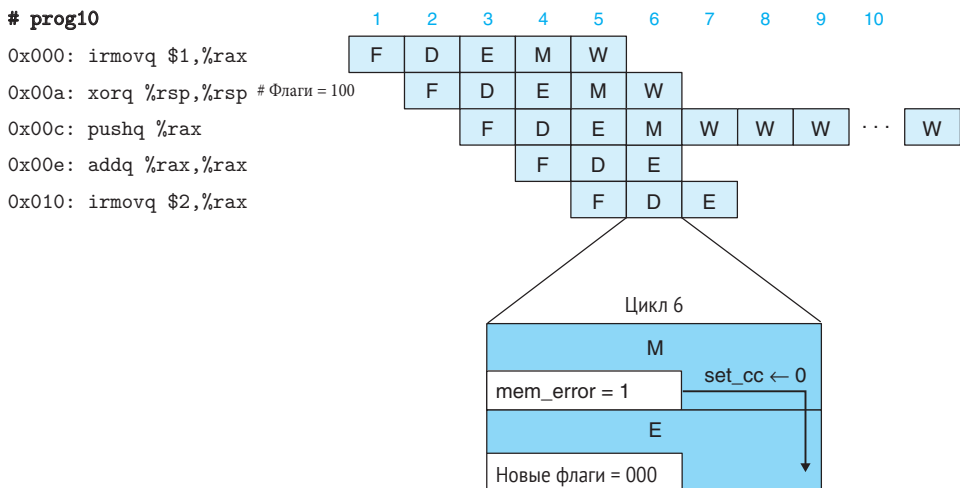
**Рис. 4.55.** Схема обработки инструкции `ret`. Этап выборки многократно извлекает инструкцию `rrmovq`, следующую за инструкцией `ret`, но затем логика управления конвейером передает «пузырек» на этап декодирования, не позволяя инструкции `rrmovq` двинуться дальше. В результате получается поведение, эквивалентное показанному на рис. 4.48

В случае с инструкциями, вызывающими исключение, конвейерная реализация должна соответствовать желаемому поведению ISA, согласно которому все предшествующие инструкции должны выполняться до конца и ни одна из последующих не должна

повлиять на состояние программы. Достижение этой цели осложняется тем фактом, что (1) исключения обнаруживаются на двух разных этапах (выборки и обращения к памяти) конвейера и (2) состояние программы обновляется на трех разных этапах (выполнения, обращения к памяти и обратной записи).

Наш проект предполагает наличие кода состояния **stat** в каждом конвейерном регистре для хранения состояния каждой инструкции по мере ее прохождения через этапы конвейера. Когда возникает исключение, процессор сохраняет эту информацию как часть состояния инструкции и продолжает выборку, декодирование и выполнение последующих инструкций, как будто ничего не случилось. Когда исключительная команда достигает этапа обращения к памяти, предпринимаются шаги для предотвращения изменения последующими инструкциями состояния, видимого программисту, путем (1) запрета изменения флагов инструкциями на этапе выполнения, (2) внедрения «пузырьков» в этап обращения к памяти, чтобы запретить любую запись в память данных и (3) остановки этапа обратной записи, когда в него поступает исключительная инструкция, чтобы остановить весь конвейер.

Временная диаграмма на рис. 4.56 показывает, как логика управления конвейером обрабатывает ситуацию, когда за инструкцией, вызывающей исключение, следует инструкция, изменяющая флаги. В цикле 6 инструкция `pushq` достигает этапа обращения к памяти и генерирует ошибку. В том же цикле инструкция `addq` на этапе выполнения генерирует новые значения для регистра флагов. Когда исключительная инструкция оказывается на этапе обращения к памяти или обратной записи, управляющая логика отключает установку флагов (проверяя сигналы **m\_stat** и **W\_stat** и затем устанавливая сигнал **set\_cc** в ноль). Также можно видеть, что на этап обращения к памяти передается «пузырек», исключительная инструкция блокируется на этапе обратной записи (инструкция `pushq` в примере на рис. 4.56) и ни одна из последующих инструкций не проходит этап выполнения.



**Рис. 4.56.** Обработка исключения обращения к недопустимому адресу в памяти.

В цикле 6 недопустимая ссылка на память в инструкции `pushq` приводит к отключению обновления флагов. Конвейер начинает вводить «пузырьки» в этап обращения к памяти и останавливать выполнение исключительной инструкции на этапе обратной записи

Управляя сигналами состояния и запрета изменения флагов, а также этапами конвейера, мы получаем желаемый эффект при появлении исключений: все инструкции,

предшествующие исключительной инструкции, выполняются до конца, и ни одна из последующих инструкций не оказывает никакого влияния на состояние, видимое программисту.

Выявление особых условий управления

В табл. 4.9 приводится обобщенный список условий, требующих особого управления конвейером. Здесь представлены HCL-выражения, описывающие условия, при которых возникают четыре особых случая.

Таблица 4.9. Условия, обнаруживаемые логикой управления конвейером. Четыре разных условия требуют изменения потока конвейера путем его остановки или отмены частично выполненных инструкций

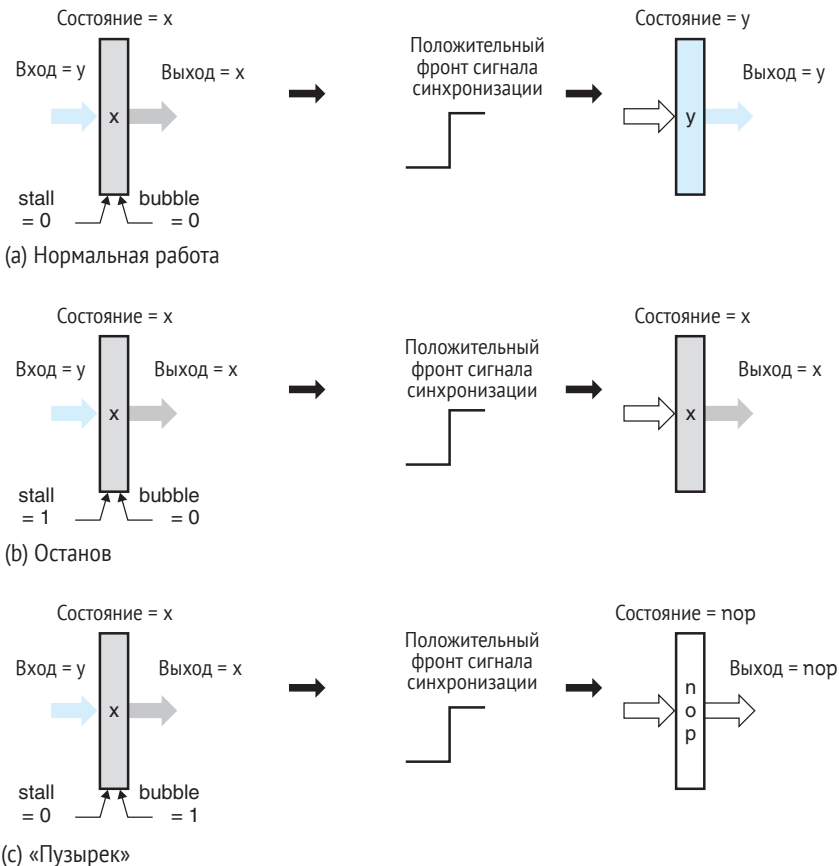
Условие	Как обнаруживается
Обработка инструкции get	$IRET \in \{D\ icode, E\ icode, M\ icode\}$
Риск загрузки/использования	$E\ icode \in \{IMRMOVQ, IPOPOPQ\} \ \&\& \ E\ dstM \in \{d\ srcA, d\ srcB\}$
Неверно предсказанное ветвление	$E\ icode = IJXX \ \&\& \ !e\ Cnd$
Исключение	$m\ stat \in \{SADR, SINS, SHLT\} \   w\ stat \in \{SADR, SINS, SHLT\}$

Эти выражения реализуются простыми блоками комбинаторной логики, которые должны генерировать свои результаты до конца цикла синхронизации, чтобы конвейерные регистры изменили свое состояние с началом следующего цикла, при прохождении положительного фронта сигнала синхронизации. В течение цикла конвейерные регистры D, E и M хранят состояния инструкций, находящихся на этапах декодирования, выполнения и обращения к памяти соответственно. По мере приближения к концу цикла в сигналах **d\_srcA** и **d\_srcB** будут установлены идентификаторы регистров операндов-источников для инструкций на этапе декодирования. Определение инструкции **get** в конвейере осуществляется по коду инструкции на этапах декодирования, выполнения и обращения к памяти. Определение рисков загрузки/использования осуществляется по типу инструкции (**rmovq** или **popq**) на этапе выполнения и путем сравнения регистра-источника и регистра-приемника на этапе декодирования. Управляющая логика конвейера должна выявить неверно спрогнозированное ветвление, пока инструкция перехода находится на этапе выполнения, чтобы создать условия, необходимые для восстановления потока выполнения после ошибочного прогноза, когда инструкция перейдет на этап обращения к памяти. Когда инструкция находится на этапе выполнения, необходимость перехода определяется сигналом **e\_Cnd**. Исключительная инструкция обнаруживается проверкой значений состояния инструкции на этапах обращения к памяти и обратной записи. На этапе обращения к памяти используется сигнал **m\_stat**, вычисляемый внутри этапа, а не **M\_stat** из конвейерного регистра. Этот внутренний сигнал учитывает возможность ошибки обращения по недопустимому адресу в памяти данных.

Механизмы управления конвейером

На рис. 4.57 показаны базовые механизмы, позволяющие управляющей логике конвейера удерживать инструкцию в конвейерном регистре или передавать «пузырек» в конвейер. В этих механизмах задействованы расширения базового синхронизированного регистра, описанного в разделе 4.2.5. Предположим, что каждый конвейерный регистр имеет два управляющих входа: **stall** (останов) и **bubble** («пузырек»). Эти сигналы определяют, как будет изменяться конвейерный регистр по положительному фронту сигнала синхронизации. При нормальном функционировании (рис. 4.57 (а)) оба входа установлены в 0, заставляя регистр загрузить входные данные, определяющие его новое состояние. Когда сигнал **stall** установлен в 1 (рис. 4.57 (b)), обновление состояния не

выполняется. То есть регистр остается в предыдущем состоянии. Это позволяет удерживать инструкцию на некотором этапе. Когда сигнал **bubble** установлен в 1 (рис. 4.57 (с)), регистр получит некоторое фиксированное *пустое состояние*, эквивалентное состоянию команды **por**. Конкретная комбинация единиц и нулей, определяющая пустое состояние, зависит от набора полей в конвейерном регистре. Например, для вставки «пузырька» в конвейерный регистр D необходимо, чтобы поле **icode** имело значение **INOP** (табл. 4.8). Для вставки «пузырька» в конвейерный регистр E поле **icode** должно иметь значение **INOP**, а поля **dstE**, **dstM**, **scrA** и **scrB** – значение **RNONE**. Определение пустого состояния – это одна из задач проектировщика, которую он должен решить при проектировании конвейерного регистра. Мы не будем углубляться в эту тему, а просто будем интерпретировать значение 1 в любом из сигналов как признак ошибки.



**Рис. 4.57.** Дополнительные операции с конвейерным регистром: (а) в нормальных условиях состояние и выход регистра устанавливаются равными значению на входе с положительным фронтом сигнала синхронизации; (б) при работе в режиме останова сохраняется прежнее состояние; (с) при внедрении «пузырька» состояние заменяется состоянием операции **por**

В табл. 4.10 показано, какие действия должны выполнять различные этапы конвейера для каждого из трех особых условий. В каждом случае выполняется некоторая комбинация операций с конвейерными регистрами при нормальных условиях, при условии останова и внедрения «пузырька». С точки зрения синхронизации, управляющие сигналы **stall** и **bubble** генерируются блоками комбинаторной логики. Эти сигналы долж-

ны иметь действительные значения к моменту прохождения положительного фронта сигнала синхронизации, чтобы каждый конвейерный регистр загрузил следующее нормальное состояние, остановился или ввел «пузырек» с началом нового цикла. Вводя подобные мелкие расширения в проект конвейерных регистров, можно реализовать полноценный конвейер, включая все его управление, с использованием базовых блоков комбинаторной логики, синхронизированных регистров и оперативной памяти.

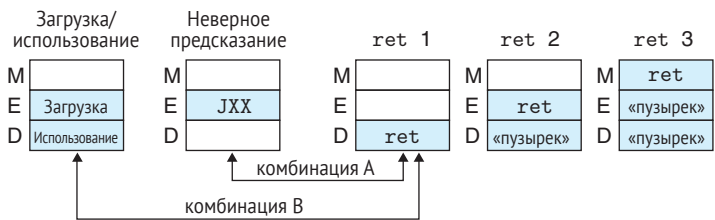
**Таблица 4.10.** Действия логики управления конвейером. Некоторые условия требуют изменения потока конвейера – его остановки или отмены частично выполненных инструкций

Условие	Конвейерные регистры				
	F	D	E	M	W
Обработка инструкции ret	останов	«пузырек»	нормальная работа	нормальная работа	нормальная работа
Риск загрузки/использования	останов	останов	«пузырек»	нормальная работа	нормальная работа
Неверное предсказание ветвления	нормальная работа	«пузырек»	«пузырек»	нормальная работа	нормальная работа

Комбинации особых случаев

До сих пор при обсуждении особых условий управления конвейером предполагалось, что в любом одном цикле синхронизации может возникнуть только один особый случай. Общей ошибкой при проектировании системы является невозможность управлять случаями, когда одновременно возникает несколько особых условий. Проанализируем некоторые из них. Мы не будем беспокоиться о случаях программных исключений, потому что уже достаточно тщательно спроектировали механизм обработки исключений с учетом наличия других инструкций в конвейере.

На рис. 4.58 показана диаграмма состояний конвейера с тремя особыми случаями. На этой диаграмме показаны блоки этапов декодирования, выполнения и обращения к памяти. Заштрихованные блоки обозначают те или иные ограничения, которые должны быть соблюдены для возникновения условия. Риск загрузки/использования определяется наличием инструкции на этапе выполнения, которая читает значение из памяти в регистр, и наличием инструкции на этапе декодирования, которая использует этот регистр в операнде-источнике. Неверно спрогнозированное ветвление определяется наличием инструкции перехода на этапе выполнения. Для инструкции ret есть три возможных варианта особого случая: данная инструкция может находиться на этапе декодирования, выполнения или обращения к памяти. По мере прохождения инструкции ret по конвейеру более ранние этапы будут заполняться «пузырьками».



**Рис. 4.58.** Состояние конвейера в особых случаях.

Пары случаев, указанные стрелками, могут возникать одновременно

На диаграмме видно, что большинство особых случаев являются взаимоисключающими. Например, особые случаи риска загрузки/использования не могут возникать одновременно с неверно спрогнозированным ветвлением, потому что в первом случае на этапе выполнения должна находиться инструкция загрузки (`movq` или `popq`), а во втором – инструкция перехода. Аналогично невозможно одновременное появление второго или третьего варианта особого случая обработки инструкции `ret` с особым случаем риска загрузки/использования или неверно спрогнозированного ветвления. Одновременно могут возникать только две комбинации, указанные стрелками.

В комбинацию А входит инструкция перехода на этапе выполнения, не выполняющая переход, и инструкция `ret` на этапе декодирования. Эта комбинация возможна, когда инструкция `ret` находится по адресу невыполненного перехода. Управляющая логика конвейера должна определить ошибочность предсказания ветвления и отменить инструкцию `ret`.

#### Упражнение 4.37 (решение в конце главы)

Напишите программу на ассемблере Y86-64, вызывающую появление комбинации А, и определите, правильно ли обрабатывается эта комбинация управляющей логикой конвейера.

Объединив действия, связанные с обработкой случаев, которые составляют комбинацию А (табл. 4.10), получаем следующие действия конвейера (предполагается, что нормальное выполнение подменяется остановом или «пузырьком»).

Условие	Конвейерные регистры				
	F	D	E	M	W
Обработка инструкции <code>ret</code>	останов	«пузырек»	нормальная работа	нормальная работа	нормальная работа
Неверное предсказание ветвления	нормальная работа	«пузырек»	«пузырек»	нормальная работа	нормальная работа
Комбинация	останов	«пузырек»	«пузырек»	нормальная работа	нормальная работа

То есть данная комбинация будет обрабатываться как неверно спрогнозированное ветвление, но с остановом на этапе выборки. К счастью, в следующем цикле логика выбора РС загрузит в счетчик инструкций адрес инструкции, следующей за инструкцией перехода, а не спрогнозированное значение, поэтому не имеет значения, что произойдет с конвейерным регистром F. Отсюда следует вывод, что конвейер корректно обрабатывает данную комбинацию.

Комбинация В включает риск загрузки/использования, когда одна инструкция загружает в регистр `%rsp` новое значение, а следующая за ней инструкция `ret` использует этот регистр в качестве операнда-источника, чтобы вытолкнуть адрес возврата из стека. Управляющая логика конвейера должна удерживать команду `ret` на этапе декодирования.

#### Упражнение 4.38 (решение в конце главы)

Напишите программу на ассемблере Y86-64, вызывающую появление комбинации В, и определите, правильно ли обрабатывается эта комбинация управляющей логикой конвейера.



Объединив действия, связанные с обработкой случаев, которые составляют комбинацию В (табл. 4.10), получаем следующие действия конвейера.

Условие	Конвейерные регистры				
	F	D	E	M	W
Обработка инструкции <code>ret</code>	останов	«пузырек»	нормальная работа	нормальная работа	нормальная работа
Риск загрузки/использования	останов	останов	«пузырек»	нормальная работа	нормальная работа
Комбинация	останов	«пузырек» + останов	«пузырек»	нормальная работа	нормальная работа
Желаемое поведение	останов	останов	«пузырек»	нормальная работа	нормальная работа

При одновременном появлении обоих случаев управляющая логика попытается приостановить инструкцию `ret`, чтобы исключить риск загрузки/использования, а также передать в этап декодирования «пузырек», обрабатывая инструкцию `ret`. Понятно, что конвейер не должен выполнять оба этих действия. Вместо этого он должен выполнить только действие, направленное на устранение риска загрузки/использования. Обработка инструкции `ret` должна быть отложена на один цикл.

Как показывает анализ, комбинация В требует особой обработки. Фактически первоначальная реализация управляющей логики в версии PIPE обрабатывает эту комбинацию неверно. Несмотря на множество проверок на имитаторе, в проекте обнаружилась совсем небольшая ошибка, выявляемая только с помощью описанного анализа. Встретившись с комбинацией В, управляющая логика установит в 1 сигналы **bubble** и **stall** для конвейерного регистра D. Этот пример наглядно показывает важность систематического анализа. Указанная ошибка вряд ли была бы обнаружена простым запуском обычных программ. Если ее не исправить, то конвейер будет неверно реализовывать ожидаемое поведение ISA.

### Реализация управляющей логики

На рис. 4.59 показана общая схема управляющей логики конвейера. На основании сигналов из конвейерных регистров и этапов управляющая логика генерирует управляющие сигналы **stall** (останов) и **bubble** («пузырек») для конвейерных регистров, а также определяет, когда можно или нельзя изменять регистр флагов. Мы можем объединить условия табл. 4.9 с действиями, перечисленными в табл. 4.10, чтобы создать HCL-описание различных сигналов, управляющих конвейером.

Конвейерный регистр F должен останавливаться в случаях риска загрузки/использования и при обработке инструкции `ret`:

```
bool F_stall =
    # Условие для риска загрузки/использования
    E_icode in { IMRMOVQ, IPOPOQ } &&
    E_dstM in { d_srcA, d_srcB } ||
    # Остановка этапа выборки, пока ret движется по конвейеру
    IRET in { D_icode, E_icode, M_icode };
```

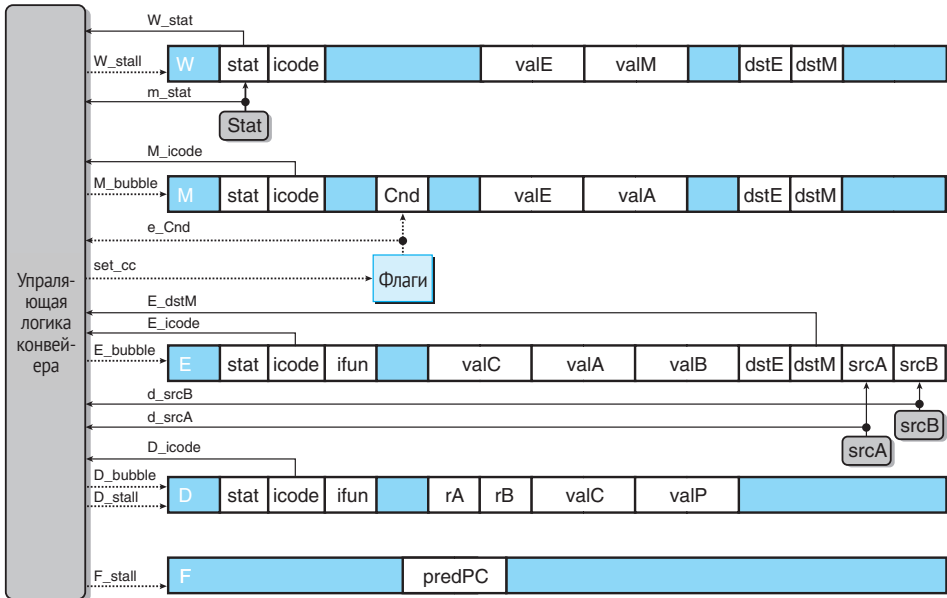
Для неверно предсказанного ветвления или инструкции `ret` в конвейерный регистр D должен быть записан «пузырек». Однако, как показывает анализ в предыдущем разделе, этого не должно происходить при наличии риска загрузки/использования в комбинации с инструкцией `ret`:

```

bool D_bubble =
    # Неверно предсказанное ветвление
    (E_icode == IJXX && !e_Cnd) ||
    # Остановить этап выборки на время прохождения инструкции ret
    # через конвейер, но только если нет риска загрузки/использования
    !(E_icode in { IMRMOVQ, IPOPOQ } && E_dstM in { d_srcA, d_srcB }) &&
    IRET in { D_icode, E_icode, M_icode };

```

Этот код охватывает все значения сигналов, управляющих обработкой особых случаев. В полном HCL-коде для PIPE все прочие управляющие сигналы установлены в 0.



**Рис. 4.59.** Логика управления конвейером PIPE. Эта логика отменяет нормальное продвижение инструкций по конвейеру для обработки особых случаев, таких как возврат из процедуры, неверное предсказание ветвления, риск загрузки/использования и программные исключения

#### Упражнение 4.39 (решение в конце главы)

Напишите HCL-код для сигнала **D\_stall** в реализации PIPE.

#### Упражнение 4.40 (решение в конце главы)

Напишите HCL-код для сигнала **E\_bubble** в реализации PIPE.

#### Упражнение 4.41 (решение в конце главы)

Напишите HCL-код для сигнала **set\_cc** в реализации PIPE. Он должен устанавливаться только для инструкций OPq и учитывать влияние программных исключений.

**Упражнение 4.42 (решение в конце главы)**

Напишите HCL-код для сигналов **M\_bubble** и **W\_stall** в реализации PIPE. Для реализации последнего придется изменить условия исключений, перечисленные в табл. 4.9.

**Тестирование проекта**

Как мы видели, практически невозможно предвидеть все ошибки заранее, даже проектируя относительно простой микропроцессор. В конвейерной обработке имеет место множество тонких взаимодействий между инструкциями на разных этапах конвейера. Мы видели, что многие проблемы проектирования связаны с необычными инструкциями (такими как выталкивание со стека указателя стека) или необычными комбинациями инструкций (такими как невыполненный переход, за которым следует `ret`). Мы также видели, что обработка исключений добавляет совершенно новое измерение в возможное поведение конвейера. Как в таких условиях можно быть уверенным в правильности проекта? Для производителей оборудования это большая проблема, потому что они не могут просто сообщить об ошибке и попросить пользователей загрузить исправления через интернет. Даже простая логическая ошибка, допущенная при проектировании, может иметь серьезные последствия, особенно учитывая, что микропроцессоры все чаще используются для управления системами, критически важными для нашей жизни и здоровья, такими как автомобильные антиблокировочные системы, кардиостимуляторы и системы управления самолетами.

Простое испытание проекта с использованием имитатора, на котором запускается серия «типичных» программ, не позволяет всесторонне протестировать систему. Для тщательного тестирования требуется разработать способы систематического создания множества тестов, которые будут использовать как можно больше различных инструкций и их комбинаций. При создании наших процессоров Y86-64 мы тоже разработали ряд сценариев тестирования, каждый из которых генерирует множество различных тестов, запускает их на имитаторе процессора и сравнивает ожидаемые значения регистров и памяти со значениями, полученными нашим имитатором набора инструкций YIS. Вот краткое описание сценариев:

- `optest`. Выполняет 49 тестов различных инструкций Y86-64 с разными регистрами-источниками и регистрами-приемниками;
- `jtest`. Выполняет 64 теста различных инструкций перехода и вызова процедур с различными комбинациями условий выполнения или невыполнения ветвления;
- `cmtest`. Выполняет 28 тестов различных инструкций условного перемещения данных с различными комбинациями условий выполнения или невыполнения;
- `htest`. Выполняет 600 тестов с различными рисками по данным, с различными комбинациями взаимозависимых инструкций и с различным количеством инструкций пор между парами взаимозависимых инструкций;
- `ctest`. Тестирует 22 различные комбинации условий, выявленные в ходе анализа, аналогичного тому, что мы провели в разделе 4.5.8;
- `etest`. Тестирует 12 различных комбинаций инструкций, когда одна может вызвать исключение, а следующие за ней – изменить видимое программисту состояние.

Ключевая идея этого метода тестирования заключается в том, чтобы как можно более систематически генерировать тесты, которые создают различные условия, способные вызвать ошибки в работе конвейера.

### Формальная проверка проекта

Даже когда проект успешно выполняет обширный набор тестов, все равно нет полной уверенности, что он будет правильно обрабатывать все возможные программы. Число возможных программ, которые можно было бы протестировать, невообразимо велико, даже если рассматривать только тесты, состоящие из коротких фрагментов кода. Однако новые методы *формальной проверки* дают нам инструменты, которые строго рассматривают все возможные варианты поведения системы и определяют наличие каких-либо ошибок проектирования.

Мы смогли применить формальную проверку к более ранней версии наших процессоров Y86-64 [13]. Мы создали платформу для сравнения поведения конвейерной архитектуры PIPE и первоначальной версии SEQ. С помощью этого метода мы смогли доказать, что при выполнении произвольной программы на машинном языке два процессора будут давать совершенно идентичное состояние, видимое программисту. Конечно, наш верификатор не может проверить работу процессоров на всех возможных программах, потому что их количество бесконечно. Поэтому в нем используется форма доказательства по индукции, демонстрирующая согласованность поведения двух процессоров цикл за циклом. Проведение этого анализа требует рассуждений об оборудовании с использованием *символических методов*, в которых все программные значения рассматриваются как произвольные целые числа, а арифметико-логическое устройство (АЛУ) интерпретируется как некий «черный ящик», вычисляющий некоторую неопределенную функцию над его аргументами. Предполагается только, что АЛУ в SEQ и PIPE вычисляют идентичные функции.

Мы использовали описания логики управления на HCL, чтобы сгенерировать логику управления для наших символьных моделей процессоров и выявить любые ошибки в коде HCL. Возможность показать, что SEQ и PIPE действуют идентично, не гарантирует, что любая из этих версий реализует архитектуру набора команд. Однако символические методы способны обнаружить любую ошибку, обусловленную неправильной конструкцией конвейера, что является основным источником ошибок проектирования.

В наших экспериментах мы проверили не только версию PIPE, аналогичную представленной в этой главе, но и несколько вариантов, разработку которых мы поручаем вам в качестве домашних заданий, поддерживающих дополнительные инструкции и возможности или использующих другие стратегии предсказания ветвлений. Интересно отметить, что во всех наших разработках мы обнаружили только одну ошибку (связанную с комбинацией B, описанной в разделе 4.5.8) – в нашем варианте решения упражнения 4.58. Это помогло нам заметить недостаток нашего режима тестирования, благодаря чему мы добавили дополнительные проверки в сценарий тестирования ctest.

Формальная проверка все еще находится на ранней стадии разработки. Эти инструменты часто сложны в использовании и не имеют возможности проверять крупномасштабные проекты. Мы смогли проверить наши процессоры отчасти из-за их относительной простоты. Но даже в этом случае потребовалось несколько недель напряженного труда и несколько сеансов тестирования, каждый из которых занимал до 8 часов машинного времени. Эта область исследований активно развивается, при этом некоторые инструменты становятся доступными на коммерческой основе, а некоторые из них даже используются такими компаниями, как Intel, AMD и IBM.

## 4.5.9. Анализ производительности

Можно отметить, что случаи, требующие особых действий со стороны управляющей логики конвейера, не дают достигнуть цели выпуска новой инструкции в каждом цикле синхронизации. Такую неэффективность можно измерить, определив частоту добавления «пузырьков» в конвейер, которые заставляют некоторые циклы выполняться вхо-

лостую. Инструкция возврата генерирует три «пузырька», риск загрузки/использования – один, а неверно предсказанное ветвление – два. Влияние этих действий на общую производительность можно выразить количественно, вычислив среднее количество циклов синхронизации, необходимых версии PIPE на выполнение каждой инструкции. Эта количественная оценка известна как CPI (cycles per instruction – циклов на инструкцию). Она оценивает значение, обратное средней пропускной способности конвейера, но измеряет время в циклах, а не в пикосекундах. Эта оценка часто используется при анализе производительности проекта.

#### Приложение в интернете ARCH:VLOG. Реализация конвейерного процессора Y86-64 на Verilog

Как уже отмечалось, современные подходы к проектированию логики включают разработку текстовых представлений архитектуры оборудования на языке *описания оборудования* (Hardware Description Language, HDL). Затем эти представления можно протестировать с помощью методов моделирования и различных инструментов формальной проверки. Когда будет достигнута полная уверенность в правильности конструкции, можно переходить к инструментам логического синтеза, чтобы воплотить проект в реальные логические схемы.

Мы разработали модели наших процессоров Y86-64 на языке описания оборудования Verilog. Эти модели объединяют модули, реализующие основные строительные блоки процессора, а также управляющую логику, созданную непосредственно из описаний HCL. Мы смогли синтезировать некоторые из этих проектов, загрузить описания логических схем в программируемую логическую интегральную схему (Field-Programmable Gate Array, FPGA) и опробовать получившиеся процессоры на реальных программах.

Если не принимать во внимание влияние исключений на производительность (которые по определению возникают редко), то на CPI можно посмотреть и с другой стороны: представить, что процессор выполняет некоторую эталонную тестовую программу, и обратить внимание на работу этапа выполнения. В каждом цикле этап выполнения будет обрабатывать либо инструкцию, которая продолжит движение через оставшиеся этапы до завершения, либо «пузырек», добавленный вследствие возникновения одного из трех особых случаев. Если этап обрабатывает  $C_i$  инструкций и  $C_b$  «пузырьков», тогда процессору потребуется порядка  $C_i + C_b$  полных циклов для выполнения  $C_i$  инструкций. Здесь использовано слово «порядка», потому что игнорируются циклы, необходимые для запуска инструкций, проходящих через конвейер. CPI для данной тестовой программы можно вычислить следующим образом:

$$CPI = \frac{C_i + C_b}{C_i} = 1,0 + \frac{C_b}{C_i}.$$

То есть значение CPI равно 1,0 плюс штрафной элемент  $C_b/C_i$ , определяющий среднее количество «пузырьков», приходящихся на одну выполненную инструкцию. Поскольку вставку «пузырька» могут вызвать только инструкции трех типов, дополнительный штрафной элемент можно разбить на три компонента:

$$CPI = 1,0 + lp + mp + rp,$$

где  $lp$  (load penalty – штраф за загрузку) – средняя частота добавления «пузырьков» на время останова для предотвращения рисков загрузки/использования;  $mp$  (mispredicted branch penalty – штраф за неверно предсказанное ветвление) – средняя частота добав-

ления «пузырьков» при отмене инструкций из-за неверно предсказанного ветвления; *gr* (return penalty – штраф за возврат) – средняя частота добавления «пузырьков» при останове на время прохождения инструкции *ret*. Каждый из этих штрафов задает общее количество «пузырьков», добавленных по объявленной причине (некоторая часть  $C_b$ ), деленное на общее число выполненных инструкций ( $C_p$ ).

Для оценки каждого из этих штрафов необходимо знать, как часто выполняются соответствующие инструкции (загрузки, условного ветвления и возврата) и как часто для каждой из них возникают особые условия. Для расчета CPI возьмем следующий набор частот (они сопоставимы с результатами измерений, приведенными в [44] и [46]):

- на долю инструкций загрузки (*movq* и *popq*) приходится 25 % от общего количества выполняемых инструкций, 20 % из них вызывают риск загрузки/использования;
- на долю инструкций условного ветвления приходится 20 % от общего количества выполняемых инструкций; для 60 % из них прогноз дается верный, для 40 % – ошибочный;
- на долю инструкций возврата приходится 2 % от общего количества выполняемых инструкций.

Величину каждого штрафа можно оценить как произведение частоты встречаемости инструкции на частоту, при которой возникает особая ситуация, и на количество «пузырьков», вставляемых в этих ситуациях:

Причина	Штраф	Частота встречаемости инструкции	Частота появления особого случая	Количество «пузырьков»	Произведение
Риск загрузки/ использования	<i>lp</i>	0,25	0,20	1	0,05
Неверное предсказание	<i>tp</i>	0,20	0,40	2	0,16
Возврат	<i>rp</i>	0,02	1,00	3	0,06
Общий штраф					0,27

Сумма трех штрафов составляет 0,27, что дает величину CPI, равную 1,27.

Нашей целью было создание конвейера, способного выпускать по одной инструкции в каждом цикле, т. е. с CPI = 1,0. Поставленная цель не была достигнута, и все же общая производительность остается достаточно высокой. Также очевидно, что дальнейшие усилия по снижению величины CPI должны быть сосредоточены на увеличении точности прогнозирования ветвления. Штраф за неверное предсказание составляет 0,16 от общей суммы штрафов в 0,27, потому что условное ветвление широко распространено в программах, а наша стратегия прогнозирования ошибается слишком часто, при этом каждая ошибка влечет отмену выполнения двух инструкций.

#### Упражнение 4.43 (решение в конце главы)

Предположим, что для прогнозирования ветвления используется стратегия «переход назад выбирается всегда, вперед – никогда» (backward taken, forward not taken, BTFNT), дающая 65 % доли успешных предсказаний (как рассказывалось в разделе 4.5.4). Как изменится величина CPI, если предположить, что все остальные частоты не изменятся?

**Упражнение 4.44 (решение в конце главы)**

Давайте проанализируем относительную производительность при использовании условной передачи данных и условной передачи управления для программ, которые вы должны были написать, решая упражнения 4.5 и 4.6. Допустим, что эти программы используются для вычисления суммы абсолютных значений очень длинного массива, поэтому общая производительность определяется в основном количеством тактов микропроцессора, необходимых для выполнения внутреннего цикла. Предположим, что наши инструкции перехода предсказываются как выполняемые всегда и что около 50 % значений массива – положительные числа.

1. Сколько инструкций в среднем выполняется во внутреннем цикле в каждой из программ?
2. Сколько пузырьков в среднем будет вводиться во внутреннем цикле в каждой из программ?
3. Какое среднее количество тактов потребуется на обработку одного элемента массива в каждой из программ?

### 4.5.10. Незаконченная работа

Мы разработали структуру конвейерного микропроцессора PIPE, спроектировали блоки управляющей логики и реализовали управляющую логику для особых случаев, когда обычного управления потоком в конвейере недостаточно. Однако версии PIPE не хватает некоторых ключевых особенностей, которые необходимы реальному микропроцессору. Некоторые из них мы обсудим далее и одновременно подскажем, что требуется для их реализации.

#### Инструкции, требующие для выполнения нескольких циклов

Все инструкции в архитектуре Y86-64 предполагают выполнение таких простых операций, как сложение, и для их обработки на этапе выполнения достаточно одного цикла синхронизации. В более полном наборе команд будут присутствовать инструкции, требующие более сложных операций, например умножения и деления целых чисел, а также операций с плавающей точкой. В процессоре со средней производительностью, таком как PIPE, обычное время выполнения подобных операций варьируется от 3–4 циклов для сложения чисел с плавающей точкой до 64 циклов для целочисленного деления. Для реализации этих инструкций требуются дополнительные аппаратные модули, выполняющие вычисления, и механизм координации обработки этих инструкций с остальными компонентами конвейера.

Один из простых подходов к реализации инструкций, требующих для выполнения нескольких циклов, заключается в добавлении модулей арифметики с целыми числами и с числами с плавающей точкой в логику этапа выполнения. Инструкция остается на этапе выполнения столько циклов синхронизации, сколько требуется, что вызывает останов этапов выборки и декодирования. Реализовать это просто, однако это отрицательно скажется на конечной производительности.

Увеличить скорость выполнения сложных операций можно с помощью специальных аппаратных модулей, работающих независимо от основного конвейера. Как правило, для выполнения умножения и деления целых чисел и операций с числами с плавающей точкой используется по одному такому модулю. Когда инструкция переходит на этап декодирования, она может быть *передана* специальному модулю, и пока модуль выполняет операцию, конвейер продолжает обработку других инструкций. Обычно модуль для выполнения операций с плавающей точкой сам по себе является конвейерным, поэтому операции можно выполнять параллельно в основном конвейере и в других разных модулях.



Выполнение операций в разных модулях необходимо синхронизировать, чтобы исключить некорректное поведение. Например, при наличии зависимостей по данным между разными операциями, обрабатываемыми разными модулями, управляющей логике может понадобиться приостановить одну часть системы до завершения вычислений в какой-то другой части. Для передачи результатов из одной части системы в другие нередко используются различные формы продвижения, подобные продвижению сигналов между различными этапами в версии PIPE. Общая структура становится сложнее, если сравнивать ее со структурой PIPE, однако, чтобы получить поведение, соответствующее последовательной модели ISA, можно использовать те же приемы останова, продвижения и управления конвейером.

### Сопряжение с системой памяти

В нашем обсуждении проекта PIPE предполагалось, что к памяти инструкций и памяти данных можно обращаться одновременно в одном цикле синхронизации. При этом мы игнорировали возможные риски, связанные с самомодифицирующимся кодом, когда одна инструкция осуществляет запись в память, откуда выбираются последующие инструкции. Более того, обращения к ячейкам памяти осуществляются в соответствии с их виртуальными адресами, требующими преобразования в физические адреса до фактического выполнения операций чтения или записи. Понятно, что выполнить все это за один цикл просто невозможно. Более того, значения в памяти, к которым осуществляется доступ, могут храниться на диске, что потребует миллионов циклов синхронизации для чтения их в память процессора.

Как будет обсуждаться в главах 6 и 9, система памяти процессора использует целый комплекс из аппаратных блоков памяти и программных средств операционной системы для управления виртуальной памятью. Система памяти организована в виде иерархии быстродействующих блоков памяти небольшой емкости, хранящих часть данных из менее быстродействующих и более емких блоков памяти. На ближайшем к процессору уровне блоки *кеш-памяти* обеспечивают быстрый доступ к данным, которые чаще всего используются программами. В типичном процессоре имеется два кеша первого уровня – один для чтения инструкций, а другой для чтения и записи данных. Другой тип кеш-памяти, известный как *буфер быстрого преобразования адреса* (Translation Look-aside Buffer, TLB), обеспечивает быстрое преобразование виртуальных адресов в физические. Комбинация TLB и кеш-памяти позволяет большую часть времени читать инструкции и читать или записывать данные за один цикл синхронизации. Таким образом, упрощенное представление механики работы нашего процессора с памятью является вполне приемлемым.

Несмотря на то что в кеш-памяти хранятся наиболее часто используемые данные, иногда возникает так называемый *промах* кеша, когда программа ссылается на значение, отсутствующее в кеше. В лучшем случае отсутствующие данные можно получить из кеша более высокого уровня или из основной памяти процессора, что потребует от 3 до 20 циклов синхронизации. На это время конвейер просто приостанавливается, удерживая инструкцию на этапе выборки или обращения к памяти, пока кеш не выполнит операцию чтения или записи. В конвейерной архитектуре это можно реализовать добавлением в управляющую логику конвейера дополнительных условий останова. Промах кеша и последующая синхронизация с конвейером обрабатываются исключительно аппаратными средствами, чтобы свести время простоя до минимума.

Иногда случается так, что ячейка виртуальной памяти, на которую ссылается программа, фактически хранится в памяти на диске. В этой ситуации аппаратные средства сигнализируют об исключительной ситуации – *ошибке обращения к отсутствующей странице*. Подобно другим исключениям, эта ошибка заставляет процессор передать управление обработчику исключительных ситуаций в операционной системе. Обработчик запускает перенос данных с диска в основную память, по завершении которого



операционная система возвращает управление первоначальной программе, где инструкция, вызывающая ошибку обращения к отсутствующей странице, будет выполнена повторно. На этот раз обращение к памяти будет успешным, даже притом что может возникнуть промах кеша. Запуск аппаратными средствами процедуры в операционной системе, которая затем возвращает управление аппаратным средствам, позволяет аппаратному и программному обеспечению взаимодействовать при обработке ошибок обращения к отсутствующим страницам. Поскольку доступ к диску может потребовать миллионов циклов синхронизации, несколько тысяч циклов, требуемых обработчику прерывания в операционной системе, не сильно повлияют на общую производительность.

### История развития индустрии проектирования процессоров

Конвейер с пятью этапами, подобный использованному в нашей версии PIPE, был разработан в середине 1980-х. В 1986-м компания MIPS Technologies вывела на рынок процессор, созданный группой исследователей Хеннесси (Hennessy, основатель компании MIPS Technologies) в Стэнфорде. В 1987-м компанией Sun Microsystems был создан процессор SPARC, основой для которого послужил прототип процессора RISC, разработанный исследовательской группой Паттерсона (Patterson) в Беркли. В обеих моделях использовались пятиэтапные конвейеры. В процессоре Intel i486 тоже применяется пятиэтапный конвейер, но с иным разделением обязанностей между этапами; в его конвейере имеется два этапа декодирования и комбинированный этап выполнения и обращения к памяти.

Такие конвейерные архитектуры имеют ограниченную пропускную способность – максимум одна инструкция за цикл синхронизации. Единица измерения CPI (cycles per instruction – циклов на инструкцию), описанная в разделе 4.5.9, никогда не может быть меньше 1.0 в таких архитектурах. Разные этапы в каждом цикле могут обрабатывать только одну инструкцию. Более поздние модели процессоров поддерживают *суперскалярные* операции, т. е. достигают CPI меньше 1.0 путем выборки, декодирования и выполнения сразу нескольких инструкций. По мере распространения суперскалярных процессоров стала реже использоваться оценка производительности CPI и чаще – среднее количество инструкций, выполняемых за один цикл синхронизации (instructions per cycle, IPC). В суперскалярных процессорах величина IPC может превосходить 1.0. В самых передовых архитектурах используется методика, известная как выполнение *не по порядку*, обеспечивающая возможность параллельного выполнения инструкций, возможно, в абсолютно ином порядке, чем в программе, но с сохранением общего поведения, предполагаемого последовательной моделью ISA. Эта форма выполнения рассматривается в главе 5, в части дискуссии об оптимизации программ.

Однако конвейерные процессоры еще не стали историей. Большинство продаваемых в мире процессоров используются во встроенных системах, управляющих работой автомобилей, бытовой техники и других устройств, где сам процессор невидим для пользователя. В таких агрегатах простота конвейерного процессора, подобного рассмотренному в этой главе, снижает стоимость и потребляемую мощность, по сравнению с более производительными процессорами.

В последнее время, когда в обиход стали входить многоядерные процессоры, появились сторонники этого направления, утверждающие, что мы могли бы увеличить их вычислительную мощность еще больше, размещая на одном кристалле много более простых процессоров вместо небольшого количества более сложных. Такие процессоры иногда называют «массово-параллельными» [10].

С точки зрения процессора, заботу о непредсказуемом поведении памяти из-за особенностей ее иерархического строения принимает на себя комбинация остановов для обработки кратковременных промахов кеша и долговременной обработки исклю-

чительных ситуаций, связанных с ошибками обращения к отсутствующим страницам памяти.

#### Приложение в интернете ARCH:HCL. HCL-описание процессора Y86-64

В этой главе мы показали фрагменты описания на языке HCL нескольких простых логических схем и управляющей логики процессоров SEQ и PIPE с архитектурой Y86-64. Для справки мы предоставляем документацию по языку HCL и полные HCL-описания логики управления для двух процессоров. Каждое из этих описаний занимает всего пять-семь страниц кода на HCL, и они стоят того, чтобы изучить их полностью.

## 4.6. Итоги

Как было показано в этой главе, архитектура набора команд (Instruction Set Architecture, ISA) обеспечивает слой абстракции, за которым скрывается фактическое поведение процессора. ISA создает полное ощущение последовательного выполнения программы, когда обработка одной инструкции целиком завершается до начала обработки следующей.

Определение набора инструкций Y86-64 мы начали с инструкций x86-64 и значительного упрощения типов данных, режимов адресации и кодировки инструкций. Полученный в результате набор можно отнести как к системе команд RISC, так и CISC. Затем мы спроектировали обработку инструкций в виде последовательности из пяти этапов, фактически выполняемые действия в которых зависят от конкретной инструкции. На основе этого проекта был сконструирован последовательный процессор SEQ, в котором каждая инструкция проходит все пять этапов, по одному в каждом цикле синхронизации.

Конвейерный режим обработки повышает производительность системы, поддерживая возможность параллельной обработки разных инструкций на разных этапах. В любой конкретный момент времени этапы конвейера обрабатывают разные инструкции. С внедрением параллельной обработки нам пришлось позаботиться о сохранности видимости последовательного выполнения программы. Мы ввели понятие конвейера, переупорядочили компоненты SEQ, получив версию SEQ+, а затем добавили конвейерные регистры и создали конвейер PIPE-.

Мы заметно увеличили производительность конвейера, добавив логику продвижения, ускоряющую передачу результатов между инструкциями. Несколько особых случаев потребовали от нас предусмотреть дополнительную логику управления конвейером, приостанавливающую или отменяющую некоторые этапы.

Наш проект предусматривает элементарные механизмы обработки исключений, при этом мы позаботились о том, чтобы на состоянии процессора, видимое программисту, влияли только инструкции, предшествующие исключительной инструкции. Полная реализация обработки исключений выглядела бы значительно сложнее. В системах с еще большим уровнем параллелизма правильная обработка исключений выглядит еще более сложной.

В этой главе мы выучили несколько важных уроков, связанных с проектированием процессоров:

- *высшим приоритетом является управление сложностью.* Основная задача – добиться оптимального использования аппаратных ресурсов и получить максимальную производительность с минимальными затратами. Мы решили эту задачу, создав очень простую и универсальную структуру обработки инструкций

всех типов. Опираясь на эту структуру, мы смогли добиться использования одних и тех же аппаратных модулей для обработки различных инструкций;

- *нет необходимости прямо следовать требованиям ISA.* Прямое следование требованиям означало бы исключительно последовательную обработку инструкций. Для достижения более высокой производительности необходимо использовать способность аппаратного обеспечения одновременно выполнять большое количество операций. Это дало возможность внедрить конвейерную организацию. Путем тщательного проектирования и анализа мы смогли избежать различных рисков и добились того, что общий эффект при выполнении программы в точности будет совпадать с эффектом, предполагаемым моделью ISA;
- *проектировщики аппаратного обеспечения должны быть очень требовательными.* Если кристалл уже готов, то исправить какие бы то ни было ошибки в нем просто невозможно. Поэтому очень важно правильно организовать процесс проектирования с самого начала. Это подразумевает подробнейший анализ различных типов инструкций и их комбинаций, даже, казалось бы, бессмысленных (например, выталкивание значения со стека в регистр указателя стека). Все решения необходимо тщательно тестировать с помощью имитации выполнения тестовых программ. При разработке управляющей логики для версии PIPE мы допустили небольшую ошибку, которая обнаружилась только после тщательного и систематического анализа управляющих комбинаций.

### 4.6.1. Имитаторы Y86-64

В материалы лабораторных работ для этой главы включены имитаторы процессоров SEQ и PIPE. Каждый имитатор представлен в двух версиях:

- версия с графическим интерфейсом отображает память, программный код и состояние процессора в графических окнах. Она дает удобную возможность наблюдать за потоком инструкций в процессоре. Панель управления позволяет интерактивно сбрасывать имитатор в исходное состояние, запускать его или действовать в пошаговом режиме;
- текстовая версия запускает тот же имитатор, но информация отображается только в текстовом виде в терминале. Данная версия не предлагает дополнительных преимуществ при отладке, но позволяет осуществлять автоматическое тестирование процессора.

Управляющая логика для имитаторов генерируется преобразованием HCL-объявлений логических блоков в код на языке C. Затем этот код компилируется и компонуется с кодом симулятора. Такая комбинация позволяет тестировать разные варианты проекта. Также доступны сценарии тестирования, которые проведут исчерпывающую проверку различных инструкций и возможных рисков.

### Библиографические заметки

Для интересующихся подробностями логического проектирования мы рекомендуем учебник по логическому проектированию Каца (Katz) и Борриелло (Borriello) [58], в котором особый упор делается на использование языков описания аппаратного обеспечения. В учебнике Хеннесси (Hennessy) и Паттерсона (Patterson), посвященном компьютерной архитектуре [46], подробно изложены вопросы проектирования процессоров, включая как простые конвейеры, подобные описанному здесь, так и более совершенные процессоры, параллельно выполняющие большее количество инструкций. Шривер (Shriver) и Смит (Smith) [101] дают очень подробное описание процессора AMD, совместимого с процессором IA32 компании Intel.

## Домашние задания

### Упражнение 4.45 ♦

В разделе 3.4.2 инструкция `pushq` в архитектуре x86-64 была описана как уменьшающая указатель стека и затем сохраняющая регистр в ячейке памяти, на которую ссылается указатель стека. То есть инструкцию вида `pushq REG`, где `REG` – некоторый произвольный регистр, можно было бы представить в виде следующей эквивалентной последовательности инструкций:

```
subq $8,%rsp      Уменьшить указатель стека
movq REG, (%rsp)  Сохранить REG на стеке
```

1. В свете анализа, выполненного в упражнении 4.7, объясните: правильно ли эта последовательность описывает поведение инструкции `pushq %rsp`?
2. Как можно переписать последовательность выше, чтобы она правильно описывала случаи, когда `REG` – это `%rsp` и любой другой регистр?

### Упражнение 4.46 ♦

В разделе 3.4.2 инструкция `popq` в архитектуре x86-64 была описана как копирующая значение с вершины стека в регистр-приемник и затем увеличивающая указатель стека. То есть инструкцию вида `popq REG`, где `REG` – некоторый произвольный регистр, можно было бы представить в виде следующей эквивалентной последовательности инструкций:

```
movq (%rsp), REG  Прочитать значение со стека в регистр REG
addq $8,%rsp      Увеличить указатель стека
```

1. В свете анализа, выполненного в упражнении 4.8, объясните: правильно ли эта последовательность описывает поведение инструкции `popq %rsp`?
2. Как можно переписать последовательность выше, чтобы она правильно описывала случаи, когда `REG` – это `%rsp` и любой другой регистр?

### Упражнение 4.47 ♦♦♦

Вам дано задание написать программу для Y86-64, выполняющую пузырьковую сортировку. Ниже для справки приводится функция на C, реализующая пузырьковую сортировку с использованием ссылок на элементы массива по индексам:

```
1 /* Пузырьковая сортировка: версия с массивом */
2 void bubble_a(long *data, long count) {
3     long i, last;
4     for (last = count-1; last > 0; last--) {
5         for (i = 0; i < last; i++)
6             if (data[i+1] < data[i]) {
7                 /* Поменять местами соседние элементы */
8                 long t = data[i+1];
9                 data[i+1] = data[i];
10                data[i] = t;
11            }
12    }
13 }
```

1. Напишите и протестируйте версию на C, которая ссылается на элементы массива посредством указателей, т. е. без использования индексов.

2. Напишите и протестируйте программу на языке ассемблера Y86-64, состоящую из функции и проверочного кода. Возможно, вам пригодится код на ассемблере x86-64, сгенерированный при компиляции вашего кода на C. Обычно сравнение указателей выполняется с использованием беззнаковой арифметики, в этом упражнении вы можете использовать арифметику со знаком.

### Упражнение 4.48 ♦♦

Измените код, который вы написали для решения упражнения 4.47, чтобы реализовать сравнение и переменную местами элементов массива в функции пузырьковой сортировки (строки 6–11) без инструкций перехода и не более чем с тремя инструкциями условного перемещения.

### Упражнение 4.49 ♦♦♦

Измените код, который вы написали для решения упражнения 4.47, чтобы реализовать сравнение и переменную местами элементов массива в функции пузырьковой сортировки (строки 6–11) без инструкций перехода и с единственной инструкцией условного перемещения.

### Упражнение 4.50 ♦♦♦

В разделе 3.6.8 мы видели, что обычно оператор `switch` реализуется как набор блоков кода, которые затем индексируются с помощью таблицы переходов. Взгляните на код C в листинге 4.4, где приводится функция `switchv` с соответствующим кодом для ее тестирования.

**Листинг 4.4.** Операторы `switch` можно транслировать в код Y86-64. Это требует реализации таблицы переходов

```
#include <stdio.h>
/* Пример использования оператора switch */

long switchv(long idx) {
    long result = 0;
    switch(idx) {
        case 0:
            result = 0xaaa;
            break;
        case 2:
        case 5:
            result = 0xbbb;
            break;
        case 3:
            result = 0xccc;
            break;
        default:
            result = 0xddd;
    }
    return result;
}

/* Тестовый код */
#define CNT 8
#define MINVAL -1

int main() {
    long vals[CNT];
```

```

long i;
for (i = 0; i < CNT; i++) {
    vals[i] = switchv(i + MINVAL);
    printf("idx = %ld, val = 0x%lx\n", i + MINVAL, vals[i]);
}
return 0;
}

```

Реализуйте `switchv` на языке ассемблера Y86-64, используя таблицу переходов. В наборе инструкций Y86-64 отсутствует инструкция косвенного перехода, но тот же эффект можно получить, поместив вычисленный адрес в стек и затем выполнив инструкцию `ret`. Реализуйте тестовый код, аналогичный коду на C, чтобы показать, что ваша реализация `switchv` будет обрабатывать все случаи, как заданные явно, так и подпадающие под случай `default`.

### Упражнение 4.51 ♦

В упражнении 4.3 была представлена инструкция `iaddq`, прибавляющая непосредственное значение к значению в заданном регистре. Опишите, какие вычисления выполняет реализация этой инструкции. Возьмите за основу вычисления для `irmovq` и `OPq` (табл. 4.3).

### Упражнение 4.52 ♦♦

Файл `seq-full.hcl` содержит HCL-описание процессора SEQ вместе с объявлением константы `IIADDQ`, имеющей шестнадцатеричное значение C – код инструкции `iaddq`. Измените HCL-описания блоков управляющей логики для реализации инструкции `iaddq`, как описано в упражнениях 4.3 и 4.51. Загляните в лабораторные материалы, чтобы узнать, как создать симулятор для вашего решения и как его протестировать.

### Упражнение 4.53 ♦♦♦

Предположим, что вам необходимо создать недорогой конвейерный процессор на основе структуры, разработанной для версии PIPE– (рис. 4.34) без продвижения. Этот процессор будет обрабатывать все зависимости по данным путем останова до момента, когда инструкция, генерирующая необходимое значение, не минует этап обратной записи.

Файл `pipe-stall.hcl` содержит модифицированную версию кода HCL для PIPE, в котором логика продвижения отключена. То есть сигналы `e_valA` и `e_valB` просто объявлены следующим образом:

```

## НЕ ИЗМЕНЯЙТЕ СЛЕДУЮЩИЙ КОД
## Продвижение отсутствует. valA получает значение valP или из блока регистров
word d_valA = [
    D_icode in { ICALL, IJXX } : D_valP; # Использовать увеличенный PC
    1 : d_rvalA; # Использовать значение из блока регистров
];

## Продвижение отсутствует. valB -- это значение из блока регистров
word d_valB = d_rvalB;

```

Измените управляющую логику конвейера в конце этого файла так, чтобы она корректно устраняла все возможные риски управления и риски по данным. В процессе проектирования вы должны проанализировать различные комбинации управляющих случаев, как мы делали это при проектировании управляющей логики конвейера PIPE. Здесь может возникнуть множество различных комбинаций, поскольку очень много ус-

ловий требуют останова конвейера. Убедитесь, что управляющая логика обрабатывает каждую комбинацию корректно. Загляните в лабораторные материалы, чтобы узнать, как создать симулятор для вашего решения и как его протестировать.

### Упражнение 4.54 ♦♦

Файл `pipe-full.hcl` содержит копию описания PIPE на языке HCL вместе с объявлением константы `IIADDQ`. Измените код в этом файле и реализуйте инструкцию `iaddq`, как описано в упражнениях 4.3 и 4.52. Загляните в лабораторные материалы, чтобы узнать, как создать симулятор для вашего решения и как его протестировать.

### Упражнение 4.55 ♦♦♦

Файл `pipe-nt.hcl` содержит копию описания PIPE на языке HCL с объявлением константы `J_YES` со значением 0 – кодом функции для инструкции безусловного перехода. Измените логику прогнозирования ветвления так, чтобы она предполагала, что условные переходы не выполняются никогда, а безусловные переходы и `call` – всегда. Вы должны придумать способ передачи **valC** – целевого адреса перехода – в конвейерный регистр M, чтобы иметь возможность восстановить нормальный ход выполнения программы после неверного предсказания перехода. Загляните в лабораторные материалы, чтобы узнать, как создать симулятор для вашего решения и как его протестировать.

### Упражнение 4.56 ♦♦♦

Файл `pipe-btfnt.hcl` содержит копию описания PIPE на языке HCL с объявлением константы `J_YES` со значением 0 – кодом функции для инструкции безусловного перехода. Измените логику прогнозирования ветвления так, чтобы она предполагала, что условные переходы выполняются, когда **valC** < **valP** (переход назад), и не выполняются, когда **valC** ≥ **valP** (переход вперед). (Поскольку Y86-64 не поддерживает беззнаковую арифметику, вы должны реализовать эту проверку, используя сравнение со знаком.) Безусловные переходы и `call` должны прогнозироваться как выполняемые всегда. Вы должны придумать способ передачи **valC** и **valP** в конвейерный регистр M, чтобы иметь возможность восстановить нормальный ход выполнения программы после неверного предсказания перехода. Загляните в лабораторные материалы, чтобы узнать, как создать симулятор для вашего решения и как его протестировать.

### Упражнение 4.57 ♦♦♦

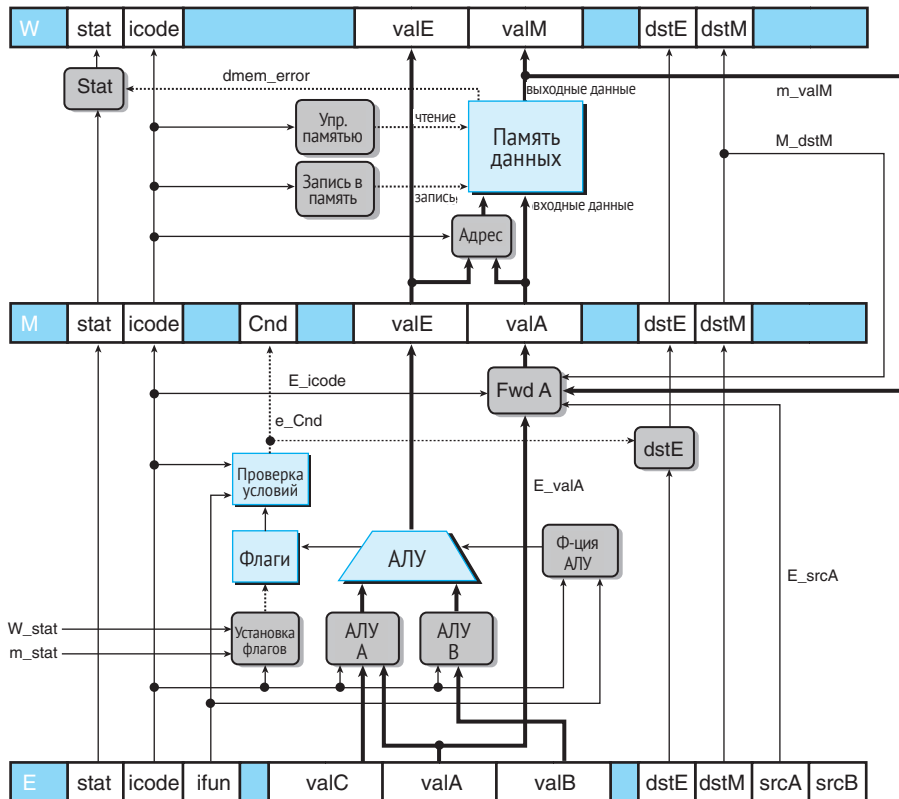
В нашем проекте PIPE мы генерируем останов всякий раз, когда одна инструкция выполняет загрузку, читая значение из памяти в регистр, а следующая за ней использует этот регистр в качестве операнда-источника. Когда источник используется на этапе выполнения, останов – единственный способ избежать риска по данным. В случае когда вторая инструкция сохраняет операнд-источник в памяти, как, например, инструкция `rmmovq` или `pushq`, останов не требуется. Взгляните на следующий пример:

```

1  rmmovq 0(%rcx),%rdx    # Загрузить 1
2  pushq  %rdx            # Сохранить 1
3  nop
4  popq   %rdx            # Загрузить 2
5  rmmovq %rax,0(%rdx)    # Сохранить 2
```

В строках 1 и 2 инструкция `rmmovq` читает значение из памяти в `%rdx`, а затем инструкция `pushq` вталкивает это значение в стек. Наш конвейер PIPE остановит команду `pushq`, чтобы избежать риска загрузки/использования. Но обратите внимание, что инструкции `pushq` значение `%rdx` не требуется, пока она не достигнет этапа обращения к памяти. Для этого случая можно добавить дополнительный обходной путь, как показано на

рис. 4.60, для продвижения выхода памяти (сигнал **m\_valM**) в поле **valA** в конвейерном регистре M. В следующем цикле синхронизации это значение можно записать в память. Данная методика называется *продвижением загрузки*.



**Рис. 4.60.** Этапы выполнения и обращения к памяти с возможностью продвижения загрузки. Добавляя обходной путь от выхода памяти к операнду-источнику valA в конвейерном регистре M, для одной из форм риска загрузки/использования можно вместо останова организовать продвижение. Это предмет задачи 4.57

Обратите внимание, что во втором случае (строки 4 и 5) не получится использовать продвижение загрузки. Значение, загруженное командой `porq`, используется в вычислении адреса следующей инструкцией, и это значение необходимо на этапе выполнения, а не на этапе обращения к памяти.

1. Напишите логическую формулу, описывающую определение риска загрузки/использования, подобную формуле в табл. 4.9, которая не будет вызывать останов в случаях, когда можно использовать продвижение загрузки.
2. Файл `pipe-lf.hcl` содержит модифицированную версию управляющей логики для PIPE. В нем имеется определение сигнала **e\_valA** для реализации блока, обозначенного как «Fwd A» (продвижение A) на рис. 4.60. Кроме того, в логике управления конвейером условие риска загрузки/использования установлено в 0, поэтому логика управления конвейером не будет обнаруживать риски по загрузке/использованию ни в какой форме. Модифицируйте это HCL-описание и реализуйте продвижение загрузки.



### Упражнение 4.58 ♦♦♦

Наша конвейерная конструкция несколько нереалистична, потому что блок регистров имеет два порта для записи, но два порта записи в блок регистров использует только инструкция `porq`. Другие инструкции могли бы использовать один порт для записи **valE** и **valM**. На рис. 4.61 показана модифицированная версия логики обратной записи, где происходит слияние обратной записи идентификаторов регистров (**W\_dstE** и **W\_dstM**) в один сигнал **w\_dstE** и обратной записи значений (**W\_valE** и **W\_valM**) также в один сигнал **w\_valE**.

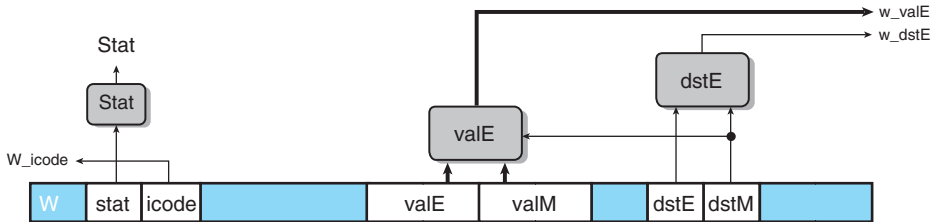


Рис. 4.61. Логика, выполняющая объединение сигналов для записи в порт блока регистров

Вот описание логики слияния на языке HCL:

```
## Установка идентификатора регистра на адресном входе порта E
word w_dstE = [
    ## запись из valM
    W_dstM != RNONE : W_dstM;
    1: W_dstE;
];

## Установка значения на входе данных порта E
word w_valE = [
    W_dstM != RNONE : W_valM;
    1: W_valE;
];
```

Управление этими мультиплексорами осуществляется сигналом **dstE**; когда он указывает на наличие определенного регистра, выбирается значение для порта E, иначе выбирается значение для порта M.

В имитационной модели можно запретить использовать порт M блока регистров, как показано ниже:

```
## Запретить использование порта M блока регистров
## Установить сигнал на адресном входе порта M блока регистров
word w_dstM = RNONE;

## Установить сигнал на входе данных порта M блока регистров
word w_valM = 0;
```

Сложность заключается в том, чтобы придумать способ обработки инструкции `porq`. Один из вариантов – использовать управляющую логику для динамической обработки инструкции `porq rA` так, чтобы она оказывала тот же эффект, что и последовательность из двух инструкций:

```
iaddq $8, %rsp
mrmovq -8(%rsp), rA
```

(Описание команды `iaddq` представлено в упражнении 4.3.) Обратите внимание на порядок этих двух инструкций, гарантирующий корректную работу `porq %rsp`. Это можно сделать с помощью управляющей логики, которая на этапе декодирования будет обрабатывать `porq` так же, как `iaddq`, за исключением того, что следующее предсказанное значение РС будет равно текущему. В следующем цикле инструкция `porq` будет выбрана повторно, но код инструкции преобразуется в особое значение `IPOR2`. Оно интерпретируется как особая инструкция, обладающая тем же поведением, что и инструкция `movq` выше.

Файл `pipe-1w.hcl` содержит модифицированную логику управления портом порта записи, описанную выше. В ней объявляется константа `IPOR2` с шестнадцатеричным значением `E`. Здесь же имеется определение сигнала `f_icode`, генерирующего поле **icode** для конвейерного регистра D. Это определение можно модифицировать для вставки кода инструкции `IPOR2` при повторной выборке инструкции `porq`. Файл `HCL` также содержит объявление сигнала `f_pc` – значение счетчика инструкций, сгенерированное на этапе выборки блоком «Выбор РС» (см. рис. 4.50).

Модифицируйте управляющую логику в этом файле для обработки команд `porq` описанным способом. Загляните в лабораторные материалы, чтобы узнать, как создать симулятор для вашего решения и как его протестировать.

### Упражнение 4.59 ♦♦

Сравните производительность трех версий пузырьковой сортировки (упражнения 4.47, 4.48 и 4.49). Объясните, почему одна из версий работает лучше.

## Решения упражнений

### Решение упражнения 4.1

Кодирование инструкций вручную – занятие достаточно утомительное, однако это укрепляет понимание, как ассемблерный код превращается в последовательность байтов. В следующем выводе ассемблера Y86-64 каждая строка содержит адрес и последовательность байтов, начинающуюся с этого адреса.

```

1 0x100:                                | .pos 0x100 # Код начинается с адреса 0x100
2 0x100: 30f30f000000000000000000    |    irmovq $15,%rbx
3 0x10a: 2031                          |    rrmovq %rbx,%rcx
4 0x10c:                                | loop:
5 0x10c: 4013fdfffffffffffff          |    rmmovq %rcx,-3(%rbx)
6 0x116: 6031                          |    addq  %rbx,%rcx
7 0x118: 700c01000000000000000000    |    jmp  loop
```

Следует отметить несколько особенностей данной кодировки:

- десятичное число 15 (строка 2) имеет шестнадцатеричное представление `0x000000000000000f`. Записав эти байты в обратном порядке, получаем `0f 00 00 00 00 00 00 00`;
- десятичное число -3 (строка 5) имеет шестнадцатеричное представление `0xfffffffffffffd`. Записав эти байты в обратном порядке, получаем `fd ff ff ff ff ff ff ff`;
- код начинается с адреса `0x100`. Первая инструкция занимает 10 байт, вторая – 2. Следовательно, метка `loop` будет иметь адрес `0x0000010c`. Записав эти байты в обратном порядке, получаем `0c 01 00 00 00 00 00 00`.

## Решение упражнения 4.2

Декодирование последовательностей байтов вручную помогает понять задачу, стоящую перед процессором. Он должен прочитать последовательность байтов и определить, какие инструкции должны быть выполнены. Далее показан ассемблерный код, полученный на основе сгенерированной последовательности байтов. Слева от ассемблерного кода указаны также адрес и последовательности байтов для каждой инструкции.

1. Некоторые операции с непосредственными данными и адресами:

```
0x100: 30f3fcffffffffffff |    irmovq $-4,%rbx
0x10a: 406300080000000000 |    rmmovq %rsi,0x800(%rbx)
0x114: 00                      |    halt
```

2. Код, включающий вызовы функций:

```
0x200: a06f                      |    pushq %rsi
0x202: 800c02000000000000 |    call proc
0x20b: 00                      |    halt
0x20c:                      | proc:
0x20c: 30f30a000000000000 |    irmovq $10,%rbx
0x216: 90                      |    ret
```

3. Код, включающий недопустимый код инструкции 0xf0:

```
0x300: 505407000000000000 |    mrmovq 7(%rsp),%rbp
0x30a: 10                      |    nop
0x30b: f0                      | .byte 0xf0 # Код недопустимой инструкции
0x30c: b01f                    |    popq %rcx
```

4. Код, содержащий инструкцию перехода:

```
0x400:                      | loop:
0x400: 6113                    |    subq %rcx, %rbx
0x402: 730004000000000000 |    je loop
0x40b: 00                      |    halt
```

5. Код, содержащий недопустимый второй байт в инструкции pushq:

```
0x500: 6362                    |    xorq %rsi,%rdx
0x502: a0                      | .byte 0xa0 # Инструкция pushq
code
0x503: f0                      | .byte 0xf0 # Недопустимый байт-спецификатор
                                # регистра
```

## Решение упражнения 4.3

Используя инструкцию `iaddq`, мы можем переписать функцию `sum` так:

```
# long sum(long *start, long count)
# start в %rdi, count в %rsi
sum:
    xorq %rax,%rax        # sum = 0
    andq %rsi,%rsi        # Установить флаги
    jmp  test

loop:
    mrmovq (%rdi),%r10     # Получить *start
    addq  %r10,%rax        # Прибавить к сумме
    iaddq $8,%rdi          # start++
    iaddq $-1,%rsi         # count--

test:
    jne  loop              # Остановить по достижении 0
    ret
```

## Решение упражнения 4.4

На машине x86-64 компилятор GCC произведет следующий код для `rsum`:

```
long rsum(long *start, long count)
start в %rdi, count в %rsi
rsum:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L9
    pushq   %rbx
    movq    (%rdi), %rbx
    subq    $1, %rsi
    addq    $8, %rdi
    call    rsum
    addq    %rbx, %rax
    popq    %rbx
.L9:
    rep; ret
```

Его легко преобразовать в код для Y86-64:

```
# long rsum(long *start, long count)
# start в %rdi, count в %rsi
rsum:
    xorq %rax,%rax          # Установить возвращаемое значение равным 0
    andq %rsi,%rsi          # Установить флаги
    je     return           # Если count == 0, вернуть 0
    pushq %rbx              # Сохранить регистры
    mrmovq (%rdi),%rbx      # Получить *start
    irmovq $-1,%r10
    addq %r10,%rsi          # count--
    irmovq $8,%r10
    addq %r10,%rdi          # start++
    call rsum
    addq %rbx,%rax          # Прибавить *start к сумме
    popq %rbx              # Восстановить регистры
return:
    ret
```

## Решение упражнения 4.5

Эта задача дает вам возможность попробовать свои силы в написании ассемблерного кода.

```
1 # long absSum(long *start, long count)
2 # start в %rdi, count в %rsi
3 absSum:
4     irmovq $8,%r8          # Константа 8
5     irmovq $1,%r9          # Константа 1
6     xorq %rax,%rax         # сумма = 0
7     andq %rsi,%rsi         # Установить флаги
8     jmp test
9 loop:
10    mrmovq (%rdi),%r10      # x = *start
11    xorq %r11,%r11         # Константа 0
12    subq %r10,%r11         # -x
13    jle pos                # Пропустить, если -x <= 0
14    rrmovq %r11,%r10       # x = -x
15 pos:
```

```

16      addq %r10,%rax      # Прибавить к сумме
17      addq %r8,%rdi      # start++
18      subq %r9,%rsi      # count--
19 test:
20      jne     loop        # Остановить по достижении 0
21      ret

```

### Решение упражнения 4.6

Эта задача дает вам возможность попробовать свои силы в написании ассемблерного кода с условными перемещениями. Мы покажем только код цикла. Остальной код совпадает с кодом из упражнения 4.5.

```

9 loop:
10      mrmovq (%rdi),%r10  # x = *start
11      xorq  %r11,%r11     # Константа 0
12      subq  %r10,%r11     # -x
13      cmovg %r11,%r10     # Если -x > 0, то x = -x
14      addq  %r10,%rax     # Прибавить к сумме
15      addq  %r8,%rdi      # start++
16      subq  %r9,%rsi      # count--
17 test:
18      jne  loop           # Остановить по достижении 0

```

### Решение упражнения 4.7

Трудно найти практическое применение для данной конкретной инструкции, тем не менее ее следует учитывать при проектировании системы, чтобы избежать неоднозначной спецификации. Необходимо определить обоснованное поведение для инструкции и убедиться, что все наши реализации придерживаются этого соглашения.

Инструкция `subq` в данном тесте сравнивает начальное значение `%rsp` со значением, которое вталкивается в стек. Если результат вычитания равен нулю, значит, в стек будет помещено старое значение `%rsp`.

### Решение упражнения 4.8

Еще труднее представить необходимость выталкивания значения со стека в указатель стека. И все же мы должны определить соглашение и следовать ему. Этот код вталкивает `0xabcd` в стек, выталкивает его в `%rsp` и возвращает вытолкнутое значение. Поскольку результат равен `0xabcd`, можно сделать вывод, что `popq %rsp` должна записать в указатель стека значение, прочитанное из памяти. Поэтому данная инструкция эквивалентна инструкции `mrmovq (%rsp), %rsp`.

### Решение упражнения 4.9

Функция EXCLUSIVE-OR (ИСКЛЮЧАЮЩЕЕ ИЛИ) требует, чтобы два бита имели противоположные значения:

```
bool xor = (!a && b) || (a && !b);
```

Вообще говоря, сигналы `eq` и `xor` будут взаимно дополняющими, т. е. один будет равен единице, а другой – нулю.

### Решение упражнения 4.10

Выходы цепей EXCLUSIVE-OR будут дополнениями значений равенства битов. С помощью законов де Моргана (приложение в интернете DATA:BOOL, упражнение 2.8) можно реализовать операцию AND (И), используя операции OR (ИЛИ) и NOT (НЕ), что дает цепь, изображенную на рис. 4.62.

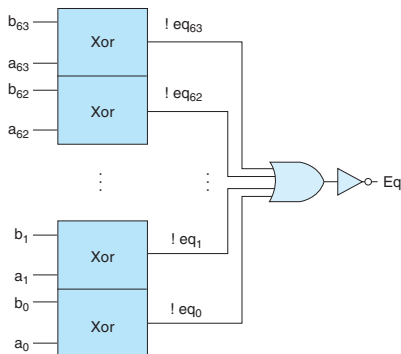


Рис. 4.62. Решение упражнения 4.10

Решение упражнения 4.11

Вторую часть case-выражения можно записать так:

```
B <= C :      B;
```

Первая строка определяет случай, когда A является минимальным элементом, поэтому во второй строке достаточно определить, какое из значений – B или C – является минимальным.

Решение упражнения 4.12

Это частный случай выбора минимального значения из трех:

```
word Med3 = [  
    A <= B && B <= C : B;  
    C <= B && B <= A : B;  
    B <= A && A <= C : A;  
    C <= A && A <= B : A;  
    1                  : C;  
];
```

Решение упражнения 4.13

Эти упражнения помогают конкретизировать вычисления, выполняемые этапами. Из объектного кода видно, что инструкция находится по адресу 0x016. Она занимает 10 байт, из которых первые два 0x30 и 0xf4. Последние 8 байт – это число 0x0000000000000080 (десятичное 128), записанное в обратном порядке.

Этап	В общем случае irmovq V, rB	Конкретно для irmovq \$128, %rsp
Выборка	icode : ifun ← M <sub>1</sub> [PC] rA :rB ← M <sub>1</sub> [PC + 1] valC ← M <sub>8</sub> [PC + 2] valP ← PC + 10	icode : ifun ← M <sub>1</sub> [0x016]= 3:0 rA :rB ← M <sub>1</sub> [0x017]= f:4 valC ← M <sub>8</sub> [0x018]= 128 valP ← 0x016 + 10 = 0x020
Декодирование		
Выполнение	valE ← 0 + valC	valE ← 0 + 128 = 128
Обращение к памяти		
Обратная запись	R[rB] ← valE	R[%rsp] ← valE = 128
Изменение PC	PC ← valP	PC ← valP = 0x020

Эта инструкция запишет 128 в регистр %rsp и увеличит PC на 10.

Решение упражнения 4.14

Инструкция находится по адресу 0x02c и состоит из двух байт со значениями 0xb0 и 0x00f. Регистр %rsp получил значение 120 после выполнения инструкции pushq (строка 6), которая сохранила в этой ячейке памяти число 9.

Этап	В общем случае irmovq V, rB	Конкретно для irmovq \$128, %rsp
Выборка	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$	$icode : ifun \leftarrow M_1[0x02c] = b:0$ $rA : rB \leftarrow M_1[0x02d] = 0:f$
	$valP \leftarrow PC + 2$	$valP \leftarrow 0x02c + 2 = 0x02e$
Декодирование	$valA \leftarrow R[\%rsp]$ $valB \leftarrow R[\%rsp]$	$valA \leftarrow R[\%rsp] = 120$ $valB \leftarrow R[\%rsp] = 120$
Выполнение	$valE \leftarrow valB + 8$	$valE \leftarrow 120 + 8 = 128$
Обращение к памяти	$valM \leftarrow M_8[valA]$	$valM \leftarrow M_8[120] = 9$
Обратная запись	$R[\%rsp] \leftarrow valE$ $R[rA] \leftarrow valM$	$R[\%rsp] \leftarrow 128$ $R[\%rax] \leftarrow 9$
Изменение PC	$PC \leftarrow valP$	$PC \leftarrow 0x02e$

Эта инструкция запишет 9 в регистр %rax, 128 – в %rsp и увеличит PC на 2.

Решение упражнения 4.15

Трассировка этапов в табл. 4.5, где **rA** равно %rsp, показывает, что на этапе обращения к памяти инструкция сохранит в памяти **valA** – первоначальное значение указателя стека, в точности как в архитектуре x86-64.

Решение упражнения 4.16

Трассировка этапов в табл. 4.5, где **rA** равно %rsp, показывает, что обе операции обратной записи изменяют %rsp. Поскольку операция, записывающая **valM**, выполняется последней, чистым эффектом от выполнения инструкции будет запись значения, прочитанного из памяти в %rsp, в точности как в архитектуре x86-64.

Решение упражнения 4.17

Для реализации условных перемещений требуется лишь незначительное изменение перемещений регистр–регистр. Мы добавляем в этап обратной записи результат проверки условия:

Этап	cmovXX rA, rB
Выборка	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 2$
Декодирование	$valA \leftarrow R[rA]$
Выполнение	$valE \leftarrow 0 + valA$ $Cnd \leftarrow Cond(CC, ifun)$
Обращение к памяти	
Обратная запись	$if (Cnd) R[rB] \leftarrow valE$
Изменение PC	$PC \leftarrow valP$

### Решение упражнения 4.18

Данная инструкция находится по адресу 0x037 и занимает 9 байт. Первый байт имеет значение 0x80; последние восемь – число 0x0000000000000041, записанное в обратном порядке, – адрес вызываемой функции. Команда `popq` (строка 7) установит указатель стека равным 128.

Этап	В общем случае <code>call Dest</code>	Конкретно для <code>call 0x041</code>
Выборка	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$	$\text{icode} : \text{ifun} \leftarrow M_1[0x037] = 8:0$
	$\text{valC} \leftarrow M_8[\text{PC} + 1]$	$\text{valC} \leftarrow M_8[0x038] = 0x041$
	$\text{valP} \leftarrow \text{PC} + 9$	$\text{valP} \leftarrow 0x037 + 9 = 0x040$
Декодирование	$\text{valB} \leftarrow R[\%rsp]$	$\text{valB} \leftarrow R[\%rsp] = 128$
Выполнение	$\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valE} \leftarrow 128 + (-8) = 120$
Обращение к памяти	$M_8[\text{valE}] \leftarrow \text{valP}$	$M_8[120] \leftarrow 0x040$
Обратная запись	$R[\%rsp] \leftarrow \text{valE}$	$R[\%rsp] \leftarrow 120$
Изменение PC	$\text{PC} \leftarrow \text{valC}$	$\text{PC} \leftarrow 0x041$

Эта инструкция запишет 120 в `%rsp`, сохранит по этому адресу значение 0x040 (адрес возврата) и запишет в PC число 0x041 (адрес вызываемой функции).

### Решение упражнения 4.19

Весь HCL-код в этом и в других упражнениях прост и понятен, однако попытка написать его самостоятельно поможет задуматься о различных инструкциях и способах их обработки. В данном упражнении можно просто взглянуть на систему команд Y86-86 (рис. 4.2) и определить, какая из них имеет поле константы.

```
bool need_valC =
    icode in { IIRMOVQ, IRMMOVQ, IMRMVQ, IJXX, ICALL };
```

### Решение упражнения 4.20

Этот код похож на код для **srcA**.

```
word srcB = [
    icode in { IOPQ, IRMMOVQ, IMRMVQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRS;
    1 : RNONE; # Регистры не используются
];
```

### Решение упражнения 4.21

Этот код похож на код для **dstE**.

```
word dstM = [
    icode in { IMRMVQ, IPOPQ } : rA;
    1 : RNONE; # Запись в регистры не выполняется
];
```

### Решение упражнения 4.22

Продолжая упражнение 4.16, для операции сохранения в регистре `%rsp` значения, прочитанного из памяти, более высокий приоритет мы должны дать записи через порт M.



### Решение упражнения 4.23

Этот код похож на код для **aluA**.

```
word aluB = [
    icode in { IRMMOVQ, IMRMVQ, IOPQ, ICALL,
               IPUSHQ, IRET, IPOPQ } : valB;
    icode in { IRRMOVQ, IIRMOVQ } : 0;
    # Другие инструкции не используют АЛУ
];
```

### Решение упражнения 4.24

Реализация условных перемещений на удивление проста: достаточно запретить запись в блок регистров, задавая регистр-приемник как **RNONE**, когда условие не выполняется.

```
word dstE = [
    icode in { IRRMOVQ } && Cnd : rB;
    icode in { IIRMOVQ, IOPQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Запись в регистры не выполняется
];
```

### Решение упражнения 4.25

Этот код похож на код для **mem\_addr**.

```
word mem_data = [
    # Значение из регистра
    icode in { IRMMOVQ, IPUSHQ } : valA;
    # Возврат PC
    icode == ICALL : valP;
    # По умолчанию: запись не выполняется
];
```

### Решение упражнения 4.26

Этот код похож на код для **mem\_read**.

```
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
```

### Решение упражнения 4.27

Для вычисления поля **Stat** требуется собрать информацию о состоянии на нескольких этапах:

```
## Определение состояния инструкции
word Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid : SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
];
```

### Решение упражнения 4.28

Это интересное упражнение на поиск оптимального баланса по набору блоков. Оно дает возможность попрактиковаться в вычислении пропускной способности и задержек в конвейерах.

1. Для конвейера с двумя этапами лучшим разделением будет размещение блоков А, В и С на первом этапе и блоков D, E и F – на втором. Время задержки на первом этапе составит 170 пс; общая продолжительность цикла –  $170 + 20 = 190$  пс. Следовательно, пропускная способность составит 5.26 GIPS, а время задержки – 380 пс.
2. Для конвейера с тремя этапами лучшим разделением будет размещение блоков А и В на первом этапе, блоков С и D – на втором и E и F – на третьем. Первые два этапа будут иметь задержку 110 пс; общая продолжительность цикла составит 130 пс, пропускная способность – 7.69 GIPS, а время задержки – 390 пс.
3. Для конвейера с четырьмя этапами лучшим разделением будет размещение блока А на первом этапе, блоков В и С – на втором, D – на третьем и блоков E и F – на четвертом. Второй этап потребует 90 пс, соответственно, общая продолжительность цикла составит 110 пс, пропускная способность – 9.09 GIPS, а время задержки – 440 пс.
4. Наиболее оптимальным будет разделение блоков на пять этапов, по одному на каждый блок, кроме блоков E и F, которые объединяются на пятом этапе. Общая продолжительность цикла в этом случае составит:  $80 + 20 = 100$  пс; пропускная способность – 10.00 GIPS и задержка – порядка 500 пс. Добавление еще одного этапа ничего не изменит, потому что конвейер не может выполнять цикл быстрее, чем за 100 пс.

### Решение упражнения 4.29

Каждый этап должен иметь комбинаторную логику со временем реакции  $300/k$  пс и конвейерный регистр со временем реакции 20 пс.

1. Общая задержка составит  $300 + 20k$  пс, а пропускная способность (в GIPS):

$$\frac{1000}{300 + 20k} - \frac{1000k}{300 + 20k}.$$

2. Если позволить  $k$  стремиться к бесконечности, то пропускная способность станет равной  $1000/20 = 50$  GIPS. Конечно, время задержки тоже приблизилось бы к бесконечности.

Это упражнение позволяет количественно оценить убывающую отдачу от увеличения глубины конвейера. При попытке разделить логику на множество этапов задержка в конвейерных регистрах становится ограничивающим фактором.

### Решение упражнения 4.30

Этот код очень похож на соответствующий код для SEQ. Единственное отличие – мы пока не можем определить, сгенерирует ли память данных сигнал ошибки для этой инструкции.

```
# Определить код состояния для выбранной инструкции
word f_stat = [
    imem_error: SADR;
    !instr_valid : SINS;
    f_icode == IHALT : SHLT;
    1 : SAOK;
];
```

### Решение упражнения 4.31

В код для SEQ нужно просто добавить префикс **D\_** в названия сигналов.

```
word d_dstE = [
    D_icode in { IRRMOVQ, IIRMOVQ, IOPQ } : D_rB;
    D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Запись в регистры не выполняется
];
```

### Решение упражнения 4.32

Инструкция `rrmovq` (строка 5) остановится на один цикл из-за риска загрузки/использования, вызванного инструкцией `popq` (строка 4). Когда `rrmovq` перейдет на этап декодирования, инструкция `popq` переместится на этап обращения к памяти, что обеспечит равенство **M\_dstE** и **M\_dstM** для `%rsp`. Если два этих случая поменять местами, то обратная запись из **M\_valE** окажется более приоритетной, что вызовет передачу увеличенного указателя стека в аргументе инструкции `rrmovq`. При этом возникнет расхождение с соглашением об обработке `popq %rsp`, которое мы определили в упражнении 4.8.

### Решение упражнения 4.33

Это упражнение дает возможность получить опыт решения одной из важнейших задач проектирования процессоров: создания тестовых программ. Вообще говоря, тестовые программы должны проверять все возможные ситуации, влекущие появление всех рисков и генерирующие некорректные результаты, если какая-то из зависимостей не обрабатывается должным образом.

Для данного примера можно воспользоваться немного модифицированной версией программы, показанной в упражнении 4.32.

```
1  irmovq $5, %rdx
2  irmovq $0x100, %rsp
3  rmmovq %rdx, 0(%rsp)
4  popq %rsp
5  nop
6  nop
7  rrmovq %rsp, %rax
```

Две инструкции `pop` перемещают инструкцию `popq` на этап обратной записи, когда инструкция `rrmovq` оказывается на этапе декодирования. Если неправильно установить приоритет для двух источников, продвигаемых с этапа обратной записи, то в регистр `%rax` будет записано увеличенное значение указателя стека, а не значение, прочитанное из памяти.

### Решение упражнения 4.34

Эта логика требует проверки пяти продвигаемых источников:

```
word d_valB = [
    d_srcB == e_dstE : e_valE; # Продвинуть valE с этапа выполнения
    d_srcB == M_dstM : m_valM; # Продвинуть valM с этапа обращения к памяти
    d_srcB == M_dstE : M_valE; # Продвинуть valE с этапа обращения к памяти
    d_srcB == W_dstM : W_valM; # Продвинуть valM с этапа обратной записи
    d_srcB == W_dstE : W_valE; # Продвинуть valE с этапа обратной записи
    1 : d_rvalB; # Использовать значение из блока регистров
];
```

### Решение упражнения 4.35

Это изменение не будет учитывать случаи, когда условное перемещение не удовлетворяет условию и устанавливает значение **dstE** равным **RNONE**. Значение результата

может быть передано следующей инструкции, даже если условное перемещение не выполняется.

```

1  irmovq $0x123,%rax
2  irmovq $0x321,%rdx
3  xorq %rcx,%rcx      # Флаги = 100
4  cmovne %rax,%rdx    # Перемещения не происходит
5  addq %rdx,%rdx      # Должно получиться 0x642
6  halt

```

Этот код инициализирует регистр `%rdx` значением `0x321`. Условное перемещение данных не происходит, поэтому последняя инструкция `addq`, удваивающая значение в `%rdx`, должна дать в результате `0x642`. Однако измененная логика продвигает значение для условного перемещения `0x123` на вход **valA** в АЛУ, а на вход **valB** передается правильное значение операнда `0x321`. Эти входные данные складываются, и получается результат `0x444`.

### Решение упражнения 4.36

Следующий код завершает вычисление кода состояния для этой инструкции:

```

## Обновить состояние
word m_stat = [
    dmem_error : SADR;
    1 : M_stat;
];

```

### Решение упражнения 4.37

Следующая тестовая программа предназначена для проверки комбинации А (см. рис. 4.58) и правильности всех операций:

```

1 # Код, генерирующий комбинацию из невыполняемого ветвления и инструкции ret
2     irmovq Stack, %rsp
3     irmovq rtnp,%rax
4     pushq %rax      # Установить указатель возврата
5     xorq %rax,%rax  # Установить флаг ZF
6     jne target     # Не выполняется (первая часть комбинации)
7     irmovq $1,%rax  # Должна выполняться эта инструкция
8     halt
9 target: ret        # Вторая часть комбинации
10    irmovq $2,%rbx   # Эта инструкция не должна выполняться
11    halt
12 rtnp: irmovq $3,%rdx # Эта инструкция не должна выполняться
13    halt
14 .pos 0x40
15 Stack:

```

Программа спроектирована так, что при возникновении каких-либо ошибок (например, при фактическом выполнении инструкции `ret`) она выполнит одну из дополнительных инструкций `irmovq` и остановится. То есть ошибка в конвейере вызовет изменение того или иного регистра. Этот код наглядно показывает необходимость со всем тщанием походить к разработке тестовых программ. Он должен создать условие для потенциальной ошибки, после чего выяснить, случится она или нет.

### Решение упражнения 4.38

Следующая тестовая программа создает условия для управляющей комбинации В (см. рис. 4.58). Имитатор автоматически выявляет ситуации, когда управляющие сигналы «пузырька» и останова для конвейерного регистра одновременно устанавливаются

в 0, поэтому тестовой программе остается только создать условия для ее выявления. Самое сложное здесь – заставить программу выполнять полезные действия при правильной обработке.

```

1 # Тестирование инструкции, модифицирующей %rsp, за которой следует ret
2     irmovq mem,%rbx
3     mrmovq 0(%rbx),%rsp # Записать в %rsp адрес точки возврата
4     ret                # Вернуться в точку возврата
5     halt               #
6 rtnpt: irmovq $5,%rsi  # Точка возврата
7     halt
8 .pos 0x40
9 mem: .quad stack       # Хранит желаемое значение указателя стека
10 .pos 0x50
11 stack: .quad rtnpt     # Вершина стека: хранит точку возврата

```

Эта программа использует два инициализированных слова в памяти. Первое (mem) хранит адрес второго (stack – желаемый указатель стека). Второе хранит адрес желаемой точки возврата для инструкции ret. Программа загружает указатель стека в %rsp и выполняет команду ret.

### Решение упражнения 4.39

Как показано в табл. 4.10, конвейерный регистр D необходимо приостановить из-за риска загрузки/использования:

```

bool D_stall =
    # Условия риска загрузки/использования
    E_icode in { IMRMVQ, IPOPQ } &&
    E_dstM in { d_srcA, d_srcB };

```

### Решение упражнения 4.40

Как показано в табл. 4.10, в конвейерный регистр E необходимо внедрить «пузырек» из-за риска загрузки/использования или неверного предсказания ветвления:

```

bool E_bubble =
    # Неверное предсказание ветвления
    (E_icode == IJXX && !e_Cnd) ||
    # Условия риска загрузки/использования
    E_icode in { IMRMVQ, IPOPQ } &&
    E_dstM in { d_srcA, d_srcB };

```

### Решение упражнения 4.41

Для этого требуется проанализировать код выполняемой инструкции и проверить исключения далее по конвейеру.

```

## Следует ли обновить флаги?
bool set_cc = E_icode == IOPQ &&
    # Состояние меняется только при нормальном выполнении
    !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };

```

### Решение упражнения 4.42

Для внедрения «пузырька» в этап обращения к памяти в следующем цикле требуется проверить наличие исключения на этапе обращения к памяти или обратной записи в текущем цикле.

```

# Внедрить «пузырек», если только имеется исключение на этапе обращения к памяти
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };

```

Для остановки этапа обратной записи достаточно просто проверить состояние инструкции на этом этапе. Если также остановить конвейер, когда исключительная инструкция находится на этапе обращения к памяти, то эта инструкция не сможет перейти на этап обратной записи.

```
bool W_stall = W_stat in { SADR, SINS, SHLT };
```

### Решение упражнения 4.43

В этом случае частота неверных прогнозов составит 0,35, подставляя ее в уравнение, получаем:  $mp = 0,20 \times 0,35 \times 2 = 0,14$ , и общее число циклов на инструкцию (CPI) составит 1,25. Такое улучшение выглядит существенным, однако оно заслуживает внимания, только если стоимость реализации новой стратегии прогнозирования ветвления не слишком высока.

### Решение упражнения 4.44

Этот упрощенный анализ, в котором мы должны сосредоточиться на внутреннем цикле, – полезный способ оценки производительности программ. Если массив достаточно велик, время, расходуемое на выполнение другого кода, будет незначительным.

1. Внутренний цикл, использующий условный переход, включает 9 инструкций, все они выполняются, когда элемент массива меньше или равен нулю, а когда элемент массива положительный, выполняется 8 инструкций. Среднее значение – 8,5. Внутренний цикл в версии, использующей условное перемещение, содержит 8 инструкций, которые выполняются всегда.
2. Переход в конце цикла будет предсказываться правильно, кроме случаев, когда цикл завершается. Для очень длинного массива влияние одного неверного предсказания пренебрежимо мало. Другой единственный источник «пузырьков» в коде, основанном на передаче управления, – это условный переход в точке проверки значения элемента массива. Неверное предсказание повлечет внедрение двух «пузырьков», но это будет происходить только в 50 % случаев, поэтому среднее значение равно 1,0. Код, основанный на условном перемещении, выполняется без внедрения «пузырьков».
3. Код на основе условной передачи управления требует в среднем  $8,5 + 1,0 = 9,5$  цикла на элемент массива (9 циклов в лучшем случае и 10 – в худшем), в то время как код на основе условного перемещения требует 8,0 цикла во всех случаях.

За неверное предсказание ветвления наш конвейер накладывает штраф всего в два цикла – это намного меньше, чем в глубоких конвейерах высокопроизводительных процессоров. В результате использование инструкции условного перемещения вместо условного перехода не сильно влияет на общую производительность программы.

# Глава 5

## Оптимизация производительности программ

- 5.1. Возможности и ограничения оптимизирующего компилятора.
  - 5.2. Выражение производительности программы.
  - 5.3. Пример программы.
  - 5.4. Устранение неэффективности в циклах.
  - 5.5. Сокращение вызовов процедур.
  - 5.6. Устранение избыточных ссылок на память.
  - 5.7. Общее описание современных процессоров.
  - 5.8. Развертывание циклов.
  - 5.9. Увеличение степени параллелизма.
  - 5.10. Обобщение результатов оптимизации комбинирующего кода.
  - 5.11. Некоторые ограничивающие факторы.
  - 5.12. Понятие производительности памяти.
  - 5.13. Жизнь в реальном мире: методы повышения производительности.
  - 5.14. Выявление и устранение узких мест производительности.
  - 5.15. Итоги.
- Библиографические заметки.
- Домашние задания.
- Решения упражнений.

**Г**лавная цель при разработке программ – правильная работа кода во всех возможных условиях. Программа, которая работает быстро, но дает неверные результаты, бесполезна. Программисты должны писать ясный и компактный код не только для того, чтобы было проще в нем разбираться, но также для того, чтобы другие могли читать и понимать код во время его проверки и когда потом потребуется внести какие-нибудь изменения.

С другой стороны, во многих случаях скорость выполнения программы тоже является важным фактором. Если программа должна обрабатывать видеокadres или сетевые пакеты в масштабе реального времени, то медленно работающая программа не сможет обеспечить необходимую функциональность. Когда вычислительная задача настолько требовательна, что для ее выполнения требуются дни или недели, то увеличение ее производительности даже на 20 % может иметь большое значение. В этой главе мы

посмотрим, как можно ускорить работу программ с помощью нескольких приемов оптимизации.

Чтобы написать эффективную программу, необходимо, во-первых, подобрать наилучший набор алгоритмов и структур данных и, во-вторых, написать код, который компилятор сможет превратить в эффективный выполняемый код. Для последнего важно понимать возможности и ограничения оптимизирующих компиляторов. Внешне незначительные изменения в исходном коде программы могут радикально повлиять на успех ее оптимизации компилятором. Одни языки программирования оптимизируются проще, другие – сложнее. Некоторые особенности С, например поддержка арифметики указателей и приведение типов, затрудняют оптимизацию. Часто программисты могут писать программы, облегчающие задачу компиляторов по генерированию эффективного кода. В-третьих (это особенно касается преимущественно вычислительных задач), необходимо разделить задачи на части, которые можно вычислять параллельно, используя преимущества многоядерных и многопроцессорных систем. Мы пока отложим этот аспект повышения производительности и вернемся к нему в главе 12. Даже при использовании параллельных вычислений важно, чтобы каждый вычислительный поток выполнялся с максимальной скоростью, поэтому сведения, что приводятся в этой главе, не теряют своей актуальности в любом случае.

В ходе разработки программы и ее оптимизации необходимо понимать, как будет использоваться ее код и какие критические факторы могут на него повлиять. В принципе, программист всегда должен искать компромиссы между простотой реализации и обслуживанием программы и ее быстродействием. На уровне алгоритмов простую сортировку вставкой можно написать за считанные минуты, но реализация и оптимизация высокоэффективной процедуры сортировки может занять несколько дней. На уровне кода многие низкоуровневые оптимизации могут ухудшать читаемость и удобство сопровождения, делать программы более чувствительными к ошибкам и затруднять возможность их модификации или расширения. Для многократно используемой программы, производительность которой является важным фактором, расширенные оптимизации обычно весьма уместны. Одна из проблем – сохранить некоторую степень элегантности и удобочитаемости кода, несмотря на обширные преобразования при оптимизации.

В этой главе описывается несколько методик повышения производительности кодов. В идеале компилятор должен принимать любой написанный нами код и генерировать максимально эффективный машинный код с заданным поведением. Современные компиляторы используют сложные виды анализа и оптимизации и продолжают совершенствоваться. Однако даже лучшим компиляторам могут помешать аспекты поведения программы, *блокирующие ее оптимизацию*, которые сильно зависят от среды выполнения. Программисты должны «помогать» компилятору, создавая такой код, который легко поддается оптимизации.

Первый шаг в оптимизации программы – устранение ненужной работы, чтобы код выполнял намеченную задачу с максимальной эффективностью. Под этим подразумевается устранение ненужных вызовов функций, условных проверок и обращений к памяти. Эти оптимизации не зависят от каких-либо конкретных свойств целевой машины.

Чтобы добиться максимальной производительности, создатель программы и компилятор должны иметь модель целевой машины, определяющую способы обработки инструкций и временных характеристик различных операций. Например, компилятор должен иметь информацию о времени выполнения, чтобы определить, что лучше использовать – инструкцию умножения или некую комбинацию сдвигов и сложений. В современных компьютерах используются очень сложные приемы обработки машинных программ с выполнением многих команд параллельно и часто в другом порядке, чем в программе. Чтобы добиться максимальной скорости выполнения своих программ,



программисты должны понимать, как работают процессоры. Далее мы рассмотрим высокоуровневую модель такой машины, основываясь на последних достижениях производителей процессоров Intel и AMD. Мы также рассмотрим приемы отображения *потоков данных* для визуализации работы процессоров.

Понимая, как работает процессор, мы сможем сделать второй шаг в оптимизации программы и использовать способность процессоров выполнять *несколько инструкций параллельно*. Мы рассмотрим несколько приемов преобразования программ, помогающих уменьшить зависимости по данным между разными этапами вычислений и увеличить степень параллелизма.

В конце главы мы обсудим вопросы, имеющие отношение к оптимизации крупных программ. Расскажем, как пользоваться *профилировщиками* – инструментами оценки производительности различных частей программы. Такой анализ может помочь обнаружить неэффективные участки кода, на которые следует обратить особое внимание в процессе оптимизации.

В представленном обсуждении оптимизация кода выглядит простым и линейным процессом применения серии преобразований в определенном порядке. На самом деле задача не так проста, и для ее решения требуется большой объем исследований и поиск оптимальных решений методом проб и ошибок. Это становится особенно очевидным на последних этапах оптимизации, когда внешне мелкие изменения могут вызывать переверот в производительности, а многообещающие методы себя абсолютно не оправдывают. Как наглядно показывают примеры в этой главе, иногда довольно трудно вразумительно объяснить, почему та или иная последовательность инструкций имеет ту или иную продолжительность выполнения. Производительность может зависеть от множества мелких особенностей процессора, которые никак или почти никак не отражены в документации. Это еще одна причина пробовать вариации и комбинации различных приемов.

Изучение ассемблерного кода – один из самых эффективных способов достичь понимания работы компилятора и особенностей выполнения сгенерированного кода. Хорошей стратегией считается сначала тщательно изучить код на наличие в нем вложенных циклов и операций, снижающих производительность, таких как избыточные ссылки на ячейки памяти. Начав с ассемблерного кода, можно предсказать, какие операции будут выполняться параллельно и насколько полно они будут использовать ресурсы процессора. Как вы увидите в этой главе, часто можно определить время (или, по крайней мере, нижнюю границу времени), необходимое для выполнения цикла, выявив *критические пути*, цепочки зависимостей данных, которые формируются во время повторных выполнений цикла, а затем вернуться и изменить исходный код, чтобы попытаться помочь компилятору выбрать эффективную реализацию.

Большинство компиляторов, включая GCC, постоянно совершенствуются, с точки зрения возможностей оптимизации. Поэтому одна из эффективных стратегий состоит в том, чтобы переписать программу и довести ее до такого состояния, когда компилятор сможет сгенерировать оптимальный код. Так можно избежать ухудшения читабельности, модульности и переносимости кода, как если бы мы были вынуждены использовать компилятор с минимальными возможностями. Эта стратегия помогает итеративно изменять программу и анализировать ее производительность, измеряя скорость выполнения и изучая сгенерированный ассемблерный код.

Начинающим программистам может показаться странной рекомендация постоянно изменять исходный код, пытаясь уговорить компилятор сгенерировать эффективный код, но на самом деле так пишутся многие высокопроизводительные программы. По сравнению с разработкой кода на языке ассемблера такой косвенный подход имеет одно важное преимущество – полученный код будет работать на других машинах, хотя, может быть, и не с максимальной производительностью.

## 5.1. Возможности и ограничения оптимизирующих компиляторов

Современные компиляторы используют сложные алгоритмы для определения вычисляемых в программе значений и особенностей их использования. Они могут применять разные способы упрощения выражений для организации единственного вычисления в нескольких частях системы и сокращения повторных вычислений. Большинство компиляторов, включая GCC, предоставляют пользователям определенный контроль над применяемыми оптимизациями. Как обсуждалось в главе 3, самым простым элементом управления является выбор *уровня оптимизации*. Например, параметр командной строки `-Og` сообщает компилятору GCC, что тот может применить только базовый набор оптимизаций.

Параметр `-O1` или выше (например, `-O2` или `-O3`) заставит GCC применить дополнительные оптимизации, которые могут еще больше повысить производительность программы, но при этом увеличить размер программы и затруднить ее отладку с применением стандартных средств. В нашем обсуждении мы в основном будем рассматривать код, скомпилированный с уровнем оптимизации `-O1`, хотя для большинства программных проектов общепринято использовать уровень `-O2`. Мы намеренно ограничиваем уровень оптимизации, чтобы показать, как разные способы оформления функций на языке C могут влиять на эффективность кода, сгенерированного компилятором. Вы увидите, что можно написать такой код на C, который при компиляции только с параметром `-O1` будет значительно превосходить более простую версию, скомпилированную с максимальным уровнем оптимизации.

Компиляторы должны соблюдать осторожность и применять только безопасные оптимизации, чтобы получившаяся программа имела такое же поведение, как и неоптимизированная версия, во всех возможных случаях, в пределах гарантий, предоставляемых стандартом языка C. Ограничение компилятора только безопасными оптимизациями устраняет возможные источники нежелательного поведения во время выполнения, но также означает, что программист должен прилагать больше усилий при разработке программ и писать такой код, чтобы компилятор мог затем преобразовать его в эффективный код машинного уровня. Чтобы понять, насколько сложно решить, какие преобразования безопасны, рассмотрим следующие две процедуры:

```

1 void twiddle1(long *xp, long *yp)
2 {
3     *xp += *yp;
4     *xp += *yp;
5 }
6
7 void twiddle2(long *xp, long *yp)
8 {
9     *xp += 2* *yp;
10 }
```

На первый взгляд кажется, что обе процедуры имеют идентичное поведение. Обе дважды прибавляют значение, хранящееся в ячейке памяти, на которую ссылается указатель `yp`, к значению в ячейке памяти, на которую ссылается указатель `xp`. Однако функция `twiddle2` более эффективна. Она только три раза ссылается на ячейки памяти (читает `*xp`, читает `*yp`, записывает в `*xp`), тогда как `twiddle1` обращается к памяти шесть раз (дважды читает `*xp`, дважды читает `*yp` и дважды записывает в `*xp`). Следовательно, если компилятору предложить скомпилировать процедуру `twiddle1`, то может показаться, что тот может сгенерировать более эффективный код, взяв за основу вычисления, выполняемые функцией `twiddle2`.

Но давайте рассмотрим случай, когда `xp` и `yp` равны. Тогда функция `twiddle1` выполнит следующие вычисления:

```
3 *xp += *xp; /* Удвоит значение в ячейке с адресом xp */
4 *xp += *xp; /* Удвоит значение в ячейке с адресом xp */
```

В результате значение в ячейке с адресом `xp` будет увеличено в 4 раза. Функция `twiddle2` выполнит следующее вычисление:

```
9 *xp += 2* *xp; /* Утроит значение в ячейке с адресом xp */
```

В результате значение в ячейке памяти, на которую ссылается указатель `xp`, будет увеличено в 3 раза. Компилятор не имеет никакой информации о том, как будет вызываться `twiddle1`, поэтому он должен предположить, что аргументы `xp` и `yp` могут быть равными. Следовательно, компилятор не может сгенерировать код в стиле `twiddle2`, пытаясь оптимизировать `twiddle1`.

Этот феномен, когда два указателя могут ссылаться на одну и ту же ячейку памяти, называют *наложением указателей*. Заботясь о безопасности оптимизации, компилятор должен предположить, что разные указатели могут ссылаться на одну и ту же ячейку памяти. Рассмотрим еще один пример программы с указателями `p` и `q`:

```
x = 1000; y = 3000;
*q = y; /* 3000 */
*p = x; /* 1000 */
t1 = *q; /* 1000 или 3000 */
```

Какое значение будет записано в переменную `t1`, зависит от равенства указателей `p` и `q` – если они равны, то `t1` получит значение 1000, иначе – 3000, но если да, то будет равно 1000. Это приводит к одному из основных препятствий оптимизации – аспектам программ, которые могут серьезно ограничить возможности компилятора генерировать оптимизированный код. Если компилятору не удастся определить, могут ли два указателя быть равными, он должен предположить, что возможен любой случай, и ограничить набор применяемых оптимизаций.

### Упражнение 5.1 (решение в конце главы)

Это упражнение иллюстрирует, как эффект наложения указателей может спровоцировать неожиданное поведение программы. Рассмотрим следующую процедуру обмена местами двух значений:

```
1 /* Меняет местами значение x, на которое ссылается указатель xp,
2  и значение y, на которое ссылается указатель yp */
3 void swap(long *xp, long *yp)
4 {
5     *xp = *xp + *yp; /* x+y      */
6     *yp = *xp - *yp; /* x+y-y = x */
7     *xp = *xp - *yp; /* x+y-x = y */
8 }
```

Что получится в результате, если этой процедуре передать один и тот же адрес в указателях `xp` и `yp`?

Второе препятствие оптимизации обусловлено обращением к функции. В качестве примера рассмотрим две следующие процедуры:

```
1 long f();
2
```

```

3 long func1() {
4     return f() + f() + f() + f();
5 }
6
7 long func2() {
8     return 4*f();
9 }

```

Может показаться, что обе процедуры вычисляют один и тот же результат, только func2 вызывает f один раз, а func1 – четыре раза. Заманчиво было бы сгенерировать для func1 код в стиле func2.

### Оптимизация вызовов функций методом встраивания

Код, включающий вызовы функций, можно оптимизировать с помощью метода, известного как *встраивание*, когда вызов функции заменяется кодом тела функции. Например, код функции func1 можно расширить, встроив в него четыре экземпляра функции f:

```

1 /* Результат встраивания функции f в func1 */
2 long func1in() {
3     long t = counter++; /* +0 */
4     t += counter++;     /* +1 */
5     t += counter++;     /* +2 */
6     t += counter++;     /* +3 */
7     return t;
8 }

```

Такая трансформация снижает накладные расходы на вызовы функций и позволяет дополнительно оптимизировать расширенный код. Например, компилятор может объединить обновления глобальной переменной counter в func1in и создать оптимизированную версию функции:

```

1 /* Оптимизация встроенного кода */
2 long func1opt() {
3     long t = 4 * counter + 6;
4     counter += 4;
5     return t;
6 }

```

Этот код точно воспроизводит поведение func1 для данного конкретного определения функции f.

Последние версии GCC пытаются использовать эту форму оптимизации при наличии параметра командной строки -finline или когда задан уровень оптимизации -O1 и выше. К сожалению, GCC пытается встраивать только функции, определенные в одном файле. То есть эта оптимизация не применяется в общем случае, когда набор библиотечных функций, определенных в одном файле, вызывается функциями из других файлов.

Иногда, однако, бывает желательно запретить компилятору выполнять встраивание. Первый такой случай – когда код предполагается отлаживать с помощью символического отладчика, такого как GDB, как описано в разделе 3.10.2. Если вызов функции будет оптимизирован методом встраивания, то любая попытка выполнить трассировку или установить точку останова в этом вызове потерпит неудачу. Второй случай – оценка производительности программы с помощью профилировщика, как обсуждается в разделе 5.14.1. Вызовы функций, которые были исключены методом встраивания, будут профилироваться неверно.

Но не будем спешить и рассмотрим следующее определение f:

```

1 long counter = 0;
2
3 long f() {
4     return counter++;
5 }

```

Эта функция имеет *побочный эффект*: она модифицирует некоторую часть глобального состояния программы. Изменение количества вызовов данной функции изменяет поведение программы. В частности, вызов func1 вернет  $0 + 1 + 2 + 3 = 6$ , тогда как вызов func2 вернет  $4 \cdot 0 = 0$ , если предположить, что обе функции вызывались со значением 0 в глобальной переменной counter.

Большинство компиляторов не пытаются определить, свободна ли функция от побочных эффектов и пригодна ли для оптимизации, как в случае с func2. Вместо этого они предполагают худший вариант и оставляют все обращения к функциям нетронутыми.

Из существующих компиляторов GCC считается наиболее адекватным, но не единственным, в том, что касается возможностей оптимизации. Он выполняет базовые оптимизации, но не выполняет радикальных преобразований программ, которые допускают более «агрессивные» компиляторы. Вследствие этого программисты, использующие GCC, должны прикладывать больше усилий при разработке программ, чтобы упростить компилятору задачу создания эффективного кода.

## 5.2. Выражение производительности программы

Далее мы будем использовать единицу измерения производительности программ CPE (Cycles Per Element – циклов на элемент). Эта единица измерения помогает весьма точно оценить производительность цикла итеративной программы. Она подходит для программ с повторяющимися вычислениями, такими как обработка пикселей в графическом изображении или вычисление элементов в произведении матриц.

Выполнение операций процессором управляется тактовым генератором, регулярно подающим сигнал синхронизации с определенной частотой, обычно измеряемой *гигагерцами* (ГГц) – миллиардами в секунду. Например, если в документации утверждается, что процессор работает на тактовой частоте 4 ГГц, то это означает, что тактовый генератор генерирует сигнал синхронизации с частотой  $4 \times 10^9$  раз в секунду. Время, необходимое для выполнения каждого цикла синхронизации, определяется как обратная величина тактовой частоте и обычно выражается в *наносекундах* (1 наносекунда равна  $10^{-9}$  с) или в *пикосекундах* (1 пикосекунда равна  $10^{-12}$  с). Например, период тактового генератора с частотой 4 ГГц составляет 0,25 нс, или 250 пс. С точки зрения программиста выражение производительности в тактовых циклах выглядит более наглядным, чем в наносекундах или пикосекундах, потому что эта единица измерения выражает количество выполненных инструкций, а не скорость работы тактового генератора.

Многие процедуры содержат цикл, выполняющий итерации через множество элементов. Например, функции psum1 и psum2, представленные в листинге 5.1, вычисляют *накопленную сумму* двух векторов с длиной  $n$ .

Для вектора  $\vec{a} = \langle a_0, a_1, \dots, a_{n-1} \rangle$  накопленная сумма  $\vec{p} = \langle p_0, p_1, \dots, p_{n-1} \rangle$  определяется так:

$$\begin{aligned}
 p_0 &= a_0, \\
 p_i &= p_{i-1} + a_i, \quad 1 \leq i < n.
 \end{aligned}
 \tag{5.1}$$

**Листинг 5.1.** Функции вычисления накопленной суммы. Эти функции наглядно демонстрируют способы выражения производительности программ

```

1 /* Вычисляет накопленную сумму вектора a */
2 void psum1(float a[], float p[], long n)

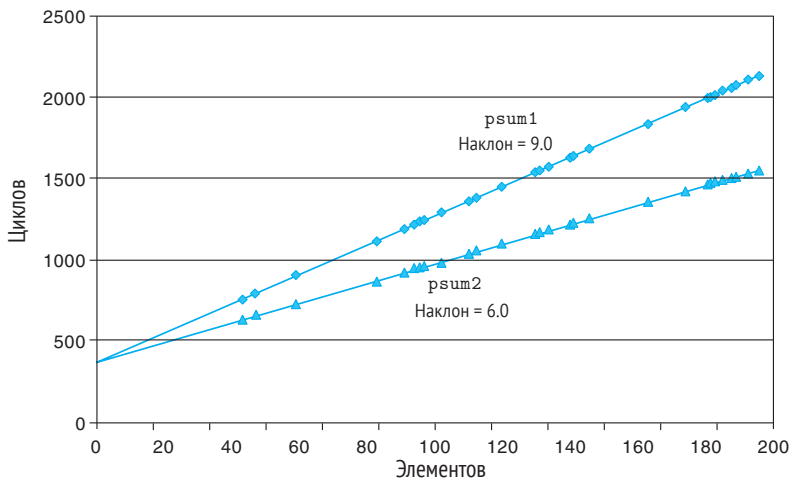
```

```

3 {
4     long i;
5     p[0] = a[0];
6     for (i = 1; i < n; i++)
7         p[i] = p[i-1] + a[i];
8 }
9
10 void psum2(float a[], float p[], long n)
11 {
12     long i;
13     p[0] = a[0];
14     for (i = 1; i < n-1; i+=2) {
15         float mid_val = p[i-1] + a[i];
16         p[i] = mid_val;
17         p[i+1] = mid_val + a[i+1];
18     }
19     /* Для нечетного значения n добавить последний элемент */
20     if (i < n)
21         p[i] = p[i-1] + a[i];
22 }

```

Функция `psum1` вычисляет один элемент вектора-результата в каждой итерации. Функция `psum2` применяет метод, известный как *развертывание цикла* (loop unrolling), чтобы вычислить два элемента в каждой итерации. Мы еще поговорим о преимуществах развертывания циклов далее в этой главе. (Дополнительный анализ оптимизации процедуры вычисления накопленной суммы вы найдете в упражнениях 5.11, 5.12 и 5.19.)



**Рис. 5.1.** Производительность функций вычисления накопленной суммы. Наклон линий отражает количество циклов, затраченных на обработку одного элемента (CPE)

Время выполнения такой процедуры можно охарактеризовать как некоторую константу, плюс коэффициент, пропорциональный количеству обработанных элементов. Например, на рис. 5.1 показана диаграмма зависимости числа циклов синхронизации, необходимого двум функциям, от количества элементов  $n$ . Используя *подгонку методом наименьших квадратов*, мы обнаружили, что значения времени выполнения функ-

ций `psum1` и `psum2` (в циклах синхронизации) можно аппроксимировать уравнениями  $368 + 9.0n$  и  $368 + 6.0n$  соответственно. Согласно этим уравнениям, 368 циклов – это накладные расходы на инициализацию и завершение процедуры, а также на подготовку цикла, и 6.0 или 9.0 цикла занимает обработка одного элемента. Для больших значений  $n$  (например, больше 200) большая часть времени выполнения будет приходиться на второе слагаемое с коэффициентом. Такие коэффициенты мы называем эффективным числом циклов на элемент и предпочитаем измерять количество циклов на элемент, а не количество циклов на итерацию, потому что такие методы, как развертывание цикла, позволяют использовать меньшее количество итераций для выполнения вычислений, но наша главная забота – насколько быстро процедура обработает вектор с данной длиной. Далее мы сосредоточим усилия на минимизации CPE. По этому показателю `psum2` с CPE 6.0 превосходит `psum1` с CPE 9.0.

### Что такое подгонка методом наименьших квадратов?

Часто бывает нужно провести прямую линию через набор точек данных  $(x_1, y_1), \dots, (x_n, y_n)$ , которая наилучшим образом аппроксимирует зависимость между значениями  $X$  и  $Y$ , имеющуюся в этих данных. Обычно для этого используется метод наименьших квадратов, помогающий найти линию вида  $y = mx + b$ , которая минимизирует следующую меру ошибки:

$$E(m, b) = \sum_{i=1, n} (mx_i + b - y_i)^2.$$

Алгоритм вычисления  $m$  и  $b$  можно получить путем нахождения производных от  $E(m, b)$  относительно  $m$  и  $b$  и установки их в 0.

### Упражнение 5.2 (решение в конце главы)

Далее в главе мы возьмем за основу одну функцию и создадим из нее много различных вариантов, обладающих тем же поведением, но разной производительностью. Для трех из этих вариантов обнаружилось, что время выполнения (в циклах синхронизации) можно аппроксимировать следующими функциями:

- $60 + 35n$ ;
- $136 + 4n$ ;
- $157 + 1.25n$ .

При каких значениях  $n$  каждая из этих версий будет выполняться быстрее остальных? Помните, что  $n$  – всегда целое число.

## 5.3. Пример программы

Чтобы показать, как из абстрактной программы можно путем систематической трансформации получить более эффективную версию, рассмотрим простую векторную структуру данных, показанную на рис. 5.2.

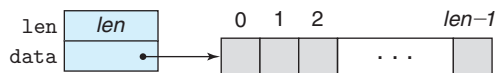


Рис. 5.2. Абстрактный векторный тип.

Вектор включает заголовок и массив указанной длины

Вектор изображен в виде двух блоков памяти: заголовка и самих данных. Заголовок объявлен, как показано ниже:

---

```
1 /* Абстрактный тип данных, представляющий вектор */
2 typedef struct {
3     long len;
4     data_t *data;
5 } vec_rec, *vec_ptr;
```

---

*code/opt/vec.h*

В этом объявлении `data_t` обозначает тип данных элементов вектора. В своих оценках мы будем измерять производительность кода при работе с целочисленными данными (`int` и `long`) и с данными с плавающей точкой (`float` и `double`). Для этого программа будет компилироваться и выполняться с разными объявлениями типа `data_t`, например с типом `long`:

```
typedef long data_t;
```

Для хранения данных программа будет распределять массив объектов типа `data_t` с длиной `len`.

В листинге 5.2 показаны некоторые базовые процедуры, которые генерируют векторы, обращаются к элементам вектора и определяют длину вектора. Обратите внимание на важную особенность: процедура доступа к элементам вектора `get_vec_element` проверяет выход за границы вектора. Этот код похож на код для работы с массивами, используемый во многих других языках, включая Java. Проверки выхода за границы вектора снижают вероятность появления программной ошибки, но, как будет видно далее, также существенно влияют на производительность программы.

**Листинг 5.2.** Реализация абстрактного векторного типа данных. В реальных программах тип данных `data_t` объявляется как `int`, `long`, `float` или `double`

---

```
1 /* Создает вектор указанной длины */
2 vec_ptr new_vec(long len)
3 {
4     /* Выделить память для заголовка */
5     vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
6     data_t *data = NULL;
7     if (!result)
8         return NULL; /* Ошибка выделения памяти */
9     result->len = len;
10    /* Выделить массив для элементов */
11    if (len > 0) {
12        data = (data_t *)calloc(len, sizeof(data_t));
13        if (!data) {
14            free((void *) result);
15            return NULL; /* Ошибка выделения памяти */
16        }
17    }
18    /* В data будет записан NULL или адрес памяти, выделенной для массива */
19    result->data = data;
20    return result;
21 }
22
23 /*
```

---

*code/opt/vec.h*



```

24 * Извлекает элемент вектора и сохраняет его в dest.
25 * Возвращает 0 (выход за границы) или 1 (успех)
26 */
27 int get_vec_element(vec_ptr v, long index, data_t *dest)
28 {
29     if (index < 0 || index >= v->len)
30         return 0;
31     *dest = v->data[index];
32     return 1;
33 }
34
35 /* Возвращает длину вектора */
36 long vec_length(vec_ptr v)
37 {
38     return v->len;
39 }

```

*code/opt/vec.h*

В качестве примера оптимизации рассмотрим код в листинге 5.3, объединяющий все элементы вектора в одно значение, выполняя определенную операцию. Используя разные определения констант времени компиляции IDENT и OP, код можно перекомпилировать для выполнения различных операций с данными. В частности, со следующими объявлениями:

```

#define IDENT 0
#define OP    +

```

вычисляется сумма элементов вектора, а с объявлениями

```

#define IDENT 1
#define OP    *

```

– их произведение.

**Листинг 5.3.** Начальная реализация объединяющей операции. Используя разные объявления нейтрального элемента IDENT и операции OP, можно измерить производительность различных операций

```

1 /* Максимально абстрактная реализация */
2 void combine1(vec_ptr v, data_t *dest)
3 {
4     long i;
5
6     *dest = IDENT;
7     for (i = 0; i < vec_length(v); i++) {
8         data_t val;
9         get_vec_element(v, i, &val);
10        *dest = *dest OP val;
11    }
12 }

```

Далее мы рассмотрим серию преобразований кода, написав разные версии объединяющей функции (функции, выполняющей указанную операцию с элементами вектора). Для целей нашего обсуждения мы измерили производительность функций на компьютере в процессором Intel Core i7 Haswell, который далее будем называть *эталонной*

машиной. Некоторые характеристики этого процессора были приведены в разделе 3.1. Полученные измерения характеризуют производительность с точки зрения выполнения программы на одной конкретной машине, поэтому нет гарантии, что сопоставимая производительность будет получена на компьютерах с процессорами других типов и/или с другими компиляторами. Тем не менее мы сравнили результаты, полученные на разных машинах и с разными компиляторами, и обнаружили, что в целом они согласуются с представленными здесь.

Демонстрируя ряд преобразований, мы покажем, что многие из них дают лишь минимальное увеличение производительности, однако некоторые производят по-настоящему драматический эффект. Выбор комбинаций преобразований для применения является частью «черной магии». Некоторые комбинации, не обеспечивающие измеримых преимуществ, действительно неэффективны, в то время как другие открывают перед компиляторами дополнительные возможности для оптимизации. Как показывает наш опыт, лучший подход – использовать комбинацию экспериментов и анализа: опробование различных подходов, измерение и изучение полученного ассемблерного кода для выявления основных узких мест в производительности.

В качестве исходной точки в следующей таблице представлены результаты измерений CPE производительности `combine1` на нашей эталонной машине с разными комбинациями операций (сложение и умножение) и типов данных (длинных целых и с плавающей точкой двойной точности). По результатам измерений на множестве разных программ мы выяснили, что операции с 32- и 64-разрядными целыми имеют равную производительность, за исключением кода, выполняющего деление. Аналогично мы выяснили, что идентичную производительность имеют также операции с числами с плавающей точкой одинарной и двойной точности. Поэтому в наших таблицах мы будем показывать результаты при обработке целочисленных данных и данных с плавающей точкой без привязки к конкретному типу.

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
<code>combine1</code>	5.3	Абстрактный неоптимизированный	22,68	20,02	19,98	20,18
<code>combine1</code>	5.3	Абстрактный -01	10,12	10,12	10,17	11,14

Как видите, наши измерения несколько неточны. Более вероятное число CPE для целочисленной суммы – 23,00, а не 22,68, а для целочисленного произведения – скорее всего, 20,0, а не 20,02. Однако мы не будем корректировать наши числа, чтобы они выглядели красиво, а представим результаты измерений, которые получили в действительности. Есть много факторов, которые усложняют надежное измерение точного количества циклов синхронизации, необходимых для выполнения некоторой последовательности кода. При проверке эти результаты полезно мысленно округлять в большую или меньшую сторону на несколько сотых тактового цикла.

Неоптимизированный код является результатом прямой трансляции исходного кода на С в машинный код, и его неэффективность очевидна. Просто добавив параметр командной строки `-01`, мы активировали применение базового набора оптимизаций, и, как видите, это значительно улучшило производительность программы – более чем в 2 раза – без каких-либо усилий со стороны программиста. В общем случае полезно взять в привычку включать некоторый уровень оптимизации. (Аналогичные результаты производительности были получены с уровнем оптимизации `-0g`.) Далее в наших экспериментах по созданию и измерению производительности программ мы будем использовать уровни оптимизации `-01` и `-02`.

## 5.4. Устранение неэффективностей в циклах

Обратите внимание, что процедура `combine1` в листинге 5.3 вызывает функцию `vec_length` для проверки условия продолжения цикла `for`. Как мы уже говорили при обсуждении трансляции циклов в машинный код (раздел 3.6.7), проверяемое условие вычисляется в каждой итерации. С другой стороны, длина вектора не меняется от итерации к итерации, поэтому длину вектора можно вычислить только однажды и использовать полученное значение в условии.

В листинге 5.4 показана модифицированная версия – функция `combine2`. Она вызывает `vec_length` в самом начале и сохраняет результат в локальной переменной `length`. Эта локальная переменная затем используется в условии цикла `for`. Удивительно, что это преобразование для одних типов данных и операций оказывает заметное влияние на общую производительность, а для других – минимальное или вообще никакое. Но в любом случае, это преобразование устраняет неэффективность, которая может стать узким местом при дальнейших попытках оптимизации.

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
<code>combine2</code>	5.4	Абстрактный -01	10,12	10,12	10,17	11,14
<code>combine2</code>	5.4	С выносом <code>vec_length</code> из цикла	7,02	9,03	9,02	11,03

Данная оптимизация является примером оптимизаций общего класса, называемых *выносом кода*. К этому классу также относится выявление вычислений, выполняемых многократно (например, в цикле), результат которых не меняется. Такие вычисления можно перенести в другой раздел кода, выполняемый не столь часто. В данном случае вызов в `vec_length` вынесен из цикла и выполняется непосредственно перед его началом.

**Листинг 5.4.** Увеличение эффективности проверки условия продолжения цикла. Убрав `vec_length` из условия продолжения цикла, мы избавились от необходимости вызывать ее в каждой итерации

```
1 /* Вынос вызова vec_length из цикла */
2 void combine2(vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(v);
6
7     *dest = IDENT;
8     for (i = 0; i < length; i++) {
9         data_t val;
10        get_vec_element(v, i, &val);
11        *dest = *dest OP val;
12    }
13 }
```

Оптимизирующие компиляторы пытаются вынести части кода из цикла. Но, к сожалению, они предельно осторожно применяют преобразования, изменяющие место вызова процедуры или количество этих вызовов. Компиляторы не могут с уверенностью определить, имеет функция побочные эффекты или нет, поэтому попросту предпола-

гают их возможное наличие. Например, если `vec_length` имеет побочный эффект, тогда `combine1` и `combine2` могут показывать разное поведение. В подобных случаях программист должен помочь компилятору, явно переместив часть кода программы.

В качестве крайнего примера неэффективности цикла, которую мы наблюдали в `combine1`, рассмотрим процедуру `lower1` (листинг 5.5). Она написана по образу и подобию процедур, представленных несколькими студентами в ходе разработки проекта сетевой программы. Цель процедуры – преобразовать все прописные буквы в строчные. Процедура перебирает символы в строке и преобразует каждый прописной символ в строчной. Преобразование заключается в сдвиге букв из диапазона от «А» до «Z» в диапазон от «а» до «z».

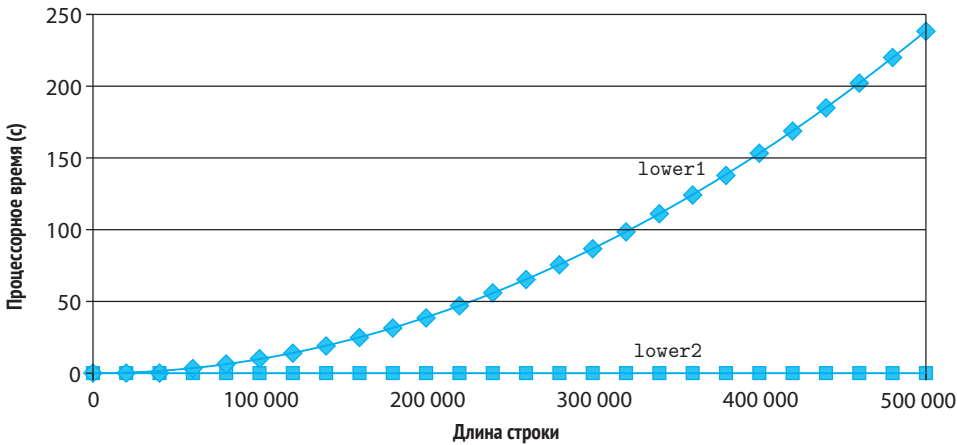
**Листинг 5.5.** Процедуры преобразования символов в нижний регистр. Эти две процедуры имеют совершенно разные характеристики

```

1 /* Преобразует символы в строке в нижний регистр: медленная версия */
2 void lower1(char *s)
3 {
4     long i;
5
6     for (i = 0; i < strlen(s); i++)
7         if (s[i] >= 'A' && s[i] <= 'Z')
8             s[i] -= ('A' - 'a');
9 }
10
11 /* Преобразует символы в строке в нижний регистр: быстрая версия */
12 void lower2(char *s)
13 {
14     long i;
15     long len = strlen(s);
16
17     for (i = 0; i < len; i++)
18         if (s[i] >= 'A' && s[i] <= 'Z')
19             s[i] -= ('A' - 'a');
20 }
21
22 /* Пример реализации библиотечной функции strlen */
23 /* Вычисляет длину строки */
24 size_t strlen(const char *s)
25 {
26     long length = 0;
27     while (*s != '\0') {
28         s++;
29         length++;
30     }
31     return length;
32 }
```

Для проверки условия продолжения цикла в каждой итерации в `lower1` вызывается библиотечная функция `strlen`. Обычно `strlen` реализуется с применением специальных строковых инструкций x86, тем не менее она выполняется практически так же, как ее упрощенная версия в листинге 5.5. Поскольку строки в языке C представляют собой последовательности символов, заканчивающиеся нулем, `strlen` вынуждена перебирать все символы в строке, пока не достигнет пустого символа (нуля). Для строки с длиной  $n$  функции `strlen` требуется время, пропорциональное  $n$ . Поскольку `strlen` вызывается в каждой из  $n$  итераций в `lower1`, общее время выполнения `lower1` пропорционально  $n^2$ .

Этот анализ подтверждается фактическими измерениями производительности функций со строками разной длины, как показано на рис. 5.3. График времени выполнения `lower1` все круче забирает вверх с увеличением длины строки. В табл. 5.1 приводятся результаты измерений времени выполнения для строк семи разных длин (не тех, что показаны на графике), каждая из которых является степенью двойки. Обратите внимание, что каждое удвоение длины строки увеличивает время выполнения `lower1` вчетверо. Это четко указывает на квадратичную зависимость. Для обработки строки длиной 1 048 576 символов функции `lower1` потребовалось 17 минут процессорного времени.



**Рис. 5.3.** Сравнительная производительность процедур преобразования символов в строке в нижний регистр. Время выполнения `lower1` имеет квадратичную зависимость от длины строки из-за неэффективной организации цикла. Время выполнения модифицированной версии `lower2` имеет линейную зависимость

**Таблица 5.1.** Результаты измерения времени выполнения функций `lower1` и `lower2`

Функция	Длина строки						
	16 384	32 768	65 536	131 072	262 144	524 288	1 048 576
lower1	0,26	1,03	4,10	16,41	65,62	262,48	1 049,89
lower2	0,0000	0,0001	0,0001	0,0003	0,0005	0,0010	0,0020

Функция `lower2` в листинге 5.7 идентична функции `lower1`. Единственное ее отличие – вызов `strlen` вынесен за пределы цикла. И как показано в табл. 5.1, это помогло радикально увеличить производительность. На обработку строки длиной 1 048 576 символов этой функции требуется всего 2 миллисекунды, т. е. более чем в 500 000 раз меньше, чем функции `lower1`. Каждое удвоение длины строки вызывает удвоение времени выполнения, что четко указывает на линейную зависимость. Для строк большей длины разница во времени выполнения будет еще больше.

В идеальном мире компилятор распознает, что каждый вызов `strlen` в проверке условия продолжения цикла возвращает один и тот же результат и вынесет его из цикла. Однако это требует очень тщательного и сложного анализа, потому что `strlen` проверяет элементы строки, а их значения изменяются с началом выполнения `lower1`. Компилятору потребуется понять, что, несмотря на изменение символов в строке, ни один из них не меняет ненулевое значение на нулевое, и наоборот. Такого рода анализ выходит далеко за пределы возможностей самых изощренных компиляторов, поэтому программисты должны выполнять подобные преобразования самостоятельно.

Этот пример иллюстрирует общую проблему программирования, когда, казалось бы, тривиальный код имеет скрытую асимптотическую неэффективность. Сложно предположить, чтобы такая обыденная процедура, как преобразование символов строк в нижний регистр, могла ограничить производительность программы. Обычно программы тестируются и анализируются на небольших наборах данных, на которых они показывают вполне приличную производительность. Однако потом, когда программа будет развернута у пользователя, ей, возможно, придется столкнуться со строкой в миллион символов. Так, ни с того ни с сего `lower1` превращается из ничем не примечательной функции в большую проблему производительности. В противовес ей функция `lower2` обрабатывает строки произвольной длины за адекватное время. Известно множество рассказов о крупных программных проектах, сталкивавшихся с проблемами подобного рода, поэтому одна из задач, стоящих перед компетентным программистом, состоит в том, чтобы не допускать появления асимптотической неэффективности.

### Упражнение 5.3 (решение в конце главы)

Взгляните на следующие функции:

```
long min(long x, long y) { return x < y ? x : y; }
long max(long x, long y) { return x < y ? y : x; }
void incr(long *xp, long v) { *xp += v; }
long square(long x) { return x*x; }
```

Они вызываются следующими тремя фрагментами кода:

```
1. for (i = min(x, y); i < max(x, y); incr(&i, 1))
    t += square(i);

2. for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
    t += square(i);

3. long low = min(x, y);
   long high = max(x, y);

   for (i = low; i < high; incr(&i, 1))
       t += square(i);
```

Предположив, что `x` равно 10, а `y` равно 100, заполните следующую таблицу. Укажите, сколько раз будет вызвана каждая из этих четырех функций в фрагментах кода 1–3.

Фрагмент кода	min	max	incr	square
1	_____	_____	_____	_____
2	_____	_____	_____	_____
3	_____	_____	_____	_____

## 5.5. Сокращение вызовов процедур

Как мы уже видели, вызовы процедур сопряжены с большим объемом накладных расходов и являются одним из факторов препятствования оптимизации. В `combine2` (листинг 5.4) мы видели, что `gen_vec_element` вызывается в каждой итерации цикла, чтобы получить следующий элемент вектора. Вызов этой процедуры обходится особенно дорого, потому что она выполняет проверку выхода за границы. Такая проверка может быть полезной при работе с произвольными массивами, однако простой анализ кода

combine2 показывает, что выход за границы вектора никогда не происходит и ссылки на элементы всегда будут действительными.

Теперь предположим, что для нашего абстрактного типа данных имеется еще одна функция – `get_vec_start`, – возвращающая начальный адрес массива данных, как показано в листинге 5.6. Используя ее, можно написать процедуру `combine3` (листинг 5.6), не вызывающую никаких функций во внутреннем цикле. Вместо вызова функции она просто получает каждый следующий элемент вектора, обращаясь непосредственно к массиву. Борцы за чистоту нравов могут заявить, что подобный шаг серьезно вредит модульности программы. В принципе, пользователю абстрактного векторного типа даже не нужно знать, что содержимое вектора хранится в виде массива, а не какой-то другой структуры, например связанного списка. Более прагматичный программист оспори́л бы преимущество данного преобразования, приведя в доказательство следующие результаты измерений.

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine2	5.4	С выносом <code>vec_length</code> из цикла	7,02	9,03	9,02	11,03
combine3	5.6	Прямой доступ к данным	7,17	9,02	9,02	11,03

Как ни странно, но никакого улучшения производительности не произошло. Более того, производительность для случая вычисления целочисленной суммы даже немного ухудшилась. Очевидно, что внутри цикла имеются другие операции, ограничивающие производительность больше, чем вызов `get_vec_element`. Мы вернемся к этой функции позже (в разделе 5.11.2) и посмотрим, почему проверка границ в `combine2` не приводит к снижению производительности. На данный момент мы можем рассматривать это преобразование как один из последовательности шагов, которые в конечном итоге приведут к значительному повышению производительности.

**Листинг 5.6.** Устранение вызовов функции внутри цикла. Полученный код не дает прироста производительности, но допускает дополнительную оптимизацию

```
1 data_t *get_vec_start(vec_ptr v) code/opt/vec.c
2 {
3     return v->data;
4 }

1 /* Прямой доступ к элементам вектора */
2 void combine3(vec_ptr v, data_t *dest) code/opt/vec.c
3 {
4     long i;
5     long length = vec_length(v);
6     data_t *data = get_vec_start(v);
7
8     *dest = IDENT;
9     for (i = 0; i < length; i++) {
10         *dest = *dest OP data[i];
11     }
12 }
```

## 5.6. Устранение избыточных ссылок на память

Функция `combine3` накапливает значение, вычисляемое комбинирующей операцией в ячейке, на которую ссылается указатель `dest`. Этот атрибут можно увидеть, заглянув в ассемблерный код цикла, сгенерированный компилятором. Ниже приводится код x86-64, сгенерированный для типа данных `double` и умножения.

```

Цикл в combine3. data_t = double, OP = *
dest в %rbx, data+i в %rdx, data+length в %rax

1 .L17:                                loop:
2 vmovsd (%rbx), %xmm0                Прочитать произведение по адресу в dest
3 vmulsd (%rdx), %xmm0, %xmm0         Умножить на data[i]
4 vmovsd %xmm0, (%rbx)                Сохранить по адресу в dest
5 addq $8, %rdx                       Увеличить data+i
6 cmpq %rax, %rdx                     Сравнить с data+length
7 jne .L17                            Если !=, то перейти к loop

```

Здесь мы видим, что адрес, соответствующий указателю `dest`, хранится в регистре `%rbx`. Кроме того, компилятор добавил указатель на  $i$ -й элемент данных, обозначенный в комментариях как `data+i`, и хранит его в регистре `%rdx`. В каждой итерации этот указатель увеличивается на 8. Условие завершения цикла проверяется сравнением этого указателя с указателем, хранящимся в регистре `%rax`. Накопленное значение извлекается из памяти и записывается в память в каждой итерации. Эти операции чтения и записи стоят довольно дорого, потому что значение, получаемое из памяти по адресу `dest` в начале каждой итерации, – это то же значение, которое было записано в конце предыдущей итерации.

Мы можем избавиться от этих избыточных операций чтения и записи в память, переписав код, как показано в листинге 5.7.

**Листинг 5.7.** Накопление результата во временной переменной. Накопленное значение хранится в локальной переменной `acc` (сокращение от «accumulator» – аккумулятор), что избавляет от необходимости извлекать его из памяти и записывать обновленное значение обратно в память в каждой итерации

```

1 /* Накопление результата в локальной переменной */
2 void combine4(vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(v);
6     data_t *data = get_vec_start(v);
7     data_t acc = IDENT;
8
9     for (i = 0; i < length; i++) {
10         acc = acc OP data[i];
11     }
12     *dest = acc;
13 }

```

Здесь вводится переменная `acc`, которая используется в цикле для накопления вычисленного значения. Результат сохраняется по адресу в `dest` только после завершения цикла. Как показывает следующий ассемблерный код, компилятор теперь может использовать регистр `%xmm0` для хранения накопленного значения. По сравнению с циклом в `combine3` количество операций с памятью сократилось с двух чтений и одной записи в каждой итерации до всего одного чтения.



```
Цикл в combine4. data_t = double, OP = *
acc в %xmm0, data+i в %rdx, data+length в %rax
1 .L25:                                loop:
2 vmulsd (%rdx), %xmm0, %xmm0        Умножить acc на data[i]
3 addq $8, %rdx                      Увеличить data+i
4 cmpq %rax, %rdx                    Сравнить с data+length
5 jne .L25                            Если !=, то перейти к loop
```

В результате производительность увеличилась существенно, как показывают результаты измерений в следующей таблице:

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine3	5.6	Прямой доступ к данным	7,17	9,02	9,02	11,03
combine4	5.7	Накопление во временной переменной	1,27	3,01	3,01	5,01

Для всех четырех комбинаций достигнуто увеличение производительности от 2,2 до 5,7 раза, при этом для целочисленного сложения потребовалось всего 1,27 цикла синхронизации на элемент.

И снова можно было бы подумать, что компилятор должен автоматически преобразовывать код combine3 в листинге 5.6 и обеспечить накопление результата в регистре, как это произошло с функцией combine4 (листинг 5.7). Однако на самом деле эти две функции могут показывать разное поведение из-за эффекта наложения указателей. Рассмотрим, к примеру, случай с перемножением целочисленных значений и единиц в качестве нейтрального элемента. Пусть  $v = [2, 3, 5]$  – вектор, состоящий из трех элементов, и в программе имеются два следующих вызова функций:

```
combine3(v, get_vec_start(v) + 2);
combine4(v, get_vec_start(v) + 2);
```

То есть указатели на результат и на последний элемент вектора совпадают. В таком случае эти две функции будут выполняться следующим образом:

Функция	Начальное состояние вектора	Перед началом цикла	$i = 0$	$i = 1$	$i = 2$	Конечный результат
combine3	[2, 3, 5]	[2, 3, 1]	[2, 3, 2]	[2, 3, 6]	[2, 3, 36]	[2, 3, 36]
combine4	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 30]

Как видите, combine3 накапливает результат в ячейке памяти, которая одновременно является последним элементом вектора. Поэтому она сначала записывает в эту ячейку 1, затем  $2 \cdot 1 = 2$ , потом  $3 \cdot 2 = 6$ . В последней итерации это значение умножается само на себя, и получается окончательный результат 36. В функции combine4 вектор остается неизменным до самого конца, и только по окончании всех вычислений в последний элемент записывается полученный результат  $1 \cdot 2 \cdot 3 \cdot 5 = 30$ .

Разумеется, этот пример, демонстрирующий различия между combine3 и combine4, во многом вымышленный. Тем не менее можно утверждать, что поведение combine4 ближе соответствует описанной цели функции. К сожалению, оптимизирующий компилятор не может анализировать ни условия, при которых могла бы использоваться функция, ни намерения программиста. Вместо этого, получив исходный код combine3, он обязан в точности сохранить последовательность операций чтения/записи, даже если при этом получится неэффективный код.

**Упражнение 5.4 (решение в конце главы)**

Скомпилировав combine3 компилятором GCC с параметром командной строки -O2, мы получили более эффективный код:

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine3	5.6	Компиляция с параметром -O1	7,17	9,02	9,02	11,03
combine3	5.6	Компиляция с параметром -O2	1,60	3,01	3,01	5,01
combine4	5.7	Накопление во временной переменной	1,27	3,01	3,01	5,01

То есть была достигнута производительность, сравнимая с производительностью combine4, за исключением случая вычисления целочисленной суммы, но даже в этом случае налицо значительное увеличение. Изучив ассемблерный код, сгенерированный компилятором, мы обнаружили интересный вариант реализации цикла:

```

Цикл в combine3. data_t = double, OP = *. Скомпилирована с параметром -O2
dest в %rbx, data+i в %rdx, data+length в %rax
Произведение накапливается в %xmm0
1 .L22:                                loop:
2  vmulsd (%rdx), %xmm0, %xmm0        Умножить произведение на data[i]
3  addq   $8, %rdx                    Увеличить data+i
4  cmpq   %rax, %rdx                  Сравнить с data+length
5  vmovsd %xmm0, (%rbx)               Сохранить произведение по адресу в dest
6  jne    .L22                        Если !=, то перейти к loop

```

Эту версию можно сравнить с версией, полученной при компиляции с параметром -O1:

```

Цикл в combine3. data_t = double, OP = *. Скомпилирована с параметром -O1
dest в %rbx, data+i в %rdx, data+length в %rax
1 .L17:                                loop:
2  vmovsd (%rbx), %xmm0              Прочитать произведение по адресу в dest
3  vmulsd (%rdx), %xmm0, %xmm0        Умножить на data[i]
4  vmovsd %xmm0, (%rbx)              Сохранить по адресу в dest
5  addq   $8, %rdx                    Увеличить data+i
6  cmpq   %rax, %rdx                  Сравнить с data+length
7  jne    .L17                        Если !=, то перейти к loop

```

Как видите, единственное отличие, кроме некоторого переупорядочивания инструкций, состоит в отсутствии в более оптимизированной версии инструкции vmovsd, реализующей чтение из памяти по адресу в dest (строка 2).

1. Чем отличаются роли регистра %xmm0 в этих двух циклах?
2. Будет ли более оптимизированная версия точно соответствовать исходному коду на C функции combine3, в том числе при наличии наложения указателя dest на вектор?
3. Объясните, почему эта оптимизация сохраняет желаемое поведение, или приведите пример, когда результаты будут отличаться от результатов, возвращаемых менее оптимизированным кодом.

Этим заключительным преобразованием была достигнута производительность на уровне всего 1,25–5 циклов синхронизации на вычисление одного элемента. Это значительное улучшение по сравнению с исходными 9–11 циклами перед включением оптимизации. Теперь давайте посмотрим, какие еще факторы ограничивают производительность кода и можно ли улучшить ситуацию еще больше.

## 5.7. Общее описание современных процессоров

До сих пор мы применяли оптимизации, не использующие какие бы то ни было особенности целевой машины. Эти оптимизации просто сокращали количество обращений к процедурам и удаляли некоторые критичные факторы, препятствующие оптимизации. Поскольку нашей задачей является максимальное увеличение производительности, мы должны рассмотреть оптимизации, использующие особенности *микроархитектуры* процессора, то есть особенности выполнения инструкций, присущие конкретным моделям процессоров. Для этого необходимо провести детальный анализ программы, а также сгенерировать код, настроенный на целевой процессор. Также можно применить некоторые базовые оптимизации, которые приведут к общему увеличению производительности на более широком классе процессоров. Подробные результаты, которые приводятся здесь, могут не соответствовать другим машинам, но общие принципы применимы к широкому диапазону типов машин.

Для понимания способов повышения производительности необходимо иметь базовые знания микроархитектуры современных процессоров. Из-за большого числа транзисторов, размещаемых на одном кристалле, в современных микропроцессорах используются комплексные аппаратные средства, способствующие повышению производительности программ. Как результат фактическое функционирование процессоров радикально отличается от того, что пользователи видят при рассмотрении ассемблерных программ. На уровне ассемблерного кода кажется, будто команды выполняются по одной за раз, каждая из них выбирает значения из регистров или памяти, выполняет операцию и сохраняет результаты в памяти или в регистре. Но на самом деле процессор выполняет сразу несколько команд – это явление называется *параллелизмом на уровне инструкций*. В некоторых архитектурах одновременно могут выполняться 100 и более инструкций. Чтобы обеспечить такое параллельное выполнение с сохранением модели последовательной семантики, используются сложные механизмы. Это одно из замечательных достижений современных микропроцессоров: они используют сложные и экзотические микроархитектуры, в которых несколько инструкций могут выполняться параллельно, при этом создавая видимость простого последовательного выполнения инструкций.

Подробный анализ устройства современного микропроцессора выходит далеко за рамки этой книги, однако для общего представления о принципах их работы достаточно понимать, как достигается параллелизм на уровне команд. Далее мы обсудим две нижние границы, от которых зависит максимальная производительность программ. *Граница задержки* начинает ограничивать производительность, когда имеется серия операций, которые должны выполняться в строгой последовательности, потому что результат одной операции должен быть получен до начала следующей. Эта граница может ограничить производительность, когда в программе имеются зависимости по данным, мешающие процессору использовать параллелизм на уровне инструкций. *Граница пропускной способности* характеризует чистую вычислительную мощность функциональных блоков процессора. Эта граница является окончательным и непреодолимым пределом производительности программ.

### 5.7.1. Общие принципы функционирования

На рис. 5.4 показано очень упрощенное представление современного микропроцессора. Архитектура нашего гипотетического процессора частично основана на структуре последних процессоров Intel. Эти процессоры называются в отрасли *суперскалярными* за их способность выполнять несколько операций в каждом цикле синхронизации и *не по порядку*, то есть порядок выполнения инструкций не обязательно будет соответствовать порядку их следования в машинной программе. Архитектура состоит из двух

основных частей: *блока управления инструкциями* (Instruction Control Unit, ICU), отвечающего за чтение последовательности инструкций из памяти и создание на их основе последовательностей элементарных операций с данными программы, и *исполнительного блока* (Execution Unit, EU), который выполняет эти операции. По сравнению с простым конвейером, который мы рассматривали в главе 4, суперскалярные процессоры имеют более сложное аппаратное устройство, зато они лучше справляются с задачей распараллеливания.

Блок управления инструкциями ICU выбирает инструкции из *кеша инструкций* – особой высокоскоростной памяти, где хранятся последние использовавшиеся инструкции. Вообще говоря, ICU выбирает инструкции с некоторым опережением, так что у него достаточно времени для их декодирования и отправки операций в исполнительный блок выполнения EU. Однако есть одна проблема: когда программа выполняет ветвление<sup>1</sup>, возникает два возможных направления дальнейшего движения. Ветвление может *выполниться* с передачей управления по адресу, указанному в инструкции ветвления, или не выполниться, и тогда выполнение продолжится со следующей инструкции. Современные процессоры используют методику *предсказания ветвления*, с помощью которой пытаются угадать, будет ли выполнено ветвление, а также предсказать целевой адрес ветвления. Используя прием, получивший название *спекулятивное выполнение*, процессор приступает к выборке и декодированию инструкций, начиная с предсказанного адреса, и даже начинает выполнение этих операций до того, как будет подтверждена верность или ошибочность предсказания. Если позже выяснится, что ветвление спрогнозировано неверно, то процессор возвращается в состояние, предшествовавшее точке ветвления, и начинает выборку и выполнение инструкций в другом направлении. Блок, подписанный на рис. 5.4 как «Управление выборкой», включает механизм предсказания ветвления, чтобы определить, какие инструкции следует выбрать.

Логика *декодирования инструкций* получает фактические инструкции, составляющие программу, и преобразует их в последовательность элементарных *операций* (иногда их также называют *микрооперациями*). Каждая из этих операций решает некоторую простую задачу, например складывает два числа, читает данные из памяти или записывает их в память. В процессорах со сложными инструкциями, таких как x86, инструкция может быть преобразована в переменное число операций. Разные процессоры имеют свои тонкости декодирования инструкций в последовательности операций, и информация о них считается коммерческой тайной. К счастью, мы можем оптимизировать свои программы, и не зная всех тонкостей конкретной аппаратной реализации.

В типичной реализации x86 инструкции, оперирующие только регистрами, такие как

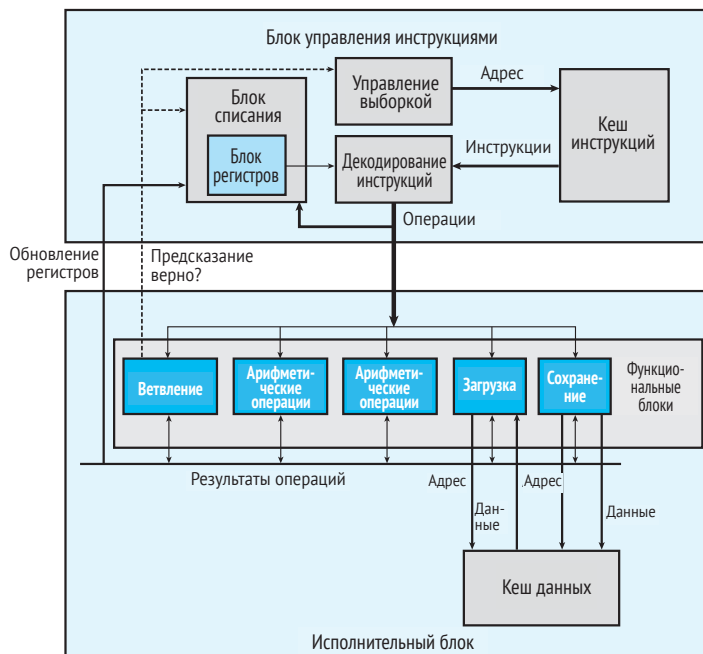
```
addq %rax,%rdx
```

преобразуются в одну операцию. С другой стороны, инструкции, выполняющие одно или несколько обращений к памяти, такие как

```
addq %rax,8(%rdx)
```

порождают несколько операций: отдельно для обращений к памяти и отдельно для выполнения арифметических действий. В частности, инструкция выше может быть декодирована в три операции: одна *загружает* значение из памяти, вторая складывает загруженное значение со значением в регистре %rax, третья *сохраняет* результат в памяти. Такое декодирование обеспечивает так называемое разделение труда между специализированными аппаратными блоками, благодаря чему эти блоки могут параллельно выполнять различные части нескольких инструкций.

<sup>1</sup> В данном случае термином «ветвление» мы обозначаем инструкции условного перехода. Другие инструкции, которые могут передавать управление, такие как возврат из процедуры и косвенные переходы, создают аналогичные проблемы для процессора.



**Рис. 5.4.** Блок-схема суперскалярного процессора. Блок управления инструкциями отвечает за чтение инструкций из памяти и создание последовательностей элементарных операций. Исполнительный блок выполняет эти операции и определяет, правильно ли были спрогнозированы ветвления

Исполнительный блок EU извлекает операции из блока управления выборкой инструкций. В каждом цикле синхронизации может извлекаться сразу несколько операций. Эти операции отправляются в *функциональные блоки*, которые фактически их выполняют. Разные функциональные блоки специализируются на обработке определенных типов операций.

Чтение из памяти и запись в память выполняются блоками загрузки и сохранения. Блок загрузки обрабатывает операции чтения данных из памяти в процессор. В нем имеется сумматор для вычисления адресов. Аналогично блок сохранения обрабатывает операции, записывающие данные из процессора в память. В нем тоже имеется сумматор для вычисления адресов. Как показано на рис. 5.4, блоки загрузки и сохранения обращаются к памяти через *кеш данных* – высокоскоростную память, содержащую последние использовавшиеся данные.

При спекулятивном выполнении операции производят вычисления, но их результаты не сохраняются в программных регистрах или в памяти данных, пока процессор не будет уверен, что эти инструкции действительно нужно было выполнить. Операции ветвления передаются в исполнительный блок не для определения пути, по которому должно пойти выполнение, а для определения верности прогноза. Если прогноз был неверным, исполнительный блок отбросит результаты, вычисленные после точки ветвления, а также сообщит блоку ветвления, что прогноз был неверным, и укажет правильный адрес инструкции, начиная с которой следует продолжить выполнение. В этом случае блок ветвления начнет выборку с нового адреса. Как было показано в разделе 3.6.6, такое неверное предсказание приводит к значительным потерям в производительности. Пройдет какое-то время, пока новые инструкции будут извлечены, декодированы и переданы функциональным блокам.

Как показано на рис. 5.4, разные функциональные блоки специализируются на выполнении определенных операций. Блоки, подписанные как «Арифметические операции», специализируются на выполнении различных комбинаций операций с целыми числами и с числами с плавающей точкой. Поскольку количество транзисторов на одном кристалле со временем увеличивалось, последовательные модели микропроцессоров увеличивали общее количество функциональных блоков, расширяли комбинации операций, которые может выполнять каждый блок, и повышали производительность каждого из этих блоков. Арифметические блоки спроектированы так, чтобы выполнять множество разных операций, потому что операции, выполняемые разными программами, сильно различаются. Например, одни программы могут включать множество целочисленных операций, а другие – множество операций с плавающей точкой. Если бы один функциональный блок специализировался только на целочисленных операциях, а другой – только на операциях с плавающей точкой, то ни одна из программ не получила бы всех преимуществ наличия нескольких функциональных блоков.

Например, в нашей эталонной машине с процессором Intel Core i7 Haswell имеется восемь функциональных блоков, пронумерованных от 0 до 7. Вот неполный список, перечисляющий некоторые из их возможностей:

0. Целочисленная арифметика, умножение с плавающей точкой, деление целочисленное и с плавающей точкой, ветвление.
1. Целочисленная арифметика, сложение с плавающей точкой, умножение целочисленное и с плавающей точкой.
2. Загрузка, вычисление адреса.
3. Загрузка, вычисление адреса.
4. Сохранение.
5. Целочисленная арифметика.
6. Целочисленная арифметика, ветвление.
7. Сохранение, вычисление адреса.

В приведенном выше списке под «целочисленной арифметикой» подразумеваются базовые операции, такие как сложение, побитовые операции и сдвиг. Умножение и деление требуют более специализированных ресурсов. Для выполнения операции сохранения используются два функциональных блока: один для вычисления адреса, а другой – для выполнения фактической операции сохранения данных. Более подробно сохранение и загрузку мы обсудим в разделе 5.12.

Такая комбинация функциональных блоков имеет возможность одновременно выполнять несколько операций одного типа. В ней имеется четыре блока, способных выполнять целочисленные операции, два – операции загрузки и два – операции умножения с плавающей точкой. Далее мы покажем, как эти ресурсы влияют на максимально достижимую нашими программами производительность.

*Блок списания (или завершения)* внутри блока управления инструкциями наблюдает за обработкой инструкций и гарантирует соблюдение семантики последовательного выполнения машинной программы. На рис. 5.4 блок *регистров* (содержащий целочисленные регистры, регистры с плавающей точкой, а также регистры SSE и AVX) показан внутри блока завершения, потому что именно он управляет обновлением регистров. После декодирования информация об инструкции помещается в очередь с дисциплиной обработки FIFO (First-In, First-Out – «первым пришел, первым ушел»). Эта информация остается в очереди до одного из двух исходов. В первом случае, когда завершается выполнение операций, предполагаемых инструкцией, и любые ветвления, ведущие к этой инструкции, подтверждаются как спрогнозированные верно, инструкцию можно *списать* (завершить) и сохранить в регистрах любые произведенные ею изменения. Во втором случае, если какое-то ветвление, приведшее к этой инструкции, окажется спрог-



нозировано неверно, то результаты, вычисленные инструкцией, будут отброшены. Благодаря этому блоку неверные прогнозы не могут изменить состояние программы.

### Обработка инструкций не по порядку

Впервые обработка инструкций не по порядку была реализована в процессоре Control Data Corporation 6600 в 1964 году. Инструкции обрабатывались десятью разными функциональными блоками, каждый из которых мог действовать независимо. В то время эта машина с тактовой частотой в 10 МГц считалась лучшей для научных вычислений.

Компания IBM впервые реализовала обработку инструкций не по порядку в процессоре IBM 360/91 в 1966 году, но только для инструкций, оперирующих числами с плавающей точкой. На протяжении 25 лет обработка не по порядку считалась экзотической технологией, реализованной только на машинах, от которых требовалась максимально возможная производительность, пока в 1990 году компания IBM не вывела на рынок линейку рабочих станций RS/6000. Их архитектура легла в основу линейки IBM/Motorola PowerPC, появившейся в 1993 году в виде модели 601, ставшей первым микропроцессором, сконструированным в виде одной микросхемы, который поддерживал обработку инструкций не по порядку. Компания Intel реализовала обработку не по порядку в своей модели Pentium Pro 1995 года, имевшей базовую микроархитектуру, аналогичную нашей эталонной машине.

Как отмечалось выше, любое обновление программных регистров производится только при списании (завершении) инструкций, а это происходит, только когда процессор убеждается, что любые ветвления, ведущие к этой инструкции, были спрогнозированы верно. Чтобы ускорить обмен результатами между инструкциями, функциональные блоки передают между собой большую часть этой информации, обозначенную на рис. 5.4 как «Результаты операций». Стрелки на схеме показывают, что исполнительные устройства могут отправлять результаты друг другу напрямую. Это более сложная форма пересылки данных, чем та, которую мы реализовали в нашем простом процессоре в разделе 4.5.5.

Наиболее широко используемый механизм управления передачей операндов между исполнительными блоками называется *переименованием регистров*. После декодирования инструкции, обновляющей регистр  $r$ , генерируется *метка*  $t$ , дающая результату операции уникальный идентификатор. В таблицу связей между программными регистрами и метками для данной операции добавляется элемент  $(r, t)$ . После декодирования последующей инструкции, которая использует регистр  $r$  в качестве операнда-источника, операция, отправленная в исполнительный блок, будет содержать  $t$  в качестве операнда. Когда некоторый исполнительный блок завершит первую операцию, он сгенерирует результат  $(v, t)$ , сообщая, что операция с меткой  $t$  выдала значение  $v$ . Затем любая операция, ожидающая  $t$ , будет использовать  $v$ . С помощью такого механизма значения могут напрямую передаваться между операциями без записи в блок регистров и чтения из него. Таблица переименования содержит записи только для регистров, для которых имеются незавершенные операции записи. Когда декодированной инструкции требуется регистр  $r$  и в таблице переименования отсутствует метка, ассоциированная с этим регистром, операнд извлекается непосредственно из блока регистров. Благодаря механизму переименования регистров вся последовательность операций может выполняться спекулятивно, даже притом что регистры обновляются только после того, как процессор будет уверен в верности ветвления.

### 5.7.2. Производительность функционального блока

В табл. 5.2 показаны характеристики некоторых арифметических операций нашей эталонной процессора Core i7 Haswell, полученные из технического справочника Intel [49] и измеренные. Эти характеристики также типичны для других процессоров.

Каждая операция характеризуется *задержкой*, под которой подразумевается количество циклов синхронизации, необходимых для ее выполнения, *временем выпуска* – минимальным количеством циклов между двумя независимыми операциями одного типа, и *емкостью* – количеством операций, которые могут выполняться одновременно.

**Таблица 5.2.** Задержка, время выпуска и характеристики емкости эталонной машины. Под задержкой подразумевается общее количество циклов синхронизации, необходимых для выполнения фактических операций, под временем выпуска – минимальное количество циклов между двумя независимыми операциями, а под емкостью – количество операций, которые могут выполняться одновременно. Время, затрачиваемое на операцию деления, зависит от значений операндов

Операция	Целочисленные			С плавающей точкой		
	Задержка	Выпуск	Емкость	Задержка	Выпуск	Емкость
Сложение	1	1	4	3	1	1
Умножение	3	1	1	5	1	2
Деление	3–30	3–30	1	3–15	3–15	1

Согласно этой таблице, операции с плавающей точкой имеют более высокие задержки, по сравнению с целочисленными операциями. Также, согласно этой таблице, все операции сложения и умножения имеют время выпуска 1, т. е. с началом каждого следующего цикла синхронизации процессор может начать выполнять новую операцию. Такое короткое время выпуска достигается за счет *конвейерной обработки*. Конвейерный функциональный блок реализован в виде серии *этапов*, каждый из которых выполняет часть операции. Например, типичный сумматор с плавающей точкой имеет три этапа (и, следовательно, задержку в три цикла синхронизации): один обрабатывает значение показателя степени, один складывает дробные части и один округляет результат. Арифметические операции могут проходить через этапы в плотной последовательности – каждая следующая операция может начинаться, не ожидая завершения предыдущей. Однако такое возможно только при выполнении последовательностей логически независимых операций. Функциональные блоки со временем выпуска в 1 цикл считаются *полностью конвейерными*: они могут начать выполнять новую операцию в каждом новом цикле. Емкость больше 1 обусловлена наличием нескольких функциональных блоков, как было описано выше.

В таблице видно, что делитель (блок деления целых чисел и чисел с плавающей точкой, а также вычисления квадратного корня с плавающей точкой) не является конвейерным – его время выпуска равно его задержке. Это означает, что делитель должен полностью завершить одну операцию деления, прежде чем начать выполнять следующую. Кроме того, задержки и время выпуска для деления даны в виде диапазонов, потому что некоторые комбинации делимого и делителя требуют больше шагов, чем другие. Большие задержка и время выпуска для деления делают эту операцию довольно дорогостоящей.

Более распространенным способом выражения времени выпуска является максимальная *пропускная способность* блока, определяемая как величина, обратная времени выпуска. Полностью конвейерный функциональный блок имеет максимальную пропускную способность 1 операция за цикл, в то время как блоки с большим временем выпуска имеют более низкую максимальную пропускную способность. Наличие нескольких функциональных блоков может увеличить пропускную способность. Для операций с емкостью  $S$  и временем выпуска  $I$  процессор потенциально может достичь пропускной способности операций  $S/I$  за цикл. Например, наша эталонная машина способна выполнять умножение с плавающей точкой со скоростью 2 операции за цикл. Далее вы



увидите, как эту возможность можно использовать для повышения производительности программы.

Разработчики цепей могут создавать функциональные блоки с широким диапазоном рабочих характеристик. Чтобы получить блок с короткой задержкой или с конвейерной обработкой, требуется больше аппаратных модулей, что особенно верно для сложных функций, таких как умножение и операции с плавающей точкой. Поскольку место на кристалле микропроцессора ограничено и на нем можно разместить не более определенного числа таких блоков, разработчики процессоров должны тщательно подбирать количество функциональных блоков и их индивидуальную производительность для достижения оптимальной общей производительности. Они оценивают производительность со множеством различных тестовых программ и стараются выделять больше ресурсов на наиболее важные операции. Как показано в табл. 5.2, при разработке процессора Core i7 Haswell наиболее важными были сочтены умножение (целочисленное и с плавающей точкой) и сложение с плавающей точкой, даже несмотря на то, что для достижения низких задержек и высокой степени конвейеризации потребовалось значительное количество аппаратных модулей. С другой стороны, деление используется относительно нечасто, и его сложно реализовать с короткой задержкой или с применением полной конвейерной обработки.

Задержки, время выпуска и емкости арифметических операций могут повлиять на производительность наших комбинирующих функций. Это влияние можно выразить с помощью двух фундаментальных ограничений CPE:

Граница	Целочисленные данные		Данные с плавающей точкой	
	+	*	+	*
Задержка	1,00	3,00	3,00	5,00
Пропускная способность	0.50	1,00	1,00	0,50

*Граница задержки* дает минимальное значение CPE для любой функции, которая должна объединять результаты операций в строгой последовательности. *Граница пропускной способности* дает минимальную границу CPE, зависящую от максимальной скорости, с которой функциональные блоки могут выдавать результаты. Например, при наличии только одного блока целочисленного умножения со временем выпуска 1 операция за цикл процессор не сможет производить более 1 умножения за цикл. С другой стороны, при наличии четырех функциональных блоков для целочисленного сложения процессор потенциально может поддерживать скорость до 4 операций за цикл. К сожалению, необходимость чтения данных из памяти дополнительно ограничивает пропускную способность. Наличие всего двух блоков загрузки не позволит процессору извлекать из памяти более 2 значений данных за цикл, из-за чего пропускная способность снизится до 0,50. Далее в этой главе вы увидите, как сказываются границы задержки и пропускной способности в различных версиях функций комбинирования.

### 5.7.3. Абстрактная модель работы процессора

В качестве инструмента анализа производительности машинной программы, выполняемой на современном процессоре, мы будем использовать представление программ в виде *потока данных*, графического представления, отражающего зависимости по данным между различными операциями, ограничивающими порядок их выполнения. Эти ограничения создают *критические пути* в графе, устанавливая нижнюю границу количества циклов синхронизации, необходимых для выполнения набора машинных инструкций.

Прежде чем перейти к техническим деталям, рассмотрим результаты измерения CPE, полученные для функции `combine4` – самой быстрой на данный момент версии:

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine4	5.7	Накопление во временной переменной	1,27	3,01	3,01	5,01
Граница задержки			1,00	3,00	3,00	5,00
Граница пропускной способности			0,50	1,00	1,00	0,50

Как видите, результаты измерений соответствуют границе задержки для процессора, за исключением случая целочисленного сложения. И это не случайное совпадение, а свидетельство того, что производительность этих функций обусловлена задержкой операции сложения или умножения. Для вычисления произведения или суммы  $n$  элементов требуется около  $L \cdot n + K$  циклов синхронизации, где  $L$  – задержка операции комбинирования, а  $K$  – накладные расходы на вызов функции, а также на запуск и завершение цикла. В результате значение CPE получается равным границе задержки  $L$ .

От машинного кода к графу потоков данных

Наше представление программ в форме потоков данных не является формальным. Мы используем его лишь для того, чтобы показать, как зависимости по данным в программе определяют ее производительность. Рассмотрим нотацию потока данных на примере функции combine4 (листинг 5.7). Чтобы не расплываться, сосредоточимся на вычислениях, выполняемых циклом, потому что именно он оказывает основное влияние на производительность при обработке больших векторов. Далее мы будем рассматривать случай перемножения данных типа double. Другие комбинации типов данных и операций дают аналогичный код. Скомпилированный код для этого цикла состоит из четырех инструкций, при этом регистр %rdx хранит указатель на  $i$ -й элемент массива, регистр %rax – указатель на конец массива, а регистр %xmm0 хранит накопленное значение acc.

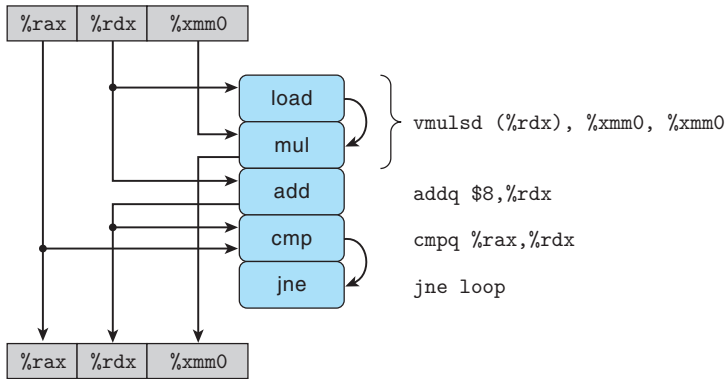
```
Цикл в combine4. data_t = double, OP = *
acc в %xmm0, data+i в %rdx, data+length в %rax
1 .L25:                                loop:
2 vmulsd (%rdx), %xmm0, %xmm0        Умножить acc на data[i]
3 addq    $8, %rdx                    Увеличить data+i
4 cmpq    %rax, %rdx                  Сравнить с data+length
5 jne     .L25                        Если !=, перейти к loop
```

Как показано на рис. 5.5, в нашем гипотетическом процессоре четыре инструкции преобразуются декодером в серию из пяти операций, при этом начальная инструкция умножения преобразуется в операции загрузки исходного операнда из памяти и умножения.

В представлении программы в виде потока данных прямоугольники и линии слева на рис. 5.5 показывают, как различные операции используют и обновляют регистры, при этом прямоугольники сверху представляют значения регистров в начале итерации, а внизу – в конце. Например, регистр %rax используется только как исходное значение операцией **cmp**, поэтому в конце итерации он имеет то же значение, что и в начале. С другой стороны, регистр %rdx используется и обновляется внутри итерации. Его начальное значение используется операциями **load** и **add**, а новое значение, созданное операцией **add**, используется затем операцией **cmp**. Регистр %xmm0 тоже обновляется внутри итерации операцией **mul**, которая сначала использует начальное значение в качестве исходного.

Некоторые операции на рис. 5.5 генерируют значения, не попадающие в регистры. Они изображены как дугообразные стрелки между операциями справа. Операция **load**

читает значение из памяти и передает его непосредственно в операцию **mul**. Так как эти две операции появляются в результате декодирования одной инструкции **vmulsd**, то промежуточное значение, передаваемое между ними, не попадает в регистры. Операция **cmp** обновляет флаги, которые затем проверяются операцией **jne**.



**Рис. 5.5.** Графическое представление цикла в `combine4`. Инструкции динамически преобразуются в одну или две операции, каждая из которых получает значения от других операций или из регистров и вычисляет значения для других операций и регистров. Цель последней инструкции показана как метка `loop`. Она выполняет переход к первой инструкции в листинге

Для сегмента кода, образующего тело цикла, мы можем разделить используемые регистры на четыре категории:

- *только для чтения* – эти регистры используются как исходные значения: либо как данные, либо для вычисления адресов памяти, но они не изменяются внутри итераций. Единственный регистр из этой категории в нашем примере – это `%rax`;
- *только для записи* – эти регистры используются как операнды-приемники в операциях перемещения данных. В нашем примере таких регистров нет;
- *локальные* – эти регистры обновляются и используются в итерациях, но их значения не зависят друг от друга между итерациями. В нашем примере таким регистром является регистр флагов: он обновляется операцией **cmp** и используется операцией **jne**, но эта зависимость сохраняется только в пределах одной итерации;
- *регистры цикла* – эти регистры используются и в роли операндов-источников, и в роли операндов-приемников, при этом значение, полученное в одной итерации, используется в другой итерации. В нашем примере к этой категории относятся регистры `%rdx` и `%xmm0`, в которых хранятся значения `data+i` и `acc`.

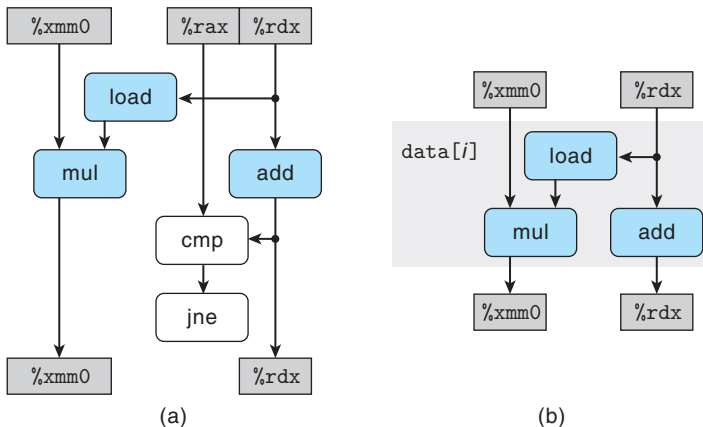
Как мы увидим далее, цепочки операций между регистрами цикла определяют зависимости по данным, ограничивающие производительность.

На рис. 5.6 показано уточненное графическое представление графа потока данных. На этой схеме присутствуют только операции и зависимости по данным, которые влияют на время выполнения программы. На рис. 5.6 (а) мы изменили расположение операций, чтобы более четко показать поток данных от регистров-источников вверху (где изображены регистры только для чтения и регистры цикла) до регистров-приемников внизу (регистров только для записи и регистров цикла).

Кроме того, операции, не являющиеся частью некоторой цепочки зависимостей между регистрами цикла, на рис. 5.6 (а) окрашены в белый цвет. В нашем примере операции

сравнения (**cmp**) и перехода (**jne**) не влияют напрямую на поток данных. Предполагается, что блок управления инструкциями все время предсказывает выполнение перехода и, следовательно, продолжение итераций. Цель операций сравнения и перехода – проверка условия перехода и уведомление блока управления инструкциями, если условие не выполняется. Предполагается, что проверка выполняется достаточно быстро и не замедляет работу процессора.

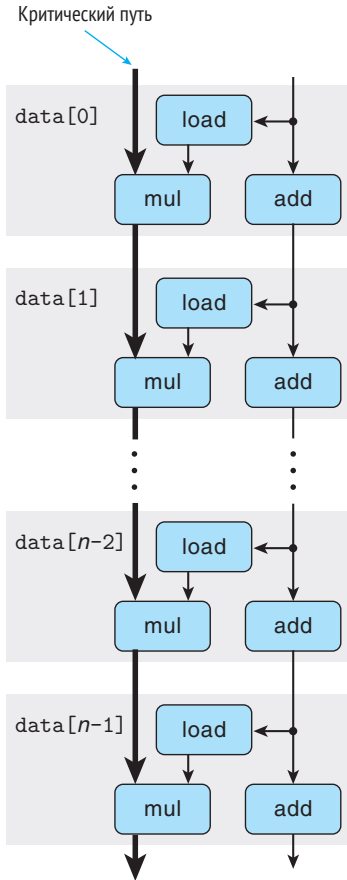
На рис. 5.6 (b) мы убрали операции, окрашенные в белый цвет на рис. 5.6 (a), и оставили только регистры цикла. У нас остался абстрактный шаблон, показывающий зависимости по данным, которые образуются между регистрами цикла. На этой диаграмме видно, что существуют две зависимости по данным между итерациями. Одна зависимость имеет место между последовательными значениями локальной переменной *ass*, хранящейся в регистре `%xmm0`. Цикл вычисляет новое значение *ass*, умножая предыдущее значение на элемент вектора, полученный операцией **load**. Другая зависимость имеет место между последовательными значениями указателя на *i*-й элемент вектора. В каждой итерации старое значение используется как адрес для операции **load** и затем увеличивается с помощью операции **add**, чтобы получить новое значение.



**Рис. 5.6.** Абстрагирование операций в цикле в функции `combine4` в виде графа потока данных. Мы переупорядочили операции, чтобы более четко показать зависимости по данным (a), а затем показываем только те операции, которые используют значения, полученные в одной итерации, для вычисления новых значений в следующей итерации (b)

На рис. 5.7 показано представление потока данных через несколько итераций цикла в функции `combine4`. Этот граф получен простым повторением шаблона, показанного на рис. 5.6 (b). Как видите, в программе имеются две цепочки зависимостей по данным, соответствующие обновлению программных значений *ass* и `data+i` операциями **mul** и **add** соответственно. Так как умножение с плавающей точкой имеет задержку в 5 циклов синхронизации, а целочисленное сложение – в 1 цикл, цепочка слева образует *критический путь*, требующий  $5n$  циклов для выполнения. Цепочка справа требует всего  $n$  циклов для выполнения, поэтому она не ограничивает производительность программы.

Диаграмма на рис. 5.7 демонстрирует, почему в `combine4` мы достигли величины CPE в 5 циклов синхронизации, равной границе задержки операции умножения с плавающей точкой. Умножение с плавающей точкой становится ограничивающим ресурсом для нашей функции. Другие необходимые операции – манипулирование указателем `data+i` и чтение данных из памяти – выполняются параллельно с умножением. По мере вычисления очередного значения *ass* оно снова используется для вычисления следующего значения, но это произойдет не раньше, чем через 5 циклов синхронизации.



**Рис. 5.7.** Представление вычислений в  $n$  итерациях в цикле в функции `combine4` в форме графа потока данных. Последовательность операций умножения образует критический путь, ограничивающий производительность программы

Поток данных для других комбинаций типов и операций идентичен показанному на рис. 5.7, только цепочку зависимостей по данным слева образует другая операция. Для всех случаев, когда операция имеет задержку  $L$  больше 1, измеренное значение CPE будет равно  $L$ , что явно указывает на цепочку, образующую критический путь, который ограничивает производительность.

### Другие факторы, влияющие на производительность

Как было показано выше, при оценке реализации цикла с целочисленным сложением мы получили значение CPE, равное 1,27, вместо 1,00, которое можно было бы спрогнозировать, опираясь на цепочки зависимостей вдоль левой или правой стороны графа на рис. 5.7. Это – иллюстрация принципа, согласно которому критические пути в потоках данных задают только *нижнюю* границу количества циклов синхронизации, которые потребуются программе. Но есть и другие факторы, способные ограничивать производительность, включая общее количество имеющихся функциональных блоков и количество значений данных, которые могут передаваться между функциональными блоками на любом заданном шаге. В случае со сложением целых чисел операция объединения выполняется достаточно быстро, из-за чего остальные операции не успевают передавать данные для нее. Чтобы точно определить, почему программе требуется 1,27 цикла синхронизации на обработку одного элемента вектора, нужно иметь более точные сведения об аппаратном устройстве процессора, чем нам доступно.

Подведем итог нашему анализу производительности combine4: абстрактное представление выполнения программы в виде потока данных показало, что combine4 имеет критический путь длиной  $L \cdot n$ , обусловленный последовательным обновлением программного значения асс, и этот путь ограничивает величину CPE, по меньшей мере, значением  $L$ . Это правило действовало для всех случаев, кроме целочисленного сложения, для которого измеренная величина CPE получилась равной 1,27 вместо 1,00 – расчетного значения длины критического пути.

Может показаться, что граница задержки образует фундаментальный предел скорости выполнения операции комбинирования. Однако это не так. Наша следующая задача – реструктурировать операции так, чтобы повысить степень параллелизма на уровне инструкций. Мы преобразуем программу так, чтобы единственным ограничением стала граница пропускной способности и величина CPE оказалась ниже или близкой к 1,00.

#### Упражнение 5.5 (решение в конце главы)

Представьте, что мы решили написать функцию вычисления многочлена степени  $n$ , который определен набором коэффициентов  $a_0, a_1, a_2, \dots, a_n$ . Для заданного значения  $x$  мы должны вычислить полином

$$a_0 + a_1x + a_2x^2 + \dots + a_nx^n. \quad (5.2)$$

Эти вычисления можно реализовать в виде функции, как показано ниже, принимающей в аргументах массив коэффициентов  $a$ , значение  $x$  и степень полинома  $degree$  (значение  $n$  в уравнении 5.2). Эта функция вычисляет последовательные члены уравнения и последовательные степени  $x$  в одном цикле:

```

1 double poly(double a[], double x, long degree)
2 {
3     long i;
4     double result = a[0];
5     double xpow = x; /* В начале цикла равно x^i */
6     for (i = 1; i <= degree; i++) {
7         result += a[i] * xpow;
8         xpow = x * xpow;
9     }
10    return result;
11 }
```

1. Сколько сложений и умножений выполняет этот код для степени  $n$ ?
2. На нашей эталонной машине с задержками арифметических операций, показанными в табл. 5.2, мы измерили и получили для этой функции величину CPE, равную 5,00. Объясните, как получилась эта величина CPE, проанализировав зависимости по данным между итерациями, которые образуют операции в строках 7–8.

#### Упражнение 5.6 (решение в конце главы)

Продолжим изучение способов решения задачи вычисления многочленов, предложенной в упражнении 5.5. Мы можем уменьшить количество умножений при вычислении многочлена, применив *метод Горнера*, названный в честь британского математика Уильяма Дж. Горнера (William G. Horner; 1786–1837). Идея состоит в том, чтобы вынести за скобки степени  $x$  и получить следующее выражение:

$$a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-1} + xa_n) \dots)). \quad (5.3)$$

Используя метод Горнера, мы реализовали вычисление полинома, как показано ниже:

```
1 /* Реализация с применением метода Горнера */
2 double polyh(double a[], double x, long degree)
3 {
4     long i;
5     double result = a[degree];
6     for (i = degree-1; i >= 0; i--)
7         result = a[i] + x*result;
8     return result;
9 }
```

1. Сколько сложений и умножений выполняет этот код для степени  $n$ ?
2. На нашей эталонной машине с задержками арифметических операций, показанными в табл. 5.2, мы измерили и получили для этой функции величину CPE, равную 8,00. Объясните, как получилась эта величина CPE, проанализировав зависимости по данным между итерациями, которые образует операция в строке 7.
3. Объясните, почему функция в упражнении 5.5 работает быстрее, несмотря на то что выполняет больше операций.

## 5.8. Развертывание циклов

Развертывание цикла – это такое преобразование программы, которое уменьшает количество итераций за счет увеличения количества вычислений в каждой итерации. Мы уже видели пример развертывания цикла в функции `psum2` (листинг 5.1), где в каждой итерации к сумме прибавлялись два элемента, за счет чего общее количество итераций уменьшилось вдвое. Развертывание цикла может улучшить производительность по двум причинам. Во-первых, уменьшается количество операций, не влияющих непосредственно на результат программы, таких как вычисление индекса и условное ветвление. Во-вторых, открываются возможности для дальнейшего преобразования кода и уменьшения количества операций в критических путях. В этом разделе мы рассмотрим простое развертывание цикла без каких-либо дальнейших преобразований.

В листинге 5.8 показана версия нашей комбинирующей функции, использующей прием, который мы будем называть «развертыванием цикла  $2 \times 1$ ». Первый цикл выполняет обход массива, извлекая по два элемента за раз. То есть индекс цикла  $i$  увеличивается на 2 в каждой итерации, и в каждой итерации операция объединения применяется к элементам  $i$  и  $i + 1$  массива.

**Листинг 5.8.** Развертывание цикла  $2 \times 1$ . Это преобразование может уменьшить накладные расходы на итерации

```
1 /* Развертывание цикла  $2 \times 1$  */
2 void combine5(vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(v);
6     long limit = length-1;
7     data_t *data = get_vec_start(v);
8     data_t acc = IDENT;
9
10    /* Комбинировать сразу по 2 элемента */
11    for (i = 0; i < limit; i+=2) {
12        acc = (acc OP data[i]) OP data[i+1];
13    }
```

```
14
15  /* Обработать оставшиеся элементы */
16  for (; i < length; i++) {
17      acc = acc OP data[i];
18  }
19  *dest = acc;
20 }
```

В общем случае длина вектора необязательно будет кратна 2. И нам нужно, чтобы наш код правильно обрабатывал векторы произвольной длины. Учтем это требование следующим образом. Прежде всего гарантируем, что первый цикл не выйдет за границы массива. Для вектора с длиной  $n$  мы установим предельное число итераций цикла равным  $n - 1$ . Это даст нам уверенность, что цикл будет продолжать выполняться, только если соблюдается условие  $i < n - 1$ , и, следовательно, максимальный индекс массива  $i + 1$  удовлетворяет условию  $i + 1 < (n - 1) + 1 = n$ .

Мы можем обобщить эту идею для развертывания циклов с любым коэффициентом  $k$ , чтобы получить в результате *развернутый цикл*  $k \times 1$ . Для этого нужно установить верхний предел равным  $n - k + 1$  и внутри цикла применить комбинирующую операцию к элементам с  $i$  по  $i + k - 1$ . Индекс цикла  $i$  в этом случае должен увеличиваться на  $k$  в каждой итерации. Тогда максимальный индекс массива  $i + k - 1$  будет меньше  $n$ . Второй дополнительный цикл выполняет обход последних нескольких элементов вектора по одному. Тело этого цикла будет выполнено от 0 до  $k - 1$  раз. Для случая  $k = 2$  можно использовать простой условный оператор, чтобы добавить последнюю итерацию, как мы это сделали в функции `rsum2` (листинг 5.1). Для случая  $k > 2$  лучше добавить второй цикл, поэтому для простоты примем это же соглашение и для случая  $k = 2$ . Мы называем это преобразование «развертыванием цикла  $k \times 1$ », потому что развертывание выполняется  $k$  раз, но накопленное значение хранится в единственной переменной `acc`.

**Упражнение 5.7 (решение в конце главы)**

Измените код `combine5` и примените развертывание цикла с коэффициентом  $k = 5$ .

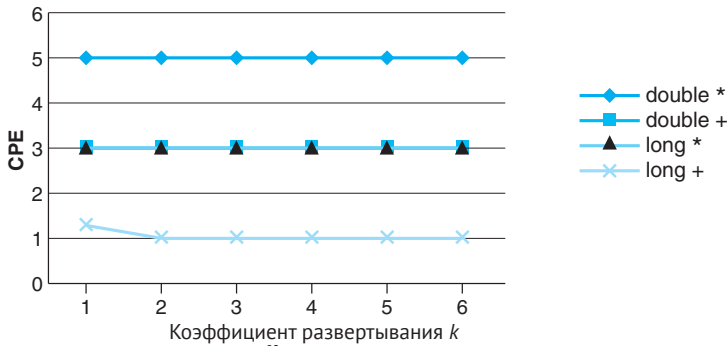
Измерив производительность кода с развернутым циклом с коэффициентами  $k = 2$  (`combine5`) и  $k = 3$ , мы получили следующие результаты:

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine4	5.7	Без развертывания цикла	1,27	3,01	3,01	5,01
combine5	5.8	С развертыванием цикла 2×1	1,01	3,01	3,01	5,01
		С развертыванием цикла 3×1	1,01	3,01	3,01	5,01
Граница задержки			1,00	3,00	3,00	5,00
Граница пропускной способности			0,50	1,00	1,00	0,50

Как видите, величина CPE для случая суммирования целых чисел улучшилась и достигла границы задержки 1,00. Можно уверенно предположить, что это улучшение произошло за счет уменьшения накладных расходов на выполнение итераций. Уменьшив количество служебных операций по отношению к количеству сложений, необходимых для вычисления суммы элементов вектора, мы смогли достичь точки, когда ограничи-



вающим фактором становится задержка целочисленного сложения. С другой стороны, дальнейшее увеличение коэффициента развертывания не дало улучшения, потому что граница задержки уже достигнута. На рис. 5.8 показан график зависимости измеренной величины CPE от коэффициента развертывания цикла до 6. Можно заметить, что тенденция к улучшению, наблюдаемая при развертывании с коэффициентами 2 и 3, сохраняется – ни в одном случае величина CPE не удаляется от границы задержки.



**Рис. 5.8.** Величина CPE с разными коэффициентами развертывания цикла  $k \times 1$ . Это преобразование увеличило производительность только целочисленного сложения

Чтобы понять, почему развертывание  $k \times 1$  не может улучшить производительность и преодолеть границу задержки, рассмотрим машинный код цикла в `combine5`, развернутый с коэффициентом  $k = 2$ . Компилятор генерирует следующий код для типа данных `double` и операции умножения:

```

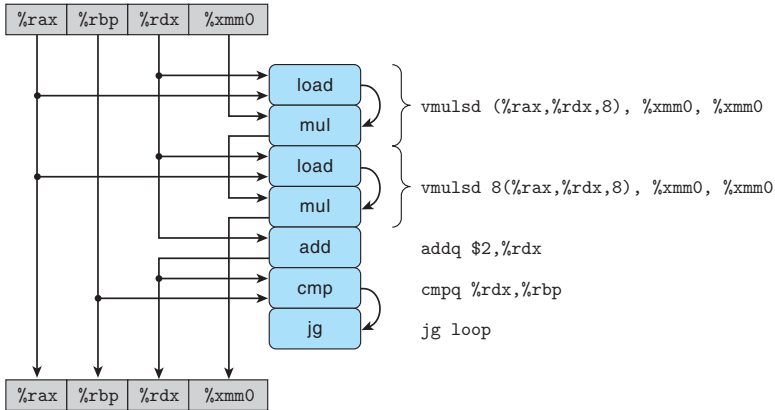
Цикл в combine5. data_t = double, OP = *
i в %rdx, data в %rax, limit в %rbp, acc в %xmm0
1 .L35:                                loop:
2 vmulsd (%rax,%rdx,8), %xmm0, %xmm0    Умножить acc на data[i]
3 vmulsd 8(%rax,%rdx,8), %xmm0, %xmm0    Умножить acc на data[i+1]
4 addq $2, %rdx                        Увеличить i на 2
5 cmpq %rdx, %rbp                     Сравнить limit:i
6 jg .L35                             Если >, перейти к loop

```

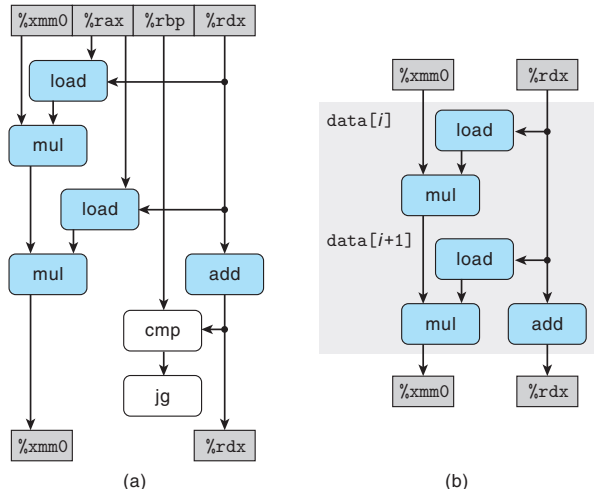
Как можно заметить, здесь GCC использует более прямолинейную трансляцию ссылок на массив, по сравнению с кодом на основе указателей, сгенерированным для `combine4`<sup>2</sup>. Индекс цикла `i` хранится в регистре `%rdx`, а адрес `data` – в регистре `%rax`. Как и раньше, накопленное значение `acc` хранится в регистре `%xmm0`. Развертывание цикла привело к созданию двух инструкций `vmulsd` – одна умножает `data[i]` на `acc`, а вторая `data[i+1]` на `acc`. На рис. 5.9 показано представление этого кода в виде графа. Каждая инструкция `vmulsd` преобразуется в две операции: одна загружает элемент массива из памяти, а вторая умножает это значение на накопленное значение. Здесь мы видим, что в каждой итерации регистр `%xmm0` читается и записывается дважды. Мы можем переупорядочить, упростить и абстрагировать этот граф, следуя процессу, изображенному на рис. 5.10 (а), чтобы получить шаблон, изображенный на рис. 5.10 (б), а затем повторить этот шаблон  $n/2$  раз, дабы получить последовательность вычислений для вектора с дли-

<sup>2</sup> В процессе работы оптимизатор GCC генерирует несколько вариантов функции, а затем выбирает тот, который, по его прогнозам, даст наилучшую производительность и наименьший размер кода. Как следствие небольшие изменения в исходном коде могут приводить к самым разным формам машинного кода. Мы обнаружили, что выбор между реализацией на основе указателей и реализацией на основе массива не влияет на производительность программ, выполняющихся на нашей эталонной машине.

ной  $n$ , как показано на рис. 5.11. В этом графе все еще существует критический путь из  $n$  операций умножения – итераций стало вдвое меньше, но каждая итерация включает две последовательные операции умножения. Поскольку критический путь был ограничивающим фактором производительности для кода без развертывания цикла, он остается таковым для кода с развертыванием цикла  $k \times 1$ .



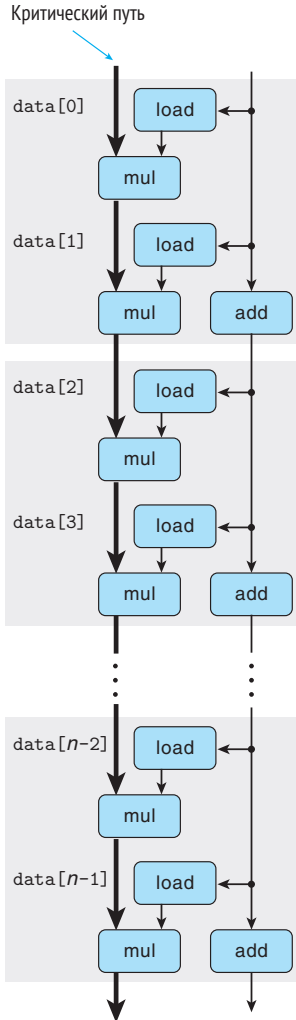
**Рис. 5.9.** Представление цикла в функции `combine5` в виде графа. В каждой итерации выполняются две инструкции `vmulsd`, каждая из которых преобразуется в операции `load` и `mul`



**Рис. 5.10.** Абстрагирование графа операций в функции `combine5`. Мы переупорядочили, упростили и абстрагировали граф на рис. 5.9, чтобы показать зависимости по данным между последовательными итерациями (a). Как можно заметить, каждая итерация выполняет две операции умножения друг за другом (b)

### Развертывание циклов компилятором

Компиляторы с легкостью справляются с развертыванием циклов. Многие включают этот прием в свои наборы оптимизаций. Компилятор GCC, например, выполняет развертывание циклов при выборе уровня оптимизации 3 или выше.



**Рис. 5.11.** Представление операций, выполняемых в `combine5` с вектором длиной  $n$ . Даже после разворачивания цикла с коэффициентом  $2n$  операций `mul` образуют критический путь

## 5.9. Увеличение степени параллелизма

Итак, наши функции достигли границы, определяемой задержками арифметических блоков. Однако, как отмечалось выше, функциональные блоки, выполняющие сложение и умножение, имеют конвейерную организацию и могут начать выполнять новую операцию с началом каждого следующего цикла синхронизации. Аппаратные модули способны выполнять операции умножения и сложения с гораздо большей скоростью, но наш код не может использовать эту возможность даже при разворачивании цикла, потому что значение накапливается в единственной переменной `acc`. Процессор не может приступить к вычислению нового значения для `acc`, пока не завершится предыдущее вычисление. Несмотря на то что функциональный блок, вычисляющий новое значение для `acc`, может начать выполнять новую операцию в каждом цикле синхронизации, он вынужден ждать  $L$  циклов, где  $L$  – задержка операции комбинирования. Давайте посмотрим, как можно разорвать эту зависимость и повысить производительность, преодолев границу, установленную задержкой.

### 5.9.1. Несколько аккумуляторов

Для ассоциативной и коммутативной комбинирующей операции, например для целочисленного сложения или умножения, производительность можно повысить, разбив набор комбинирующих операций на две или более частей, и комбинировать их результаты в конце. Например, пусть  $P_n$  обозначает произведение элементов  $a_0, a_1, \dots, a_{n-1}$ :

$$P_n = \prod_{i=0}^{n-1} a_i.$$

Если предположить, что  $n$  – четное число, эту формулу можно записать как  $P_n = PE_n \times PO_n$ , где  $PE_n$  – произведение элементов с четными индексами, а  $PO_n$  – произведение элементов с нечетными индексами:

$$PE_n = \prod_{i=0}^{n/2-1} a_{2i},$$

$$PO_n = \prod_{i=0}^{n/2-1} a_{2i+1}.$$

В листинге 5.9 показан код, реализующий этот метод. Он использует не только разворачивание цикла с коэффициентом 2, чтобы скомбинировать больше элементов в одной итерации, но и увеличивает степень параллелизма, накапливая элементы с четными индексами в переменной `acc0` и с нечетными – в переменной `acc1`. Мы называем это «развертыванием цикла 2×2». Как и прежде, в функцию добавлен второй цикл, накапливающий любые оставшиеся элементы массива на случай, если длина вектора не будет кратна 2. И в самом конце выполняется комбинирующая операция с аккумуляторами `acc0` и `acc1` для вычисления окончательного результата.

**Листинг 5.9.** Применение развертывания цикла 2×2. Благодаря накоплению результатов в двух аккумуляторах эта версия эффективнее использует преимущество наличия нескольких функциональных блоков и их способность обрабатывать операции конвейерным способом

```

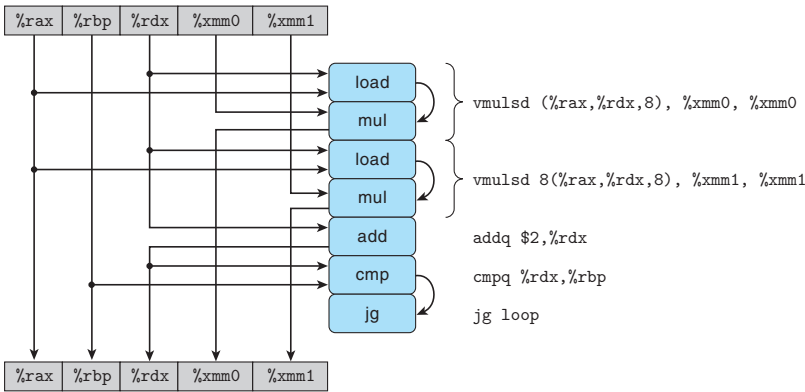
1 /* Развертывание цикла 2x2 */
2 void combine6(vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(v);
6     long limit = length-1;
7     data_t *data = get_vec_start(v);
8     data_t acc0 = IDENT;
9     data_t acc1 = IDENT;
10
11     /* Комбинировать по 2 элемента в каждой итерации */
12     for (i = 0; i < limit; i+=2) {
13         acc0 = acc0 OP data[i];
14         acc1 = acc1 OP data[i+1];
15     }
16
17     /* Обработать оставшиеся элементы */
18     for (; i < length; i++) {
19         acc0 = acc0 OP data[i];
20     }
21     *dest = acc0 OP acc1;
22 }
```

Для сравнения реализаций с простым развертыванием цикла и с развертыванием цикла и двойным параллелизмом мы измерили производительность и получили следующие результаты:

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine4	5.7	Без развертывания цикла	1,27	3,01	3,01	5,01
combine5	5.8	С развертыванием цикла 2×1	1,01	3,01	3,01	5,01
combine6	5.9	С развертыванием цикла 2×2	0,81	1,51	1,51	2,51
Граница задержки			1,00	3,00	3,00	5,00
Граница пропускной способности			0,50	1,00	1,00	0,50

Как видите, производительность улучшилась во всех случаях: производительность вариантов с целочисленным умножением, сложением с плавающей точкой и умножением с плавающей точкой улучшилась практически в 2 раза, производительность варианта с целочисленным сложением тоже улучшилась, хотя и не в 2 раза. Но что особенно важно, мы преодолели барьер, установленный границей задержки. Процессору больше не нужно откладывать начало одной операции комбинирования до завершения предыдущей.

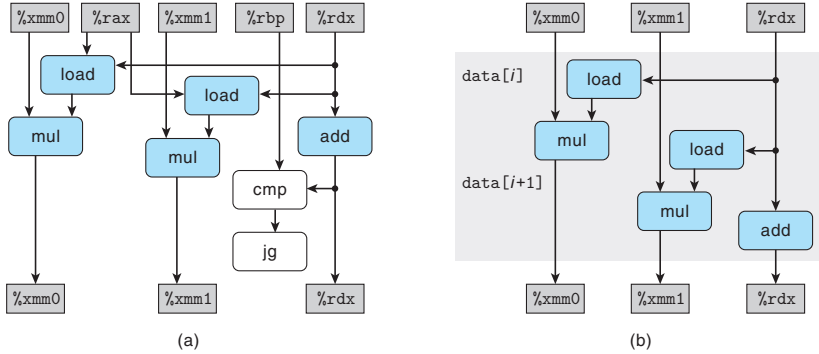
Чтобы разобраться, как достигается такая производительность в combine6, нарисуем базовый граф с последовательностью операций (рис. 5.12).



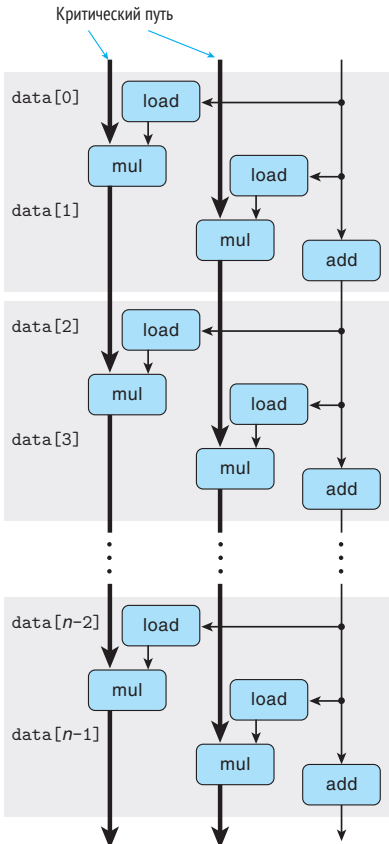
**Рис. 5.12.** Представление цикла в функции combine6 в виде графа. В каждой итерации выполняются две инструкции `vmulsd`, каждая из которых преобразуется в операции `load` и `mul`

Из него вытекает шаблон, показывающий зависимости по данным между итерациями, как показано на рис. 5.13. Так же как в combine5, в каждой итерации в combine6 выполняются две инструкции `vmulsd`, но в этой версии инструкции преобразуются в операции `mul`, которые читают и записывают отдельные регистры, не образуя зависимость по данным между собой (рис. 5.13 (b)). Повторив этот шаблон  $n/2$  раз (рис. 5.14), получаем поток обработки вектора с длиной  $n$ . Как видите, теперь в графе два критических пути: один соответствует вычислению произведения четных элементов (значение `acc0`), второй – нечетных элементов (значение `acc1`). Каждый из этих критических путей выполняет всего  $n/2$  операций, что дает величину CPE около  $5,00 / 2 = 2,50$ . Приведенный анализ объясняет наблюдаемое значение CPE, близкое к  $L/2$  для операций с задержкой  $L$ . Получив возможность использовать преимущества наличия

нескольких функциональных блоков, программа увеличила производительность в 2 раза. Единственное исключение – версия с целочисленным сложением. Да, величина CPE уменьшилась до уровня ниже 1,0, но накладные расходы на итерации все еще слишком велики в сравнении с комбинирующей операцией, чтобы программа могла достичь теоретического предела 0,50.



**Рис. 5.13.** Абстрагирование графа операций в функции `combine6`. Мы переупорядочили, упростили и абстрагировали граф на рис. 5.12, чтобы показать зависимости по данным между последовательными итерациями (а). Как можно заметить, операции `mul` не зависят друг от друга (b)



**Рис. 5.14.** Представление операций, выполняемых в `combine6` с вектором длины  $n$ . Теперь в графе присутствуют два критических пути, в каждом из которых выполняется  $n/2$  операций

**Приложение в интернете OPT:SIMD.** Увеличение степени параллелизма с применением векторных инструкций

Как рассказывалось в разделе 3.1, в 1999 году компания Intel реализовала инструкции SSE, где SSE расшифровывается как Streaming SIMD Extensions (поточковые расширения SIMD), а аббревиатура SIMD (произносится как «сим-ди») в свою очередь расшифровывается как «single instruction, multiple data» (одна команда, много данных). За эти годы сменилось несколько поколений SSE, и последние версии стали называться AVX (Advanced Vector eXensions – усовершенствованные векторные расширения). Модель выполнения SIMD предполагает применение инструкций к вектору данных целиком. Для этого векторы должны храниться в специальном наборе векторных регистров с именами %ymm0–%ymm15. Текущие векторные регистры AVX имеют длину 32 байта, т. е. каждый может хранить восемь 32-разрядных или четыре 64-разрядных числа, которые могут быть целыми числами или значениями с плавающей точкой. Инструкции AVX могут выполнять векторные операции с этими регистрами, такие как параллельное сложение или умножение восьми или четырех наборов значений. Например, если YMM-регистр %ymm0 хранит восемь чисел с плавающей точкой одинарной точности, которые мы обозначим как  $a_0, \dots, a_7$ , а %rcx содержит адрес памяти, где располагается последовательность из восьми чисел с плавающей точкой одинарной точности, которые мы обозначим как  $b_0, \dots, b_7$ , то инструкция

```
vmulps (%rcx), %ymm0, %ymm1
```

прочитает восемь значений из памяти, параллельно выполнит восемь умножений  $a_i \cdot b_i$  для  $0 \leq i < 8$  и сохранит восемь произведений в векторном регистре %ymm1. Как видите, одна инструкция может выполнить вычисления с несколькими значениями данных, откуда и термин «SIMD».

GCC поддерживает расширения языка C, позволяющие программистам выражать в программах векторные операции, которые могут быть скомпилированы в векторные инструкции AVX (или в более ранние инструкции SSE). Этот прием предпочтительнее написания кода непосредственно на языке ассемблера, потому что GCC также может генерировать векторные инструкции, поддерживаемые другими процессорами.

Используя комбинацию из инструкций GCC, развертывания цикла и нескольких аккумуляторов, можно достичь следующей производительности в наших функциях комбинирования:

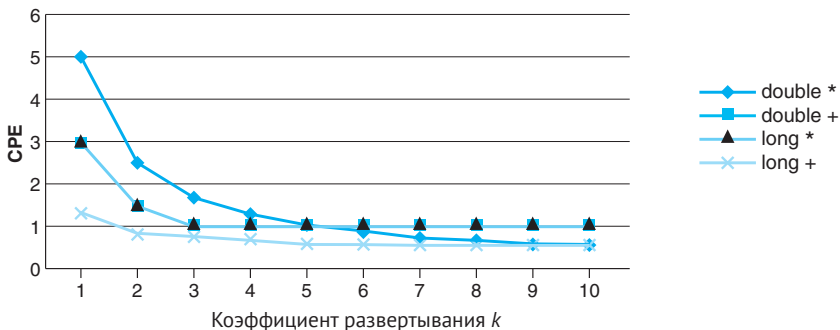
Метод	Целочисленные данные				Данные с плавающей точкой			
	int		long		float		double	
	+	*	+	*	+	*	+	*
Скаляр 10×10	0,54	1,01	0,55	1,00	1,01	0,51	1,01	0,52
Граница пропускной способности для скаляров	0,50	0,50	1,00	1,00	1,00	1,00	0,50	0,50
Вектор 8×8	0,05	0,24	0,13	1,51	0,12	0,08	0,25	0,16
Граница пропускной способности для векторов	0,06	0,12	0,12	–	0,12	0,06	0,25	0,12

Первый набор чисел отражает производительность обычного, *скалярного* кода, написанного в стиле combine6, с развертыванием цикла 10×10. Второй набор чисел отражает производительность кода, написанного в форме, которую GCC может скомпилировать в векторный код AVX. Помимо использования векторных операций, эта версия применяет развертывание цикла 8×8. В таблице показаны результаты как для 32-, так и для 64-разрядных чисел, потому что векторные инструкции параллельно выполняют 8 операций в первом случае и только 4 во втором.

Первый набор чисел отражает производительность обычного, *скалярного* кода, написанного в стиле `combine6`, с разворачиванием цикла  $10 \times 10$ . Второй набор чисел отражает производительность кода, написанного в форме, которую GCC может скомпилировать в векторный код AVX. Помимо использования векторных операций, эта версия применяет разворачивание цикла  $8 \times 8$ . В таблице показаны результаты как для 32-, так и для 64-разрядных чисел, потому что векторные инструкции параллельно выполняют 8 операций в первом случае и только 4 во втором.

Как видите, векторный код обеспечивает почти восьмикратное улучшение во всех четырех случаях для 32-разрядных чисел и четырехкратное в трех из четырех случаев для 64-разрядных. Только умножение длинных целых чисел не достигает максимальной производительности в векторном коде. Набор команд AVX не имеет инструкции для параллельного умножения 64-разрядных целых, поэтому GCC не может сгенерировать векторный код для этого случая. Векторные инструкции создают новую границу пропускной способности для операций комбинирования. Она ниже скалярной границы в восемь раз для 32-разрядных операций и в четыре раза – для 64-разрядных. Наш код приблизился к этим границам для нескольких комбинаций типа данных и вида операции.

Мы можем обобщить это преобразование на произвольное множество аккумуляторов, чтобы развернуть цикл с коэффициентом  $k$  и накапливать  $k$  значений параллельно, т. е. получить разворачивание цикла  $k \times k$ . На рис. 5.15 показан эффект применения этого преобразования для значений до  $k = 10$ . Как видите, при достаточно больших значениях  $k$  программа способна вплотную приблизиться к границе пропускной способности для всех вариантов. Для целочисленного сложения при  $k = 7$  достигается производительность  $CPE = 0,54$  – довольно близко к границе пропускной способности 0,50, которую обеспечивают два блока загрузки из памяти. Для целочисленного умножения и сложения с плавающей точкой при  $k \geq 3$  достигается производительность  $CPE = 1,01$  – совсем рядом с границей пропускной способности 1,00, которую имеют функциональные блоки. Для умножения с плавающей точкой при  $k \geq 10$  достигается производительность  $CPE = 0,51$  – почти на границе пропускной способности 0,50 двух блоков умножения и двух блоков загрузки из памяти. Обратите внимание, что наш код может достичь почти вдвое большей пропускной способности при использовании умножения с плавающей точкой, чем при сложении с плавающей точкой, хотя умножение является более сложной операцией.



**Рис. 5.15.** Зависимость величины производительности CPE от величины коэффициента разворачивания цикла  $k \times k$ . Все варианты комбинирования улучшают CPE при применении этого преобразования и практически достигают границы пропускной способности

В общем случае программа может достичь предела пропускной способности для некоторой операции, только если сумеет заполнить конвейеры всех функциональных блоков, способных выполнять эту операцию. Для операции с задержкой  $L$  и емкостью  $C$



это потребует реализовать развертывание  $k \geq C \cdot L$ . Например, умножение с плавающей точкой имеет  $C = 2$  и  $L = 5$ , поэтому коэффициент развертывания должен быть  $k \geq 10$ . Сложение с плавающей точкой имеет  $C = 1$  и  $L = 3$ , и для этой операции максимальная пропускная способность достигается при  $k \geq 3$ .

Выполняя развертывание  $k \times k$ , необходимо учитывать сохранность функциональности исходного кода. В главе 2 мы видели, что арифметика в дополнительном коде коммутативна и ассоциативна, даже когда возникает переполнение. Следовательно, для целочисленного типа данных результат функции `combine6` будет идентичен результату функции `combine5` при всех возможных условиях. То есть оптимизирующий компилятор потенциально может преобразовать код функции `combine4` сначала в вариант, показанный в `combine5`, развернув цикл с коэффициентом 2, а затем в вариант `combine6`, добавив параллелизм. Некоторые компиляторы действуют именно так или похожим образом, чтобы повысить производительность обработки целочисленных данных.

С другой стороны, умножение и сложение с плавающей точкой не ассоциативны. Поэтому `combine5` и `combine6` могут давать разные результаты из-за ошибок округления или переполнения. Представьте, например, вычисление произведения, в котором все элементы с четными индексами являются числами с очень большими абсолютными значениями, а элементы с нечетными индексами близки к 0,0. В таком случае при вычислении произведения  $PE_n$  может произойти переполнение, а при вычислении произведения  $PO_n$  – потеря точности, даже притом что вычисление общего произведения  $P_n$  протекает нормально и дает точный результат. Однако в большинстве реальных приложений такие закономерности маловероятны. Поскольку большинство физических явлений непрерывны, числовые данные обычно достаточно гладкие. Даже имеющиеся разрывы, как правило, не порождают периодических закономерностей, способных привести к ситуации, описанной выше. Маловероятно, что перемножение элементов в строгом порядке даст лучшую точность, чем перемножение двух групп независимо с последующим умножением произведений, полученных для групп. Для большинства приложений двукратное увеличение производительности оправдывает риск получения разных результатов для данных с необычной структурой. И все же разработчик программы должен проконсультироваться с потенциальными пользователями, чтобы узнать, существуют ли определенные условия, которые могут сделать усовершенствованный алгоритм неприемлемым. Большинство компиляторов не предпринимают попыток таких преобразований для операций с плавающей точкой, потому что они не могут оценить риски подобных преобразований, которые могут изменить поведение программы.

### 5.9.2. Переупорядочение операций

Теперь исследуем другой способ разорвать зависимости между последовательными итерациями и тем самым повысить производительность, преодолев границу задержки. Мы видели, что развертывание цикла  $k \times 1$  для `combine5` не меняло набор операций, выполняемых при комбинировании элементов вектора. Однако, лишь немного изменив код, можно кардинально поменять способ выполнения комбинирования, а также значительно повысить производительность программы.

В листинге 5.10 показана функция `combine7`, которая отличается от `combine5` (листинг 5.8) только порядком комбинирования элементов в цикле. В `combine5` комбинирование выполняется так:

```
12     acc = (acc OP data[i]) OP data[i+1];
```

а в `combine7` так:

```
12     acc = acc OP (data[i] OP data[i+1]);
```

Отличие состоит только в размещении двух круглых скобок. Мы называем это преобразование *переупорядочением операций*, потому что круглые скобки изменяют порядок, в котором элементы вектора объединяются с накопленным значением асс, давая форму развертывания цикла, которую мы называем « $2 \times 1a$ ».

Неопытному глазу эти два выражения могут показаться практически одинаковыми, но, измерив CPE, мы получаем удивительный результат:

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine4	5.7	Без развертывания цикла	1,27	3,01	3,01	5,01
combine5	5.8	С развертыванием цикла $2 \times 1$	1,01	3,01	3,01	5,01
combine6	5.9	С развертыванием цикла $2 \times 2$	0,81	1,51	1,51	2,51
combine7	5.10	С развертыванием цикла $2 \times 1a$	1,01	1,51	1,51	2,51
Граница задержки			1,00	3,00	3,00	5,00
Граница пропускной способности			0,50	1,00	1,00	0,50

**Листинг 5.10.** Применение развертывания  $2 \times 1a$ . Изменив порядок выполнения операций, мы увеличили количество операций, выполняемых параллельно

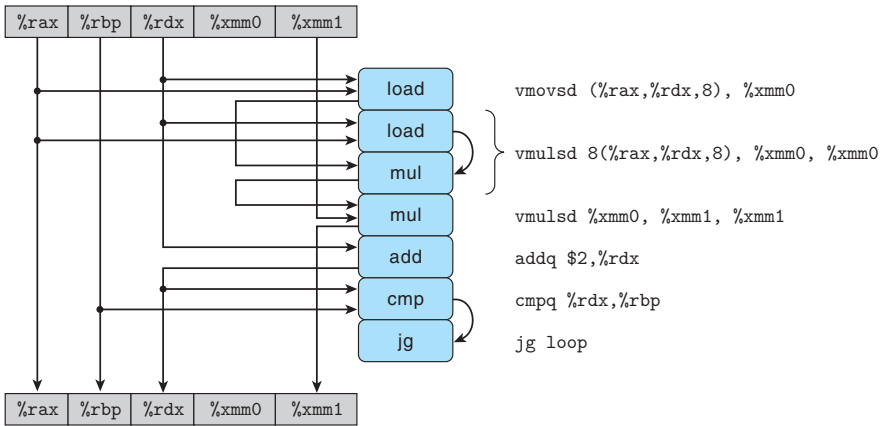
```

1 /* Развертывание цикла  $2 \times 1a$  */
2 void combine7(vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(v);
6     long limit = length-1;
7     data_t *data = get_vec_start(v);
8     data_t acc = IDENT;
9
10    /* Комбинировать по 2 элемента в каждой итерации */
11    for (i = 0; i < limit; i+=2) {
12        acc = acc OP (data[i] OP data[i+1]);
13    }
14
15    /* Обработать оставшиеся элементы */
16    for (; i < length; i++) {
17        acc = acc OP data[i];
18    }
19    *dest = acc;
20 }
```

Вариант с целочисленным сложением по производительности соответствует версии с развертыванием  $k \times 1$  (combine5), в то время как производительность трех других вариантов соответствуют производительности версий с параллельными аккумуляторами (combine6), удваивая производительность по сравнению с развертыванием  $k \times 1$ . Эти варианты преодолели барьер, установленный границей задержки.

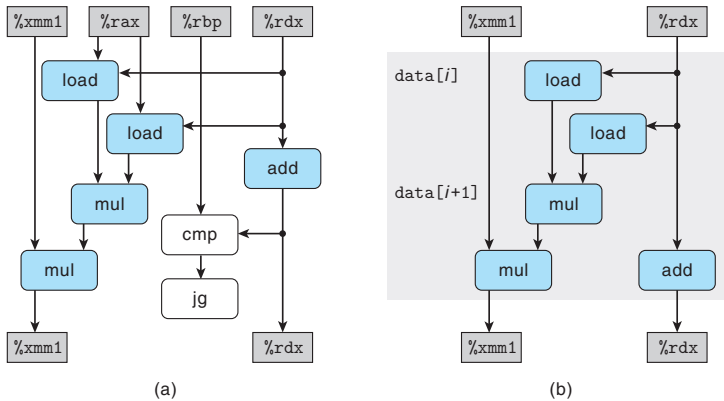
На рис. 5.16 показано, как код цикла в combine7 (в варианте с типом данных double и умножением в качестве операции комбинирования) декодируется в операции и зависимости по данным. Как видите, операции **load**, являющиеся результатом декодирования инструкции vmovsd и первой инструкции vmulsd, загружают элементы вектора  $i$

и  $i + 1$  из памяти, а первая операция **mul** перемножает их. Затем вторая операция умножения умножает полученный результат на накопленное значение в асс.

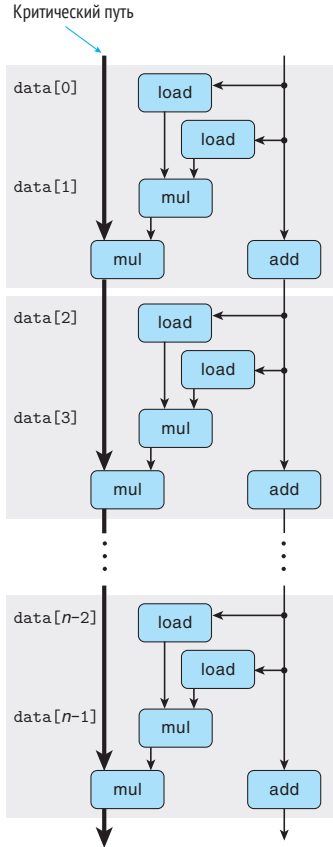


**Рис. 5.16.** Представление цикла в функции `combine7` в виде графа. Инструкции в каждой итерации декодируются так же, как инструкции в `combine5` или `combine6`, но с разными зависимостями по данным

На рис. 5.17 (а) показано, как мы переупорядочили, упростили и абстрагировали граф, изображенный на рис. 5.16, чтобы получить шаблон, представляющий зависимости по данным для одной итерации (рис. 5.17 (b)). Так же как в случае с шаблонами для `combine5` и `combine7`, мы имеем две операции **load** и две операции **mul**, но только одна из операций **mul** образует цепочку зависимости по данным между итерациями. Затем, скопировав этот шаблон  $n/2$  раз, чтобы показать вычисления, выполняемые при перемножении  $n$  элементов вектора (рис. 5.18), мы видим, что в критическом пути находятся только  $n/2$ . Первое умножение в каждой итерации может выполняться, не дожидаясь, когда будет готово накопленное значение, вычисление которого началось в предыдущей итерации. В результате этой простой манипуляции со скобками мы уменьшили величину CPE почти в 2 раза.

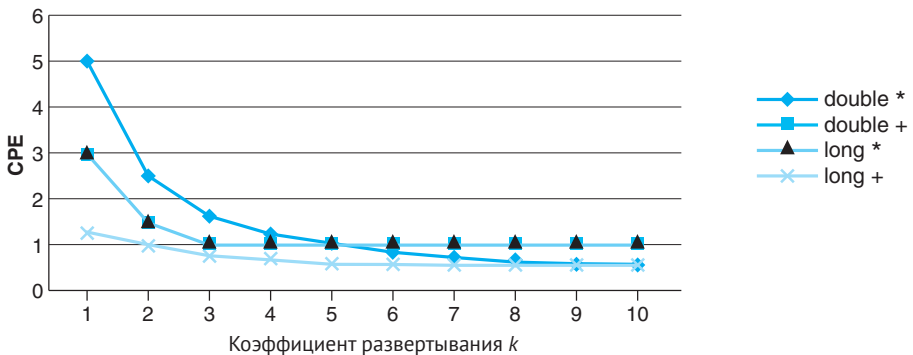


**Рис. 5.17.** Абстрагирование графа операций в функции `combine7`. Мы переупорядочили, упростили и абстрагировали граф на рис. 5.16, чтобы показать зависимости по данным между последовательными итерациями. Верхняя операция **mul** перемножает два элемента вектора, а нижняя – умножает результат на аккумулятор асс



**Рис. 5.18.** Представление операций, выполняемых в combine7 с вектором длиной  $n$ . Здесь только один критический путь, но он содержит  $n/2$  операций

На рис. 5.19 показан эффект переупорядочения операций для достижения того, что мы называем разворачиванием цикла  $k \times 1a$ , со значениями до  $k = 10$ . Как видите, это преобразование дает эффект, похожий на эффект, который производит разворачивание цикла  $k \times k$ . Во всех вариантах программа приближается к границе пропускной способности функциональных блоков.



**Рис. 5.19.** Зависимость величины производительности CPE от величины коэффициента разворачивания цикла  $k \times 1a$ . Все варианты комбинирования улучшают CPE при применении этого преобразования и практически достигают границы пропускной способности

Применяя переупорядочение операций, мы также меняем порядок комбинирования элементов вектора. Для целочисленного сложения и умножения, вследствие их ассоциативности, это переупорядочение не повлияет на результат. Для сложения и умножения с плавающей точкой мы должны, как и прежде, оценить, может ли такое переупорядочение существенно повлиять на результат. Мы со своей стороны готовы поспорить, что для большинства приложений разница будет несущественной.

Итак, переупорядочение операций может уменьшить количество операций в критическом пути вычислений и повысить производительность за счет более полного использования нескольких функциональных блоков и их возможностей конвейерной обработки. Большинство компиляторов не пытаются переупорядочивать операции с плавающей точкой, потому что их ассоциативность не гарантируется. Текущие версии GCC действительно выполняют переупорядочение целочисленных операций, но это не всегда дает положительный эффект. Мы обнаружили, что развертывание цикла и параллельное накопление результатов в нескольких аккумуляторах является более надежным способом повышения производительности программы.

#### Упражнение 5.8 (решение в конце главы)

Взгляните на следующую функцию, вычисляющую произведение элементов типа `double` массива с длиной `n`. Здесь применен прием развертывания цикла с коэффициентом `k = 3`.

```
double aprod(double a[], long n)
{
    long i;
    double x, y, z;
    double r = 1;
    for (i = 0; i < n-2; i+= 3) {
        x = a[i]; y = a[i+1]; z = a[i+2];
        r = r * x * y * z; /* Вычисление произведения */
    }
    for (; i < n; i++)
        r *= a[i];
    return r;
}
```

В строке с комментарием «Вычисление произведения» можно использовать круглые скобки, чтобы упорядочить выполнение операций пятью разными способами:

```
r = ((r * x) * y) * z; /* A1 */
r = (r * (x * y)) * z; /* A2 */
r = r * ((x * y) * z); /* A3 */
r = r * (x * (y * z)); /* A4 */
r = (r * x) * (y * z); /* A5 */
```

Допустим, что эти функции вызываются на машине, где умножение с плавающей точкой имеет задержку в 5 циклов синхронизации. Определите нижнюю границу CPE, определяемую зависимостями по данным в операциях умножения. (*Подсказка*: нарисуйте граф потока данных, представляющий вычисление `r` в каждой итерации.)

## 5.10. Обобщение результатов оптимизации комбинирующего кода

Наши усилия, направленные на максимальное увеличение производительности процедуры, складывающей или перемножающей элементы вектора, явно окупились. В следующей таблице обобщены результаты, которых мы добились с использованием

скалярного кода, т. е. без использования векторных инструкций AVX, обеспечивающих параллельное выполнение операций:

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine1	5.3	Абстрактный –01	10,12	10,12	10,17	11,14
combine6	5.9	С разворачиванием цикла 2×2	0,81	1,51	1,51	2,51
		С разворачиванием цикла 10×10	0,55	1,00	1,01	0,52
Граница задержки			1,00	3,00	3,00	5,00
Граница пропускной способности			0,50	1,00	1,00	0,50

Используя несколько оптимизаций, мы смогли достичь значений CPE, близких к границам пропускной способности 0,50 и 1,00 функциональных блоков. Мы смогли добиться 10–20-кратного улучшения производительности, причем меняя только исходный код на C и используя стандартный компилятор. Применение новых расширений в языке C, направленных на использование инструкций SIMD, дало дополнительный прирост производительности от 4 до 8 раз. Например, для умножения с одинарной точностью величина CPE упала с исходных 11,14 цикла на элемент до 0,06, что дало общий прирост производительности более чем в 180 раз. Этот пример демонстрирует, что современные процессоры обладают значительными вычислительными мощностями, но нам может потребоваться приложить некоторые усилия, чтобы вытянуть из них эту мощь, разрабатывая программы особым образом.

## 5.11. Некоторые ограничивающие факторы

Как мы видели, критический путь в представлении программы в виде графа потока данных помогает определить фундаментальную нижнюю границу времени, необходимого для выполнения программы. То есть если в программе есть цепочка зависимостей по данным, в которой сумма всех задержек равна  $T$ , то программе потребуется не менее  $T$  циклов для выполнения.

Мы также видели, что границы пропускной способности функциональных блоков тоже задают нижнюю границу времени выполнения программы. То есть если в общей сложности программе требуется выполнить некоторую операцию  $N$  раз, микропроцессор имеет  $C$  функциональных блоков, способных выполнять эту операцию, и время выполнения одной операции одним блоком равно  $I$ , то на выполнение этих операций потребуется не менее  $N \cdot I / C$  циклов синхронизации.

В этом разделе мы рассмотрим еще несколько факторов, ограничивающих производительность программ на реальных машинах.

### 5.11.1. Вытеснение регистров

Выгоды от параллельного выполнения операций в циклах ограничены возможностью выразить вычисления в ассемблерном коде. Если программа имеет степень параллелизма  $P$ , превышающую количество доступных регистров, то компилятор прибегает к приему *вытеснения регистров* (register spilling), сохраняя некоторых временных значений в памяти, обычно выделяя место в стеке. Например, ниже показаны результаты расширения схемы с несколькими аккумуляторами в combine6 до  $k = 10$  и  $k = 20$ :

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine6	5.9	С развертыванием цикла 10×10	0,55	1,00	1,01	0,52
		С развертыванием цикла 20×20	0,83	1,03	1,02	0,68
Граница пропускной способности			0,50	1,00	1,00	0,50

Как видите, ни один из показателей CPE не улучшился с увеличением коэффициентов развертывания, а некоторые даже ухудшились. Современные процессоры x86-64 имеют 16 целочисленных регистров и могут использовать 16 регистров УММ для хранения данных с плавающей точкой. Если количество переменных цикла превышает количество доступных регистров, то программа вынуждена разместить некоторые из них в стеке.

Следующий фрагмент иллюстрирует, как реализуется обновление аккумулятора `acc0` в коде, реализованном с применением развертывания цикла 10×10:

Обновление аккумулятора `acc0` в коде с развертыванием цикла 10×10

```
vmulsd  (%rdx), %xmm0, %xmm0  acc0 *= data[i]
```

В этом примере аккумулятор хранится в регистре `%xmm0`, поэтому программа может просто читать `data[i]` из памяти и умножать их на значение в этом регистре.

Аналогичный исходный код с развертыванием цикла 20×20 дает совершенно иной ассемблерный код:

Обновление аккумулятора `acc0` в коде с развертыванием цикла 20×20

```
vmovsd  40(%rsp), %xmm0
vmulsd  (%rdx), %xmm0, %xmm0
vmovsd  %xmm0, 40(%rsp)
```

Аккумулятор хранится в локальной переменной в стеке со смещением 40 от адреса в указателе стека. Программа должна прочесть из памяти уже два значения, аккумулятор и `data[i]`, перемножить их и сохранить результат обратно в память.

Как только возникает необходимость прибегнуть к вытеснению регистров, любые преимущества использования нескольких аккумуляторов тут же теряются. К счастью, в x86-64 достаточно регистров, поэтому пропускная способность большинства циклов будет ограничена еще до того, как возникнет необходимость в вытеснении регистров.

### 5.11.2. Прогнозирование ветвлений и штрафы за ошибки предсказания

Как показали эксперименты в разделе 3.6.6, условное ветвление может привести к *значительному штрафу* за неверное предсказание, когда логика, предсказывающая ветвления, дает неверный прогноз. Теперь, имея некоторое представление о работе процессоров, мы можем понять, откуда возникает этот штраф.

Современные процессоры работают с некоторым опережением, извлекая новые инструкции из памяти и декодируя их, чтобы определить, какие операции выполнять и с какими операндами. Такая *конвейерная обработка инструкций* прекрасно справляется со своей задачей, пока инструкции следуют в простой последовательности. Обнаружив ветвление, процессор должен угадать, в каком направлении продолжится выполнение. То есть, встретив условный переход, он должен угадать, будет ли выполнен этот переход. Для такой инструкции, как косвенный переход (как мы видели в коде, реализующем пе-

реходы по таблице переходов) или возврат из процедуры, это означает предсказание целевого адреса. В этом обсуждении мы сосредоточимся на условных ветвях.

При использовании *спекулятивного выполнения* процессор начинает выполнение инструкций из спрогнозированной ветви. При этом ни один фактический регистр или ячейка памяти не изменяется, пока не станет окончательно ясно, будет ли выполнен переход. Если прогноз оказался верным, то процессор просто «подтверждает» результаты спекулятивного выполнения инструкций, сохраняя их в регистрах или в памяти. Если прогноз оказался ошибочным, то процессор отбросит все полученные результаты и начнет процесс выборки инструкций с верного адреса. При этом возникает значительный штраф, потому что конвейер инструкций должен снова наполниться, прежде чем он даст полезные результаты.

В разделе 3.6.6 рассказывалось, что последние версии процессоров x86, включая все процессоры с архитектурой x86-64, поддерживают инструкции условного перемещения. GCC может генерировать код с этими инструкциями вместо традиционной передачи управления. Основная идея трансляции в код, использующий условные перемещения, состоит в том, чтобы выполнить вычисления или инструкции, составляющие условное выражение, и затем использовать условное перемещение для передачи желаемого значения. В разделе 4.5.7 мы видели, что инструкции условного перемещения можно использовать в конвейерной обработке, не гадая, будет ли выполнено условие, и, следовательно, избежать штрафа за неверный прогноз.

Но может ли программист на C убедиться, что штрафы за неверное предсказание ветвления не снижают эффективность программы? Учитывая 19-тактный штраф за неверное предсказание, который мы получили в результате измерений на эталонной машине, ставки очень высоки. На этот вопрос нет простого ответа, но можно использовать следующие общие принципы.

### Не беспокойтесь о предсказуемости ветвления

Как мы видели, штраф за неверное предсказание ветвления может быть очень высоким, но это не означает, что все ветвления будут замедлять программу. Фактически логика прогнозирования ветвлений в современных процессорах хорошо распознает типичные шаблоны ветвления. Например, ветвления, закрывающие цикл в наших процедурах комбинирования, обычно предсказываются как выполняемые, и, следовательно, ошибка предсказания возникнет только в конце последней итерации.

Вот еще один пример: рассмотрим результаты, которые мы наблюдали при переходе от `combine2` к `combine3`, когда вынесли функцию `get_vec_element` из цикла:

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
<code>combine2</code>	5.4	С выносом <code>vec_length</code> из цикла	7,02	9,03	9,02	11,03
<code>combine3</code>	5.6	Прямой доступ к данным	7,17	9,02	9,02	11,03

Показатель CPE не улучшился, даже притом что из каждой итерации мы убрали два условия, которые проверяют, находится ли индекс в пределах границ вектора. В этой функции проверки всегда успешны, и, следовательно, они хорошо предсказуемы.

Чтобы измерить влияние проверки границ на производительность, рассмотрим код в листинге 5.11, в котором мы изменили цикл в `combine4`, заменив обращение к элементу вектора результатом выполнения встроенного кода `get_vec_element`. Назовем эту новую версию `combine4b`. Этот код выполняет проверку границ, а также ссылается на элементы вектора через структуру.



**Листинг 5.11.** Накопление результата во временной переменной.  
В цикл включена проверка границ

```
1 /* В цикл включена проверка границ */
2 void combine4b(vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(v);
6     data_t acc = IDENT;
7
8     for (i = 0; i < length; i++) {
9         if (i >= 0 && i < v->len) {
10             acc = acc OP v->data[i];
11         }
12     }
13     *dest = acc;
14 }
```

Теперь можно напрямую сравнить CPE для функций с проверкой границ и без нее:

Функция	Листинг	Метод	Целочисленные данные		Данные с плавающей точкой	
			+	*	+	*
combine4	5.7	Без проверки границ	1,27	3,01	3,01	5,01
combine4b	5.11	С проверкой границ	2,02	3,01	3,01	5,01

Версия с проверкой границ немного медленнее в варианте целочисленного сложения, но в остальных трех вариантах показывает такую же производительность. Производительность этих трех вариантов ограничивается задержками соответствующих операций комбинирования. Дополнительные вычисления, необходимые для проверки границ, могут выполняться параллельно с операциями комбинирования. Процессор верно предсказывает результаты ветвления, поэтому проверка не оказывает большого влияния на выборку и обработку инструкций, образующих критический путь в выполнении программы.

### Пишите код, более подходящий для использования инструкций условного перемещения

Предсказание ветвлений надежно работает только для типичных шаблонов. Но многие проверки в программах совершенно непредсказуемы и зависят от особенностей обрабатываемых данных, например от знака чисел. В этих случаях логика предсказания ветвления будет плохо справляться со своей задачей. При наличии таких непредсказуемых по своей природе случаев производительность программы может быть значительно повышена за счет использования инструкций условного перемещения вместо инструкций условной передачи управления. Программист на C не может управлять этим напрямую, но может использовать некоторые приемы выражения условного поведения, помогающие компилятору принять решение об использовании инструкций условного перемещения.

Мы обнаружили, что GCC генерирует код с инструкциями условного перемещения, когда код написан в более «функциональном» стиле, где условные операции вычисляют значения и затем эти значения используются для обновления состояния программы, в противовес более «императивному» стилю, когда условные выражения используются для выборочного обновления состояния программы.

Нет строгих правил оформления программ в этих двух стилях, поэтому мы проиллюстрируем их на примере. Предположим, нам даны два массива целых чисел  $a$  и  $b$ , и для каждой позиции  $i$  нужно в  $a[i]$  записать минимальное значение из  $a[i]$  и  $b[i]$  и в  $b[i]$  – максимальное.

В императивном стиле выполняется проверка каждой позиции  $i$  и по ее результатам два элемента меняются местами:

```

1 /* Переупорядочение двух векторов такое, что в любой позиции  $b[i] \geq a[i]$  */
2 void minmax1(long a[], long b[], long n) {
3     long i;
4     for (i = 0; i < n; i++) {
5         if (a[i] > b[i]) {
6             long t = a[i];
7             a[i] = b[i];
8             b[i] = t;
9         }
10    }
11 }
```

Согласно проведенным измерениям, эта функция показала величину CPE около 13,5 для случайных данных и 2,5–3,5 – для предсказуемых данных, что соответствует штрафу за ошибочное предсказание примерно в 20 циклов.

Эта же функция, написанная в функциональном стиле, вычисляет минимальное и максимальное значения в каждой позиции  $i$ , а затем присваивает эти значения элементам  $a[i]$  и  $b[i]$  соответственно:

```

1 /* Переупорядочение двух векторов такое, что в любой позиции  $b[i] \geq a[i]$  */
2 void minmax2(long a[], long b[], long n) {
3     long i;
4     for (i = 0; i < n; i++) {
5         long min = a[i] < b[i] ? a[i] : b[i];
6         long max = a[i] < b[i] ? b[i] : a[i];
7         a[i] = min;
8         b[i] = max;
9     }
10 }
```

Согласно проведенным измерениям, эта функция показала величину CPE около 4,0 независимо от предсказуемости или непредсказуемости данных. (Мы также исследовали сгенерированный ассемблерный код, чтобы убедиться, что в нем действительно используются инструкции условного перемещения.)

Как обсуждалось в разделе 3.6.6, не всякое условное поведение можно реализовать с помощью условного перемещения данных, поэтому неизбежны случаи, когда код, написанный программистом, будет компилироваться в инструкции условной передачи управления, с прогнозированием которых процессор плохо справляется. Но, как мы показали, немного сообразительности со стороны программиста иногда может сделать код более простым для трансляции в инструкции условного перемещения. Конечно, для этого придется поэкспериментировать, написать несколько разных версий функции, а затем изучить сгенерированный ассемблерный код и измерить его производительность.

#### Упражнение 5.9 (решение в конце главы)

Традиционная реализация этапа слияния алгоритма сортировки слиянием требует трех циклов [98]:

```

1 void merge(long src1[], long src2[], long dest[], long n) {
2     long i1 = 0;
```

```

3   long i2 = 0;
4   long id = 0;
5   while (i1 < n && i2 < n) {
6       if (src1[i1] < src2[i2])
7           dest[id++] = src1[i1++];
8       else
9           dest[id++] = src2[i2++];
10  }
11  while (i1 < n)
12      dest[id++] = src1[i1++];
13  while (i2 < n)
14      dest[id++] = src2[i2++];
15 }

```

Ветвления, обусловленные сравнением переменных `i1` и `i2` с `n`, имеют хорошие характеристики прогнозирования – алгоритм прогнозирования ошибается, только когда условия впервые становятся ложными. С другой стороны, сравнение значений `src1[i1]` и `src2[i2]` (строка 6) крайне непредсказуемо для типичных данных. Это сравнение управляет условным ветвлением, давая на случайных данных величину CPE (когда количество элементов равно  $2n$ ) около 15,0.

Перепишите код так, чтобы условный оператор в первом цикле (строки 6–9) транслировался в инструкции условного перемещения.

## 5.12. Понятие производительности памяти

Весь код, написанный нами до сих пор, и все тесты, которые мы запускали, требовали относительно небольшого объема памяти. Например, производительность процедур комбинирования оценивалась на векторах, содержащих 1000 элементов и занимавших не более 8000 байт памяти. Все современные процессоры имеют один или более *кешей*, обеспечивающих быстрый доступ к таким маленьким объемам памяти. В этом разделе мы продолжим исследовать производительность программ, включающих операции загрузки (чтение из памяти в регистры) и сохранения (запись из регистров в память), рассматривая только случаи, когда все данные хранятся в кеше. В главе 6 мы подробно рассмотрим работу кеш-памяти, ее характеристики производительности и дополнительно покажем, как писать код, максимально использующий кеш.

Как показано на рис. 5.4, современные процессоры имеют отдельные функциональные блоки, выполняющие операции загрузки и сохранения, и эти блоки имеют внутренние буферы для хранения запросов, ожидающих выполнения операций с памятью. Например, наша эталонная машина имеет два блока загрузки, каждый из которых может хранить до 72 ожидающих запросов на чтение, и только один блок сохранения с буфером на 42 запроса. Каждый из этих блоков может выполнять 1 операцию в каждом цикле синхронизации.

### 5.12.1. Производительность операций загрузки

Производительность программы, выполняющей операции загрузки, зависит не только от производительности конвейера, но и от производительности блока загрузки. В наших экспериментах с функциями комбинирования на эталонной машине мы заметили, что величина CPE никогда не опускалась ниже 0,50 для любой комбинации типа данных и операции, за исключением случаев использования операций SIMD. Одним из факторов, ограничивающих CPE в наших примерах, является необходимость чтения одного значения из памяти для каждого вычисляемого элемента. С двумя модулями загрузки, каждый из которых может выполнять не более 1 операции загрузки в каждом

цикле синхронизации, CPE не может быть меньше 0,50. В приложениях, загружающих  $k$  значений для каждого вычисляемого элемента, невозможно достичь CPE ниже  $k/2$  (см., например, упражнение 5.15).

В примерах, приводившихся до сих пор, мы не видели никакого влияния задержек загрузки на производительность. Адреса для операций загрузки в этих примерах зависят только от индекса цикла  $i$ , поэтому операции загрузки не являются частью критического пути, ограничивающего производительность.

Чтобы определить задержку, создаваемую операциями загрузки, можно организовать вычисления, включающие последовательность операций загрузки, в которых результат одной операции определяет адрес следующей. Рассмотрим функцию `list_len` в листинге 5.12, которая вычисляет длину связанного списка.

**Листинг 5.12.** Функция определения длины связанного списка. Ее производительность ограничена задержкой операции загрузки

```

1  typedef struct ELE {
2      struct ELE *next;
3      long data;
4  } list_ele, *list_ptr;
5
6  long list_len(list_ptr ls) {
7      long len = 0;
8      while (ls) {
9          len++;
10         ls = ls->next;
11     }
12     return len;
13 }
```

В цикле в этой функции каждое следующее значение переменной `ls` зависит от значения, полученного обращением к памяти по указателю `ls->next`. Как показали результаты измерений, функция `list_len` имеет величину CPE, равную 4,00, что, как можно утверждать, является прямым следствием задержки в операциях загрузки. Чтобы убедиться в этом, рассмотрим ассемблерный код цикла:

```

    Цикл в list_len
    ls в %rdi, len в %rax
1  .L3:                                loop:
2      addq    $1, %rax                Увеличить len
3      movq    (%rdi), %rdi           ls = ls->next
4      testq   %rdi, %rdi             Проверить ls
5      jne     .L3                    Если не null, то перейти к loop
```

Узким местом в этом цикле является инструкция `movq` в строке 3. Каждое следующее значение регистра `%rdi` зависит от результата операции загрузки с адресом в `%rdi`. То есть операция загрузки в следующей итерации не может начаться, пока не будет выполнена операция загрузки в предыдущей итерации. Величина CPE, равная 4,00 для этой функции, определяется задержкой операции загрузки. В действительности этот результат соответствует документированному времени доступа к кешу L1, занимающему 4 цикла для эталонной машины, как обсуждается в разделе 6.4.

### 5.12.2. Производительность операций сохранения

Во всех предыдущих примерах мы исследовали только функции, выполняющие преимущественно операции загрузки для чтения значений из памяти в регистры. Про-

тивоположная ей операция *сохранения* записывает значение регистра в память. Производительность этой операции, особенно в комплексе с операцией загрузки, имеет «щекотливые» моменты.

Так же как операция загрузки, в большинстве случаев операция сохранения может действовать в конвейерном режиме и сохранять одно значение в каждом цикле синхронизации. Например, рассмотрим функцию в листинге 5.13, обнуляющую элементы массива *dest* с длиной *n*. Измеренная величина CPE составила 1,0. Это лучший показатель, которого можно добиться от машины с единственным блоком сохранения.

**Листинг 5.13.** Функция, обнуляющая элементы массива. Этот код имеет величину CPE, равную 1,0.

```
1 /* Обнуляет элементы массива */
2 void clear_array(long *dest, long n) {
3     long i;
4     for (i = 0; i < n; i++)
5         dest[i] = 0;
6 }
```

В отличие от других операций, операция сохранения не влияет на значения регистров. То есть операции сохранения по своей природе не создают зависимостей по данным. И только операция загрузки может зависеть от результата операции сохранения, потому что только операция загрузки может читать обратно значение из памяти, записанное операцией сохранения.

Функция *write\_read* в листинге 5.14 иллюстрирует потенциальную зависимость между загрузкой и сохранением. А на рис. 5.20 показаны два примера выполнения этой функции, когда она вызывается для обработки двухэлементного массива *a* с начальными значениями –10 и 17 и с аргументом *cnt*, равным 3. Эти диаграммы иллюстрируют некоторые тонкости операций загрузки и сохранения.

**Листинг 5.14.** Функция, записывающая и читающая ячейки памяти. Эта функция подчеркивает зависимость между операциями сохранения и загрузки, когда аргументы *src* и *dst* равны

```
1 /* Пишет в dest, читает из src */
2 void write_read(long *src, long *dst, long n)
3 {
4     long cnt = n;
5     long val = 0;
6
7     while (cnt) {
8         *dst = val;
9         val = (*src)+1;
10        cnt--;
11    }
12 }
```

В примере А на рис. 5.20 аргумент *src* ссылается на элемент массива *a*[0], а аргумент *dst* – на элемент *a*[1]. В этом случае каждая операция загрузки значения из памяти по ссылке *\*src* будет возвращать значение –10. То есть после двух итераций состояние массива зафиксировано и будет хранить значения –10 и –9. Операция чтения из *src* не влияет на результат записи в *dst*. В этом примере измеренная величина CPT составила 1,3.

Пример A: `write_read(&a[0], &a[1], 3)`

	Начальное состояние	Итерация 1	Итерация 2	Итерация 3								
cnt	3	2	1	0								
a	<table><tr><td>-10</td><td>17</td></tr></table>	-10	17	<table><tr><td>-10</td><td>0</td></tr></table>	-10	0	<table><tr><td>-10</td><td>-9</td></tr></table>	-10	-9	<table><tr><td>-10</td><td>-9</td></tr></table>	-10	-9
-10	17											
-10	0											
-10	-9											
-10	-9											
val	0	-9	-9	-9								

Пример B: `write_read(&a[0], &a[0], 3)`

	Начальное состояние	Итерация 1	Итерация 2	Итерация 3								
cnt	3	2	1	0								
a	<table><tr><td>-10</td><td>17</td></tr></table>	-10	17	<table><tr><td>0</td><td>17</td></tr></table>	0	17	<table><tr><td>1</td><td>17</td></tr></table>	1	17	<table><tr><td>2</td><td>17</td></tr></table>	2	17
-10	17											
0	17											
1	17											
2	17											
val	0	1	2	3								

Рис. 5.20. Диаграммы выполнения функции из листинга 5.14

В примере B оба аргумента, `src` и `dest`, ссылаются на элемент `a[0]`. В этом случае каждая операция загрузки значения из памяти по ссылке `*src` будет возвращать значение, записанное предыдущей операцией сохранения по ссылке `*dest`.

Как следствие в этом элементе будут сохраняться все увеличивающиеся значения. В общем случае, если функцию `write_read` вызывать с аргументами `src` и `dest`, ссылающимися на одну и ту же ячейку памяти, и с аргументом `cnt`, имеющим некоторое значение  $n > 0$ , то в конечном итоге функция просто запишет в данную ячейку значение  $n - 1$ . Этот пример иллюстрирует явление, которое в книге называется *зависимостью между записью и чтением*: результат чтения из памяти зависит от последней операции записи в память. В этом примере измеренная величина CPT составила 7.3. Зависимость между записью и чтением замедляет обработку примерно на 6 циклов синхронизации.

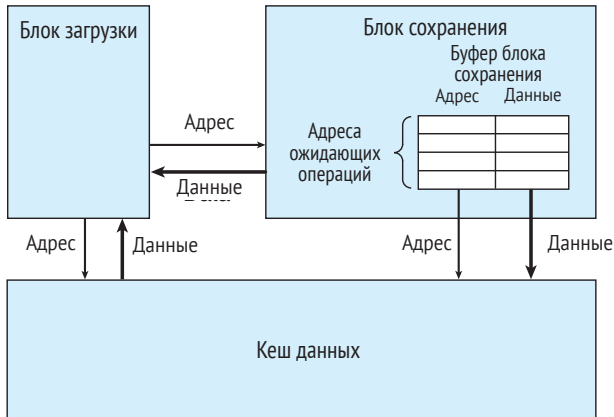
Чтобы понять, как процессор различает эти два случая и почему один работает медленнее другого, необходимо более детально рассмотреть работу блоков загрузки и сохранения, как показано на рис. 5.21. Блок сохранения имеет *буфер сохранения* с адресами и данными операций сохранения, переданными блоку сохранения, но еще не завершенных, потому что для их завершения необходимо обновить кеш данных. Этот буфер устроен так, что последовательность операций сохранения может выполняться без ожидания, пока каждая операция обновит кеш. Когда процессор встречает операцию загрузки, он должен проверить записи в буфере сохранения на предмет совпадения адресов. Если совпадение найдено (т. е. любой из записываемых байтов имеет тот же адрес, что и читаемый байт), процессор извлекает соответствующую запись из буфера и возвращает ее как результат операции загрузки.

GCC генерирует следующий код для цикла в функции `write_read`:

```

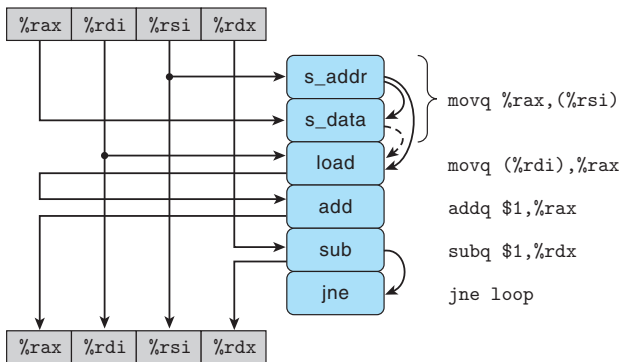
Цикл в write_read
src в %rdi, dst в %rsi, val в %rax
.L3:                                loop:
movq  %rax, (%rsi)                 Записать val в dst
movq  (%rdi), %rax                 t = *src
addq  $1, %rax                     val = t+1
subq  $1, %rdx                     cnt--
jne   .L3                          Если != 0, перейти к loop

```



**Рис. 5.21.** Схема работы блоков загрузки и сохранения. Блок сохранения поддерживает буфер ожидающих операций записи. Блок загрузки должен сверить свой адрес с адресами в буфере блока сохранения, чтобы обнаружить зависимость записи/чтения

На рис. 5.22 показан граф потока данных, представляющий этот код. Инструкция `movq %rax, (%rsi)` преобразуется в две операции: **s\_addr** вычисляет адрес для операции сохранения, создает запись в буфере сохранения и устанавливает поле адреса в этой записи. Операция **s\_data** устанавливает поле данных в записи. Как будет показано далее, тот факт, что эти два вычисления выполняются независимо, может иметь большое значение для производительности программы. Это объясняет разделение функциональных блоков для данных операций в эталонной машине.

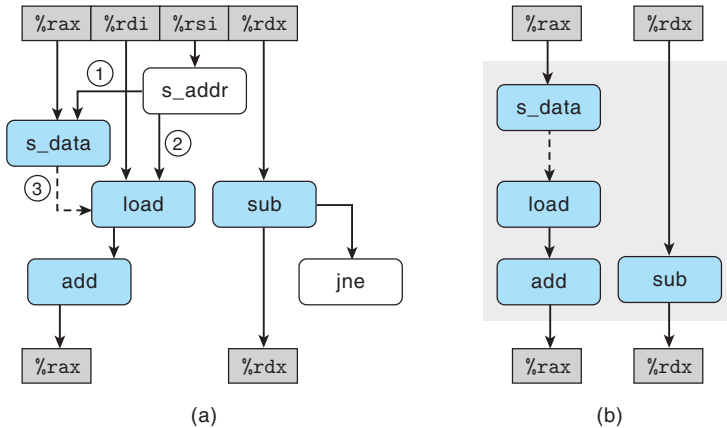


**Рис. 5.22.** Граф операций, выполняемых циклом в функции `write_read`. Первая инструкция `movl` декодируется в операции вычисления адреса хранилища и сохранения данных в памяти

Помимо зависимостей по данным между операциями записи и чтения регистров, дугообразные стрелки справа обозначают неявные зависимости между этими операциями. В частности, вычисление адреса операции **s\_addr** обязательно должно предшествовать операции **s\_data**. Кроме того, операция **load**, сгенерированная при декодировании инструкции `movq (%rdi), %rax`, должна проверять адреса любых ожидающих операций сохранения, что создает зависимость по данным между ней и операцией **s\_addr**. На рис. 5.22 операции **s\_data** и **load** соединяет пунктирная стрелка, обозначающая условную зависимость: если два адреса совпадают, операция **load** должна ждать, пока **s\_data**

не поместит свой результат в буфер сохранения, но если адреса различаются, то операции могут выполняться независимо.

На рис. 5.23 показаны зависимости по данным между операциями в цикле в функции `write_read`. Слева (рис. 5.23 (a)) операции были переупорядочены, чтобы сделать зависимости более явными. Здесь мы особо выделили три зависимости между операциями загрузки и сохранения. Стрелка с меткой «1» представляет требование, обязывающее вычислять адрес сохранения до записи данных. Стрелка с меткой «2» представляет требование, обязывающее операцию **load** сравнить свой адрес с адресами всех ожидающих операций сохранения. Наконец, пунктирная стрелка с меткой «3» представляет условную зависимость по данным, которая возникает при совпадении адресов загрузки и хранения.



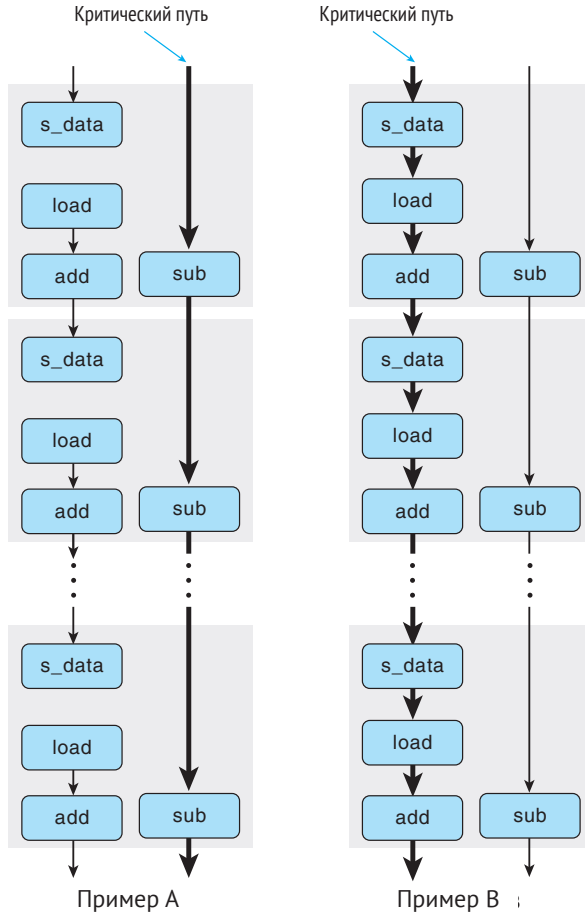
**Рис. 5.23.** Абстрагирование операций в функции `write_read`. Сначала мы переупорядочили операции, изображенные на рис. 5.23 (a), а затем оставили только те, которые используют значения в одной итерации, чтобы создать новые значения для следующей (b)

Справа (рис. 5.23 (b)) показан тот же граф, но без операций, которые не влияют напрямую на поток данных между итерациями. На этом графе видны только две цепочки зависимостей: одна слева, где значения сохраняются, загружаются и увеличиваются (только в случае совпадения адресов), и одна справа, где уменьшается переменная `spt`.

Теперь характеристики производительности функции `write_read` более очевидны. На рис. 5.24 показаны зависимости по данным между несколькими итерациями. В примере А на рис. 5.20, с разными адресами источника и приемника, операции загрузки и сохранения могут выполняться независимо, и, следовательно, единственный критический путь образует операция уменьшения переменной `spt`, накладывающая ограничение на величину CPE, равное 1,0. В примере В с совпадающими адресами источника и приемника зависимость по данным между операциями **s\_data** и **load** образует критический путь, включающий данные, которые хранятся, загружаются и увеличиваются. Согласно измерениям, проведенным на нашей эталонной машине, этим трем последовательным операциям требуется в общей сложности около 7 циклов синхронизации.

Как показано в этих двух примерах, реализация операций с памятью таит множество тонкостей. Выполняя операции с регистрами, процессор может определить, какие инструкции повлияют друг на друга на этапе декодирования. Однако, выполняя операции с памятью, процессор лишен такой возможности, пока не будут вычислены адреса загрузки и сохранения. Поскольку операции с памятью составляют значительную часть программы, подсистема памяти использует множество оптимизаций, в том числе и параллельное выполнение операций, не зависящих друг от друга.





**Рис. 5.24.** Граф потока данных в функции `write_read`. Когда два адреса не совпадают, образуется критический путь, обусловленный операцией уменьшения `cnt` (пример А).

Если адреса совпадают, образуется критический путь, обусловленный цепочкой операций сохранения, загрузки и увеличения (пример В)

#### Упражнение 5.10 (решение в конце главы)

Вот еще один пример кода с потенциальной зависимостью операций загрузки от операций – функция копирования одного массива в другой:

```

1 void copy_array(long *src, long *dest, long n)
2 {
3     long i;
4     for (i = 0; i < n; i++)
5         dest[i] = src[i];
6 }
```

Предположим, что `a` – массив с длиной 1000, инициализированный так, что каждый элемент `a[i]` равен `i`.

1. Что получится в результате вызова `soru_agray(a+1, a, 999)`?
2. Что получится в результате вызова `soru_agray(a, a+1, 999)`?
3. Измерения производительности показали, что первый вызов имеет величину CPE 1,2, а второй – величину CPE 5.0. Какие факторы обусловили такую разницу в производительности?
4. Какую производительность можно ожидать от вызова `soru_agray(a, a, 999)`?

#### Упражнение 5.11 (решение в конце главы)

Измерения производительности функции суммирования `psum1` (листинг 5.1) дали величину CPE 9.00 на машине, где основная операция – сложение с плавающей точкой – имеет задержку всего 3 цикла синхронизации. Давайте попробуем понять, почему функция имеет худшую производительность.

Вот ассемблерный код цикла внутри функции:

Цикл в <code>psum1</code>		
	<code>a в %rdi, i в %rax, cnt в %rdx</code>	
1	<code>.L5:</code>	<code>loop:</code>
2	<code>vmovss -4(%rsi,%rax,4), %xmm0</code>	Получить <code>p[i-1]</code>
3	<code>vaddss (%rdi,%rax,4), %xmm0, %xmm0</code>	Прибавить <code>a[i]</code>
4	<code>vmovss %xmm0, (%rsi,%rax,4)</code>	Сохранить в <code>p[i]</code>
5	<code>addq \$1, %rax</code>	Увеличить <code>i</code>
6	<code>cmpq %rdx, %rax</code>	Сравнить <code>i:cnt</code>
7	<code>jne .L5</code>	Если <code>!=</code> , перейти к <code>loop</code>

Проанализируйте этот код, подобно тому, как мы делали это с функциями `combine3` (рис. 5.6) и `write_read` (рис. 5.23), нарисуйте граф зависимостей по данным, создаваемых этим циклом, и определите критический путь. Объясните, почему CPE имеет такую большую величину.

#### Упражнение 5.12 (решение в конце главы)

Перепишите функцию `psum1` (листинг 5.1) так, чтобы ей не приходилось многократно извлекать значение `p[i]` из памяти. Для этого не требуется использовать разворачивание цикла. Мы написали такой код и, измерив его производительность, получили величину CPE 3.00, обусловленную задержкой операции сложения чисел с плавающей точкой.

## 5.13. Жизнь в реальном мире: методы повышения производительности

Выше мы рассмотрели довольно ограниченный набор примеров и, тем не менее, можем сделать важные выводы о том, как писать эффективный код. В ходе обсуждения было представлено несколько базовых стратегий оптимизации производительности программ:

*Высокоуровневое проектирование.*

Выбирайте алгоритмы и структуры данных, подходящие для решаемой задачи. Особо следите, чтобы в число выбранных не попали алгоритмы, имеющие асимптотическую низкую производительность.

*Базовые принципы программирования.*

Избегайте приемов, препятствующих оптимизации, чтобы компилятор имел возможность сгенерировать эффективный код:

- устраняйте чрезмерные обращения к функциям. По возможности, выносите вычисления за пределы циклов. Для повышения эффективности рассматривайте различные варианты компоновки программы;
- устраняйте ненужные ссылки на память. Используйте временные переменные для хранения промежуточных результатов. Сохраняйте в массиве или в глобальной переменной только окончательный результат вычислений.

*Низкоуровневая оптимизация:*

Структурируйте код так, чтобы извлечь максимальную выгоду из возможностей оборудования:

- разворачивайте циклы, чтобы уменьшить накладные расходы и открыть возможность применения других оптимизаций;
- ищите способы повысить степень параллелизма на уровне инструкций, используя такие приемы, как накопление результата в нескольких аккумуляторах и переупорядочение операций;
- записывайте условные операции в функциональном стиле, чтобы позволить компилятору использовать инструкции условного перемещения данных.

И последний совет: избегайте ошибок, когда будете переписывать программы, чтобы добиться большей производительности. Очень легко допустить ошибку, добавляя новые переменные, изменяя границы цикла и усложняя код в целом. Обязательно тестируйте производительность каждой версии функции по мере ее оптимизации, чтобы исключить просачивание ошибок во время этого процесса. Набор надежных тестов поможет проверить новые версии функции и гарантирует, что они будут возвращать те же результаты. Набор тестов должен быть шире для высокооптимизированного кода, потому что есть множество нетривиальных случаев, которые важно учитывать. Например, тест для кода, использующего разворачивание цикла, должен проверить множества различных границ цикла, чтобы убедиться, что в конце обрабатывается правильное количество завершающих итераций.

## 5.14. Выявление и устранение узких мест производительности

До настоящего момента мы рассматривали оптимизацию только небольших программ, в которых имелись места, явно требующие оптимизации. При работе с крупными программами затруднение может вызвать даже сам поиск места, на котором следует сосредоточить усилия по оптимизации. В этом разделе мы поговорим об использовании *профилировщиков* – инструментов анализа, собирающих данные о производительности программы в процессе ее выполнения, – а также обсудим общие принципы оптимизации, включая следствия известного закона Амдала (раздел 1.9.1).

### 5.14.1. Профилирование программ

*Профилирование* программы подразумевает запуск версии программы, в которую встроен специальный код, определяющий количество времени, затрачиваемое на выполнение разными частями программы. Этот метод помогает выявлять части программы, на которые следует обратить внимание с точки зрения оптимизации. Одной из

сильных сторон профилирования является возможность оценить производительность «настоящей» программы, обрабатывающей вполне реальные исходные данные.

В системах Unix имеется программа-профилировщик GPROF. Она генерирует два вида информации. Во-первых, определяет количество процессорного времени, потраченного на выполнение каждой функции. Во-вторых, вычисляет количество вызовов каждой функции, группируя их по точкам, откуда были сделаны вызовы. И те, и другие сведения могут быть весьма полезными. Замеры времени помогают определить относительный вклад различных функций в общее время выполнения. Информация о количестве вызовов позволяет понять динамику поведения программы.

Профилирование с GPROF выполняется в три этапа, как показано на примере программы `prog.c`, запускаемой с аргументом командной строки `file.txt`.

1. Программа должна быть скомпилирована и скомпонована для профилирования. С GCC (и другими компиляторами языка C) для этого достаточно выполнить компиляцию с параметром командной строки `-pg`:

```
linux> gcc -Og -pg prog.c -o prog
```

2. Следующий этап: запуск программы как обычно:

```
linux> ./prog file.txt
```

Программа выполняется немного (примерно в 2 раза) медленнее, чем обычно, в остальном единственное отличие заключается в том, что она генерирует файл `gmon.out`.

3. Для анализа данных в `gmon.out` используется профилировщик GPROF:

```
linux> gprof prog
```

В первой части отчета профилировщика приводятся значения времени, потраченного на выполнение различных функций, в порядке убывания. Например, в следующем листинге представлены первые три записи из этой части отчета:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
97.58	203.66	203.66	1	203.66	203.66	sort_words
2.32	208.50	4.85	965027	0.00	0.00	find_ele_rec
0.14	208.81	0.30	12511031	0.00	0.00	Strlen

В каждой строке приводится суммарное время выполнения определенной функции. В первом столбце дается доля всего времени в процентах, затраченного на выполнение функции. Во втором – накопленное время, затраченное всеми функциями, до включения этой функции в отчет в эту строку. В третьем столбце – общее время выполнения этой функции, а в четвертом – количество вызовов функции (не считая рекурсивных вызовов). В нашем примере функция `sort_words` вызывалась только один раз, но этот вызов длился 203,66 секунды, тогда как функция `find_ele_rec` была вызвана 965 027 раз и общее время ее выполнения составило 4,85 секунды. Функция `Strlen` в этой программе вычисляет длину строки, вызывая библиотечную функцию `strlen`. Вызовы библиотечных функций обычно не отображаются в отчетах GPROF. Их время чаще включается во время вызывающих их функций. Создав «функцию-обертку» `Strlen`, мы смогли отследить вызовы `strlen`: она была вызвана 12 511 031 раз, но на все эти вызовы было потрачено всего 0,30 секунды.

Во второй части отчета профилировщика показана история вызовов функций. Например, вот как выглядит история вызовов `find_ele_rec`:

```

4.85    0.10  158655725  find_ele_rec [5]
          965027/965027  insert_string [4]
```

[5]	2.4	4.85	0.10	965027+158655725	find_ele_rec [5]
		0.08	0.01	363039/363039	save_string [8]
		0.00	0.01	363039/363039	new_ele [12]
				158655725	find_ele_rec [5]

История вызовов показывает, какие функции вызывали `find_ele_rec` и какие функции вызывала она сама. Первые две строки сообщают, что функция рекурсивно вызывала сама себя 158 655 725 раз и 965 027 раз ее вызывала функция `insert_string` (которая сама вызывалась 965 027 раз). В свою очередь, функция `find_ele_rec` вызывала две другие функции: `save_string` и `new_ele`, по 363 039 раз каждую.

Из этой информации о вызовах часто можно извлечь полезную информацию о поведении программы. Например, функция `find_ele_rec` – это рекурсивная процедура, просматривающая связный список в поисках конкретной строки. Сравнивая количество рекурсивных вызовов с количеством вызовов функции из других функций, можно оценить длину просматриваемых списков. В данном случае эти числа относятся как 164,4:1, откуда следует, что в среднем программа просматривала 164 элемента списка, прежде чем находила искомое.

Вот некоторые важные свойства GPROF:

- замеры времени не отличаются высокой точностью. Они основаны на простой схеме *подсчета интервалов*, когда при компиляции в каждую функцию программы встраивается счетчик, хранящий время, потраченное на выполнение этой функции. Операционная система прерывает выполнение через некоторые регулярные интервалы времени  $\delta$ , обычно от 1,0 до 10,0 мс. Определяет, какую функцию выполняла программа в этот момент, и увеличивает счетчик этой функции на  $\delta$ . Конечно, может случиться так, что функция только начала выполняться и очень скоро будет завершена, однако на ее счет будет записано время, прошедшее с момента последнего прерывания. Какая-то другая функция может успеть выполниться между двумя прерываниями, но профилировщик не заметит этого. Для функций, выполняющихся длительное время, эта схема дает вполне правдоподобные значения. Статистически каждой функции должно быть присвоено значение, согласующееся с фактическим временем ее выполнения. Но вообще для программ, выполняющихся меньше одной секунды, эти оценки следует рассматривать как весьма приблизительные;
- информация о количестве вызовов вполне надежна, если компилятор не применил оптимизацию встраиванием тела функции в точку вызова. Скомпилированная программа поддерживает счетчик для каждой пары вызывающей и вызываемой функций, который увеличивается при каждом соответствующем вызове;
- по умолчанию время выполнения библиотечных функций не показывается. Время их выполнения прибавляется ко времени выполнения вызывающей функции.

### 5.14.2. Использование профилировщика при выборе кода для оптимизации

Чтобы показать, как пользоваться информацией, предоставляемой профилировщиком при выборе кода для оптимизации, мы написали программу-приложение, решающую несколько задач с использованием нескольких структур данных. Эта программа читает текстовый файл и подсчитывает количество *n-грамм* – последовательностей из  $n$  слов, – встречающихся в файле. Для  $n = 1$  подсчитываются отдельные слова, для  $n = 2$  – пары слов и т. д. Для заданного значения  $n$  программа создает таблицу уникальных  $n$ -грамм и затем сортирует ее в порядке убывания количества вхождений.

Для оценки программе был передан файл с полным собранием сочинений Шекспира, содержащий 965 028 слов, 23 706 из которых уникальны. Мы выяснили, что для  $n = 1$  даже программа, написанная на скорую руку, способна обработать весь файл менее чем за 1 с, поэтому мы установили  $n = 2$ , чтобы усложнить задачу. В случае  $n = 2$   $n$ -граммы называются *биграммами*. Мы определили, что собрание сочинений Шекспира содержит 363 039 уникальных биграмм. Чаше всего встречается биграмма «I am» – 1892 раза. Самая, пожалуй, известная биграмма «to be» встречается 1020 раз. 266 018 биграмм встречаются только один раз.

Ниже перечислены части, из которых состоит программа. Для каждой части мы создали несколько версий, начав с простых алгоритмов, и затем заменяли их более сложными:

1. Каждое слово, прочтенное из файла, преобразуется в нижний регистр. Начальная версия этой части использовала функцию `lower1` (листинг 5.5), которая, как известно, имеет квадратичную сложность из-за многократного вызова `strlen`.
2. Для создания числа в диапазоне от 0 до  $s - 1$  для хеш-таблицы с  $s$  разделами к строке применялась хеш-функция. Начальная версия функции просто суммировала ASCII-коды символов по модулю  $s$ .
3. Каждый раздел хеш-таблицы организован в виде связанного списка. В процессе поиска программа просматривает соответствующий список и при обнаружении совпадения увеличивает количество встреченных вхождений искомого слова. Если поиск не увенчался успехом, то в список добавляется новый элемент. Начальная версия выполняла эту операцию рекурсивно, вставляя новые элементы в конец списка.
4. После создания таблицы программа выполняет сортировку элементов таблицы по количеству вхождений. Начальная версия использовала сортировку методом вставки.

На рис. 5.25 показаны результаты профилирования разных версий программы анализа  $n$ -грамм. Для каждой версии мы разделили время на следующие категории:

- **Сортировка.** Сортировка  $n$ -грамм по количеству вхождений.
- **Поиск.** Поиск совпадения в связанном списке со вставкой нового элемента при необходимости.
- **Преобразование.** Преобразование строки в нижний регистр.
- **Длина.** Вычисление длины строки.
- **Хеширование.** Вычисление хеш-функции.
- **Остальное.** Сумма времени, затраченного на выполнение всех остальных функций.

Как показано на рис. 5.25 (а), начальная версия выполнялась в течение 3,5 мин, причем большая часть этого времени была потрачена на сортировку. И это не удивительно, потому что сортировка методом вставки имеет квадратичную сложность, а программа сортирует 363 039 значений.

В следующей версии сортировка выполнялась с помощью библиотечной функции `qsort`, базирующейся на алгоритме быстрой сортировки [98]. Ожидаемое время работы этого алгоритма составляет  $O(n \log n)$ . На схеме эта версия обозначена как «Быстрая сортировка». Применение более эффективного алгоритма сортировки уменьшило время, затрачиваемое на сортировку, до ничтожной величины, а общее время выполнения – до 5,4 с. На рис. 5.25 (б) показано время для остальных версий программы в масштабе, позволяющем четко видеть распределение времени между частями программы.

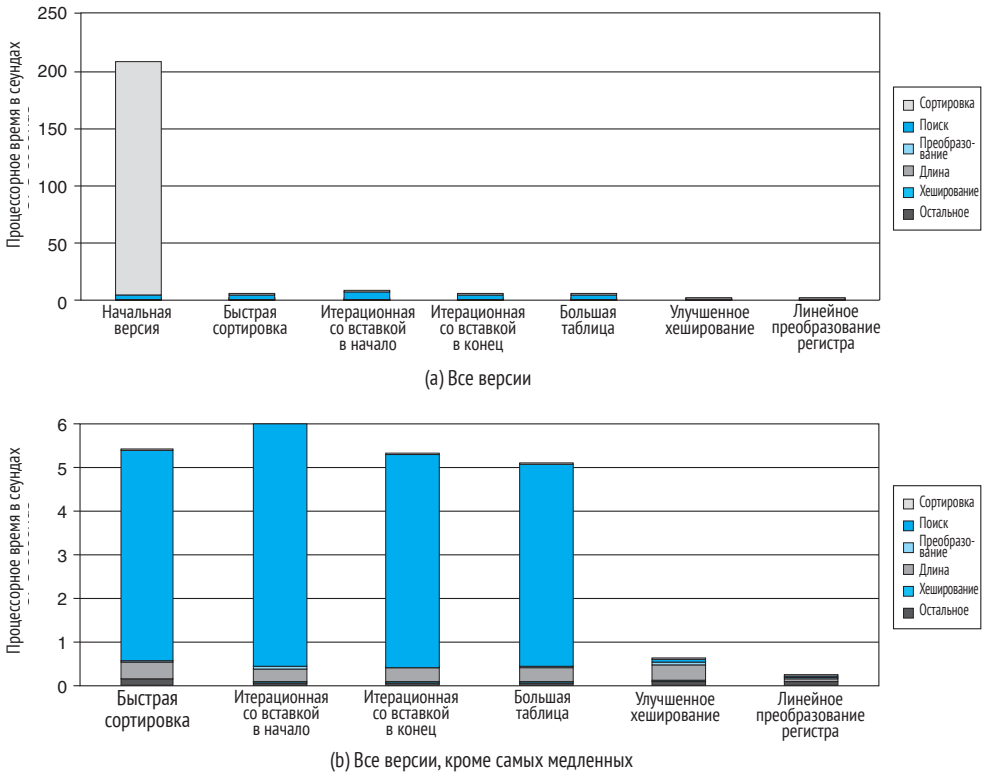


Рис. 5.25. Результаты профилирования различных версий программы подсчета биграмм.

Время распределяется в соответствии с различными основными операциями в программе

После применения более совершенного алгоритма наиболее затратной частью становится поиск в списке. Предположив, что неэффективность связана с рекурсивной природой функции, мы заменили рекурсивный алгоритм итерационным. Эта версия подписана на схеме как «Итерационная со вставкой в начало». Удивительно, но время выполнения увеличилось до 7,5 с. При более скрупулезном изучении мы обнаружили небольшое расхождение между двумя функциями поиска в списке. Рекурсивная версия добавляла новые элементы в конец списка, а итерационная – в начало. Для увеличения производительности необходимо, чтобы часто встречающиеся  $n$ -граммы вставлялись ближе к началу списка. В этом случае функция будет быстрее находить часто встречающиеся  $n$ -граммы. Если предположить, что  $n$ -граммы распределены в документе равномерно, то можно ожидать, что первое вхождение более частой  $n$ -граммы встретится раньше, чем менее частой. Вставкой новых  $n$ -грамм в конец списка первая функция стремилась располагать слова в порядке убывания их частоты, тогда как вторая функция – наоборот. Поэтому мы создали третью функцию, просматривающую список итерационным способом, но вставляющую новые элементы в конец. С этой версией время сократилось до 5,3 с, что немного лучше по сравнению с рекурсивной версией. Это лишний раз доказывает важность экспериментов с программой в процессе оптимизации. Мы изначально предполагали, что преобразование рекурсивного кода в итеративный улучшит его производительность, но не учли различия между добавлением в конец или в начало списка.

Теперь перейдем к хеш-функции. Начальная версия имела 1021 раздел (обычно количество разделов выбирается равным простому числу, чтобы обеспечить более рав-



номерное распределение ключей по разделам). Для таблицы с 363 039 в каждый раздел в среднем попадало  $363,039 / 1,021 = 355,6$   $n$ -граммы. Это объясняет, почему так много времени тратится на поиск в списке: в процессе поиска выполнялось сравнение кандидата со значительным количеством  $n$ -грамм. Это также объясняет повышенную чувствительность выполнения к упорядочиванию списка. Мы увеличили количество разделов до 199 999, что уменьшило среднее количество  $n$ -грамм в разделах до 1,8. Однако общее время выполнения уменьшилось всего на 0,2 с – до 5,1 с.

Продолжив исследование, мы заметили, что незначительность прироста производительности после увеличения размера таблицы связана с плохим выбором хеш-функции. Простое суммирование ASCII-кодов символов не дает достаточно широкого диапазона значений. В частности, максимальный код буквы имеет значение 122, поэтому строка из  $n$  символов будет генерировать сумму не более  $122n$ . Самая длинная биграмма в нашем документе, «honorificabilitudinitatibus thou», дает сумму всего 3371, поэтому большинство разделов в нашей хеш-таблице просто пусты. Кроме того, коммутативная хеш-функция, такая как сложение, не делает различий между  $n$ -граммами с разным порядком символов. Например, слова «rat» и «tar» будут давать одинаковые суммы.

Мы решили выбрать хеш-функцию, использующую операции сдвига и ИСКЛЮЧАЮЩЕЕ ИЛИ. В этой версии, обозначенной на схеме как «Улучшенное хеширование», время сократилось до 0,6 с. Если заняться этим вопросом всерьез, то стоило бы более тщательно изучить распределение ключей между разделами и убедиться, что оно приближается к равномерному.

Наконец, время выполнения сократилось до точки, когда большая часть времени тратится на выполнение `strlen`, и большинство вызовов этой функции производится в функции преобразования в нижний регистр. Мы уже видели, что функция `lower1` имеет квадратичную производительность, особенно для длинных строк. Слова в этом документе достаточно короткие, чтобы квадратичная производительность могла иметь катастрофические последствия; самая длинная биграмма состоит всего из 32 символов. Тем не менее переключение на использование `lower2` (на схеме эта версия обозначена как «Линейное преобразование регистра») дало значительное улучшение, а общее время уменьшилось примерно до 0,2 с.

В этом упражнении мы хотели показать, как профилирование помогает сократить время выполнения простой программы с 3,5 мин до 0,2 с – в 1000 раз! Профилировщик помогает сосредоточить внимание на наиболее затратных по времени частях программы, а также предоставляет полезную информацию о структуре вызовов процедур. Некоторые из узких мест в нашем коде, такие как использование процедуры сортировки с квадратичной сложностью, легко заметить, однако другие, такие как добавление элементов в начало или в конец списка, обнаруживаются только после тщательного анализа.

Профилирование – полезный инструмент, достойный для включения в арсенал каждого программиста, однако не следует забывать и о других инструментах. Приемы измерения времени выполнения несовершенны, особенно для коротких интервалов (меньше одной секунды). Результаты измерений применимы только к конкретным данным, использовавшимся для тестирования.

Например, если бы мы опробовали начальную версию на данных, содержащих меньше длинных строк, то обнаружили бы, что основным узким местом является процедура преобразования в нижний регистр. Хуже того, при профилировании на документах, содержащих только короткие слова, мы вовсе не обнаружили скрытые узкие места, такие как квадратичная производительность `lower1`. Вообще говоря, профилирование может помочь в оптимизации *типичных* случаев, предполагая, что программа тестируется с использованием репрезентативных данных, но мы также должны убедиться, что программа будет иметь достойную производительность во всех возможных случаях. В основном это включает отказ от алгоритмов (таких как сортировка вставкой) и методов



программирования (например, `lower1`), имеющих низкую асимптотическую производительность.

Закон Амдала, описанный в разделе 1.9.1, помогает оценить, какой дополнительный прирост производительности можно получить с помощью целевой оптимизации. Замена сортировки вставкой в нашей программе подсчета  $n$ -грамм быстрой сортировкой помогла, как мы видели, уменьшить общее время выполнения с 209,0 до 5,4 с. Первоначальная версия тратила 203,7 с из 209,0 на выполнение сортировки вставкой, что дает  $\alpha = 0,974$  – долю времени, на которую можно повлиять. Применение алгоритма быстрой сортировки сократило время, расходуемое на сортировку, оно стало незначительным, дав прогнозируемое ускорение  $209/\alpha = 39,0$  – довольно близко к измеренному ускорению 38,5. Нам удалось добиться столь существенного ускорения, потому что сортировка составляла очень большую долю от общего времени выполнения. Однако после устранения одного узкого места возникло новое, и поэтому для получения дополнительного ускорения необходимо сосредоточить внимание на других частях программы.

## 5.15. Итоги

Во многих книгах, посвященных обсуждению вопросов оптимизации, отмечается способность компиляторов генерировать эффективный код, однако очень многое зависит от программиста, которому надлежит оказывать компилятору всяческую «помощь». Никакой компилятор не заменит неэффективный алгоритм или структуру данных на более качественные, поэтому эти аспекты программы должны всегда находиться в поле пристального внимания разработчиков. Мы также видели, что некоторые факторы, препятствующие оптимизации, такие как наложение указателей и обращения к процедурам, серьезно ограничивают способность компиляторов к применению расширенных видов оптимизации. И снова основная ответственность за их устранение лежит на плечах программиста. Использование этих приемов должно стать частью повседневной практики программирования, потому что они помогают устранять ненужную работу.

Настройка производительности сверх базового уровня требует некоторого понимания микроархитектуры процессора и знания основных механизмов, с помощью которых процессор реализует свою архитектуру набора команд. Простое знание, что процессор может выполнять операции не по порядку, величин задержек и пропускной способности функциональных блоков дает прочный фундамент для прогнозирования производительности программы.

В этой главе мы изучили несколько приемов – развертывание циклов, выполнение итераций с несколькими аккумуляторами и переупорядочение операций, – которые позволяют использовать параллелизм на уровне инструкций, поддерживаемый современными процессорами. По мере углубления в изучение приемов оптимизации особую важность приобретают изучение сгенерированного ассемблерного кода и попытки понять, как машина выполняет вычисления. Многого можно добиться, выявив в программе критические пути, определяемые зависимостями по данным, особенно между итерациями цикла. Также можно вычислить границу пропускной способности на основе количества операций, которые должны быть выполнены, и количества блоков, выполняющих эти операции.

Программы, включающие условные ветвления или комплексные взаимодействия с системой памяти, сложнее для анализа и оптимизации, чем простые программы с циклами, которые мы рассмотрели. Основная стратегия их оптимизации заключается в том, чтобы сделать ветвления предсказуемыми или дать компилятору возможность использовать инструкции условного перемещения данных. Также необходимо уделить внимание зависимостям между операциями сохранения и загрузки. Храните промежуточные значения в регистрах, это поможет компилятору использовать регистры.

При работе с крупными программами особую важность приобретает сосредоточение усилий на частях системы, выполнение которых занимает большую часть времени. Профилировщики и аналогичные инструменты помогут вам последовательно оценить и улучшить производительность программы. В этой главе мы описали GPROF – стандартный инструмент профилирования в Unix. Кроме него доступны другие, более совершенные профилировщики, такие как система разработки программ VTUNE от Intel и VALGRIND, широко доступный в Linux. Эти инструменты могут разбить время выполнения до уровня отдельных процедур и помочь оценить производительность *элементарных блоков* программы. (Элементарный блок – это последовательность инструкций, которая всегда выполняется целиком и не имеет условных переходов в середине.)

## Библиографические заметки

В этой главе основное наше внимание было сосредоточено на описании оптимизации кода с точки зрения программиста, демонстрации приемов, способных упростить для компиляторов генерирование эффективного кода. В статье Челлаппы (Chellappa), Франчетти (Franchetti) и Пушеля (Püschel) [19] описывается аналогичный подход, но более подробно рассматриваются характеристики процессора.

Существует множество публикаций, описывающих оптимизацию кода с точки зрения компилятора, где разъясняются приемы, при применении которых компиляторы могут генерировать более эффективный код. Наиболее полной в этом отношении считается книга Мучника (Muchnick) [80]. Книга Уэдли (Wadleigh) и Кроуфорда (Crawford) по оптимизации программного обеспечения [115] включает описание некоторых приемов, представленных нами, но также охватывает процесс достижения высокой производительности на параллельных машинах. Ранняя статья Малке (Mahlke) и др. [75] описывает, как методы, разработанные для компиляторов, отображающих программы на параллельные машины, можно адаптировать для использования параллелизма на уровне инструкций, поддерживаемого современными процессорами. В этой статье рассматриваются также представленные нами преобразования кода, в том числе развертывание цикла, использование нескольких аккумуляторов (которое они называют *расширением набора переменных-аккумуляторов*) и переупорядочение операций (которое они называют *уменьшением высоты дерева*).

Мы довольно кратко и абстрактно описали способность процессора выполнять операции не по порядку. Более полное описание можно найти в руководствах по продвинутой компьютерной архитектуре, например в книге Хеннесси (Hennessy) и Паттерсона (Patterson) [46, гл. 2–3]. В книге Шена (Shen) и Липасты (Lipasti) [100] подробно рассматривается устройство современного процессора.

## Домашние задания

### Упражнение 5.13 ♦♦

Предположим, что нам понадобилось написать процедуру, вычисляющую скалярное произведение двух векторов, *u* и *v*. Абстрактная версия этой функции имеет CPE 14–18 на архитектуре x86-64 как для целочисленных данных, так и для данных с плавающей точкой. Выполнив те же преобразования, которые применялись к абстрактной функции `combine1`, чтобы получить более эффективную версию `combine4`, получаем следующий код:

```

1 /* Скалярное произведение. Накопление во временной переменной */
2 void inner4(vec_ptr u, vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(u);
6     data_t *udata = get_vec_start(u);
```

```

7   data_t *vdata = get_vec_start(v);
8   data_t sum = (data_t) 0;
9
10  for (i = 0; i < length; i++) {
11      sum = sum + udata[i] * vdata[i];
12  }
13  *dest = sum;
14 }

```

Измерив производительность этой версии, мы получили значение CPI 1,50 для целочисленных данных и 3,00 для данных с плавающей точкой. Вот как выглядит ассемблерный код цикла в версии, обрабатывающей данные типа double:

```

Цикл в inner4. data_t = double, OP = *
udata в %rbp, vdata в %rax, sum в %xmm0
i в %rcx, limit в %rbx
1 .L15:                                loop:
2 vmovsd 0(%rbp,%rcx,8), %xmm1        Получить udata[i]
3 vmulsd (%rax,%rcx,8), %xmm1, %xmm1  Умножить на vdata[i]
4 vaddsd %xmm1, %xmm0, %xmm0         Прибавить к sum
5 addq $1, %rcx                     Увеличить i
6 cmpq %rbx, %rcx                   Сравнить i:limit
7 jne .L15                          Если !=, перейти к loop

```

Допустим, что функциональные блоки имеют характеристики, перечисленные в табл. 5.2.

1. Нарисуйте диаграмму декодирования этой последовательности инструкций в операции и покажите, как зависимости по данным между ними создают критический путь, подобно тому, как было показано на рис. 5.5 и 5.6.
2. Какая нижняя граница CPE определяется критическим путем для типа данных double?
3. Предположив, что для целочисленной версии генерируется та же последовательность инструкций, определите нижнюю границу CPE, которая определяется критическим путем.
4. Объясните, почему версия для данных с плавающей точкой может иметь величину CPE, равную 3,00, несмотря на то что для операции умножения требуется 5 циклов синхронизации.

### Упражнение 5.14 ♦

Напишите версию процедуры вычисления скалярного произведения, описанной в упражнении 5.13, в которой применяется развертывание цикла  $6 \times 1$ . На нашей эталонной машине с архитектурой x86-64 эта версия показала величину CPE 1,07 с целочисленными данными, но в варианте с данными с плавающей точкой она осталась на уровне 3,01.

1. Объясните, почему на процессоре Intel Core i7 Haswell любая (не векторная) версия процедуры вычисления скалярного произведения не способна показать величину CPE меньше 1,00.
2. Объясните, почему развертывание цикла не повышает производительность для данных с плавающей точкой.

### Упражнение 5.15 ♦

Напишите версию процедуры вычисления скалярного произведения векторов, описанной в упражнении 5.13, в которой применяется развертывание цикла  $6 \times 6$ . На нашей эталонной машине с архитектурой x86-64 эта версия показала величину CPE 1,06 с це-

лочисленными данными и 1,01 с данными с плавающей точкой. Какой фактор ограничивает CPE величиной 1,00?

### Упражнение 5.16 ♦

Напишите версию процедуры вычисления скалярного произведения векторов, описанной в упражнении 5.13, в которой применяется развертывание цикла  $6 \times 1a$ . На нашей эталонной машине с архитектурой x86-64 эта версия показала величину CPE 1,10 с целочисленными данными и 1,05 с данными с плавающей точкой.

### Упражнение 5.17 ♦

Библиотечная функция `memset` имеет следующий прототип:

```
void *memset(void *s, int c, size_t n);
```

Эта функция заполняет  $n$  байт памяти, начиная с  $s$ , копиями младшего байта в  $c$ . Ее можно использовать, например, для заполнения области памяти нулевыми байтами, передав в  $c$  значение 0, но можно передавать и другие значения.

Вот простая реализация `memset`:

```
1 /* Простая реализация memset */
2 void *basic_memset(void *s, int c, size_t n)
3 {
4     size_t cnt = 0;
5     unsigned char *schar = s;
6     while (cnt < n) {
7         *schar++ = (unsigned char) c;
8         cnt++;
9     }
10    return s;
11 }
```

Реализуйте более эффективную версию функции, используя переменную типа `unsigned long` для упаковки восьми копий  $c$ , которая заполняет область памяти 8-байтными словами. Возможно, вам пригодится прием развертывания цикла. На нашей эталонной машине мы смогли уменьшить величину CPE с 1,00 для простой реализации до 0,127. То есть программа может записывать 8 байт в каждом цикле синхронизации.

Вот еще несколько требований (для переносимости пусть  $K$  обозначает значение `sizeof(unsigned long)` для машины, на которой запускается программа):

- вам запрещается вызывать какие-либо библиотечные функции;
- ваш код должен работать с произвольными значениями  $n$ , в том числе не кратными  $K$ . Вы можете добавить несколько заключительных итераций, как это делали мы в наших примерах развертывания цикла;
- код должен компилироваться и работать правильно на любой машине, независимо от значения  $K$ . Используйте для этого операцию `sizeof`;
- на некоторых машинах запись по адресам, не кратным размеру машинного слова, может выполняться намного медленнее, чем по кратным адресам. (Хуже того, на некоторых архитектурах, отличных от x86, запись целых слов по не кратным адресам может даже вызвать ошибку сегментации.) Напишите код так, чтобы он сначала выполнял запись байтов по одному, пока адрес назначения не станет кратным  $K$ , затем выполнял запись словами, а потом (при необходимости) заканчивал запись байтами по одному;
- будьте внимательны со случаями, когда значение `cnt` слишком мало и верхние границы некоторых циклов могут стать отрицательными. Выражения, включающие оператор `sizeof`, могут выполнять проверку с использованием беззнаковой арифметики (см. раздел 2.2.8 и упражнение 2.72).

### Упражнение 5.18 ♦♦♦

В упражнениях 5.5 и 5.6 мы рассмотрели задачу вычисления многочлена как методом прямого вычисления, так и методом Горнера. Попробуйте написать более быстрые версии функции, используя изученные методы оптимизации, включая развертывание цикла, использование нескольких аккумуляторов и переупорядочение операций. С помощью этих методов оптимизации вы найдете множество различных способов смешивания схемы Горнера и прямого вычисления.

В идеале вы должны достичь величины CPE, близкой к пределу пропускной способности вашей машины. Наша лучшая версия на нашей эталонной машине достигла CPE 1,07.

### Упражнение 5.19 ♦♦♦

В упражнении 5.12 мы смогли уменьшить CPE реализации суммирования элементов вектора до 3,00, достигнув предела задержки операции сложения чисел с плавающей точкой. Простое развертывание цикла не улучшает ситуацию.

Попробуйте объединить развертывание цикла и переупорядочение операций и напишите код с величиной CPE меньше, чем задержка сложения чисел с плавающей точкой на вашем компьютере. Для этого потребуется увеличить количество выполняемых сложений. Например, наша версия с развертыванием цикла с коэффициентом 2 выполняет три сложения в каждой итерации, а версия с развертыванием с коэффициентом 4 – пять. Наша лучшая реализация показала величину CPE 1,67 на эталонной машине.

Определите, как пределы пропускной способности и задержки на вашей машине ограничивают величину CPE, которую можно достичь в реализации суммирования элементов вектора.

## Решения упражнений

### Решение упражнения 5.1

Это упражнение иллюстрирует некоторые эффекты, возникающие при наложении указателей.

Результатом выполнения приведенного кода будет обнуление `xr`:

```
4  *xr = *xr + *xr; /* 2x */
5  *xr = *xr - *xr; /* 2x-2x = 0 */
6  *xr = *xr - *xr; /* 0-0 = 0 */
```

Этот пример наглядно показывает, что поведение программы может отличаться от предполагаемого нами. Мы обычно считаем, что `xr` и `ur` имеют разные значения, упуская из виду случай, когда они могут быть равны. Ошибки часто возникают из-за возникновения условий, о возможности которых программист даже не догадывается.

### Решение упражнения 5.2

Это упражнение иллюстрирует связь между CPE и абсолютной производительностью. Оно решается с помощью элементарной алгебры. Версия 1 оказывается самой быстрой для случая  $n \leq 2$ . Версия 2 – самая быстрая для случая  $3 \leq n \leq 7$ , а версия 3 – для случая  $n \geq 8$ .

### Решение упражнения 5.3

Это простое упражнение, но позволяет понять, что четыре элемента цикла `for` – инициализация, проверка условия, обновление переменной цикла и тело – выполняются разное количество раз.

Фрагмент кода	min	max	incr	square
1	1	91	90	90
2	91	1	90	90
3	1	1	90	90

## Решение упражнения 5.4

Этот ассемблерный код демонстрирует интересную оптимизацию, обнаруженную в GCC. Постарайтесь внимательно изучить этот код, чтобы понять тонкость оптимизации кода.

1. В менее оптимизированном коде регистр `%xmm0` используется как временное значение, которое изменяется и используется в каждой итерации цикла. В более оптимизированном коде он используется, скорее, как переменная `acc` в `combine4`, накапливая произведение элементов вектора. Однако, в отличие от `combine4`, здесь адрес `dest` обновляется в каждой итерации второй инструкцией `vmoovsd`.

Вот как можно выразить эту оптимизированную версию на языке C:

```

1 /* Гарантирует изменение dest в каждой итерации */
2 void combine3w(vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(v);
6     data_t *data = get_vec_start(v);
7     data_t acc = IDENT;
8
9     /* Инициализировать на случай, если length <= 0 */
10    *dest = acc;
11
12    for (i = 0; i < length; i++) {
13        acc = acc OP data[i];
14        *dest = acc;
15    }
16 }
```

2. Две версии `combine3` будут работать идентично, даже при наложении указателей.
3. Это преобразование можно выполнить без изменения поведения программы, потому что, за исключением первой итерации, значение, прочитанное из `dest` в начале каждой итерации, будет таким же значением, записанным в этот регистр в конце предыдущей итерации. Следовательно, инструкция комбинирования может просто использовать значение, уже записанное в `%xmm0` в начале цикла.

## Решение упражнения 5.5

Решение многочлена – основной метод решения многих задач. Например, функции решения многочлена обычно используются в математических библиотеках для аппроксимации тригонометрических функций.

1. Функция выполнит  $2n$  умножений и  $n$  сложений.
2. В этом случае вычислением, ограничивающим производительность, является повторное вычисление выражения `xrwr = x * xrwr`. Для этого требуется выполнить умножение с плавающей точкой (5 циклов синхронизации), при этом вычисление в следующей итерации не может начаться, пока не будет завершено вычисление в предыдущей итерации. Обновление результата требует только сложения чисел с плавающей точкой (3 цикла синхронизации) между последовательными итерациями.

## Решение упражнения 5.6

Это упражнение наглядно показывает, что иногда уменьшение количества операций в вычислении может не приводить к улучшению производительности.

1. Функция выполнит  $n$  умножений и  $n$  сложений, то есть в два раза меньше умножений, чем в исходной функции `polyh`.
2. В этом случае вычислением, ограничивающим производительность, является повторное вычисление выражения `result = a[i] + x * result`. Чтобы получить значение для текущей реализации, мы должны сначала умножить значение `result`, полученное в предыдущей итерации, на  $x$  (5 циклов синхронизации), а затем прибавить результат к `a[i]` (3 цикла). Таким образом, в каждой итерации имеет место минимальная задержка в 8 циклов, что в точности соответствует измеренной величине CPE.
3. Каждая итерация в функции `poly` выполняет два умножения, а не одно, но только одно умножение оказывается в критическом пути.

## Решение упражнения 5.7

Следующий код прямо следует правилам, которые мы сформулировали для развертывания цикла с некоторым коэффициентом  $k$ :

```

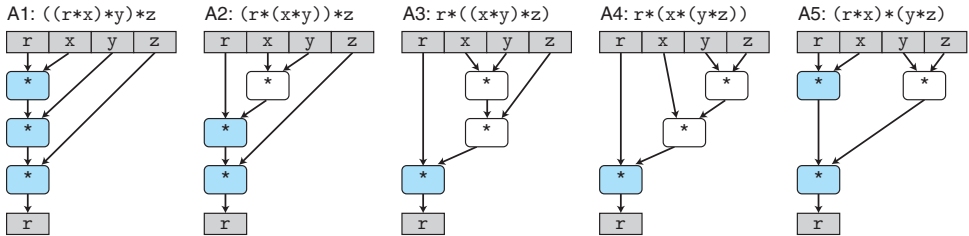
1 void unroll5(vec_ptr v, data_t *dest)
2 {
3     long i;
4     long length = vec_length(v);
5     long limit = length-4;
6     data_t *data = get_vec_start(v);
7     data_t acc = IDENT;
8
9     /* Объединять сразу по 5 элементов */
10    for (i = 0; i < limit; i+=5) {
11        acc = acc OP data[i] OP data[i+1];
12        acc = acc OP data[i+2] OP data[i+3];
13        acc = acc OP data[i+4];
14    }
15
16    /* Обработать оставшиеся элементы */
17    for (; i < length; i++) {
18        acc = acc OP data[i];
19    }
20    *dest = acc;
21 }
```

## Решение упражнения 5.8

Это упражнение демонстрирует, как небольшие изменения в программе могут приводить к существенным отличиям в производительности, особенно на процессоре, поддерживающем выполнение инструкций не по порядку. На рис. 5.26 показаны три операции умножения, выполняемые в одной итерации функции. На этом рисунке операции, изображенные в виде синих прямоугольников, расположены вдоль критического пути — они вычисляются последовательно и дают в результате новое значение для переменной цикла  $r$ . Операции, изображенные в виде белых прямоугольников, могут вычисляться параллельно с операциями критического пути. Для цикла с  $P$  операциями вдоль критического пути каждая итерация потребует минимум  $5P$  циклов синхронизации и будет вычислять произведение трех элементов, что дает нижнюю границу CPE, равную  $5P/3$ . Отсюда вытекает нижняя оценка 5,00 для A1, 3,33 для A2 и A5 и 1,67 для A3 и A4. Мы



опробовали данные функции на процессоре Intel Core i7 Haswell и обнаружили, что они достигают этих значений CPE.



**Рис. 5.26.** Зависимости по данным между операциями умножения для случаев в упражнении 5.8. Операции в синих прямоугольниках образуют критические пути итераций

## Решение упражнения 5.9

Это еще одна демонстрация, как небольшое изменение стиля программирования может значительно упростить компилятору возможность использования инструкций условного перемещения:

```
while (i1 < n && i2 < n) {
    long v1 = src1[i1]; long v2 = src2[i2];
    long take1 = v1 < v2;
    dest[id++] = take1 ? v1 : v2;
    i1 += take1;
    i2 += (1-take1);
}
```

Мы измерили CPE для этого кода и получили значение около 12,0 – небольшое увеличение, по сравнению с исходным значением CPE 15,0.

## Решение упражнения 5.10

Для решения этого упражнения нужно проанализировать потенциальные влияния операций загрузки и сохранения в программе.

1. Каждый элемент  $a[i]$  получит значение  $i + 1$  для  $0 \leq i \leq 998$ .
2. Каждый элемент  $a[i]$  получит значение 0 для  $1 \leq i \leq 999$ .
3. Во втором случае загрузка в одной итерации зависит от результата сохранения в предыдущей итерации. То есть между операциями записи и чтения в последовательных итерациях существует зависимость.
4. Этот вызов даст CPE 1,2, как и в примере 1, потому что операция сохранения и последующая за ней операция загрузки не зависят друг от друга.

## Решение упражнения 5.11

Эта функция, как можно заметить, имеет зависимые операции записи и чтения в последовательных итерациях – целевое значение  $r[i]$  в одной итерации совпадает с исходным значением  $r[i-1]$  в следующей. Поэтому итерации образуют критический путь, состоящий из операции сохранения (в предыдущей итерации), загрузки и сложения чисел с плавающей точкой. Измеренная величина CPE 9,0 согласуется с величиной CPE 7,3, измеренной для `write_read` при наличии зависимости по данным, потому что `write_read` включает целочисленное сложение (задержка в 1 цикл синхронизации), а `psum1` включает сложение с плавающей точкой (задержка в 3 цикла синхронизации).



## Решение упражнения 5.12

Вот переделанная версия функции:

```

1 void psum1a(float a[], float p[], long n)
2 {
3     long i;
4     /* last_val хранит p[i-1]; val хранит p[i] */
5     float last_val, val;
6     last_val = p[0] = a[0];
7     for (i = 1; i < n; i++) {
8         val = last_val + a[i];
9         p[i] = val;
10        last_val = val;
11    }
12 }
```

Мы добавили локальную переменную `last_val`. В начале  $i$ -й итерации она хранит значение  $p[i-1]$ . Затем мы вычисляем `val` как значение  $p[i]$  и новое значение для `last_val`.

Эта версия компилируется в следующий ассемблерный код:

Цикл в <code>psum1a</code>	
<code>a в %rdi, i в %rax, cnt в %rdx, last_val в %xmm0</code>	
1 <code>.L16:</code>	<code>loop:</code>
2 <code>vaddss (%rdi,%rax,4), %xmm0, %xmm0</code>	<code>last_val = val = last_val + a[i]</code>
3 <code>vmovss %xmm0, (%rsi,%rax,4)</code>	<code>Сохранить val в p[i]</code>
4 <code>addq \$1, %rax</code>	<code>Увеличить i</code>
5 <code>cmpq %rdx, %rax</code>	<code>Сравнить i:cnt</code>
6 <code>jne .L16</code>	<code>Если !=, то перейти к loop</code>

Этот код хранит `last_val` в `%xmm0` и тем самым избегает необходимости читать  $p[i-1]$ , что устраняет зависимость между операциями записи и чтения, наблюдаемую в `psum1`.

## Иерархия памяти

- 6.1. Технологии хранения информации.
  - 6.2. Локальность.
  - 6.3. Иерархия памяти.
  - 6.4. Кеш-память.
  - 6.5. Как писать код, адаптированный для использования кеш-памяти.
  - 6.6. Влияние кеша на производительность программ.
  - 6.7. Итоги.
- Библиографические заметки.  
Домашние задания.  
Решения упражнений.

В предыдущих главах при изучении компьютерных систем мы рассматривали простую модель центрального процессора, выполняющего инструкции и обладающего системой памяти, хранящей эти инструкции и данные. В этой простой модели память представляет собой линейный массив байтов, а время обращения к любой ячейке памяти постоянно. Несмотря на довольно высокую правдоподобность этой модели, она все-таки не отражает истинного устройства и принципов работы современных компьютерных систем.

В действительности *система памяти* являет собой иерархию запоминающих устройств разной емкости и стоимости и с разным временем доступа. В регистрах процессора хранятся наиболее часто используемые данные. Быстродействующие устройства кеш-памяти небольшой емкости, встроенные в процессор, выполняют функции промежуточных хранилищ для данных и инструкций, хранимых в сравнительно медленной основной памяти. Последняя организует данные, хранящиеся на больших, медленнеедействующих дисках, которые, в свою очередь, служат промежуточными хранилищами для данных, находящихся на дисках или лентах других машин, объединенных в сети.

Иерархическая организация памяти оправдывает себя, потому что хорошо написанные программы стремятся обращаться к хранилищу на любом конкретном уровне чаще, чем к хранилищу на следующем более низком уровне. Как следствие хранилище на следующем уровне может быть медленнее и стоить дешевле в расчете на бит хранимой информации. В результате мы имеем большой пул памяти со стоимостью недорогого хранилища в нижней части иерархии, но обслуживающий данные для программ со скоростью быстрого хранилища, находящегося на вершине иерархии.

Как программист вы должны понимать иерархическую организацию памяти, потому что она серьезно влияет на производительность приложений. Если необходимые для программы данные хранятся в регистрах процессора, то они будут доступны выполня-

емой инструкции немедленно. Если данные хранятся в кеше, то для их извлечения потребуется от 4 до 75 циклов синхронизации. Чтобы получить данные из основной памяти, потребуются сотни циклов, а чтобы прочитать с диска – десятки миллионов циклов!

Фундаментальная и незыблемая идея любой компьютерной системы заключается в том, что, понимая, как система перемещает данные вверх и вниз по иерархии памяти, можно писать программы так, что они будут хранить свои данные на более высоких уровнях иерархии, обеспечивая более короткое время доступа.

Эта идея тесно связана с базовым принципом создания компьютерных программ, получившим название *локальность*. Программы с хорошей локальностью снова и снова обращаются к одному и тому же множеству элементов данных, хранящихся рядом друг с другом. Программы с хорошей локальностью стремятся чаще использовать элементы данных из верхних уровней иерархии памяти, чем программы с плохой локальностью, и поэтому работают быстрее. Например, в нашей системе с процессором Core i7 время перемножения матриц в программах с разной степенью локальности может отличаться в 40 раз!

В этой главе мы представим основные технологии хранения информации – в памяти SRAM, в памяти DRAM, в памяти ROM и на дисках, а также опишем их организацию в виде иерархии. В частности, мы уделим много внимания кеш-памяти, играющей роль промежуточного хранилища между процессором и основной памятью, потому что она оказывает наибольшее влияние на производительность программ. Мы покажем, как проверить локальность программ, а также познакомимся с некоторыми методиками ее повышения. Вы узнаете интересный способ представления иерархии памяти конкретной машины в виде «горы памяти», позволяющий изобразить время доступа как функцию локальности.

## 6.1. Технологии хранения информации

Успех информационных технологий во многом обусловлен невероятным прогрессом технологии хранения информации. Первые компьютеры имели всего несколько килобайт ОЗУ. Самые первые персональные компьютеры IBM не имели даже жесткого диска. Все изменилось с появлением в 1982 году модели IBM PC-XT с диском на 10 Мбайт. К 2015 году самый обычный компьютер имел в 300 000 раз больше дисковой памяти, и каждые пару лет этот показатель увеличивается в 2 раза.

### 6.1.1. Память с произвольным доступом

*Память с произвольным доступом* (Random Access Memory, RAM), или ОЗУ, выпускается в двух вариантах: статическом и динамическом. *Статическая память* (Static RAM, SRAM) обладает более высоким быстродействием, но и стоит дороже, чем *динамическая память* (Dynamic RAM, DRAM). SRAM используется как кеш-память, которая встраивается в сам процессор и/или размещается на материнской плате рядом с ним. DRAM используется в роли основной памяти и буфера графической системы. Как правило, обычные настольные системы имеют не более нескольких десятков мегабайт SRAM, но сотни или тысячи мегабайт DRAM.

#### Статическая память с произвольным доступом

SRAM хранит каждый бит в *бистабильной* (с двумя устойчивыми состояниями) ячейке памяти. Каждая ячейка реализуется в виде цепи из шести транзисторов. Эта цепь обладает способностью бесконечно долго поддерживать любой из двух уровней напряжения, которые называют *состояниями*. Любое промежуточное состояние будет неустойчивым: попав в него, цепь быстро перейдет в одно из устойчивых состояний. Такая ячейка памяти аналогична перевернутому маятнику, показанному на рис. 6.1.



**Рис. 6.1.** Обратный маятник. Подобно ячейке SRAM, обратный маятник имеет только два устойчивых состояния

Такой маятник устойчив, когда находится в крайнем левом или правом положении. Оказавшись в любом другом положении, маятник упадет в ту или иную сторону. В принципе, маятник может оставаться сбалансированным в вертикальном положении на неопределенное время, но это *метастабильное* состояние: малейшее возмущение заставит его упасть, а после его падения он никогда не вернется в вертикальное положение.

Благодаря своей бистабильной природе ячейка памяти SRAM будет хранить свое значение неопределенно долго, пока на нее подается питание. Даже когда возмущение, такое как электрический шум, вызывает возмущение напряжения, схема вернется в стабильное состояние, когда возмущение исчезнет.

### Динамическая память с произвольным доступом

DRAM хранит каждый бит как заряд конденсатора. Конденсатор имеет очень маленькую емкость, порядка 30 фемтофард ( $30 \times 10^{-15}$  фарад). Но не забывайте, что 1 фарад – это гигантская емкость. Память DRAM допускает возможность очень плотного размещения ячеек, каждая из которых состоит из конденсатора и одного транзистора. В отличие от SRAM, ячейки памяти DRAM очень чувствительны к любым возмущениям. При изменении напряжения конденсатор не сможет сам восстановить его. Величина напряжения заряда конденсатора может измениться даже под воздействием солнечных лучей. На самом деле светочувствительные матрицы в цифровых фотоаппаратах и видеокамерах являются массивами ячеек DRAM.

Из-за естественных причин ячейки DRAM разряжаются в течение порядка 10–100 мс. К счастью, для компьютеров, в которых длительность циклов синхронизации измеряется наносекундами, этот факт не является существенным. Система памяти периодически обновляет хранимые биты, читая и перезаписывая их. В некоторых системах также применяются технологии хранения с исправлением ошибок, когда в каждое компьютерное слово добавляется несколько дополнительных битов (например, 32-битное слово быть представлено 38 битами), с помощью которых схема может выявить и исправить любой одиночный ошибочный бит в слове.

В табл. 6.1 представлены сравнительные характеристики памяти SRAM и DRAM. Память SRAM устойчиво хранит информацию, пока ячейки находятся под напряжением. В отличие от DRAM, обновлять их необязательно. Доступ к SRAM требует меньше времени, чем к DRAM. Память SRAM не восприимчива к таким возмущениям, как свет и электрические помехи. Но для каждой ячейки SRAM требуется больше транзисторов, чем для ячейки DRAM, их нельзя разместить так же плотно, как ячейки DRAM, поэтому память получается более дорогостоящей и потребляет больше электроэнергии.

**Таблица 6.1.** Сравнительные характеристики устройств памяти SRAM и DRAM

	Транзисторов на бит	Относительное время доступа	Надежность хранения	Чувствительность	Относительная стоимость	Применение
SRAM	6	1×	Да	Нет	1000×	Кеш-память
DRAM	1	10×	Нет	Да	1×	Основная память, буферы графических систем

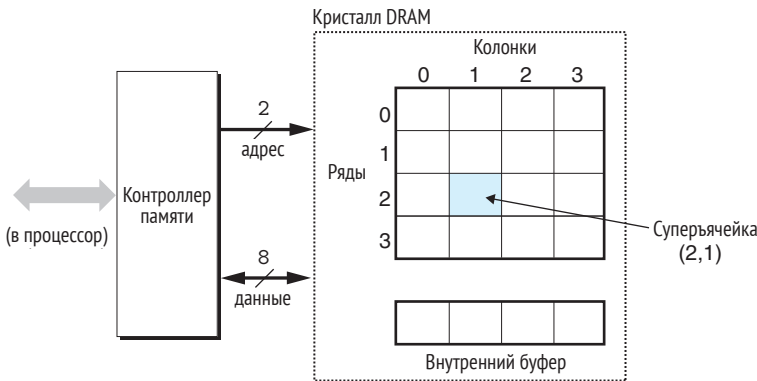
**Примечание по терминологии**

Разработчики устройств памяти не смогли договориться о едином названии элемента массива DRAM. Компьютерные архитекторы называют его «ячейкой», перегружая термин «ячейка памяти DRAM». Схемотехники называют его «словом», перегружая термин «слово основной памяти». Чтобы избежать путаницы, мы решили использовать менее двусмысленный термин «суперъячейка».

**Стандартная динамическая память с произвольным доступом**

Ячейки (биты) в DRAM разделены на  $d$  суперъячеек, каждая из которых состоит из  $w$  ячеек DRAM. Всего DRAM может хранить  $d \times w$  бит информации. Суперъячейки организованы в виде прямоугольного массива из  $r$  рядов и  $c$  колонок, где  $rc = d$ . Каждая суперъячейка имеет адрес в форме  $(i, j)$ , где  $i$  обозначает номер ряда, а  $j$  – номер колонки.

Например, на рис. 6.2 показана организация кристалла DRAM с размером  $16 \times 8$ , где  $d = 16$  суперъячеек,  $w = 8$  бит на суперъячейку,  $r = 4$  ряда и  $c = 4$  колонки. Заштрихованная клетка обозначает суперъячейку с адресом  $(2, 1)$ . Информация поступает в кристалл и покидает его через внешние соединители, называемые *контактами*. Каждый контакт передает 1-битный сигнал. На рис. 6.2 показаны два набора контактов: 8 контактов данных для переноса одного байта в кристалл или из него, и 2 контакта адреса, передающих 2-битный номер ряда и колонки, образующих адрес суперъячейки. Другие контакты, передающие управляющую информацию, здесь не показаны.



**Рис. 6.2.** Обобщенное представление 128-битного кристалла DRAM размером  $16 \times 8$

Каждый кристалл DRAM подключен к определенной цепи, называемой *контроллером памяти*, по которой за единицу времени может передаваться  $w$  бит в кристалл DRAM и из него. Для чтения содержимого суперъячейки  $(i, j)$  контроллер памяти посылает в DRAM сначала адрес ряда  $i$ , а вслед за ним адрес колонки  $j$ . DRAM в ответ посылает контроллеру содержимое суперъячейки  $(i, j)$ . Адрес ряда  $i$  называется *запросом RAS* (Row Access Strobe – строб адреса ряда), а адрес колонки  $j$  – *запросом CAS* (Column Access Strobe – строб адреса колонки). Обратите внимание, что запросы RAS и CAS посылаются через одни и те же контакты адреса DRAM.

Например, чтобы прочитать содержимое суперъячейки  $(2, 1)$  из памяти DRAM  $16 \times 8$ , изображенной на рис. 6.2, контроллер памяти посылает адрес ряда 2 (рис. 6.3 (а)). DRAM в ответ копирует содержимое всего ряда 2 во внутренний буфер. Далее контроллер па-

мяти посылает адрес колонки 1, как показано на рис. 6.3 (b). DRAM в ответ посылает контроллеру памяти 8 бит из суперъячейки во внутреннем буфере, соответствующей суперъячейке (2,1).

Одна из причин, почему разработчики устройств памяти организовали DRAM в виде двумерных, а не линейных массивов, – сокращение числа адресных контактов. Например, если бы уже упоминавшаяся 128-битная память DRAM была организована в виде линейного массива, состоящего из 16 суперъячеек с адресами от 0 до 15, то для адресации суперъячеек потребовалось бы четыре адресных контакта, а не два, правда, при этом адрес должен отправляться в два этапа, что увеличивает время доступа.

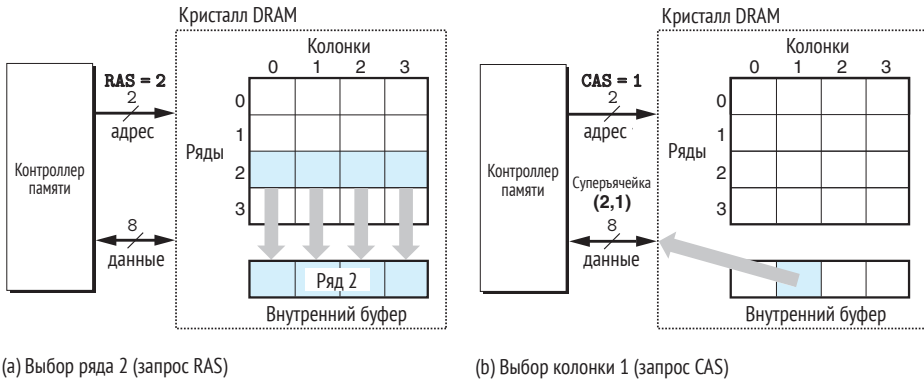


Рис. 6.3. Чтение содержимого суперъячейки DRAM

## Модули памяти

Кристаллы DRAM объединяются в пакеты – *модули памяти*, – подключаемые к разъемам на материнской плате. В системах на основе Core i7 используются 240-контактные *модули памяти с двухрядным расположением контактов* (Dual Inline Memory Module, DIMM), обменивающиеся данными с контроллером памяти 64-битными порциями.

На рис. 6.4 показана обобщенная схема устройства модуля памяти. Модуль в этом примере может хранить всего 64 Мбайт в восьми 64-Мбитных (8M×8) кристаллах DRAM, пронумерованных от 0 до 7. Каждая суперъячейка хранит один байт, а каждое 64-битное слово с адресом  $A$  байт представлено восемью суперъячейками. В примере на рис. 6.4 DRAM 0 хранит первый байт (младшего разряда), DRAM 1 – следующий байт и т. д.

Чтобы прочитать 64-битное слово с адресом  $A$ , контроллер памяти преобразует  $A$  в адрес суперъячейки  $(i, j)$  и посылает его в модуль памяти, который затем передает  $i$  и  $j$  в каждый кристалл DRAM. В ответ кристаллы DRAM возвращают содержимое своих 8-битных суперъячеек  $(i, j)$ . Цепь модуля памяти собирает эти данные, формирует 64-битное слово и возвращает его контроллеру памяти.

Основную память можно скомпоновать, подключив к контроллеру несколько модулей памяти. В этом случае, когда контроллер получит адрес  $A$ , он выберет модуль  $k$ , соответствующий адресу  $A$ , преобразует  $A$  в форму  $(i, j)$  и отправит  $(i, j)$  в модуль  $k$ .

### Упражнение 6.1 (решение в конце главы)

Пусть  $r$  – количество рядов в массиве DRAM,  $c$  – количество колонок,  $b_r$  – количество битов, необходимых для адресации рядов, и  $b_c$  – количество битов, необходимых для адресации колонок. Для каждого из следующих кристаллов DRAM определите размеры массива, кратные степени двойки, минимизирующие  $\max(b_r, b_c)$  – максимальное число битов, необходимое для адресации рядов и колонок массива.

Организация	$r$	$c$	$b_r$	$b_c$	$\max(b_r, b_c)$
16×1					
16×4					
128×8					
512×4					
1024×4					

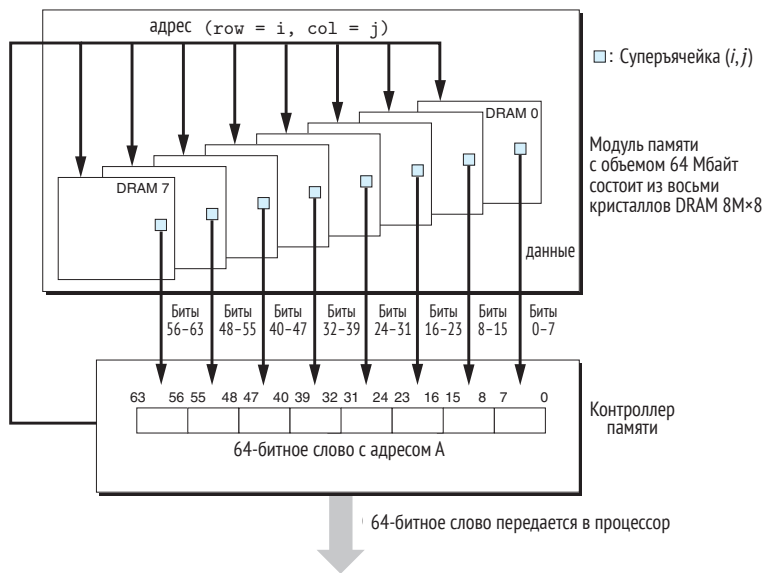


Рис. 6.4. Чтение данных из модуля памяти

### Расширенная динамическая память с произвольным доступом

Существует много типов памяти DRAM, и регулярно появляются новые, по мере того как производители пытаются не отставать от постоянно совершенствующихся процессоров. Все они основаны на использовании стандартных ячеек DRAM, но с разными оптимизациями, увеличивающими скорость доступа.

*Быстрая память DRAM со страничным режимом (Fast page mode DRAM, FPM DRAM).* Традиционная память DRAM, которая копирует весь ряд суперъячеек во внутренний буфер, использует одну суперъячейку, а остальные отбрасывает. Для ускорения FPM DRAM обеспечивает возможность обслуживания последующих запросов из того же буфера. Например, чтобы прочитать четыре суперъячейки из ряда  $i$ , контроллер традиционной памяти DRAM должен отправить четыре запроса RAS/CAS, даже притом что адрес ряда  $i$  в этих случаях не изменяется. Чтобы прочитать суперъячейки из того же ряда, контроллер памяти FPM DRAM отправит первоначальный запрос RAS/CAS, за которым последуют три запроса CAS. В ответ на первоначальный запрос RAS/CAS память DRAM скопирует ряд  $i$  в буфер и вернет суперъячейку, соответствующую запросу CAS. Следующие три суперъячейки будут извлечены непосредственно из буфера и несколько быстрее, чем первоначальная суперъячейка.

*DRAM с расширенными возможностями вывода* (Extended data out DRAM, EDO DRAM). Расширенный вариант FPM DRAM, позволяющий сближать во времени отдельные сигналы CAS.

*Синхронная DRAM* (Synchronous DRAM, SDRAM). Традиционные FPM и EDO DRAM действуют асинхронно, в том смысле, что они взаимодействуют с контроллером памяти посредством явных управляющих сигналов. SDRAM замещает многие из этих управляющих сигналов нарастающими фронтами внешнего синхронизирующего сигнала для управления контроллером памяти. Не вдаваясь в детали, можно сказать, что благодаря этому SDRAM выдает содержимое суперъядеек с большей скоростью, чем асинхронные эквиваленты.

*Синхронная DRAM с удвоенной скоростью обработки данных* (Double Data-Rate Synchronous DRAM, DDR DRAM). DDR DRAM – это усовершенствованный вариант SDRAM, действующий с удвоенной скоростью за счет использования для управления обоих фронтов синхронизирующего сигнала.

*Видеопамять* (Video RAM, VRAM). Применяется в буферах кадров графических систем. По духу VRAM близка к FPM DRAM. Два основных отличия: выходные данные VRAM создаются сдвигом всего содержимого внутреннего буфера, и VRAM обеспечивает параллельные чтение и запись. То есть система может заполнять экран пикселями в кадровом буфере (чтение) и параллельно записывать новые значения для следующего обновления (запись).

#### **История развития технологии DRAM**

До 1995 года большинство персональных компьютеров оснащались FPM DRAM. С 1996 по 1999 год на рынке доминировали EDO DRAM, а FPM DRAM практически сошли со сцены. SDRAM впервые появились в 1995 году в высокотехнологичных системах, а к 2002 году большинство компьютеров уже оснащались устройствами SDRAM и DDR SDRAM. К 2010 году большинство серверов и настольных компьютеров оснащались памятью DDR3 SDRAM. Кстати, Intel Core i7 поддерживает исключительно DDR3 SDRAM.

### **Энергонезависимая память**

Память DRAM и SRAM является *энергозависимой*, т. е. при выключении питания информация в ней теряется. *Энергонезависимая память*, напротив, сохраняет информацию даже при отключении питания. Существует множество типов энергонезависимой памяти. По историческим причинам они носят общее название *постоянной памяти* (Read-Only Memory, ROM), хотя некоторые типы ROM позволяют не только читать информацию, но и записывать. Разные устройства постоянной памяти различаются количеством раз, сколько их можно перепрограммировать (записывать), а также механизмами такого перепрограммирования.

*Программируемую постоянную память* (Programmable ROM, PROM) можно запрограммировать только один раз. Ячейки памяти в PROM напоминают плавкие предохранители, которые могут «перегореть» при подаче большого тока.

*Стираемая программируемая постоянная память* (Erasable Programmable ROM, EPROM) имеет прозрачное кварцевое окошко, через которое ячейки можно облучить ультрафиолетовым светом. Информация в ячейках EPROM стирается при попадании на них ультрафиолетовых лучей через окошко. Программирование EPROM осуществляется с помощью специального устройства – программатора. EPROM выдерживает до 1000 циклов стирания/программирования. Электричес-



ки стираемая постоянная память (Electrically Erasable PROM, EEPROM) сходна с EPROM, но не требует отдельного программатора и может перепрограммироваться на месте, без извлечения из печатной платы. EEPROM можно перепрограммировать до  $10^5$  раз.

**Флеш-память** – это разновидность энергонезависимой памяти на базе EEPROM, которая стала важнейшей технологией хранения данных. Флеш-память получила большое распространение как быстрое и надежное энергонезависимое хранилище, пригодное для использования во множестве электронных устройств, включая цифровые камеры, сотовые телефоны и музыкальные плееры, а также портативные, настольные и серверные компьютерные системы. В разделе 6.1.3 мы подробно рассмотрим новую форму флеш-накопителя, известную как *твердотельный диск* (Solid State Disk, SSD), – более быструю, более надежную и менее энергоемкую альтернативу обычным вращающимся дискам.

Программы, сохраняемые в ROM, часто называются «защитными». При подаче питания компьютерная система запускает зашитую программу, хранящуюся в ROM. В некоторых системах зашитые программы включают небольшие наборы элементарных функций ввода и вывода, например процедуры BIOS (Basic Input/Output System – базовая система ввода/вывода). Сложные устройства, такие как видеокарты и дисковые накопители, также имеют вшитое программное обеспечение для преобразования запросов ввода/вывода, получаемых от центрального процессора.

## Доступ к основной памяти

Между процессором и основной памятью DRAM постоянно передаются данные туда и обратно через совместно используемые электрические соединения, называемые *шинами*. Каждая передача данных между процессором и памятью выполняется в несколько этапов, называемых *транзакцией шины*. *Транзакция чтения* передает данные из основной памяти в процессор. *Транзакция записи* передает данные из процессора в основную память.

**Шина** – это набор проводников, по которым передаются адрес, данные и управляющие сигналы. В зависимости от конструкции шины данные и адреса могут передаваться по одним и тем же или по разным проводникам. Также одна шина может совместно использоваться несколькими устройствами. Управляющие сигналы передаются по отдельным проводникам, они синхронизируют транзакцию и идентифицируют тип выполняемой транзакции. Например, с каким устройством выполняется транзакция – с основной памятью или каким-либо другим устройством ввода/вывода; тип транзакции – чтение или запись; тип информации, передаваемой по шине, – адрес или данные.

На рис. 6.5 показана конфигурация обычной компьютерной системы. Основными компонентами являются процессор, набор интерфейсных микросхем, который называют *мостом ввода/вывода* (куда входит и контроллер памяти), и модули памяти DRAM, составляющие основную память. Эти компоненты соединены парой шин: *системной шиной*, связывающей процессор с мостом ввода/вывода, и *шиной памяти*, связывающей мост ввода/вывода с основной памятью. Мост ввода/вывода преобразует электрические сигналы системной шины в электрические сигналы шины памяти. Как будет показано далее, мост ввода/вывода также связывает системную шину и шину памяти с *шиной ввода/вывода*, совместно используемой устройствами ввода/вывода, такими как диски и графические карты. Но пока мы сосредоточим основное внимание только на шине памяти.

### Об архитектуре шин

Шины имеют сложную и быстро меняющуюся архитектуру. Разные производители разрабатывают разные архитектуры шин для своих продуктов. Например, в некоторых системах Intel используются наборы микросхем, известные как *северный мост* и *южный мост*, связывающие процессор с памятью и устройствами ввода/вывода соответственно. В старых системах Pentium и Core 2 для связи процессора с северным мостом использовалась *передняя шина* (Front Side Bus, FSB). В системах AMD вместо системной шины используется высокопроизводительная шина *HyperTransport*, а в новых системах Intel Core i7 используют шину *QuickPath*. Обсуждение подробностей этих архитектур шин выходит за рамки нашей книги, поэтому мы не будем их рассматривать, а вместо этого будем строить обсуждение, опираясь на обобщенную архитектуру, изображенную на рис. 6.5. Это простая, но полезная абстракция, позволяющая быть достаточно конкретными. Она отражает основные идеи, не будучи слишком привязанным к техническим деталям каких-либо проприетарных архитектур.

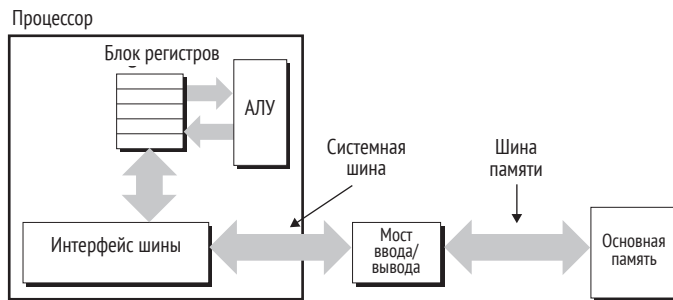


Рис. 6.5. Типичная архитектура шин, связывающая процессор с основной памятью

Давайте посмотрим, что происходит, когда процессор выполняет такую операцию загрузки данных из памяти, как

```
movq A,%rax
```

которая загружает содержимое ячейки памяти с адресом *A* в регистр *%rax*. Блок процессора, который называется *интерфейсом шины*, инициирует на шине транзакцию чтения. Она выполняется в три этапа. Во-первых, процессор помещает адрес *A* на системную шину. Мост ввода/вывода передает сигнал на шину памяти (рис. 6.6 (а)). Основная память распознает сигнал адреса на шине памяти, читает его, выбирает слово данных из DRAM и записывает данные в шину памяти. Мост ввода/вывода преобразует сигнал шины памяти в сигнал системной шины и передает его на системную шину (рис. 6.6 (b)). Наконец, процессор распознает появление данных на системной шине, читает их и копирует в регистр *%rax* (рис. 6.6 (c)).

Напротив, когда процессор выполняет команду сохранения, такую как

```
movq %rax,A
```

которая записывает содержимое регистра *%rax* в ячейку с адресом *A*, процессор инициирует транзакцию записи. Она тоже выполняется в три этапа. Сначала процессор помещает адрес на системную шину. Память читает адрес с шины памяти и ждет появления данных (рис. 6.8 (а)). Процессор копирует слово данных из *%rax* на системную шину (рис. 6.7 (b)), а основная память читает это слово с шины памяти и сохраняет в DRAM (рис. 6.7 (c)).

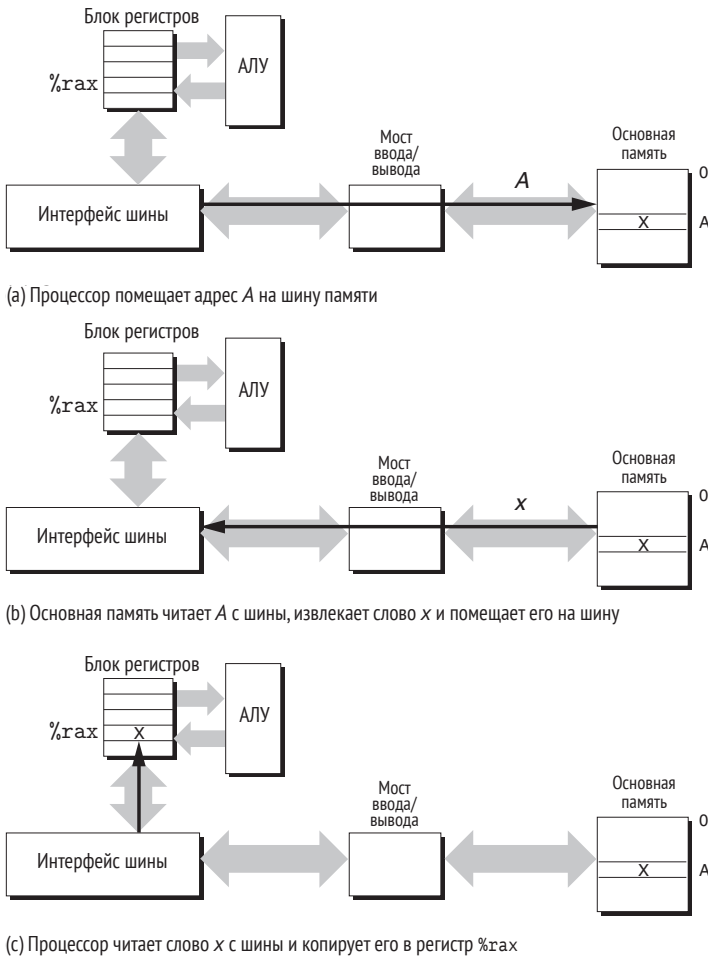


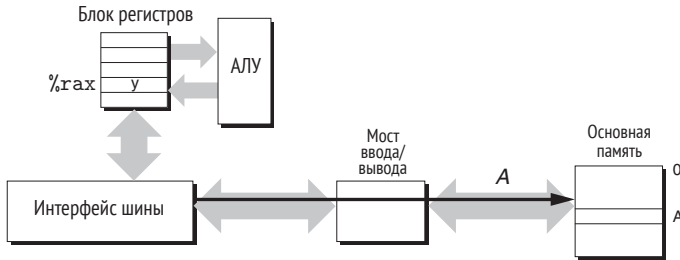
Рис. 6.6. Транзакция чтения из памяти для операции загрузки `movq A, %rax`

### 6.1.2. Диски

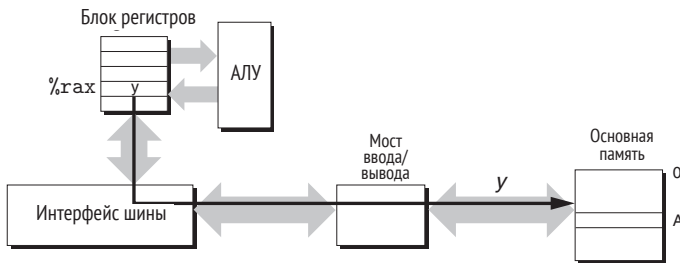
Диски являются основными устройствами хранения информации. Они способны хранить огромные объемы данных – сотни и тысячи гигабайт, в отличие от основной памяти, объем которой ограничивается тысячами мегабайт. Однако время чтения информации с диска измеряется миллисекундами – в сотни тысяч раз больше времени чтения из DRAM и в миллионы раз больше времени чтения из SRAM.

#### Геометрия диска

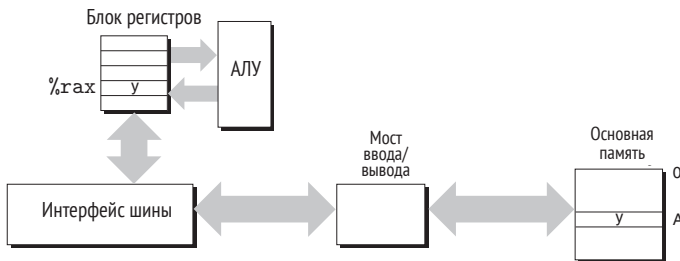
Диски состоят из *пластин*. Каждая пластина с двух сторон, или *поверхностей*, покрыта магнитным материалом. *Шпиндель* в центре пластины вращает ее с фиксированной скоростью вращения, обычно 5400 или 15 000 оборотов в минуту (об/мин). Дисковое устройство обычно содержит несколько таких пластин, закрытых в герметичном корпусе.



(a) Процессор помещает адрес `A` на шину памяти. Основная память читает адрес и ждет появления слова данных



(b) Процессор помещает слово данных `y` на шину



(c) Основная память читает слово `y` с шины и сохраняет его в ячейке памяти с адресом `A`

**Рис. 6.7.** Транзакция записи в память для операции сохранения `movq %rax, A`

На рис. 6.8 (а) показана геометрия поверхности типичного диска. Каждая поверхность делится на множество концентрических колец, называемых *дорожками*. Каждая дорожка разделена на множество *секторов*. Каждый сектор содержит равное количество данных (обычно 512 байт), записанных на магнитный материал в секторе. Секторы разделены *промежутками*, в которых не хранится никаких данных. Промежутки используются для хранения идентифицирующей информации о секторах.

Пластины одеваются на общий шпиндель друг над другом – стопкой, как показано на рис. 6.8 (b). Всю конструкцию целиком часто называют *дисковым приводом*, но мы будем называть ее просто *диском*. Иногда мы будем называть диски *вращающимися дисками*, чтобы отличить их от твердотельных дисков (Solid State Disk, SSD) на основе флеш-памяти, у которых нет движущихся частей.

Производители дисков часто описывают геометрию дисков с несколькими пластинами, используя термин *цилиндры*, где цилиндром называется набор равноудаленных от центра дорожек на всех поверхностях. Например, если диск имеет три пластины и шесть поверхностей и дорожки на всех поверхностях пронумерованы единообразно, то цилиндр  $k$  будет представлять набор из дорожек с номером  $k$  на шести поверхностях.

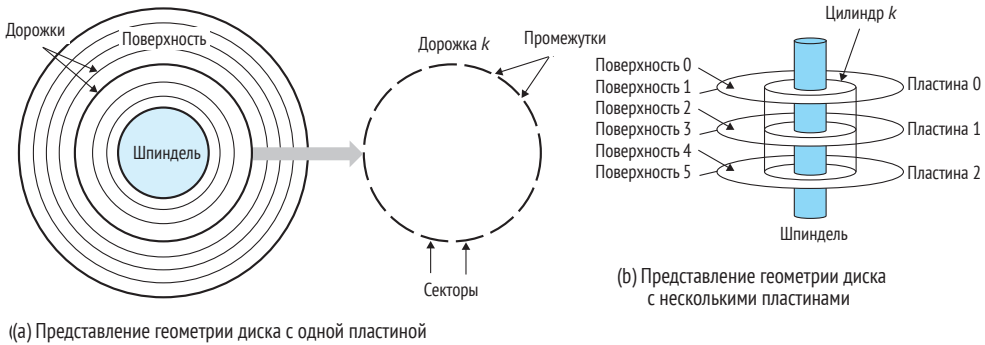


Рис. 6.8. Геометрия диска

### 1 гигабайт – это сколько?

К сожалению, значение таких приставок, как «кило» (К), «мега» (М), «гига» (Г) и «тера» (Т), зависит от контекста. Когда речь идет о емкости DRAM и SRAM, обычно принимается:  $K = 2^{10}$ ,  $M = 2^{20}$ ,  $G = 2^{30}$  и  $T = 2^{40}$ . Когда речь идет о емкости устройств ввода/вывода (дисков и сетей), принимаются немного другие значения:  $K = 10^3$ ,  $M = 10^6$ ,  $G = 10^9$  и  $T = 10^{12}$ . Эти же значения приставок используются для оценки быстродействия и пропускной способности.

К счастью, для упрощенных расчетов, которые обычно используются на практике, подходят любые из перечисленных значений. Например, относительная разница между  $2^{30}$  и  $10^9$  не особенно велика:  $(2^{30} - 10^9) / 10^9 \approx 7\%$ . Аналогично:  $(2^{40} - 10^{12}) / 10^{12} \approx 10\%$ .

## Емкость диска

Максимальный объем информации, который можно записать на диск, называется его *максимальной емкостью* (или просто *емкостью*). Емкость диска определяется следующими технологическими факторами:

- *плотность записи* (бит/дюйм) – количество битов, которые можно записать на однодюймовом сегменте дорожки;
- *плотность размещения дорожек* (дорожек/дюйм) – количество дорожек, которые можно уместить на однодюймовом отрезке радиуса пластины;
- *поверхностная плотность записи* (бит/дюйм<sup>2</sup>) – произведение плотности записи и плотности дорожки.

Производители дисков неустанно работают над повышением поверхностной плотности записи (и, следовательно, емкости), которая удваивается каждые пару лет. В первых дисках, разработанных в эру малой поверхностной плотности записи, каждая дорожка была разделена на одинаковое количество секторов, которое можно записать на самую близкую к центру дорожку. Для поддержки фиксированного количества секторов в каждой дорожке секторы на дальних от центра дорожках разделялись большими промежутками. В ту пору это считалось разумным подходом. Однако с увеличением поверхностной плотности записи промежутки между секторами (в которых не хранится почти никакой полезной информации) превратились в недопустимую роскошь. По этой причине в современных дисках стала применяться технология *многозонной записи*, когда набор цилиндров разбивается на несвязанные подмножества *зон записи*. Каждая зона объединяет несколько соседних цилиндров. Каждый цилиндр в зоне имеет одинаковое количество секторов, которое можно уместить на самом внутреннем цилиндре в зоне.

Емкость диска определяется следующей формулой:

$$\text{Емкость} = \frac{\text{кол. байт}}{\text{сектор}} = \frac{\text{сред. кол. секторов}}{\text{дорожка}} = \frac{\text{кол. дорожек}}{\text{поверхность}} = \frac{\text{кол. поверхностей}}{\text{пластина}} = \frac{\text{кол. пластин}}{\text{диск}}.$$

Например, пусть у нас имеется диск с пятью пластинами, секторами по 512 байт, 20 000 дорожками на каждой поверхности и 300 секторами на дорожку в среднем. Емкость такого диска можно вычислить следующим образом:

$$\begin{aligned} \text{Емкость} &= \frac{512 \text{ байт}}{\text{сектор}} = \frac{300}{\text{дорожка}} = \frac{20\,000}{\text{поверхность}} = \frac{2}{\text{поверхности}} = \frac{5}{\text{пластина}} = \frac{5}{\text{диск}} \\ &= 30\,720\,000\,000 \text{ байт} \\ &= 30,72 \text{ Гбайт.} \end{aligned}$$

Обратите внимание, что производители выражают емкость диска в гигабайтах (Гбайт) или терабайтах (Тбайт), где 1 Гбайт =  $10^9$  байт, а 1 Тбайт =  $10^{12}$  байт.

#### Упражнение 6.2 (решение в конце главы)

Какую емкость имеет диск с 2 пластинами, 10 000 цилиндров, 400 секторами на дорожку в среднем и 512 байтами на сектор?

### Работа диска

Диски читают и записывают биты с помощью магнитной головки чтения/записи, закрепленной на конце *рычага актуатора*, как показано на рис. 6.9 (а). Перемещая рычаг вперед и назад вдоль радиуса, дисковое устройство может менять положение головки относительно любой дорожки на поверхности. Это механическое движение называется *позиционированием* (seek). Как только головка оказывается над нужной дорожкой, то по мере прохождения под ней каждого бита головка либо читает, либо записывает его значение. Диски с несколькими пластинами имеют отдельные головки для каждой поверхности, как показано на рис. 6.9 (б). Головки выровнены вертикально и двигаются все одновременно. В любой момент времени все головки расположены в одном и том же цилиндре.

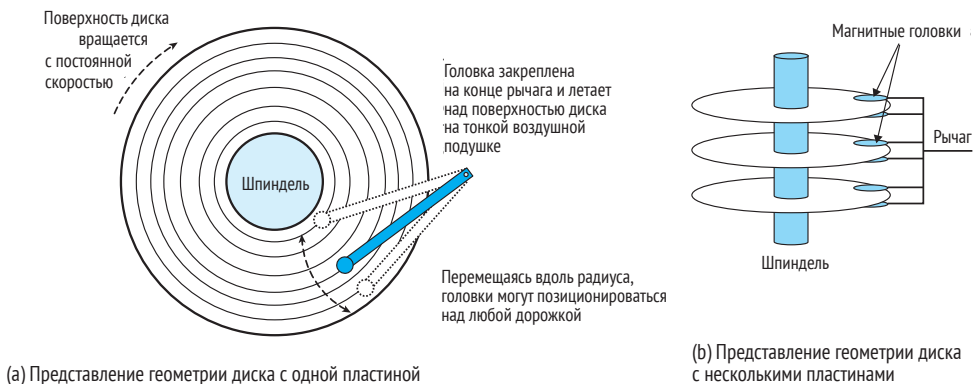


Рис. 6.9. Динамика диска

Головка на конце рычага буквально летает на тончайшей воздушной подушке над поверхностью диска на высоте порядка 0,1 микрона со скоростью примерно 80 км/ч. Это сродни тому, как если бы небоскреб положили на бок и заставили его облететь Землю на высоте 2,5 см (1 дюйм) над поверхностью, при этом на каждый виток уходило бы всего 8 с! При таких допусках крошечная пылинка на поверхности покажется огромным валуном. Если бы головка ударилась об один из этих валунов, то прекратила бы свой полет и врезалась в поверхность (произошла бы так называемая авария головки). По этой причине корпуса дисков всегда делаются герметичными.

Диски читают и записывают данные блоками с размером сектора. *Время доступа* к сектору складывается из трех составляющих: *времени позиционирования, задержки из-за вращения и времени передачи*:

- *время позиционирования*. Для чтения содержимого сектора рычаг сначала располагает головку над дорожкой, где находится сектор. Время, необходимое для перемещения рычага, называется *временем позиционирования*. Время позиционирования  $T_{seek}$  зависит от предыдущего положения головки и от скорости перемещения рычага. Среднее время позиционирования в современных дисках  $T_{avg seek}$ , измеренное по нескольким тысячам операций позиционирования, обычно составляет порядка 3–9 мс. Максимальное время позиционирования  $T_{max seek}$  может достигать 20 мс;
- *задержка из-за вращения диска*. После того как головка окажется над дорожкой, диск ждет, пока под головкой окажется первый бит требуемого сектора. Продолжительность этого этапа зависит от угла поворота поверхности в момент, когда головка достигла целевой дорожки, и от скорости вращения диска. В худшем случае головка оказалась над нужной дорожкой, когда целевой сектор только-только миновал ее позицию, и теперь диск должен дожидаться, пока пластина совершит полный оборот. Максимальная задержка из-за вращения диска вычисляется по следующей формуле:

$$T_{max rotation} = \frac{1}{\text{об/мин}} \times \frac{60 \text{ с}}{1 \text{ мин}}.$$

Средняя задержка из-за вращения диска  $T_{avg rotation}$  для простоты принимается равной половине  $T_{max rotation}$ :

- *время передачи*. Когда первый бит целевого сектора оказывается под головкой, дисковод начинает читать или записывать содержимое сектора. Время передачи одного сектора зависит от скорости вращения и количества секторов на одной дорожке. То есть среднее время передачи одного сектора можно приблизительно вычислить по следующей формуле:

$$T_{avg transfer} = \frac{1}{\text{об/мин}} \times \frac{1}{(\text{ср. кол. секторов на дорожке})} \times \frac{60 \text{ с}}{1 \text{ мин}}.$$

Среднее время доступа к содержимому сектора диска можно вычислить как сумму среднего времени позиционирования, средней задержки из-за вращения диска и среднего времени передачи. Например, рассмотрим диск со следующими параметрами:

Параметр	Значение
Скорость вращения	7200 об/мин
Среднее время позиционирования ( $T_{avg seek}$ )	9 мс
Среднее количество секторов на дорожке	400

Для этого диска средняя задержка из-за вращения диска (в мс) составит:

$$\begin{aligned} T_{avg\ rotation} &= 1/2 \times T_{max\ rotation} \\ &= 1/2 \times (60\ c / 7200\ об/мин) \times 1000\ мс/с \\ &\approx 4\ мс. \end{aligned}$$

Среднее время передачи:

$$\begin{aligned} T_{avg\ transfer} &= 60\ c / 7200\ об/мин \times 1/400\ секторов\ на\ дорожке \times 1000\ мс/с \\ &\approx 0,02\ мс. \end{aligned}$$

Сложив, получаем общее оценочное время доступа:

$$\begin{aligned} T_{access} &= T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \\ &= 9\ мс + 4\ мс + 0,02\ мс \\ &= 13,02\ мс. \end{aligned}$$

Из этого примера следует несколько важных выводов:

- во времени доступа к 512 байтам дискового сектора большую часть занимают время позиционирования и задержка из-за вращения диска. Доступ к первому байту сектора занимает сравнительно много времени, однако доступ к остальным байтам почти не занимает времени;
- поскольку время позиционирования и задержка из-за вращения диска имеют сопоставимые величины, то удвоение времени позиционирования можно рассматривать как простую аппроксимацию общего времени доступа к диску;
- время доступа к 64-разрядному слову в SRAM составляет приблизительно 4 нс, в DRAM – 60 нс. То есть время чтения 512-байтного блока из памяти SRAM составит приблизительно 256 нс и из памяти DRAM – 4000 нс. Время доступа к диску (приблизительно 10 мс) в 40 000 раз больше времени доступа к SRAM и примерно в 2500 раз больше времени доступа к DRAM.

### Упражнение 6.3 (решение в конце главы)

Вычислите среднее время доступа (в мс) к сектору для следующего диска:

Параметр	Значение
Скорость вращения	15 000 об/мин
Среднее время позиционирования ( $T_{avg\ seek}$ )	8 мс
Среднее количество секторов на дорожке	500

## Логические блоки диска

Как уже отмечалось, современные диски имеют весьма сложную геометрию со множеством поверхностей, разделенных на разные зоны записи. Чтобы скрыть эту сложность от операционной системы, современные диски дают упрощенное представление о своей геометрии в виде последовательности *В логических блоков* размером с сектор, пронумерованных как 0, 1, ...,  $B - 1$ . Все необходимые преобразования между логическими блоками и физическими секторами выполняет маленькое программно-аппаратное устройство внутри диска.

Когда операционной системе требуется выполнить операцию ввода/вывода, например прочитать сектор в основную память, то она посылает команду контроллеру диска



с запросом прочитать определенный логический блок с определенным номером. Защищая в контроллер программа выполняет быстрый поиск по номеру логического блока в таблице и определяет триаду (*поверхность, дорожка, сектор*), соответствующую физическому сектору. Аппаратная часть контроллера перемещает головки в соответствующий цилиндр, дожидается прохождения нужного сектора, записывает прочитанные головкой биты в небольшой буфер и затем копирует их в основную память.

#### Емкость отформатированного диска

Перед тем как диск можно будет использовать для хранения данных, его необходимо *отформатировать* с помощью контроллера диска. Эта процедура заключается в заполнении промежутков между секторами информацией, идентифицирующей секторы, выявлении любых цилиндров с дефектами поверхности и выделении резервных цилиндров в каждой зоне, чтобы их можно было использовать, если один или более цилиндров в зоне выйдет из строя в течение срока эксплуатации диска. Емкость отформатированного диска, указанная производителем, меньше максимальной емкости из-за наличия этих резервных цилиндров.

#### Упражнение 6.4 (решение в конце главы)

Предположим, что файл размером 1 Мбайт, состоящий из логических блоков размером 512 байт, хранится на диске со следующими характеристиками:

Параметр	Значение
Скорость вращения	10 000 об/мин
Среднее время позиционирования ( $T_{avg\ seek}$ )	5 мс
Среднее количество секторов на дорожке	1000
Поверхности	4
Размер сектора	512 байт

Для каждого из приведенных ниже случаев предположим, что программа читает логические блоки файла последовательно один за другим и время позиционирования головок над первым блоком равно  $T_{avg\ seek} + T_{avg\ rotation}$ .

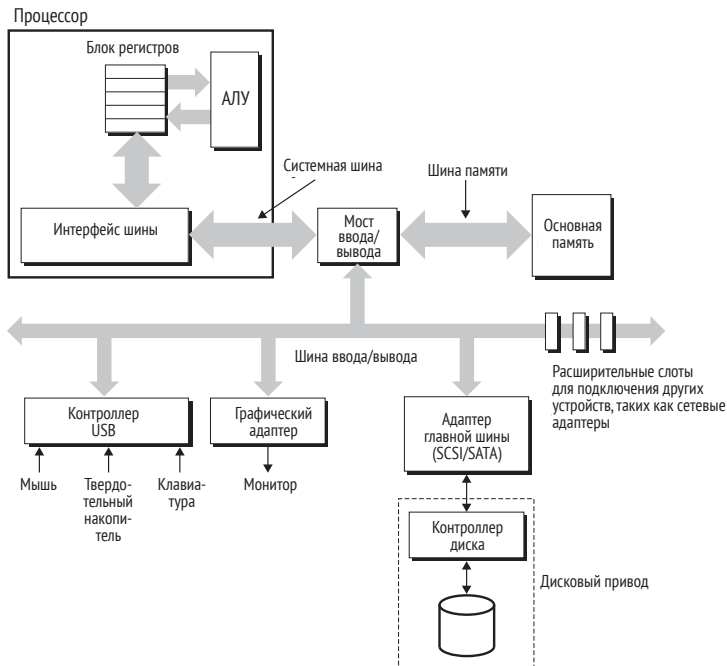
1. *Лучший случай*: оцените оптимальное время (в мс), необходимое для чтения файла, с учетом наилучшего возможного отображения логических блоков в секторы диска (т. е. последовательного).
2. *Произвольный случай*: оцените время (в мс), необходимое для чтения файла, если блоки отображаются в случайные секторы диска.

## Подключение устройств ввода/вывода

Устройства ввода/вывода, такие как видеокарты, мониторы, мыши, клавиатуры и диски, подключаются к процессору и основной памяти с помощью *шины ввода/вывода*. В отличие от системной шины и шины памяти, архитектура которых зависит от процессора, шины ввода/вывода спроектированы так, чтобы не зависеть от процессора. На рис. 6.10 показана типичная шина ввода/вывода, которая связывает процессор, основную память и устройства ввода/вывода.

### Новые достижения в разработке архитектур шин ввода/вывода

Шина ввода/вывода на рис. 6.10 – это простая абстракция, которая позволяет добавить конкретики в обсуждение без привязки к деталям какой-то конкретной системы. Эта абстракция основана на шине для *подключения периферийных компонентов* (Peripheral Component Interconnect, PCI), которая была популярна примерно до 2010 года. В модели PCI все устройства, имеющиеся в системе, совместно используют шину, но только одно может получить доступ к ее проводникам в каждый момент времени. В современных системах общая шина PCI была заменена шиной *PCI Express* (PCIe), которая представляет набор высокоскоростных последовательных соединений точка–точка, установленных посредством коммутаторов, которые напоминают коммутаторы сетей Ethernet (с ними вы познакомитесь в главе 11). Шина PCIe с максимальной пропускной способностью 16 Гбайт/с на порядок быстрее шины PCI, максимальная пропускная способность которой составляет 533 Мбайт/с. За исключением большей пропускной способности, другие различия между этими шинами не видны прикладным программам, поэтому далее в книге мы будем использовать простую абстракцию общей шины.



**Рис. 6.10.** Структура типичной шины, связывающей процессор, основную память и устройства ввода/вывода

Шина ввода/вывода действует медленнее, чем системная шина и шина памяти, зато к ней можно подключить широкий спектр устройств ввода/вывода. Например, к шине на рис. 6.10 подключены устройства трех разных типов.

- Контроллер *универсальной последовательной шины* (Universal Serial Bus, USB) служит каналом для устройств, подключаемых к шине USB, которая широко используется для подключения различных периферийных устройств ввода/вывода, включая клавиатуры, мыши, модемы, цифровые камеры, игровые контроллеры,

принтеры, внешние и твердотельные диски. Шина USB 3.0 имеет максимальную пропускную способность 625 Мбит/с, а USB 3.1 – 1250 Мбит/с.

- *Графическая карта* (или *адаптер*) содержит аппаратную и программную логику, отвечающую за вывод пикселей на экран монитора «от лица» процессора.
- *Адаптер главной шины* (host bus adapter), посредством которого можно подключить несколько дисков к шине ввода/вывода с использованием протокола связи, определяемого конкретным *интерфейсом главной шины*. Два самых популярных таких интерфейса для дисков – это SCSI (произносится как «скази») и SATA (произносится как «сата»). Диски SCSI обычно быстрее и дороже, чем диски SATA. Адаптер главной шины SCSI (его часто называют *контроллером SCSI*) может поддерживать несколько дисков, в отличие от адаптеров SATA, которые поддерживают возможность подключения только одного диска.

Дополнительные устройства, такие как *сетевые адаптеры*, можно подключить к шине ввода/вывода путем простого включения адаптера в свободный разъем *расширительного слота* на материнской плате, обеспечивающий прямое электрическое подключение к шине.

## Доступ к дискам

Подробное описание особенностей работы устройств ввода/вывода и их программирования выходит за рамки этой книги, но мы считаем важным познакомить вас с общей идеей. Например, на рис. 6.11 показано, какие этапы выполняет процессор при чтении данных с диска.

Процессор посылает команды устройствам ввода/вывода, используя прием, называемый *ввод/вывод с отображением в память* (рис. 6.11 (а)). В системе, поддерживающей ввод/вывод с отображением в память, часть адресного пространства зарезервирована для взаимодействий с устройствами ввода/вывода. Каждый из этих адресов называется *портом ввода/вывода*. Каждое устройство отображается в один или несколько портов, когда подключается к шине.

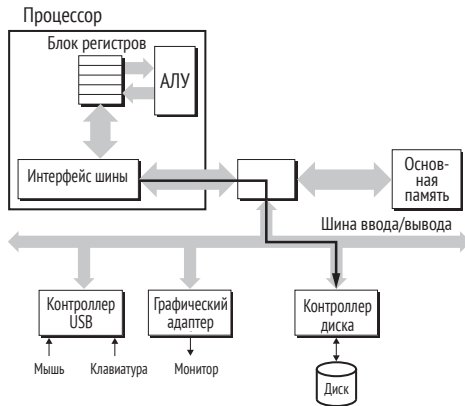
Например, предположим, что контроллер диска отображается в порт 0xа0. Процессор может инициировать операцию чтения с диска, выполнив три инструкции сохранения в память с адресом 0xа0: первая из этих инструкций записывает слово команды, информирующее диск о начале операции чтения, и другие параметры, такие как необходимость сгенерировать прерывание по окончании чтения. (Прерывания обсуждаются в разделе 8.1.) Вторая инструкция записывает номер логического блока, который нужно прочитать. Третья инструкция сообщает адрес в основной памяти, куда следует поместить содержимое прочитанного сектора.

Обычно после отправки запроса, пока диск выполняет чтение, процессор переходит к решению других задач. Напомню, что процессор с тактовой частотой в 1 ГГц и продолжительностью цикла синхронизации 1 нс теоретически может выполнить 16 млн инструкций за 16 мс, которые необходимы диску, чтобы прочитать сектор. Простое ожидание, пока операция выполнится, привело бы к напрасному расходованию вычислительных ресурсов.

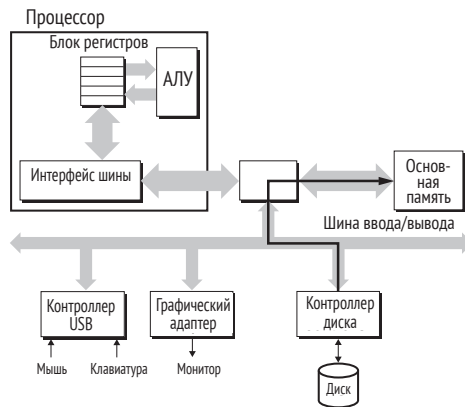
Получив команду от процессора, контроллер диска преобразует номер логического блока в адрес сектора, читает содержимое этого сектора и переносит содержимое непосредственно в основную память без вмешательства процессора (рис. 6.11 (b)). Этот процесс, когда устройство самостоятельно выполняет транзакцию чтения или записи, называется *прямым доступом к памяти* (Direct Memory Access, DMA). А передача данных называется *передачей DMA*.

По завершении передачи DMA и благополучного сохранения содержимого дискового сектора в основной памяти контроллер диска уведомляет процессор, генерируя сигнал прерывания (рис. 6.11 (c)). После этого процессор прекращает работу над текущей за-

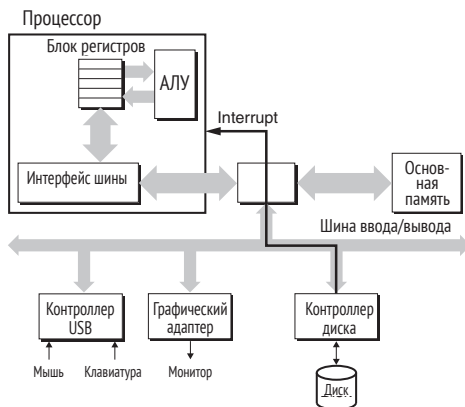
дачей и переходит к выполнению системной процедуры обработки прерывания. Эта процедура фиксирует факт завершения ввода/вывода и возвращает управление в точку, где работа процессора была прервана.



(а) Процессор инициирует чтение с диска, записывая команду, номер логического блока и адрес назначения в памяти в порт ввода/вывода, отображенный в память, который связан с диском



(б) Контроллер диска читает сектор и выполняет передачу DMA, сохраняя прочитанные данные в основную память



(с) Когда передача DMA завершается, контроллер диска генерирует прерывание, чтобы уведомить процессор о завершении операции

**Рис. 6.11.** Чтение сектора диска

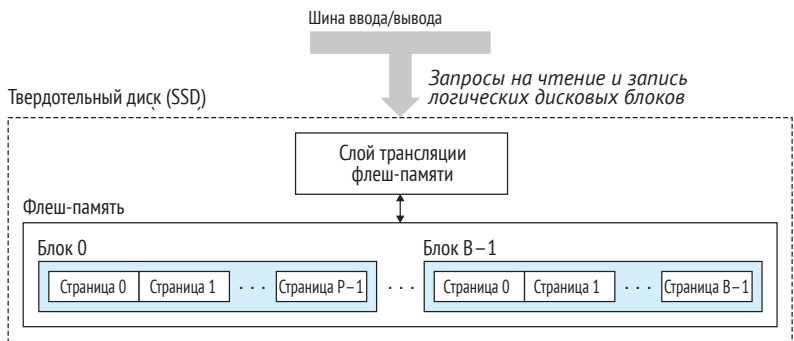
**Характеристики коммерческого диска**

Производители дисков публикуют на своих сайтах массу полезной технической информации. Например, если зайти на сайт компании Seagate, то там можно найти следующую информацию (и не только!) о популярных дисках Barracuda 7400 ([Seagate.com](http://Seagate.com)).

Атрибут геометрии	Значение	Атрибут геометрии	Значение
Диаметр пластин	3.5 дюйма	Скорость вращения	7200 об/мин
Емкость отформатированного диска	3 Тбайт	Средняя задержка из-за вращения	4.16 мс
Пластина	3	Среднее время позиционирования	8.5 мс
Поверхности	6	Время позиционирования между соседними дорожками	1.0 мс
Логические блоки	5 860 533 168	Средняя скорость передачи	156 Мбайт/с
Размер логического блока	512 байт	Максимальная устойчивая скорость передачи	210 Мбайт/с

**6.1.3. Твердотельные диски**

Твердотельный диск (SSD) – это технология хранения, основанная на флеш-памяти (раздел 6.1.1), которая в некоторых ситуациях является довольно привлекательной альтернативой обычному вращающемуся диску. На рис. 6.12 показано общее устройство такого диска. Устройство SSD включается в стандартный слот для дисков на шине ввода/вывода (обычно USB или SATA) и ведет себя как любой другой диск, обрабатывая запросы от процессора на чтение и запись логических дисковых блоков. Устройство SSD состоит из одной или нескольких микросхем флеш-памяти, которые заменяют механический вращающийся диск, и *устройства трансляции адресов флеш-памяти* (Flash Translation Layer, FTL) – аппаратно-программного микроустройства, играющего ту же роль, что и контроллер диска, и преобразующего запросы с номерами логических блоков в физические адреса в базовом устройстве.



**Рис. 6.12.** Твердотельный диск (SSD)

В табл. 6.2 перечислены рабочие характеристики типичного твердотельного диска. Обратите внимание, что для SSD скорость чтения выше скорости записи. Эта разни-

ца обусловлена фундаментальным свойством базовой флеш-памяти. Как показано на рис. 6.12, флеш-память состоит из последовательности  $B$  блоков, каждый из которых имеет  $P$  страниц. Обычно страницы имеют размер от 512 байт до 4 Кбайт, а блок состоит из 32–128 страниц и имеет общий размер от 16 Кбайт до 512 Кбайт. Чтение и запись данных происходят целыми страницами. Страница может быть записана только после *стирания* всего блока, которому она принадлежит (обычно это означает установку всех битов в блоке в 1). Однако после стирания блока во все остальные его страницы можно записать данные без стирания. Обычно блок выдерживает порядка 100 000 циклов записи. После этого блок изнашивается и становится непригодным для использования.

**Таблица 6.2.** Технические характеристики коммерческого твердотельного диска. Источник: спецификация продукта Intel SSD 730 [53]. IOPS – количество операций ввода/вывода в секунду. Показатели пропускной способности предполагают чтение/запись блоками по 4 Кбайт. (Спецификация продукта Intel SSD 730. Корпорация Intel. 52)

Чтение		Запись	
Пропускная способность при чтении последовательных блоков	550 Мбайт/с	Пропускная способность при записи последовательных блоков	470 Мбайт/с
Пропускная способность при чтении произвольных блоков	89 000 IOPS	Пропускная способность при записи произвольных блоков	74 000 IOPS
Пропускная способность при чтении произвольных блоков	365 Мбайт/с	Пропускная способность при записи произвольных блоков	303 Мбайт/с
Среднее время доступа при чтении последовательных блоков	50 мкс	Среднее время доступа при записи последовательных блоков	60 мкс

Запись произвольных блоков выполняется медленнее по двум причинам. Во-первых, стирание блока занимает относительно много времени, примерно 1 мс, что на порядок больше, чем требуется для доступа к странице. Во-вторых, если операция записи пытается изменить страницу  $p$ , содержащую данные (т. е. не все страницы в блоке), то любые страницы в этом блоке, содержащие данные, необходимо скопировать в новый (предварительно стертый) блок. Производители разработали сложную логику в слое трансляции флеш-памяти, которая пытается компенсировать высокую стоимость стирания блоков и минимизировать количество копирований при записи, но маловероятно, что когда-нибудь запись в произвольные блоки будет работать так же быстро, как и чтение.

SSD имеют ряд преимуществ перед вращающимися дисками. Они сконструированы на основе полупроводниковой памяти и не содержат движущихся частей, поэтому имеют гораздо меньшее время доступа, чем вращающиеся диски, потребляют меньше электроэнергии и надежнее. Однако есть и недостатки. Первый – износ блоков флеш-памяти при перезаписи. Логика компенсации в слое трансляции флеш-памяти пытается максимально увеличить срок службы каждого блока за счет равномерного распределения стирания по всем блокам. И на самом деле эта логика работает настолько хорошо, что до полного износа SSD можно эксплуатировать много лет (упражнение 6.5). Во-вторых, стоимость хранения байта в твердотельных накопителях примерно в 30 раз выше, чем во вращающихся дисках, и их емкость намного меньше. Однако цены на твердотельные диски быстро снижаются по мере роста их популярности, и разрыв между ними сокращается.

Твердотельные диски полностью заменили вращающиеся диски в портативных музыкальных устройствах, широко используются в ноутбуках и даже начали появляться в настольных компьютерах и серверах. Вращающиеся диски еще долго никуда не денутся, но уже ясно, что твердотельные диски являются важной альтернативой.

**Упражнение 6.5 (решение в конце главы)**

Как отмечалось выше, одним из недостатков твердотельных дисков является износ флеш-памяти. Например, для SSD, характеристики которого перечислены в табл. 6.2, компания Intel гарантирует возможность записи на диск до 128 петабайт ( $128 \times 10^{15}$  байт) до того, как наступит износ. Учитывая это утверждение, оцените срок службы (в годах) этого твердотельного диска для следующих рабочих нагрузок:

1. *Худший случай для последовательной записи:* запись на SSD производится непрерывно со скоростью 470 Мбайт/с (средняя пропускная способность устройства при последовательной записи).
2. *Худший случай для произвольной записи:* запись на SSD производится непрерывно со скоростью 303 Мбайт/с (средняя пропускная способность устройства при произвольной записи).
3. *Средний случай:* запись на SSD производится со скоростью 20 Гбайт/день (средняя дневная скорость записи, предполагаемая некоторыми производителями компьютеров при моделировании рабочей нагрузки мобильных компьютеров).

## 6.1.4 Тенденции развития технологий хранения

Из нашего обсуждения технологий хранения вытекает несколько важных идей.

- *Разные технологии хранения имеют свои достоинства и недостатки в смысле цены и производительности.* SRAM несколько быстрее DRAM, а DRAM намного быстрее дисков. С другой стороны, чем быстрее хранилище, тем оно дороже. Стоимость хранения байта в SRAM выше, чем в DRAM. Стоимость хранения байта в DRAM намного выше, чем на диске. SSD занимают промежуточное положение между DRAM и вращающимися дисками.
- *Цена и производительность различных технологий хранения данных меняются с разной скоростью.* В табл. 6.3 перечислены цены и производительность технологий хранения данных с 1985 года, вскоре после появления первых ПК. Цифры были взяты из прошлых выпусков отраслевых журналов и из интернета. Они были собраны в рамках неофициального опроса, но все же достаточно наглядно показывают некоторые интересные тенденции.

С 1985 года стоимость и производительность технологии SRAM улучшались практически равномерно. Время доступа и стоимость хранения мегабайта снизились примерно в 100 раз (табл. 6.3 (а)). Однако в отношении DRAM и дисков тенденции гораздо более существенные и имеют расхождения. Так, стоимость хранения мегабайта в DRAM снизилась в 44 000 раз (более чем на четыре порядка!), а время доступа к DRAM уменьшилось всего в 10 раз (табл. 6.3 (b)). Тенденции развития дисковых технологий примерно повторяют тенденции развития DRAM, но более драматично. Так, стоимость хранения мегабайта в дисковом хранилище с 1985 года упала более чем в 3 млн раз (более чем на шесть порядков!), а время доступа улучшилось намного меньше, всего в 25 раз (табл. 6.3 (с)). Эти поразительные долгосрочные тенденции подчеркивают основную истину технологий памяти и дисков: увеличить плотность (и, соответственно, снизить стоимость) намного проще, чем уменьшить время доступа.

- *Производительность DRAM и дисков отстает от производительности процессора.* Как показано в табл. 6.3 (d), время цикла синхронизации ЦП уменьшилось в 500 раз между 1985 и 2010 гг. Если посмотреть на *эффективное время цикла*, которое определяется как время цикла отдельного процессора, деленное на количество ядер, то улучшение между 1985 и 2010 гг. еще больше – в 2000 раз.

**Таблица 6.3.** Тенденции развития технологий хранения и обработки. В Core i7 2010 года используется ядро процессора Nehalem. В Core i7 2015 года используется ядро Haswell

**(а) Тенденции развития SRAM**

Метрика	1985	1990	1995	2000	2005	2010	2015	2015:1985
\$/Мбайт	2900	320	256	100	75	60	25	116
Доступ (нс)	150	35	15	3	2	1.5	1.3	115

**(b) Тенденции развития DRAM**

Метрика	1985	1990	1995	2000	2005	2010	2015	2015:1985
\$/Мбайт	880	100	30	1	0.1	0.06	0.02	44 000
Доступ (нс)	200	100	70	60	50	40	20	10
Типичный размер (Мбайт)	0.256	4	16	64	2000	8000	16 000	62 500

**(с) Тенденции развития вращающихся дисков**

Метрика	1985	1990	1995	2000	2005	2010	2015	2015:1985
\$/Гбайт	100 000	8000	300	10	5	0.3	0.03	3 333 333
Среднее время позиционирования (мс)	75	28	10	8	5	3	3	25
Типичный размер (Гбайт)	0.01	0.16	1	20	160	1500	3000	300 000

**(d) Тенденции развития процессоров**

Метрика	1985	1990	1995	2000	2003	2005	2010	2015	2015:1985
Процессоры Intel	80286	80386	Pentium	Pentium III	Pentium 4	Core 2	Core i7 (n)	Core i7 (h)	–
Тактовая частота (МГц)	6	20	150	600	3300	2000	2500	3000	500
Время цикла (нс)	166	50	6	1.6	0.3	0.5	0.4	0.33	500
Ядро	1	1	1	1	1	2	4	4	4
Эффективное время цикла (нс)	166	50	6	1.6	0.3	0.25	0.1	0.08	2075

Расщепление кривой производительности процессоров примерно в 2003 году отражает появление многоядерных процессоров (как описывается во врезке «Когда уменьшение времени цикла прекратилось: появление многоядерных процессоров» ниже). После этого продолжительность цикла отдельных ядер фактически немного увеличилась и только спустя годы снова начала уменьшаться, хотя и не так быстро, как раньше.

Обратите внимание, что, несмотря на отставание темпов роста производительности SRAM, эта технология почти не уступает производительности процессоров. А вот разрыв между производительностью DRAM и дисков и производительностью процессоров фактически увеличивается. До появления многоядерных процессоров примерно в 2003 году этот разрыв был функцией задержки – время доступа к DRAM и диску уменьшалось медленнее, чем время цикла синхронизации отдельного процессора. Однако с появлением многоядерных процессоров этот разрыв в производительности в большей



степени стал зависеть от пропускной способности, когда несколько ядер отправляют запросы к DRAM и диску параллельно.

Различные тенденции достаточно хорошо видны на рис. 6.13, где в полулогарифмическом масштабе показаны время доступа и продолжительность цикла синхронизации из табл. 6.3.

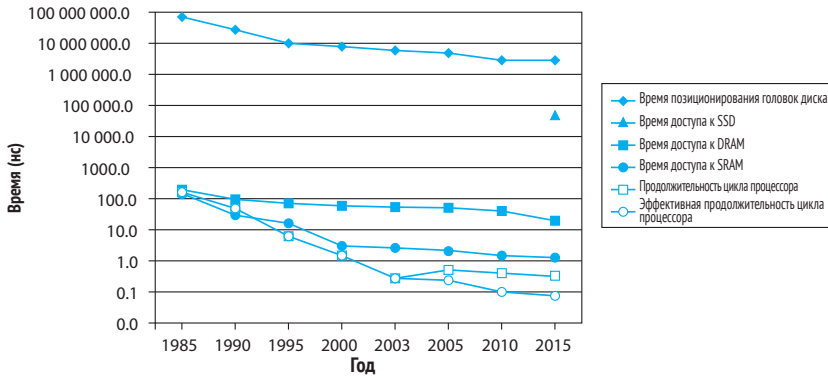


Рис. 6.13. Увеличивающийся разрыв между быстродействием DRAM, дисков и процессоров

#### Когда уменьшение времени цикла прекратилось: появление многоядерных процессоров

В истории развития вычислительной техники имели место некоторые необычные события, которые привели к глубоким изменениям в отрасли и в мире. Интересно отметить, что эти события происходили примерно раз в десятилетие: разработка Fortran в 1950-х годах, появление IBM 360 в начале 1960-х, зарождение интернета (тогда называвшегося ARPANET) в начале 1970-х, появление IBM PC в начале 1980-х и создание Всемирной паутины (World Wide Web) в начале 1990-х.

Последнее такое событие случилось в начале XXI века, когда производители компьютеров уперлись в так называемую «стену потребляемой мощности», препятствующую дальнейшему увеличению тактовой частоты процессоров, потому что это было сопряжено с потреблением слишком большого количества энергии. Тогда было решено продолжить наращивать производительность за счет размещения на одном кристалле нескольких процессорных ядер, каждое из которых является полноценным процессором, способным выполнять программы независимо и параллельно с другими ядрами. Такой *многоядерный* подход действительно стал решением проблемы, потому что общая мощность, потребляемая процессором, пропорциональна  $P = fCV^2$ , где  $f$  – тактовая частота,  $C$  – емкость, а  $V$  – напряжение. Емкость  $C$  примерно пропорциональна площади, поэтому мощность, потребляемая несколькими ядрами, остается постоянной, пока общая площадь не изменяется. Пока, согласно закону Мура, размеры функциональных блоков продолжают сокращаться с экспоненциальной скоростью, количество ядер в процессорах и их эффективная производительность будут продолжать расти.

С этого момента компьютеры будут работать быстрее не за счет увеличения тактовой частоты, а за счет увеличения количества ядер в каждом процессоре, а также за счет внедрения архитектурных инноваций, повышающих эффективность программ, выполняющихся на этих ядрах. Эта тенденция отчетливо видна на рис. 6.13. Продолжительность цикла синхронизации достигла своего минимума в 2003 году, после чего немного увеличилась и снова начала снижаться, но медленнее, чем раньше. Однако благодаря появлению многоядерных процессоров (двухъядерных в 2004 и четырехъядерных в 2007) эффективная продолжительность цикла продолжает уменьшаться и почти достигла прежнего уровня.

Как рассказывается в разделе 6.4, в современных компьютерах широко используются кэши на основе SRAM, чтобы хоть немного сократить разрыв между производительностью процессора и памяти. Этот прием дает положительный результат благодаря фундаментальному свойству прикладных программ, известному как *локальность*, которое мы обсудим далее.

#### Упражнение 6.6 (решение в конце главы)

Опираясь на динамику цен с 2005 по 2015 год, что приводится в табл. 6.3 (с), оцените, в каком году вы сможете купить вращающийся диск емкостью в один петабайт ( $10^{15}$  байт) за 500 долларов, если предположить, что доллар не будет подвержен инфляции.

## 6.2. Локальность

Хорошо написанные компьютерные программы, как правило, имеют хорошую *локальность*. То есть они стремятся снова и снова обращаться к одним и тем же данным или к данным, хранящимся в памяти по соседству. Эта тенденция, называемая *принципом локальности*, оказывает значительное влияние на архитектуру и производительность аппаратного и программного обеспечения.

Локальность имеет две формы: *временная локальность* и *пространственная локальность*. В программе с хорошей временной локальностью одни и те же элементы данных многократно используются в течение короткого периода времени. В программе с хорошей пространственной локальностью чаще всего используются данные, расположенные по соседству с уже использовавшимися данными.

Программисты должны понимать принцип локальности, потому что *программы с хорошей локальностью обычно выполняются быстрее, чем программы с плохой локальностью*. Все уровни современных компьютерных систем, от аппаратного обеспечения до операционных систем и отдельных приложений, спроектированы с учетом использования принципа локальности. На уровне аппаратного обеспечения принцип локальности позволяет проектировщикам ускорить доступ к основной памяти за счет внедрения небольших быстродействующих устройств памяти, называемых *кэшами*, в которых хранятся последние использовавшиеся блоки инструкций и данных. На уровне операционных систем принцип локальности позволяет системе использовать основную память как кэш для хранения последних использовавшихся блоков памяти из виртуального адресного пространства. Аналогично операционная система использует основную память для кеширования дисковых блоков. Принцип локальности также играет ключевую роль при проектировании прикладных программ. Например, веб-браузеры используют временную локальность, сохраняя в кеше на локальном диске последние использовавшиеся документы. В высокопроизводительных веб-серверах последние запрашивавшиеся документы сохраняются в специализированных дисковых кэшах, откуда они передаются запрашивающим их пользователям без участия самого сервера.

### 6.2.1. Локальность обращений к данным программы

Рассмотрим простую функцию в листинге 6.1, суммирующую элементы вектора. Обладает ли эта функция хорошей локальностью? Для ответа на этот вопрос нужно исследовать, как она обращается к каждой переменной. В данном примере обращения к переменной `sum` происходят в каждой итерации, то есть налицо хорошая временная локальность. С другой стороны, поскольку `sum` – скаляр, в отношении ее отсутствует пространственная локальность.

**Листинг 6.1.** Функция с хорошей локальностью

```

1 int sumvec(int v[N])
2 {
3     int i, sum = 0;
4
5     for (i = 0; i < N; i++)
6         sum += v[i];
7     return sum;
8 }
```

Как показано в табл. 6.4, элементы вектора  $v$  читаются последовательно, один за другим, в том порядке, в каком они хранятся в памяти (для удобства предполагается, что первый элемент массива хранится в ячейке памяти с адресом 0). То есть относительно переменной  $v$  функция имеет хорошую пространственную локальность, но плохую временную локальность, потому что доступ к каждому элементу вектора происходит только один раз. Таким образом, функция `sumvec` имеет хорошую временную и пространственную локальность в отношении той или иной переменной.

**Таблица 6.4.** Порядок использования вектора  $v$  ( $N = 8$ ). Обратите внимание, что доступ к элементам вектора осуществляется в том же порядке, в каком они хранятся в памяти

Адрес	0	4	8	12	16	20	24	28
Содержимое	$v_0$	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$
Порядок доступа	1	2	3	4	5	6	7	8

Говорят, что функция, такая как `sumvec`, последовательно обращающаяся к каждому элементу вектора, следует *шаблону обращений с шагом 1* (с учетом размера элемента). Такой шаблон обращений с шагом 1 мы будем иногда называть *шаблоном последовательных обращений*. Шаблон с последовательными обращениями к каждому  $k$ -му элементу вектора называется *шаблоном обращений с шагом  $k$* . Шаблон обращений с шагом 1 – широко распространенный и важный источник пространственной локальности в программах. Вообще говоря, с увеличением шага пространственная локальность ухудшается.

Величина шага также является важным аспектом для программ, обращающихся к многомерным массивам. Рассмотрим функцию `sumarrayrows` в листинге 6.2, суммирующую элементы двумерного массива. Двойной вложенный цикл читает элементы массива по строкам. То есть внутренний цикл читает элементы первой строки, затем второй и т. д. Функция `sumarrayrows` обладает хорошей пространственной локальностью, потому что обращается к элементам массива в том же порядке, в каком они хранятся в памяти (табл. 6.5). То есть она следует шаблону последовательных обращений с шагом 1 – великопная пространственная локальность.

**Листинг 6.2.** Еще одна функция с хорошей локальностью

```

1 int sumarrayrows(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (i = 0; i < M; i++)
6         for (j = 0; j < N; j++)
7             sum += a[i][j];
8     return sum;
9 }
```

**Таблица 6.5.** Порядок использования массива  $a$  ( $M = 2, N = 3$ ). Этот шаблон дает хорошую пространственную локальность, потому что доступ к элементам массива осуществляется в том же порядке, в каком они хранятся в памяти

Адрес	0	4	8	12	16	20
Содержимое	$a_{00}$	$a_{01}$	$a_{02}$	$a_{10}$	$a_{11}$	$a_{12}$
Порядок доступа	1	2	3	4	5	6

Внешне тривиальные изменения в программе могут оказать серьезное влияние на ее локальность. Например, функция `sumarraycols` (листинг 6.3) вычисляет тот же результат, что и функция `sumarrayrows` в листинге 6.2. Единственное – циклы по строкам и столбцам поменялись местами. Как это повлияет на локальность программы?

**Листинг 6.3.** Функция с плохой пространственной локальностью

```

1 int sumarraycols(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (j = 0; j < N; j++)
6         for (i = 0; i < M; i++)
7             sum += a[i][j];
8     return sum;
9 }
```

Функция `sumarraycols` имеет плохую пространственную локальность, потому что просматривает массив по столбцам, а не по строкам. Поскольку массивы в языке C хранятся в памяти построчно, получается, что эта функция следует шаблону обращений с шагом  $N$ , как показано в табл. 6.6.

**Таблица 6.6.** Порядок использования массива  $a$  ( $M = 2, N = 3$ ). Этот шаблон дает плохую пространственную локальность, потому что доступ к элементам массива осуществляется с шагом  $N$

Адрес	0	4	8	12	16	20
Содержимое	$a_{00}$	$a_{01}$	$a_{02}$	$a_{10}$	$a_{11}$	$a_{12}$
Порядок доступа	1	3	5	2	4	6

### 6.2.2. Локальность выборки инструкций

Инструкции программы хранятся в памяти и должны выбираться (читаться) процессором, поэтому локальность программы можно также оценить с точки зрения последовательности выборки инструкций. Например, в листинге 6.1 инструкции в теле цикла `for` выполняются последовательно, как они хранятся в памяти, следовательно, этот цикл обладает хорошей пространственной локальностью. Учитывая, что тело цикла выполняется многократно, он также обладает хорошей временной локальностью.

Важным свойством кода, отличающим его от данных, является невозможность его модификации во время выполнения – процессор только читает инструкции из памяти, но никогда не изменяет их.

### 6.2.3. В заключение о локальности

В этом разделе мы обсудили фундаментальное понятие локальности и обозначили некоторые простые правила оценки локальности в программе:

- программы, многократно обращающиеся к одной и той же переменной, обладают хорошей локальностью;
- для программ с шаблоном обращений с шагом  $k$ : чем меньше шаг, тем лучше пространственная локальность. Программы с шаблоном обращений с шагом 1 имеют хорошую пространственную локальность. Программы, «перепрыгивающие» через ячейки памяти, обладают худшей пространственной локальностью;
- циклы имеют хорошую временную и пространственную локальность в отношении выборки инструкций. Чем короче тело цикла и больше число итераций цикла, тем лучше локальность.

Далее в этой главе, после обсуждения устройства кеш-памяти и особенностей ее работы, мы расскажем, как определяется локальность с точки зрения попаданий и промахов кеша. Мы объясним, почему программы с хорошей локальностью, как правило, выполняются быстрее, чем программы с плохой локальностью. Как бы то ни было, программистам важно знать и понимать, как оценивать исходный код с точки зрения принципа локальности.

#### Упражнение 6.7 (решение в конце главы)

Поменяйте местами циклы в следующей функции, чтобы она просматривала трехмерный массив, следуя шаблону обращений с шагом 1:

```

1 int sumarray3d(int a[N][N][N])
2 {
3     int i, j, k, sum = 0;
4
5     for (i = 0; i < N; i++) {
6         for (j = 0; j < N; j++) {
7             for (k = 0; k < N; k++) {
8                 sum += a[k][i][j];
9             }
10        }
11    }
12    return sum;
13 }
```

#### Упражнение 6.8 (решение в конце главы)

Три функции в листингах 6.4–6.7 выполняют ту же операцию, но с разными степенями пространственной локальности. Упорядочьте эти функции по возрастанию пространственной локальности. Обоснуйте выбранный порядок.

**Листинг 6.4.** Массив структур

```

1 #define N 1000
2
3 typedef struct {
4     int vel[3];
5     int acc[3];
6 } point;
7
8 point p[N];
```

**Листинг 6.5.** Функция clear1

```

1 void clear1(point *p, int n)
2 {
3     int i, j;
4
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < 3; j++)
7             p[i].vel[j] = 0;
8         for (j = 0; j < 3; j++)
9             p[i].acc[j] = 0;
10    }
11 }
```

Листинг 6.6. Функция clear2

```
1 void clear2(point *p, int n)
2 {
3     int i, j;
4
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < 3; j++) {
7             p[i].vel[j] = 0;
8             p[i].acc[j] = 0;
9         }
10    }
11 }
```

Листинг 6.7. Функция clear3

```
1 void clear3(point *p, int n)
2 {
3     int i, j;
4
5     for (j = 0; j < 3; j++) {
6         for (i = 0; i < n; i++)
7             p[i].vel[j] = 0;
8         for (i = 0; i < n; i++)
9             p[i].acc[j] = 0;
10    }
11 }
```

### 6.3. Иерархия памяти

В разделах 6.1 и 6.2 описываются некоторые фундаментальные свойства технологии хранения информации и приемы разработки программного обеспечения.

*Технологии хранения*

Различные технологии хранения информации характеризуются широким диапазоном времени доступа. Более быстросействующие технологии стоят дороже менее быстросействующих и имеют меньшую емкость. Расхождение между производительностью процессоров и основной памяти увеличивается.

*Программное обеспечение*

Хорошо продуманные программы имеют лучшую локальность.

При удачном совпадении всех этих фундаментальных свойств аппаратное и программное обеспечение прекрасно дополняют друг друга. Их комплементарная природа предполагает подход к организации систем хранения, который называется *иерархией памяти* и используется во всех современных компьютерных системах. На рис. 6.14 показана типичная иерархия памяти.



Рис. 6.14. Иерархия памяти

Вообще говоря, чем ниже устройства хранения находятся в иерархии, тем они медленнее, дешевле и объемнее. На самом высоком уровне (L0) находится небольшое число быстродействующих регистров процессора, доступ к которым процессор может получить за один цикл синхронизации. Далее следуют устройства кеш-памяти SRAM небольшого объема, доступ к которым происходит за несколько циклов синхронизации. Потом следует основная память DRAM большого объема, доступ к которой происходит за десятки или сотни циклов синхронизации. Затем идут медленные локальные диски огромной емкости. Наконец, некоторые системы имеют дополнительный уровень дисковых хранилищ на удаленных серверах, доступных по сети. Например, распределенные файловые системы AFS (Andrew File System) и NFS (Network File System) позволяют программам обращаться к файлам на удаленных сетевых серверах. Аналогично Всемирная паутина позволяет программам использовать удаленные файлы, хранящиеся на веб-серверах, разбросанных по всему миру.

### Другие иерархии памяти

Мы показали вам один пример иерархии памяти, но есть и другие возможные комбинации. Например, многие сайты, в том числе центры обработки данных Google, создают резервные копии локальных дисков на архивных магнитных лентах. На некоторых из этих площадок операторы вручную меняют ленты в ленточных накопителях по мере необходимости. На других площадках эту задачу решают роботы. В любом случае набор лент тоже образует уровень иерархии памяти ниже уровня локальных дисков, и к нему применяются те же общие принципы. Ленты обходятся дешевле, чем диски, что позволяет архивировать содержимое локальных дисков. Но за дешевизну приходится платить большим временем доступа к лентам. Другой пример: твердотельные диски играют все более важную роль в иерархии памяти, заполняя пропасть между DRAM и вращающимися дисками.

## 6.3.1. Кеширование в иерархии памяти

*Кеш* – небольшое и быстродействующее устройство памяти, играющее роль промежуточного хранилища для данных, хранящихся на более емких и медленных устройствах. Процесс использования кеш-памяти называется *кешированием*.

Основная идея иерархии памяти заключается в том, что более быстрое и менее емкое устройство хранения служит кешем для менее быстрого и более емкого устройства уровнем ниже. Иными словами, каждый следующий уровень в иерархии кеширует данные, хранящиеся уровнем ниже. Например, локальный диск служит кешем для файлов (например, веб-страниц), находящихся на дисках удаленных серверов в сети; основная память служит кешем для данных, хранящихся на локальных дисках и т. д. до самого маленького кеша из всех существующих – набора регистров процессора.

На рис. 6.15 схематически показана общая идея кеширования в иерархии памяти. Устройство хранения на уровне  $k + 1$  разделено на смежные фрагменты с данными, называемые *блоками*. Каждый блок обладает уникальным адресом или именем, отличающим его от других блоков. Блоки могут быть фиксированного (обычный случай) или переменного размера (например, HTML-файлы, хранимые на веб-серверах). Например, устройство хранения, показанное на рис. 6.15 на уровне  $k + 1$ , поделено на 16 блоков фиксированного размера с номерами от 0 до 15.

Аналогично устройство хранения на уровне  $k$  делится на небольшие блоки того же размера, что и блоки на уровне  $k + 1$ . В любой момент кеш на уровне  $k$  содержит копии некоторых блоков из уровня  $k + 1$ . Например, на рис. 6.15 кеш на уровне  $k$  хранит четыре блока – в данный момент блоки 4, 9, 14 и 3.

Данные копируются между уровнями  $k$  и  $k + 1$  *целыми блоками*. Важно понимать, что размер блоков между одной парой уровней может не совпадать с размерами блоков

между другими парами. Например, между уровнями L0 и L1 на рис. 6.14 данные копируются блоками размером в одно слово. А между уровнями L4 и L3 – блоками с размером в сотни или тысячи байт. Вообще говоря, устройства хранения на более низких уровнях в иерархии (и отстоящие дальше от процессора) требуют больше времени для доступа к ним, и поэтому обмен с ними выполняется блоками большего размера, чтобы хоть немного компенсировать затраты на доступ.

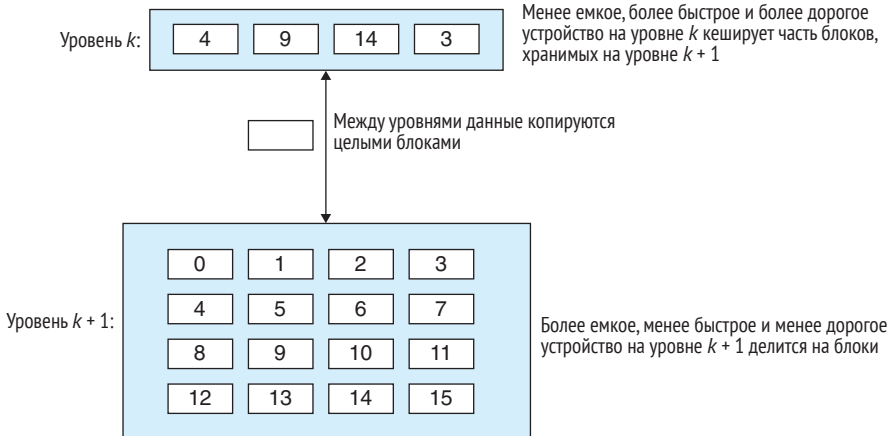


Рис. 6.15. Общая идея кеширования в иерархии памяти

### Попадание в кеш

Когда программе требуется определенный фрагмент данных  $d$ , хранящийся на уровне  $k+1$ , то сначала будет предпринята попытка отыскать  $d$  в одном из блоков, хранящихся в данный момент на уровне  $k$ . Если фрагмент  $d$  обнаруживается в кеше на уровне  $k$  (это называется *попаданием в кеш*), то программа получает  $d$  непосредственно из кеша на уровне  $k$ , причем, благодаря природе иерархии памяти, намного быстрее, чем при чтении с уровня  $k+1$ . Например, программе с хорошей временной локальностью может потребоваться получить фрагмент данных из блока 14 (рис. 6.15), результатом чего становится попадание в кеш уровня  $k$ .

### Промех кеша

С другой стороны, если фрагмент данных  $d$  отсутствует в кеше на уровне  $k$ , тогда возникает так называемый *промах кеша*. В случае промаха кеш на уровне  $k$  извлекает блок с фрагментом  $d$  из кеша на уровне  $k+1$ , возможно, с затиранием существующего блока, если кеш уровня  $k$  полон.

Процесс затирания существующего блока называется *заменой*, или *вытеснением*, блока. Вытесненный блок иногда называется *блоком-жертвой*. Выбор блока для вытеснения определяется *стратегией замены*. Например, кеш с *произвольной стратегией замены* выберет случайный блок-жертву. Кеш со стратегией вытеснения наиболее давно использовавшихся блоков (Least Recently Used, LRU) вытеснит блок, к которому не было обращений дольше всего.

После того как кеш на уровне  $k$  получит блок с уровня  $k+1$ , программа сможет прочитать  $d$  с уровня  $k$ , как и прежде. Например, на рис. 6.15 показано, как происходит чтение фрагмента данных из блока 12. При обращении к кешу на уровне  $k$  произойдет промах, потому что блок 12 в данный момент отсутствует в кеше уровня  $k$ . Как только блок 12 будет скопирован с уровня  $k+1$  на уровень  $k$ , он останется там в ожидании последующих обращений.



## Виды промахов кеша

Иногда имеет смысл разделять промахи кеша по видам. Если кеш на уровне  $k$  пуст, то при обращении к любым данным произойдет промах кеша. Пустой кеш иногда называют *холодным*, и промахи в пустом кеше называются *неизбежными*, или *холодными*, промахами. Отличать холодные промахи важно, потому что они часто являются источниками переходных событий, которые могут не возникать в устойчивом состоянии *разогрева* кеша многократными обращениями к памяти.

Всякий раз, когда промах, кеш на уровне  $k$  должен реализовать некоторую *стратегию размещения*, определяющую место для размещения блока, полученного с уровня  $k + 1$ . Наиболее гибкой является стратегия, позволяющая размещать блок с уровня  $k + 1$  в любом блоке на уровне  $k$ . Для уровней выше в иерархии (находящихся ближе к процессору) и реализованных на аппаратном уровне, где быстроедействие является основным приоритетом, эта стратегия неприменима, так как реализация произвольного размещения блоков в аппаратных кешах требует больших затрат.

По этой причине в аппаратных кешах обычно реализуется более простая стратегия размещения, ограничивающая выбор блока для сохранения данных с предыдущего уровня иерархии небольшим подмножеством (иногда одним элементом) блоков на уровне  $k$ . Например, представьте, что в ситуации на рис. 6.15 блок  $i$  с уровня  $k + 1$  должен помещаться в блок  $(i \bmod 4)$  на уровне  $k$ . Тогда блоки 0, 4, 8 и 12 с уровня  $k + 1$  будут помещаться в блок 0 на уровне  $k$ , блоки 1, 5, 9 и 13 – в блок 1 и т. д. Кстати, в нашем примере на рис. 6.15 кеш на уровне  $k$  использует именно эту стратегию.

Подобные ограничивающие стратегии размещения приводят к промахам, которые называют *конфликтными промахами* – кеш имеет достаточный размер для хранения всех запрашиваемых данных, но из-за ограничений стратегии размещения все равно могут возникать промахи. Например, как показано на рис. 6.15, если программа запросит блок 0, затем блок 8, потом блок 0 и затем опять блок 8, то каждый раз будет возникать промах кеша на уровне  $k$ , даже если этот кеш может хранить до 4 блоков.

Программы часто выполняются как последовательность этапов (циклов), когда каждый этап получает доступ к более или менее постоянному множеству блоков в кеше. Например, вложенный цикл может многократно обращаться к элементам одного и того же массива. Это множество блоков называется *рабочим множеством* этапа. Когда размер рабочего множества превышает размер кеша, то последний испытывает то, что называется *промахами емкости*. То есть емкости кеша недостаточно для хранения данных из конкретного рабочего множества.

## Управление кешем

Как уже отмечалось, в иерархии памяти каждый вышестоящий уровень играет роль кеша для нижестоящего уровня. На каждом уровне должна иметься определенная логика, управляющая кешем, то есть логика, разделяющая нижестоящее устройство хранения на блоки, копирующая блоки между уровнями, определяющая попадание в кеш или промах кеша и обрабатывающая всю эту информацию. Логика управления кешем может быть реализована аппаратными или программными средствами либо их комбинацией.

Например, компилятор управляет блоком регистров – самым верхним уровнем в иерархии. Он определяет, когда генерировать команду загрузки, когда возникает промах, а также выбирает регистры для хранения данных. Кешы на уровнях L1 и L2 управляются встроенной аппаратной логикой. В системе с виртуальной памятью основная память DRAM служит кешем для блоков данных, хранящихся на диске, и управляется комбинацией программных средств операционной системы и аппаратных средств преобразования адресов в процессоре. Для машины с распределенной файловой системой (например, AFS) локальный диск служит кешем, который управляется клиентским процессом AFS, действующим на локальной машине. В большин-

стве случаев кеш работает автоматически и не требует особых или явных действий со стороны программы.

6.3.2. В заключение об иерархии памяти

В заключение можно еще раз отметить, что иерархии памяти базируются на понятии кеширования, потому что медленные устройства дешевле быстрых, а также потому, что программы стремятся обеспечить локальность.

Использование временной локальности

Суть временной локальности заключается в многократном использовании одних и тех же данных. Как только фрагмент данных окажется в кеше после первого промаха, можно ожидать некоторого количества последующих обращений к этим же данным. Кеш работает быстрее, чем устройство хранения уровнем ниже, поэтому эти последующие обращения будут обслуживаться на-много быстрее, чем первоначальный промах.

Использование пространственной локальности

В блоках обычно содержится множество объектов данных. Если программа имеет хорошую пространственную локальность, то можно ожидать, что затраты на ко-пирование блока после промаха будут компенсированы за счет более короткого времени доступа в последующих обращениях к другим объектам в этом же блоке.

В современных системах кеш используется повсеместно: в процессорах, в операци-онных системах, в распределенных файловых системах, а также во Всемирной паутине. Они создаются различными комбинациями аппаратных и программных средств и ими же управляются. Обратите внимание, что в табл. 6.7 упоминаются некоторые еще не описанные термины и аббревиатуры. Мы включили их только для демонстрации ти-пичных устройств кеш-памяти.

**Таблица 6.7.** Повсеместное кеширование в современных компьютерных системах. Аббревиатуры: TLB: Translation Lookaside Buffer (буфер ассоциативной трансляции); MMU: Memory Management Unit (блок управления памятью); OS: Operating System (операционная система); NFS: Network File System (сетевая файловая система)

Тип	Что кеширует	Где кеширует	Задержка (в циклах)	Управляется
Регистры процессора	4- или 8-байтные слова	Регистры в процессоре	0	Компилятором
TLB	Таблицы адресов	В буфере ассоциатив-ной трансляции	0	Аппаратным модулем MMU
Кеш L1	64-байтные блоки	Аппаратный кеш L1	4	Аппаратно
Кеш L2	64-байтные блоки	Аппаратный кеш L2	10	Аппаратно
Кеш L3	64-байтные блоки	Аппаратный кеш L3	50	Аппаратно
Виртуальная память	Страницы по 4 Кбайт	Основная память	200	Аппаратно + OS
Буферный кеш	Фрагменты файлов	Основная память	200	OS
Дисковый кеш	Секторы диска	Контроллер диска	100 000	Аппаратурой контроллера
Сетевой кеш	Фрагменты файлов	Локальный диск	10 000 000	Клиент NFS
Кеш браузера	Веб-страницы	Локальный диск	10 000 000	Веб-браузером
Веб-кеш	Веб-страницы	Диски на удаленных серверах	1 000 000 000	Прокси-сервером

## 6.4. Кеш-память

Иерархии памяти первых компьютерных систем состояли всего из трех уровней: регистров процессора, основной памяти DRAM и дисковых устройств. Однако из-за все увеличивающейся разницы в быстродействии между процессорами и основной памятью проектировщики были вынуждены добавить *кеш-память* SRAM небольшого объема – *кеш L1* (кеш первого уровня) – между блоком регистров процессора и основной памятью, как показано на рис. 6.16. Время доступа к кешу L1 сопоставимо со временем доступа к регистрам и составляет примерно 4 цикла синхронизации.

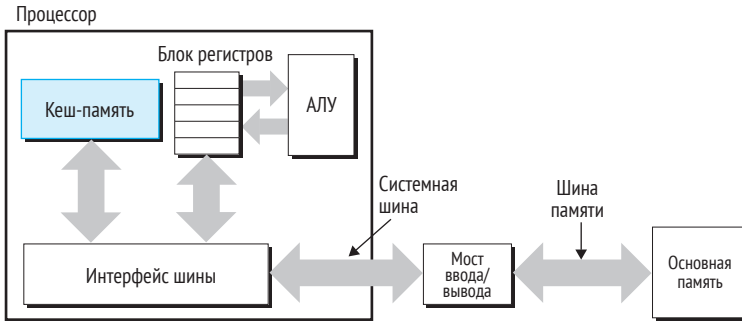


Рис. 6.16. Типичная организация кешей в процессоре

На продолжающееся увеличение разницы в быстродействии между процессорами и основной памятью проектировщики отреагировали добавлением дополнительной кеш-памяти большего объема – *кеша L2* (кеш второго уровня) – между кешем L1 и основной памятью со временем доступа порядка 10 циклов синхронизации. Во многих современных системах имеется еще один кеш, еще большего объема, располагающийся между кешем L2 и основной памятью, который называется *кешем L3* (кешем третьего уровня) и имеет время доступа порядка 50 циклов синхронизации. Несмотря на большое разнообразие способов организации кеш-памяти, общие принципы остаются неизменными. Для целей нашего обсуждения в следующем разделе мы будем предполагать простую иерархию памяти с одним кешем L1 между процессором и основной памятью.

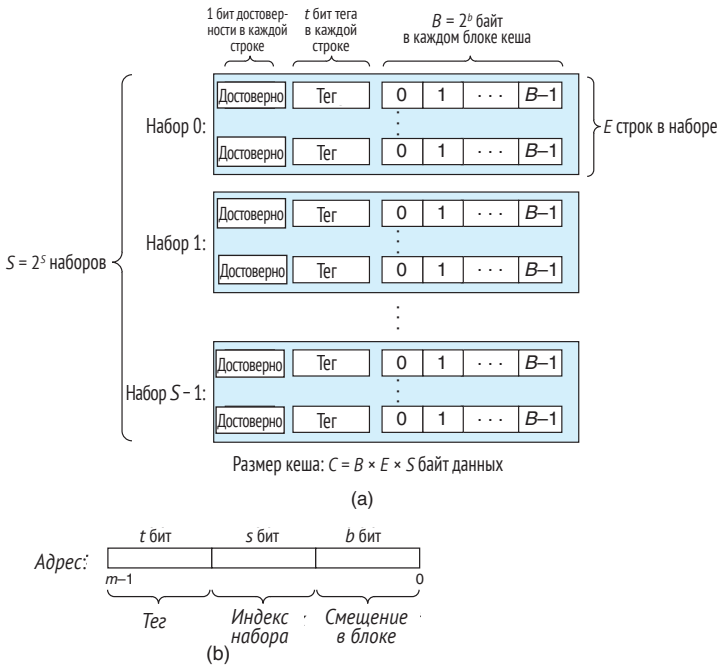
### 6.4.1. Обобщенная организация кеш-памяти

Рассмотрим компьютерную систему, в которой каждый адрес памяти состоит из  $m$  бит, образующих  $M = 2^m$  уникальных адресов. Как показано на рис. 6.17 (а), кеш-память для такой системы организована в виде массива  $S = 2^s$  наборов *кеша*. Каждый блок состоит из  $E$  строк *кеша* (cache line). Каждая строка включает блок данных из  $B = 2^b$  байт, *бита достоверности*, указывающего, содержит ли строка значащую информацию,  $t = m - (b + s)$  разрядов тега (подмножества битов из адреса текущего блока в памяти), которые уникально идентифицируют блок, хранимый в строке кеша.

Организацию кеш-памяти можно охарактеризовать кортежем  $(S, E, B, b)$ . Размер (или емкость) кеша  $C$  определяется совокупным размером всех блоков. Биты тегов и бит достоверности при этом не учитываются. То есть  $C = S \times E \times B$ .

Когда процессор встречается инструкцию загрузки слова из ячейки памяти с адресом  $A$ , он передает адрес в кеш. Если кеш содержит копию слова с адресом  $A$ , то немедленно возвращает его процессору. Но как кеш определяет наличие в нем копии слова? Кеш-память устроена так, что может найти запрошенное слово, просто исследовав биты адреса, как это происходит в хеш-таблицах с предельно простой хеш-функцией. Давайте рассмотрим этот механизм подробнее.

Параметры  $S$  и  $B$  управляют разделением  $m$  бит адреса на три поля, как показано на рис. 6.17 (b). Первыми анализируются биты  $s$  в адресе  $A$ , которые задают индекс в массиве из  $S$  наборов. Первый набор имеет индекс 0, второй – 1 и т. д. Эти биты интерпретируются как целое без знака и определяют, в каком наборе должно храниться слово. После определения набора анализируются биты тега  $t$  в  $A$ , которые определяют строку в наборе с искомым словом (если она существует). Строка содержит слово, только если установлен бит достоверности и биты тега в строке совпадают с битами тега в адресе  $A$ . После того как нужная строка будет найдена, анализируются биты  $b$  смещения, которые задают смещение слова в блоке данных, состоящем из  $B$  байт.



**Рис. 6.17.** Обобщенная организация кеш-памяти ( $S, E, B, m$ ). (a) Массив наборов. Каждый набор содержит одну или несколько строк. Каждая строка содержит бит достоверности, несколько битов идентификации и блок данных. (b) Кеш отображает  $m$  адресных бит в  $t$  бит тега,  $s$  бит индекса набора и  $b$  бит смещения в блоке

Как вы наверняка заметили, в описании механики работы кеша используется множество символов. Их расшифровка приводится в табл. 6.8.

**Таблица 6.8.** Параметры, характеризующие особенности работы кеша

Параметр	Описание
Основные параметры	
$S = 2^s$	Количество наборов
$E$	Количество строк в наборе
$B = 2^b$	Размер блока в байтах
$m = \log_2(M)$	Количество битов физического адреса (в основной памяти)

Параметр	Описание
Производные параметры	
$M = 2^m$	Максимальное число уникальных адресов в памяти
$s = \log_2(S)$	Количество битов индекса набора
$b = \log_2(B)$	Количество битов смещения в блоке
$t = m - (s + b)$	Количество битов тега
$C = B \times E \times S$	Размер кеша в байтах без учета служебной информации (битов тега и достоверности)

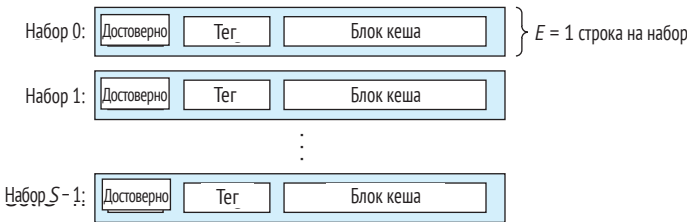
**Упражнение 6.9 (решение в конце главы)**

В следующей таблице даны параметры для некоторых разных кешей. Определите для каждого количество наборов ( $S$ ), количество битов тега ( $t$ ), количество битов индекса набора ( $s$ ) и количество битов смещения в блоке ( $b$ ).

Кеш	$m$	$C$	$B$	$E$	$S$	$t$	$s$	$b$
1	32	1024	4	1	_____	_____	_____	_____
2	32	1024	8	4	_____	_____	_____	_____
3	32	1024	32	32	_____	_____	_____	_____

**6.4.2. Кеш с прямым отображением**

Кеши делятся на классы по значению  $E$  – количеству строк в наборе. Кеш с одной строкой в наборе ( $E = 1$ ) называется кешем с *прямым отображением* (рис. 6.18). Это самый простой кеш, поэтому мы используем его для иллюстрации некоторых общих принципов работы кешей.



**Рис. 6.18.** Кеш с прямым отображением ( $E = 1$ ).  
Содержит ровно одну строку в наборе

Пусть имеется система с процессором, блоком регистров, кешем L1 и основной памятью. Встретив инструкцию чтения слова  $w$  из памяти, процессор посылает запрос кешу L1. Если в нем присутствует кешированная копия  $w$ , то имеет место попадание в кеш L1 – кеш быстро извлекает  $w$  и возвращает его процессору. В противном случае имеет место промах кеша и процессор должен дожидаться, пока кеш L1 запросит из основной памяти копию блока, содержащего  $w$ . Когда нужный блок будет получен из памяти, кеш L1 сохранит его в одной из своих строк, извлечет слово  $w$  из сохраненного блока и вернет процессору. Определение промаха или попадания и извлечение запрошенного слова выполняются в три этапа: (1) *выбор набора*; (2) *сопоставление строки*; (3) *извлечение слова*.

### Выбор набора в кеше с прямым отображением

На этом этапе кеш извлекает  $s$  бит индекса набора из середины адреса слова  $w$  и интерпретирует их как целое число без знака – номер набора. То есть если рассматривать кеш как одномерный массив наборов, то биты индекса формируют индекс элемента в этом массиве. На рис. 6.19 показано, как происходит выбор набора в кеше с прямым отображением. В этом примере биты индекса  $00001_2$  интерпретируются как целочисленный индекс, соответствующий набору с порядковым номером 1.

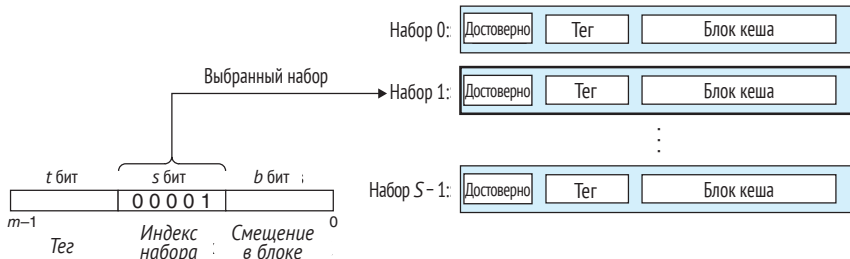


Рис. 6.19. Выбор множества в кеше с прямым отображением

### Сопоставление строк в кеше с прямым отображением

Следующий этап после выбора набора – определение наличия слова  $w$  в одной из его строк. Для кеша с прямым отображением эта проверка выполняется просто и быстро, потому что в наборе имеется только одна строка. Копия  $w$  присутствует в строке, только если установлен бит достоверности и тег в строке кеша совпадает с тегом в адресе слова  $w$ .

На рис. 6.20 показано, как происходит сопоставление строк в кеше с прямым отображением. В данном примере в выбранном множестве имеется только одна строка. Бит достоверности в этой строке установлен, поэтому биты в теге и блоке являются значимыми. Биты тега в строке совпадают с битами тега в адресе, следовательно, копия нужного слова действительно хранится в этой строке. Иначе говоря, имеет место попадание в кеш. Если бы бит достоверности был сброшен или биты тега не совпадали, то имел бы место промах кеша.

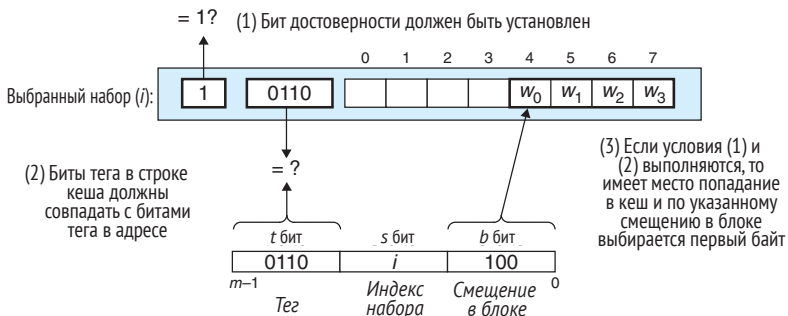


Рис. 6.20. Сопоставление строк и извлечение слова в кеше с прямым отображением. Внутри блока кеша  $w_0$  обозначает младший байт слова  $w$ ,  $w_1$  – следующий байт и т. д.

### Извлечение слова в кеше с прямым отображением

В случае попадания в кеш известно, что  $w$  находится где-то в блоке. Этот последний этап определяет, где искомое слово начинается в блоке. Как показано на рис. 6.20, биты

смещения в блоке задают местоположение первого байта этого слова. Подобно представлению кеша в виде массива строк, блок можно представить как массив байтов, а смещение – как индекс в этом массиве. В нашем примере биты смещения в блоке  $100_2$  указывают, что искомое слово  $w$  находится в блоке, начиная с 4-го байта. (Предполагается, что слова имеют длину 4 байта.)

### Вытеснение строк при промахх в кеше с прямым отображением

В случае промаха кеша блок с требуемым словом необходимо извлечь из нижележащего уровня в иерархии памяти и сохранить его в одной из строк в наборе, который определяется битами индекса набора. В общем случае, если кеш заполнен достоверными строками, тогда одну из них необходимо вытеснить. В кеше с прямым отображением, где каждый набор состоит из единственной строки, стратегия вытеснения тривиальна: текущая строка просто заменяется новой.

### Все вместе: кеш с прямым отображением в действии

Механизмы, используемые кешем для выбора наборов и идентификации строк, на редкость просты. И это не удивительно, потому что аппаратная часть должна выполнять их всего за несколько наносекунд. Впрочем, такие манипуляции с битами выглядят просто для машины, но не для человека. Давайте подробнее рассмотрим весь процесс на конкретном примере. Предположим, что имеется кеш с прямым отображением:

$$(S, E, B, m) = (4, 1, 2, 4).$$

То есть данный кеш имеет четыре набора, в каждом наборе одна строка, в каждом блоке 2 байта, а адреса имеют длину в 4 бита. Также предположим, что слово имеет размер 1 байт. Такие предположения, разумеется, далеки от реальности, но они помогут предельно упростить пример.

При первом знакомстве с кешами нелишним будет пронумеровать все адресное пространство и определить разбивку битов, как мы сделали это в табл. 6.9 для нашего примера. В части нумерации адресного пространства следует отметить несколько интересных моментов:

- совокупность битов тега и индекса набора однозначно идентифицирует каждый блок памяти. Например, блок 0 включает адреса 0 и 1, блок 1 – адреса 2 и 3, блок 2 – адреса 4 и 5 и т. д.;
- поскольку всего в памяти помещается восемь блоков, а наборов в кеше всего четыре, в один и тот же набор отображается несколько блоков (в данном случае два), в том смысле, что они имеют одно и то же значение индекса набора. Например, блоки 0 и 4 отображаются в набор 0, блоки 1 и 5 отображаются в набор 1 и т. д.;
- блоки, отображающиеся в один и тот же набор кеша, уникально идентифицируются тегом. Например, блок 0 имеет значение 0 в бите тега, а блок 4 – значение 1, аналогично блок 1 имеет значение 0 в бите тега, а блок 5 – значение 1.

**Таблица 6.9.** 4-битное адресное пространство для примера кеша с прямым отображением

Адрес (десятичный)	Биты адреса			Номер блока (десятичный)
	Биты тега ( $t = 1$ )	Биты индекса ( $s = 2$ )	Биты смещения ( $b = 1$ )	
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1

Адрес (десятичный)	Биты адреса			Номер блока (десятичный)
	Биты тега ( $t = 1$ )	Биты индекса ( $s = 2$ )	Биты смещения ( $b = 1$ )	
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	0	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Смоделируем работу кеша при выполнении процессором последовательности операций чтения.

Напомним: в этом примере предполагается, что процессор читает 1-байтные слова. Подобное моделирование вручную – дело довольно утомительное, и многим из вас захочется его пропустить, однако, как показывает опыт поколений студентов, понять принцип работы кеша сложно, если не выполнить хотя бы несколько таких примеров.

Первоначально кеш пуст и все биты достоверности сброшены в 0:

Набор	Бит достоверности	Тег	блок[0]	блок[1]
0	0			
1	0			
2	0			
3	0			

Каждая строка в таблице представляет строку кеша. Первый столбец обозначает номер набора, которому принадлежит строка, однако следует помнить, что этот столбец приведен только для удобства и не является частью кеша. Оставшиеся четыре столбца представляют фактические биты в каждой строке кеша. Посмотрим, что происходит, когда процессор выполняет последовательность операций чтения.

1. **Чтение слова с адресом 0.** Поскольку бит достоверности для набора 0 сброшен в 0, имеет место промах кеша. Кеш выбирает блок 0 из памяти (или кеша уровня ниже) и сохраняет его в наборе 0. Затем кеш возвращает  $m[0]$  (содержимое ячейки памяти с адресом 0) из блока[0] вновь выбранной строки кеша.

Набор	Бит достоверности	Тег	блок[0]	блок[1]
0	1	0	$m[0]$	$m[1]$
1	0			
2	0			
3	0			

2. **Чтение слова с адресом 1.** В этом случае имеет место попадание в кеш. Последний немедленно возвращает  $m[1]$  из блока[1] строки кеша. Состояние кеша не меняется.



3. **Чтение слова с адресом 13.** Поскольку строка кеша в наборе 2 недостоверна, налицо промах кеша. Кеш выбирает блок 6 из памяти, сохраняет его в наборе 2 и возвращает  $m[13]$  из блока[1] новой строки кеша.

Набор	Бит достоверности	Тег	блок[0]	блок[1]
0	1	0	$m[0]$	$m[1]$
1	0			
2	1	1	$m[12]$	$m[13]$
3	0			

4. **Чтение слова с адресом 8.** Промах кеша. Строка кеша в наборе 0 достоверна, однако теги не соответствуют друг другу. Кеш выбирает блок 4, сохраняет его в наборе 0 (замещая строку, оставшуюся там после чтения блока с адресом 0) и возвращает  $m[8]$  из блока[0] новой строки кеша.

Набор	Бит достоверности	Тег	блок[0]	блок[1]
0	1	1	$m[8]$	$m[9]$
1	0			
2	1	1	$m[12]$	$m[13]$
3	0			

5. **Чтение слова с адресом 0.** Еще один промах кеша вследствие неудачного стечения обстоятельств (блок с адресом 0 был замещен предыдущей операцией чтения блока с адресом 8). Такой промах, когда в кеше еще много свободного места, но приходится вытеснять хранящиеся в нем блоки, отображаемые в один и тот же набор, является примером конфликтного промаха.

Набор	Бит достоверности	Тег	блок[0]	блок[1]
0	1	0	$m[0]$	$m[1]$
1	0			
2	1	1	$m[12]$	$m[13]$
3	0			

### Конфликтные промахи в кеше с прямым отображением

Конфликтные промахи в реальных программах – обычное дело и могут вызывать проблемы с производительностью. Конфликтные промахи в кешах с прямым отображением случаются, когда программы обрабатывают массивы с размерами, кратными степени двойки. Например, рассмотрим функцию, вычисляющую скалярное произведение двух векторов:

```

1 float dotprod(float x[8], float y[8])
2 {
3     float sum = 0.0;
4     int i;
5
6     for (i = 0; i < 8; i++)
7         sum += x[i] * y[i];
8     return sum;
9 }
```

Данная функция обладает хорошей пространственной локальностью относительно  $x$  и  $y$ , поэтому можно ожидать, что она будет часто попадать в кеш. К сожалению, это не всегда верно.

Пусть значения типа `float` занимают 4 байта, массив `x` хранится в непрерывном фрагменте памяти размером 32 байта, начиная с адреса 0, а массив `y` находится сразу за массивом `x`, начиная с адреса 32. Для простоты предположим также, что блок имеет размер 16 байт (достаточно большой, чтобы в нем можно было сохранить четыре значения типа `float`), а кеш имеет два набора с общим размером 32 байта. Допустим, что переменная `sum` хранится в регистре процессора и потому обращение к ней не влечет обращения к памяти. При всех указанных допущениях элементы `x[i]` и `y[i]` будут отображаться в один и тот же набор кеша:

Элемент	Адрес	Индекс набора	Элемент	Адрес	Индекс набора
<code>x[0]</code>	0	0	<code>y[0]</code>	32	0
<code>x[1]</code>	4	0	<code>y[1]</code>	36	0
<code>x[2]</code>	8	0	<code>y[2]</code>	40	0
<code>x[3]</code>	12	0	<code>y[3]</code>	44	0
<code>x[4]</code>	16	1	<code>y[4]</code>	48	1
<code>x[5]</code>	20	1	<code>y[5]</code>	52	1
<code>x[6]</code>	24	1	<code>y[6]</code>	56	1
<code>x[7]</code>	28	1	<code>y[7]</code>	60	1

В первой итерации цикла при обращении к `x[0]` произойдет промах кеша, который вызовет загрузку блока с элементами `x[0]–x[3]` в набор 0. Затем, при обращении к `y[0]`, произойдет еще один промах кеша, который вызовет загрузку блока с элементами `y[0]–y[3]` в набор 0 с вытеснением элементов из массива `x`, которые были загружены предыдущей операцией чтения. В следующей итерации при обращении к `x[1]` вновь произойдет промах, который вызовет загрузку блока с элементами `x[0]–x[3]` в набор 0 с вытеснением элементов `y[0]–y[3]`. На этот раз налицо конфликтный промах, и фактически каждое последующее обращение к массивам `x` и `y` будет вызывать конфликтный промах и пробуксовку между блоками `x` и `y`. Термин *пробуксовка* описывает любую ситуацию, когда кеш многократно загружает и вытесняет одни и те же наборы блоков.

Итак, несмотря на то что программа обладает хорошей пространственной локальностью и в кеше достаточно места для хранения блоков `x[i]` и `y[i]`, при каждом обращении к массивам будет возникать конфликтный промах, потому что блоки массивов отображаются в один и тот же набор кеша. Результатом такой пробуксовки будет замедление цикла в 2 или даже в 3 раза. Также имейте в виду, что, несмотря на простоту примера, решение данной проблемы, характерной для кешей с прямым отображением, представляет определенную сложность.

К счастью, программисты могут избавиться от пробуксовки, понимая суть происходящего. Одно из простейших решений – размещение  $B$  байт (заполненных незначащей информацией) в конце каждого массива. Например, массив `x` можно объявить не как `float x[8]`, а как `float x[12]`. Тогда, если допустить, что массив `y` располагается в памяти вслед за массивом `x`, получается следующее отображение элементов массивов в наборы:

Элемент	Адрес	Индекс набора	Элемент	Адрес	Индекс набора
<code>x[0]</code>	0	0	<code>y[0]</code>	48	1
<code>x[1]</code>	4	0	<code>y[1]</code>	52	1
<code>x[2]</code>	8	0	<code>y[2]</code>	56	1
<code>x[3]</code>	12	0	<code>y[3]</code>	60	1
<code>x[4]</code>	16	1	<code>y[4]</code>	64	0
<code>x[5]</code>	20	1	<code>y[5]</code>	68	0
<code>x[6]</code>	24	1	<code>y[6]</code>	72	0
<code>x[7]</code>	28	1	<code>y[7]</code>	76	0

Благодаря дополнительным незначащим элементам в конце  $x$ ,  $x[i]$  и  $y[i]$  будут отображаться в разные наборы, а пробуксовка вместе с конфликтными промахами исчезнет.

### Упражнение 6.10 (решение в конце главы)

Какую долю от общего количества обращений к  $x$  и  $y$  в предыдущем примере `dotprod` составят попадания в кеш после добавления дополнительных элементов в конец массива  $x$ ?

#### Почему индекс определяется битами из середины адреса?

У многих из вас наверняка возник вопрос: почему индекс наборов в кеше определяется битами из середины адреса, а не, скажем, старшими битами? Тому есть свое объяснение, как показано на рис. 6.21. Если в качестве индекса использовать старшие биты, тогда некоторые блоки, хранящиеся в памяти по соседству, будут отображаться в тот же набор кеша. Например, на рис. 6.21 показано, что первые четыре блока отображаются в первый набор кеша, вторые четыре блока – во второй набор и т. д. Если программа обладает хорошей пространственной локальностью и просматривает элементы массива последовательно, тогда в любой момент времени кеш может хранить только часть массива размером с блок. Такое использование кеша неэффективно. Сравните этот случай с индексированием битами из середины, когда смежные блоки всегда отображаются в разные наборы кеша. В этой ситуации кеш может хранить фрагмент массива размера  $C$ , где  $C$  – размер кеша.

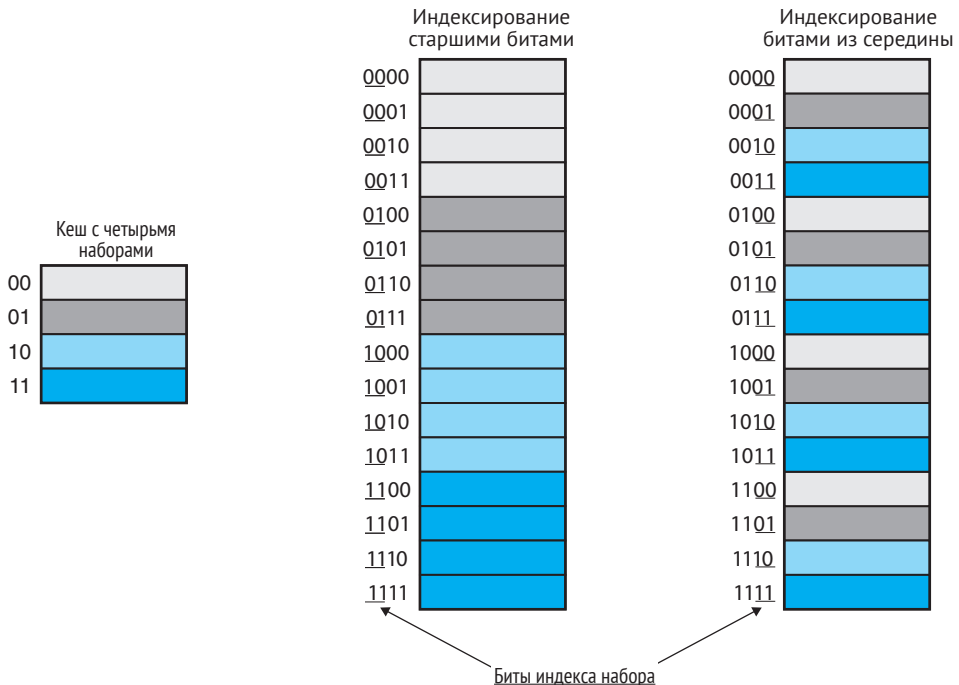


Рис. 6.21. Почему индекс определяется битами из середины адреса

**Упражнение 6.11 (решение в конце главы)**

Представьте гипотетический кеш, использующий старшие  $s$  бит адреса в качестве индекса набора. В таком кеше соседние фрагменты памяти будут отображаться в один и тот же набор.

1. Сколько блоков уместится в каждом из этих смежных фрагментов?
2. Взгляните на следующий код, выполняющийся в системе с кешем  $(S, E, B, m) = (512, 1, 32, 32)$ :

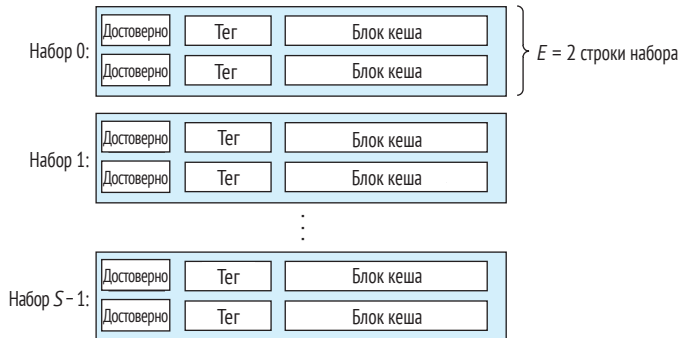
```
int array[4096];

for (i = 0; i < 4096; i++)
    sum += array[i];
```

Какое максимальное число блоков массива может храниться в кеше в любой момент времени?

**6.4.3. Ассоциативные кеши**

Проблема с конфликтными промахами в кешах с прямым отображением обусловлена наличием только одной строки в каждом наборе (в нашей терминологии  $E = 1$ ). Ассоциативные кеши ослабляют это ограничение за счет наличия в каждом наборе дополнительных строк. Кеш с  $1 < E < C/B$  часто называют  $E$ -строчным ассоциативным кешем. В следующем разделе мы рассмотрим особый случай, когда  $E = C/B$ . На рис. 6.22 показано устройство 2-строчного ассоциативного кеша.



**Рис. 6.22.** Ассоциативный кеш ( $1 < E < C/B$ ). В ассоциативном кеше каждый набор содержит несколько строк. В частности, здесь изображен 2-строчный ассоциативный кеш

**Выбор набора в ассоциативном кеше**

Выбор набора в ассоциативном кеше происходит так же, как в кеше с прямым отображением, – с использованием битов индекса набора. Механика выбора показана на рис. 6.23.

**Сопоставление строк и извлечение слова в ассоциативном кеше**

Сопоставление строк в ассоциативных кешах организовано сложнее, чем в кешах с прямым отображением, потому что требуется проверять биты тегов и достоверности в нескольких строках, чтобы определить присутствие запрошенного слова в наборе. Традиционная память организована как массив значений; она принимает адрес на входе и возвращает значение, хранящееся по этому адресу. Ассоциативная память – это массив пар (ключ, значение); она принимает на входе ключ и возвращает значение из пары (ключ, значение), ключ которой совпал с ключом на входе. Каждый набор в ассоциатив-

ном кеше можно рассматривать как ассоциативную память небольшого объема, роль ключей в которой играют комбинации битов тега и достоверности, а роль значений – содержимое блоков.

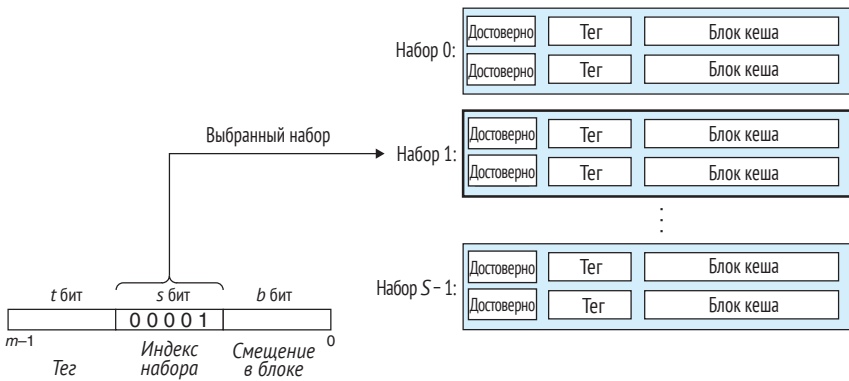


Рис. 6.23. Выбор набора в ассоциативном кеше

На рис. 6.24 показана механика сопоставления строк в ассоциативном кеше. Здесь важно понимать, что любая строка в наборе может содержать любой из блоков памяти, отображающихся в это множество. Поэтому кеш должен проверить каждую строку в наборе, чтобы найти достоверную строку, тег которой совпадает с тегом в адресе. Если такая строка имеется, то имеет место попадание в кеш, и далее искомое слово выбирается с использованием смещения в блоке.

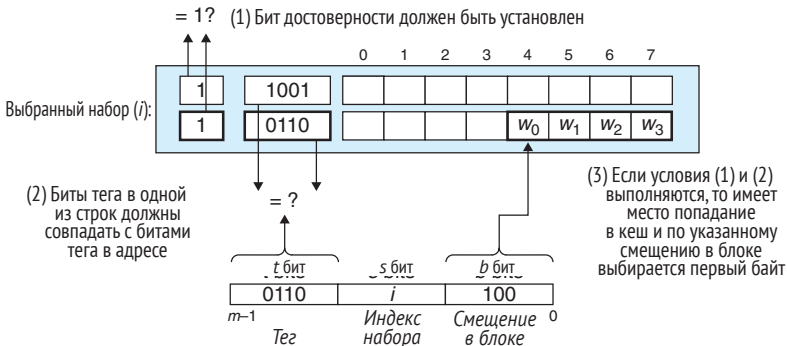


Рис. 6.24. Сопоставление строк и извлечение слова в ассоциативном кеше

### Вытеснение строк при промахх в ассоциативном кеше

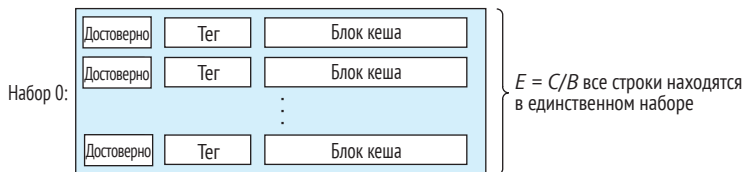
Если запрошенное слово не найдено ни в одной строке набора, то имеет место промах кеша, и необходимо получить из памяти блок, содержащий искомое слово. Но какую строку должен заменить кеш? Разумеется, если есть пустая (недействительная) строка, тогда она и будет подходящим кандидатом на вытеснение. Но если в наборе пустых строк нет, то необходимо выбрать одну из заполненных и надеяться, что процессор никогда не запросит вытесненную строку.

При разработке программ программистам сложно учесть все нюансы стратегии вытеснения строк, поэтому мы не будем вдаваться в детали. Самой простой стратегией замены является выбор произвольной строки. В других более сложных стратегиях используется принцип локальности, чтобы минимизировать вероятность обращения к

вытесненной строке в будущем. Например, стратегия вытеснения *наименее часто используемого* (Least Frequently Used, LFU) вытеснит строку, к которой было меньше всего обращений в течение определенного периода времени в прошлом. Стратегия вытеснения *наиболее давно использовавшегося* (Least Recently Used, LRU) вытеснит строку, к которой не было обращений дольше всего. Все эти стратегии требуют дополнительно времени и аппаратных средств. Однако чем дальше от процессора находится кеш в иерархии памяти, тем дороже обходятся промахи, поэтому минимизация промахов с помощью действенных стратегий вытеснения имеет определяющее значение.

#### 6.4.4. Полностью ассоциативные кеши

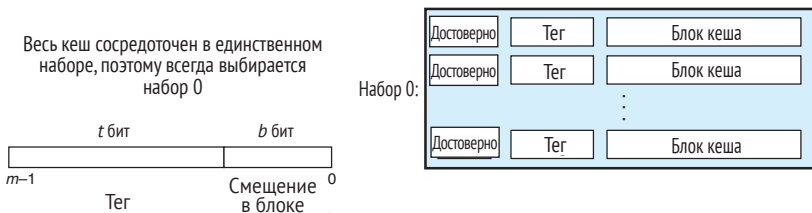
Полностью ассоциативный кеш состоит из одного набора ( $E = C/B$ ), содержащего все строки кеша. Структура такого кеша показана на рис. 6.25.



**Рис. 6.25.** Полностью ассоциативный кеш ( $E = C/B$ ). В полностью ассоциативном кеше имеется только один набор, включающий все строки кеша

#### Выбор набора в полностью ассоциативных кешах

Выбор набора в полностью ассоциативном кеше производится тривиально просто, потому что в кеше имеется только один набор, как показано на рис. 6.26. Обратите внимание на отсутствие в адресе битов индекса набора; адрес делится только на тег и смещение в блоке.



**Рис. 6.26.** Выбор набора в полностью ассоциативном кеше. Обратите внимание на отсутствие битов индекса набора

#### Упражнение 6.12 (решение в конце главы)

Следующие задачи помогут закрепить понимание особенностей работы кешей. Допустим следующее:

- память поддерживает адресацию на уровне байтов;
- доступ к памяти осуществляется как к 1-байтным словам (не 4-байтным);
- адреса имеют длину 13 бит;
- кеш – 2-строчный ассоциативный ( $E = 2$ ) с размером блока 4 байта ( $B = 4$ ) и 8 наборами ( $S = 8$ ).

Ниже показано содержимое кеша (все числа приведены в шестнадцатеричном формате):

Кеш с 2-строчными ассоциативными наборами												
Индекс набора	Строка 0						Строка 1					
	Тег	Достов.	Байт 0	Байт 1	Байт 2	Байт 3	Тег	Достов.	Байт 0	Байт 1	Байт 2	Байт 3
0	09	1	86	30	3F	10	00	0	–	–	–	–
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	–	–	–	–	0B	0	–	–	–	–
3	06	0	–	–	–	–	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	–	–	–	–
6	91	1	A0	B7	26	2D	F0	0	–	–	–	–
7	46	0	–	–	–	–	DE	1	12	C0	88	37

Следующая таблица иллюстрирует структуру адреса (одна ячейка – один бит). Покажите в этой таблице (поставив метки), какие биты будут использоваться для:

- определения смещения в блоке (CO);
- определения индекса набора (CI);
- определения тега (CT).

12	11	10	9	8	7	6	5	4	3	2	1	0

Сопоставление строк и извлечение слова в полностью ассоциативных кешах

Сопоставление строк и извлечение слова в полностью ассоциативном кеше происходят так же, как в ассоциативном кеше (рис. 6.27). Разница лишь в масштабе.

По причине того, что цепи кеша должны осуществлять параллельный поиск тега в большом количестве строк, построить большой и быстродействующий полностью ассоциативный кеш трудно и дорого. По этой причине полностью ассоциативная архитектура используется только для создания небольших кешей, таких как буфер ассоциативной трансляции адресов (Translation Lookaside Buffer, TLB) в системах виртуальной памяти, кеширующих записи из таблицы страниц (раздел 9.6.2).

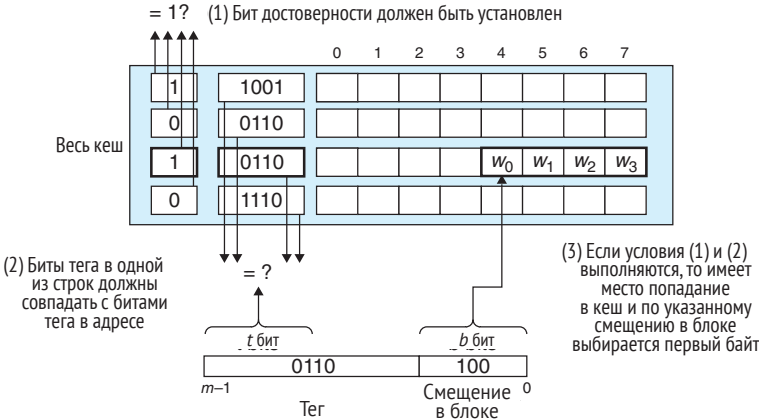


Рис. 6.27. Сопоставление строк и извлечение слова в полностью ассоциативном кеше

**Упражнение 6.13 (решение в конце главы)**

Представьте, что программа, выполняющаяся на машине из упражнения 6.12, обращается к 1-байтовому слову с адресом 0x0E34. Укажите, из какой строки в кеше и какой байт (в шестнадцатеричном виде) будет извлечен. Если возникнет промах кеша, то в графу «Байт из кеша» впишите прочерк.

1. Формат адреса (одна ячейка – один бит):

12	11	10	9	8	7	6	5	4	3	2	1	0	

2. Обращение к ячейке памяти:

Параметр	Значение
Смещение в блоке (CO)	0x_____
Индекс набора (CI)	0x_____
Тег (CT)	0x_____
Попадание в кеш? (Д/Н)	_____
Байт из кеша	0x_____

**Упражнение 6.14 (решение в конце главы)**

Решите задачу из упражнения 6.13 для адреса 0x0DD5.

1. Формат адреса (одна ячейка – один бит):

12	11	10	9	8	7	6	5	4	3	2	1	0	

2. Обращение к ячейке памяти:

Параметр	Значение
Смещение в блоке (CO)	0x_____
Индекс набора (CI)	0x_____
Тег (CT)	0x_____
Попадание в кеш? (Д/Н)	_____
Байт из кеша	0x_____

**Упражнение 6.15 (решение в конце главы)**

Решите задачу из упражнения 6.13 для адреса 0x1FE4.

1. Формат адреса (одна ячейка – один бит):

12	11	10	9	8	7	6	5	4	3	2	1	0	

2. Обращение к ячейке памяти:

Параметр	Значение
Смещение в блоке (CO)	0x_____
Индекс набора (CI)	0x_____
Тег (CT)	0x_____
Попадание в кеш? (Д/Н)	_____
Байт из кеша	0x_____



**Упражнение 6.16 (решение в конце главы)**

Для кеша, описанного в упражнении 6.13, перечислите все шестнадцатеричные адреса памяти, при обращении к которым будет иметь место попадание в кеш в набор 3.

### 6.4.5. Проблемы с операциями записи

Как мы видели, при выполнении операций чтения кеш действует прямолинейно. Сначала выполняется поиск копии требуемого слова  $w$  в кеше. При попадании в кеш слово  $w$  сразу возвращается процессору. При промахе из нижестоящего уровня в иерархии памяти выбирается блок, содержащий слово  $w$ , затем этот блок сохраняется в определенной строке кеша (с возможным вытеснением ранее сохраненной строки) и слово  $w$  возвращается процессору.

Ситуация с операциями записи немного сложнее. Предположим, что процессор записывает уже кешированное слово  $w$  (имеет место *попадание при записи*). Но что должно произойти дальше, после того как кеш обновит копию  $w$ ? В простейшем случае, известном как *сквозная запись* (write-through), блок с измененным словом  $w$  тут же записывается в нижестоящий уровень в иерархии памяти. Однако, несмотря на простоту, метод сквозной записи имеет существенный недостаток – он вызывает передачу данных по шине при каждой записи. Другой подход, называемый *обратной записью* (write-back), откладывает запись измененного блока в нижестоящий уровень на максимально длительное время, до вытеснения из кеша алгоритмом замещения. Благодаря локальности метод обратной записи может значительно сократить количество операций с шиной, но он тоже имеет недостаток – увеличенная сложность реализации. Кеш должен поддерживать дополнительный *бит изменения* для каждой строки кеша, служащий признаком изменения блока кеша.

Еще одна проблема: как обрабатывать промахи при записи. Одно из решений, называемое *записью с размещением* (write-allocate), загружает нужный блок из нижестоящего уровня в иерархии памяти в кеш, после чего обновляет блок в кеше. Метод записи с размещением пытается использовать пространственную локальность операций записи, однако любой промах будет вызывать перемещение блока из нижестоящего уровня в иерархии в кеш. Альтернативный прием, известный как *запись без размещения* (no-write-allocate), заключается в записи слова непосредственно в нижестоящий уровень в иерархии в обход кеша. Кеши со сквозной записью обычно выполняют запись без размещения строк. Кеши с обратной записью, как правило, выполняют запись с размещением.

Оптимизация кеша для операций записи – дело весьма деликатное и сложное, поэтому мы не будем углубляться в обсуждение этой проблемы. Конкретные детали зависят от особенностей каждой конкретной системы, как правило, запатентованных и скупой описанных в документации. Программистам, желающим писать программы, «оптимально использующие кеш», можно предложить остановить выбор на модели кеша с обратной записью и с размещением. Подобный выбор объясняется несколькими причинами: вероятнее всего, кеши на нижних уровнях иерархии памяти будут использовать стратегию обратной записи из-за значительного времени, требуемого на перенос. Например, в системах виртуальной памяти (использующих основную память в качестве кеша для дисковых блоков) применяется исключительно стратегия обратной записи. Однако с увеличением плотности размещения логических элементов сложность обратной записи перестает быть серьезной помехой, и можно с полной уверенностью ожидать использования кешей с обратной записью на всех уровнях современных компьютерных систем. Поэтому данное допущение соответствует современным тенденциям. Другой причиной выбора модели кеша с обратной записью и с размещением является симметричность обработки операций записи и чтения, в том смысле, что обратная запись с

размещением хорошо укладывается в понимание локальности. То есть мы можем просто писать программы с хорошей временной и пространственной локальностью и не задумываться об оптимизации под конкретную систему организации памяти.

6.4.6. Устройство реальной иерархии кешей

До сих пор предполагалось, что в кеше хранятся только данные. Однако в действительности, помимо данных, в кеше могут также храниться инструкции. Кеш, хранящий только инструкции, называется *i-cache* (кеш инструкций). Кеш, хранящий только данные, называется *d-cache* (кеш данных). Кеш, хранящий инструкции и данные, называется *универсальным*. Современные процессоры имеют отдельные кеши инструкций и данных. Тому есть свои причины. При наличии двух отдельных кешей процессор может одновременно читать слова инструкций и данных. Кеши инструкций обычно доступны только для чтения и, соответственно, устроены проще. Обычно кеши этих двух видов оптимизируются с учетом разных закономерностей доступа и могут иметь разные размеры блоков, ассоциативность и емкость. Кроме того, наличие отдельных кешей гарантирует, что обращение к данным не повлечет конфликтные промахи при выборке инструкций, и наоборот.

На рис. 6.28 показана иерархия кеш-памяти процессора Intel Core i7. Микросхема процессора имеет четыре ядра. Каждое ядро имеет свой кеш инструкций L1, кеш данных L1 и универсальный кеш L2. Все ядра используют общий кеш L3. Интересная особенность этой иерархии состоит в том, что вся кеш-память SRAM находится внутри процессора.

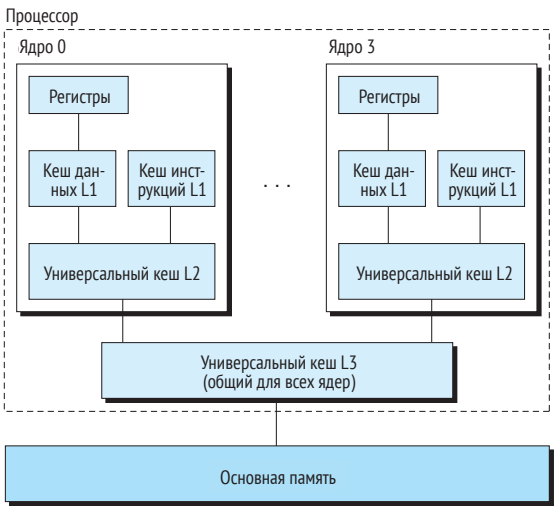


Рис. 6.28. Иерархия кешей процессора Intel Core i7

В табл. 6.10 перечислены основные характеристики кешей в процессоре Intel Core i7.

Таблица 6.10. Характеристики иерархии кешей в процессоре Intel Core i7

Тип кеша	Время доступа (циклы)	Размер кеша (С)	Ассоциативность (Е)	Размер блока (В)	Количество наборов (S)
Кеш инструкций L1	4	32 Кбайт	8	64 байт	64
Кеш данных L1	4	32 Кбайт	8	64 байт	64
Универсальный кеш L2	10	256 Кбайт	8	64 байт	512
Универсальный кеш L3	40–75	8 Мбайт	16	64 байт	8192

**Строки, наборы и блоки: чем они отличаются**

Строки кеша, наборы и блоки легко перепутать. Давайте рассмотрим их подробнее, чтобы устранить любое недопонимание:

- *блок* – пакет информации фиксированного размера, перемещающийся между кешем и основной памятью (или нижележащим кешем);
- *строка* – это контейнер в кеше, хранящий блок и другую служебную информацию: бит достоверности и биты тега;
- *набор* – это набор из одной или более строк. Наборы в кешах с прямым отображением состоят из одной строки. Наборы в ассоциативных и полностью ассоциативных кешах состоят из нескольких строк.

В кешах с прямым отображением наборы и строки эквивалентны. Но в ассоциативных кешах наборы и строки различаются, и эти термины нельзя использовать взаимозаменяемо.

Поскольку в строке всегда хранится только один блок, термины «строка» и «блок» часто используются взаимозаменяемо. Например, профессионалы обычно используют термин «размер строки» кеша, когда на самом деле подразумевается размер блока. Такое использование вполне обычно и не должно вызывать путаницы, пока есть понимание, что такое блок, а что – строка.

**6.4.7. Влияние параметров кеша на производительность**

Производительность кеша оценивается по нескольким показателям:

*Частота промахов*

Доля обращений к памяти во время выполнения программы или ее части, при которых возникает промах кеша. Частота промахов вычисляется как *количество промахов / количество обращений*.

*Частота попаданий*

Доля попаданий в кеш при обращении к памяти. Вычисляется как  $1 - \text{частота промахов}$ .

*Время обработки попадания*

Время, необходимое для передачи слова из кеша в процессор, включая время выбора набора, поиска строки и выборки слова. Время попадания для кеша L1 обычно составляет несколько циклов синхронизации.

*Штраф за промах*

Дополнительное время, расходуемое в случае промаха на получение слова из кеша нижестоящего уровня. Штраф за промах кеша L1, влекущий обращение к кешу L2, обычно составляет около 10 циклов синхронизации. Штраф за промах кеша L2, влекущий обращение к кешу L3, составляет примерно 50 циклов. Штраф за промах кеша L3, влекущий обращение к основной памяти, составляет примерно 200 циклов.

Оптимизация компромиссов между стоимостью и производительностью кеш-памяти – сложная задача, требующая обширного тестирования на реалистичных примерах, и потому мы не будем рассматривать ее в этой книге. Однако есть возможность идентифицировать некоторые качественные альтернативы.

**Влияние размера кеша**

С одной стороны, с увеличением размера кеша увеличивается частота попаданий. С другой стороны, увеличить быстродействие устройств памяти большого объема всег-

да сложнее. В результате с увеличением кеша увеличивается и время попадания. Это объясняет, почему кеш L1 меньше кеша L2, а кеш L2 меньше кеша L3.

### Влияние размера блока

Выгоды от использования блоков большого размера сомнительны. С одной стороны, большие блоки могут помочь увеличить частоту попаданий при хорошей пространственной локальности программы. Однако чем крупнее блок, тем меньше строк в кеше, что может отрицательно сказаться на частоте попаданий в программах с хорошей временной, но плохой пространственной локальностями. Также крупные блоки отрицательно влияют на штраф за промах, потому что для переноса крупных блоков требуется больше времени. В современных системах, таких как Core i7, блоки содержат 64 байта.

### Влияние ассоциативности

В этом случае проблема заключается в выборе параметра  $E$  – количества строк в каждом наборе. Чем выше ассоциативность (т. е. чем выше значение  $E$ ), тем выше чувствительность кеша к пробуксовке из-за конфликтных промахов. К тому же высокая ассоциативность влечет за собой увеличение затрат на ее обслуживание. Высокая ассоциативность требует больше затрат на реализацию, а увеличить ее быстродействие достаточно трудно. Она требует большего количества битов тега на строку, дополнительных битов состояния LRU на строку, а также дополнительной управляющей логики. Высокая ассоциативность может увеличить время попадания из-за дополнительных накладных расходов, обусловленных повышенной сложностью выбора строки для вытеснения.

Выбор степени ассоциативности сводится к поиску компромисса между временем попадания и накладными расходами. Традиционно для высокопроизводительных систем, где важна высокая тактовая частота, предпочтение отдается кешу L1 с низкой ассоциативностью (для которого штраф за промах составляет всего несколько циклов) и кешам более низких уровней с высокой ассоциативностью, в которых штраф за промах намного выше. Например, в Intel Core i7 кеши L1 и L2 имеют ассоциативность, равную 8, а кеш L3 – 16.

### Влияние стратегии записи

Кеши со сквозной записью просты в реализации и могут использовать *буфер записи*, обновляющий память и работающий независимо от кеша. Более того, штраф за промах чтения в таком кеше оказывается ниже, потому что операции записи исключаются из числа источников причин промахов. С другой стороны, кеши с обратной записью меньше нагружают шину, что способствует увеличению пропускной способности памяти для устройств ввода/вывода, использующих технологию прямого доступа к памяти (DMA). Кроме того, сокращение количества операций с шиной тем важнее, чем ниже находится кеш в иерархии памяти и чем больше времени требуется на перенос данных. В общем случае кеши, расположенные ниже в иерархии, с большей вероятностью будут использовать обратную запись, чем сквозную.

## 6.5. Разработка программ, эффективно использующих кеш

В разделе 6.2 мы познакомились с понятием локальности и в общих чертах обсудили, от чего зависит хорошая локальность. Теперь, понимая особенности работы кеш-памяти, мы можем осветить некоторые дополнительные подробности. Программы с хорошей локальностью будут иметь более низкую частоту промахов, а программы с низкой частотой промахов будут выполняться быстрее, чем программы с высокой частотой промахов. Таким образом, хороший программист должен всегда стараться писать программы,

*эффективно использующие кеш*, то есть с хорошей локальностью. Вот основные правила, следование которым гарантирует более или менее эффективное использование кеша.

1. *Обеспечьте максимальную производительность наиболее частого случая.* Нередко при выполнении программ значительная часть времени тратится на выполнение небольшого числа основных функций. Эти функции зачастую большую часть времени тратят на выполнение циклов. Поэтому сосредоточьте свое внимание на этих циклах в основных функциях и не отвлекайтесь на остальные.
2. *Сведите к минимуму количество промахов кеша во всех внутренних циклах.* При прочих равных условиях, таких как общее число операций загрузки и сохранения, циклы с меньшей частотой промахов будут работать быстрее.

Для понимания, как это происходит на практике, рассмотрим функцию `sumvec` из раздела 6.2:

```

1 int sumvec(int v[N])
2 {
3     int i, sum = 0;
4
5     for (i = 0; i < N; i++)
6         sum += v[i];
7     return sum;
8 }
```

Насколько эффективно эта функция использует кеш? Во-первых, обратите внимание, что в отношении переменных `i` и `sum` в теле цикла наблюдается хорошая временная локальность. Фактически любой оптимизирующий компилятор постарается хранить локальные переменные в регистрах – на самом верхнем уровне иерархии памяти. Теперь рассмотрим последовательный перебор элементов вектора `v`. В общем случае, если размер блока в кеше составляет  $B$  байт, то обход элементов вектора с шагом  $k$  (где  $k$  измеряется в словах) приводит в среднем к числу промахов в каждой итерации, которое вычисляется как  $\min(1, (\text{размер слова} \times k)/B)$ . То есть минимальное число промахов достигается при  $k = 1$ , и, соответственно, наиболее эффективное использование кеша имеет место при последовательном обходе элементов вектора с шагом 1. Например, предположим, что `v` размещен в памяти с выравниванием по границе блока, слова содержат 4 байта, блоки кеша – 4 слова и кеш изначально пуст (холодный кеш). Тогда, независимо от организации кеша, обход элементов `v` выльется в следующую комбинацию промахов и попаданий:

<code>v[i]</code>	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
Порядок доступа ([h] – попадание; [m] – промах)	1[m]	2[h]	3[h]	4[h]	5[m]	6[h]	7[h]	8[h]

В этом примере обращение к `v[0]` приводит к промаху и соответствующий блок, содержащий `v[0]–v[3]`, загружается в кеш из памяти. Поэтому следующие три обращения будут заканчиваться попаданием в кеш. Обращение к `v[4]` вызывает очередной промах и загрузку в кеш следующего блока, после чего три обращения к последующим элементам вектора заканчиваются попаданиями, и т. д. Три из четырех обращений приведут к попаданию, и это лучшее, чего можно добиться с холодным кешем.

Наш простой пример с функцией `sumvec` иллюстрирует два важных правила, способствующих эффективному использованию кеша:

- многократное использование локальных переменных выгодно, потому что компилятор может кешировать их в регистрах (временная локальность);

- последовательный обход элементов с шагом 1 – самый оптимальный, потому что кеши на всех уровнях иерархии сохраняют данные в виде непрерывных блоков (пространственная локальность).

Пространственная локальность особенно важна в программах, обрабатывающих многомерные массивы. Рассмотрим для примера функцию `sumarrayrows`, описанную в разделе 6.2, которая суммирует элементы двумерного массива, выполняя обход элементов по строкам:

```
1 int sumarrayrows(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (i = 0; i < M; i++)
6         for (j = 0; j < N; j++)
7             sum += a[i][j];
8     return sum;
9 }
```

В языке C массивы сохраняются в памяти по строкам, поэтому вложенный цикл в этой функции выполняет такой же последовательный обход элементов с шагом 1, что и цикл в `sumvec`. Если сделать те же допущения в отношении кеша, что были сделаны для `sumvec`, то получится следующая комбинация промахов и попаданий:

$a[i][j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
$i = 0$	1[m]	2[h]	3[h]	4[h]	5[m]	6[h]	7[h]	8[h]
$i = 1$	9[m]	10[h]	11[h]	12[h]	13[m]	14[h]	15[h]	16[h]
$i = 2$	17[m]	18[h]	19[h]	20[h]	21[m]	22[h]	23[h]	24[h]
$i = 3$	25[m]	26[h]	27[h]	28[h]	29[m]	30[h]	31[h]	31[h]

Но давайте посмотрим, что получится, если внести на первый взгляд безобидное изменение – переставить циклы местами:

```
1 int sumarraycols(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (j = 0; j < N; j++)
6         for (i = 0; i < M; i++)
7             sum += a[i][j];
8     return sum;
9 }
```

В этом случае массив просматривается по столбцам, а не по строкам. При удачном стечении обстоятельств, если весь массив уместится в кеше, то частота промахов не превысит 1/4. Но если размер массива превышает размер кеша (что весьма вероятно), то каждое обращение к `a[i][j]` будет приводить к промаху!

$a[i][j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
$i = 0$	1[m]	5[m]	9[m]	13[m]	17[m]	21[m]	25[m]	29[m]
$i = 1$	2[m]	6[m]	10[m]	14[m]	18[m]	22[m]	26[m]	30[m]
$i = 2$	3[m]	7[m]	11[m]	15[m]	19[m]	23[m]	27[m]	31[m]
$i = 3$	4[m]	8[m]	12[m]	16[m]	20[m]	24[m]	28[m]	32[m]

Высокая частота промахов существенно повлияет на время выполнения. Например, на нашем настольном компьютере `sumarrayrows` обрабатывает большие массивы в 25 раз быстрее, чем `sumarraycols`. Учитывая такую большую разницу, можно сделать вывод, что программист просто обязан заботиться о локальности своих программ и в полной мере использовать ее.

### Упражнение 6.17 (решение в конце главы)

Транспонирование матриц (перемена местами строк и столбцов) широко используется в приложениях обработки сигналов и научных вычислений. Но эта операция также интересна с точки зрения локальности, потому что обход матриц выполняется как по строкам, так и по столбцам. Рассмотрим, к примеру, следующую процедуру транспонирования:

```
1 typedef int array[2][2];
2
3 void transpose1(array dst, array src)
4 {
5     int i, j;
6
7     for (i = 0; i < 2; i++) {
8         for (j = 0; j < 2; j++) {
9             dst[j][i] = src[i][j];
10        }
11    }
12 }
```

Предположим, что этот код выполняется на машине со следующими свойствами:

- `sizeof (int) = 4`;
  - массив `src` начинается с адреса 0, а массив `dst` – с адреса 16 (десятичного);
  - существует один кеш данных L1 с прямым отображением, сквозной записью, размещением строк при записи и с размером блока 8 байт;
  - кеш имеет общий размер 16 байт данных и изначально пуст;
  - обращения к массивам `src` и `dst` являются единственными источниками промахов чтения и записи.
1. Укажите для каждого значения `row` и `col`, вызывает ли обращение к `src[row][col]` и `dst[row][col]` попадание (h) или промах (m). Например, чтение из `src[0][0]` вызывает промах, как и запись в `dst[0][0]`.

массив dst			массив src		
	столбец 0	столбец 1		столбец 0	столбец 1
строка 0	m	_____	строка 0	m	_____
строка 1	_____	_____	строка 1	_____	_____

2. Решите это же упражнение для размера кеша 32 байта.

### Упражнение 6.18 (решение в конце главы)

Ядром игры *SimAquarium* является плотный цикл, вычисляющий среднее положение 256 водорослей. Производительность кеша оценивается на машине с 1024-байтным кешем данных с прямым отображением и с 16-байтными блоками ( $B = 16$ ). У вас есть следующие определения:

```
1 struct algae_position {
2     int x;
```

```

3   int y;
4 };
5
6 struct algae_position grid[16][16];
7 int total_x = 0, total_y = 0;
8 int i, j;

```

Предположим, что игра выполняется на машине со следующими свойствами:

- `sizeof (int) = 4`;
- массив `grid` начинается с адреса 0;
- изначально кеш пуст;
- единственные обращения к памяти – операции доступа к элементам массива `grid`.  
Переменные `i, j, total_x` и `total_y` хранятся в регистрах.

Определите производительность кеша для следующего кода:

```

1 for (i = 0; i < 16; i++) {
2     for (j = 0; j < 16; j++) {
3         total_x += grid[i][j].x;
4     }
5 }
6
7 for (i = 0; i < 16; i++) {
8     for (j = 0; j < 16; j++) {
9         total_y += grid[i][j].y;
10    }
11 }

```

1. Сколько всего операций чтения будет выполнено?
2. Сколько промахов чтения возникнет?
3. Определите частоту промахов.

#### Упражнение 6.19 (решение в конце главы)

Исходя из предположений, перечисленных в упражнении 6.18, определите производительность кеша для следующего кода:

```

1 for (i = 0; i < 16; i++){
2     for (j = 0; j < 16; j++) {
3         total_x += grid[j][i].x;
4         total_y += grid[j][i].y;
5     }
6 }

```

1. Сколько всего операций чтения будет выполнено?
2. Сколько промахов чтения возникнет?
3. Определите частоту промахов.
4. Как изменится частота промахов, если размер кеша увеличить вдвое?

#### Упражнение 6.20 (решение в конце главы)

Исходя из предположений, перечисленных в упражнении 6.18, определите производительность кеша для следующего кода:

```

1 for (i = 0; i < 16; i++){
2     for (j = 0; j < 16; j++) {

```



```

3         total_x += grid[i][j].x;
4         total_y += grid[i][j].y;
5     }
6 }

```

1. Сколько всего операций чтения будет выполнено?
2. Сколько промахов чтения возникнет?
3. Определите частоту промахов.
4. Как изменится частота промахов, если размер кеша увеличить вдвое?

## 6.6. Все вместе: влияние кеша на производительность программ

В этом разделе подводится итог обсуждения иерархии памяти и дается краткий обзор влияния кеша на производительность программ, выполняемых на реальных машинах.

### 6.6.1. Гора памяти

Частота следования операций чтения данных из памяти называется *пропускной способностью чтения*, или *шириной полосы чтения*. Если программа читает  $n$  байт за  $s$  секунд, то пропускная способность чтения за этот период составит  $n/s$ . Обычно она выражается в мегабайтах в секунду (Мбайт/с).

Если программа выполняет операции чтения в плотном цикле, то измерение пропускной способности чтения может дать некоторое понимание производительности системы памяти для конкретной последовательности операций чтения. В листинге 6.8 показана пара функций, измеряющих пропускную способность для определенной последовательности операций чтения.

**Листинг 6.8.** Функции, измеряющие и вычисляющие полосу пропускания операций чтения. Чтобы сгенерировать гору памяти для конкретного компьютера, нужно вызвать `run` с разными значениями `size` (соответствует временной локальности) и `stride` (соответствует пространственной локальности)

*ccode/mem/mountain/mountain.c*

```

1 long data[MAXELEM]; /* Глобальный массив для обхода */
2
3 /* test - выполняет обход первых elems элементов массива data
4  * с шагом stride, использует разворачивание цикла 4x4.
5  */
6 int test(int elems, int stride)
7 {
8     long i, sx2 = stride*2, sx3 = stride*3, sx4 = stride*4;
9     long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
10    long length = elems;
11    long limit = length - sx4;
12
13    /* Сложить сразу 4 элемента */
14    for (i = 0; i < limit; i += sx4) {
15        acc0 = acc0 + data[i];
16        acc1 = acc1 + data[i+stride];
17        acc2 = acc2 + data[i+sx2];
18        acc3 = acc3 + data[i+sx3];
19    }

```

```

20
21 /* Прибавить к сумме остальные элементы */
22 for (; i < length; i += stride) {
23     acc0 = acc0 + data[i];
24 }
25 return ((acc0 + acc1) + (acc2 + acc3));
26 }
27
28 /* run - вызывает test(elems, stride) и возвращает ширину полосы пропускания
29 * чтений (Мбайт/с). size измеряется в байтах, stride -- в элементах,
30 * a Mhz -- это тактовая частота процессора в МГц.
31 */
32 double run(int size, int stride, double Mhz)
33 {
34     double cycles;
35     int elems = size / sizeof(long);
36
37     test(elems, stride); /* Разогреть кеш */
38     cycles = fcyc2(test, elems, stride, 0); /* Вызвать test(elems, stride) */
39     return (size/stride) / (cycles/Mhz); /* Преобразовать циклы в Мбайт/с */
40 }

```

ccode/mem/mountain/mountain.c

Функция `test` генерирует последовательность чтений, просматривая первые `elems` элементы массива с шагом `stride`. Чтобы увеличить способность аппаратуры выполнять операции параллельно, во вложенном цикле используется прием развертывания  $4 \times 4$  (раздел 5.9). Функция `run` – это функция-обертка, которая вызывает функцию `test` и возвращает измеренную пропускную способность чтения. Вызов функции `test` в строке 37 разогревает кеш. Функция `fcyc2`, что вызывается в строке 18, оценивает время выполнения функции `test` в циклах синхронизации. Обратите внимание, что аргумент `size` функции `run` измеряется в байтах, а аргумент `elems` функции `test` – в элементах массива. Также в строке 39 пропускная способность пересчитывается в Мбайт/с ( $10^6$  байт/с) как  $10^6$  байт/с вместо  $2^{20}$  байт/с.

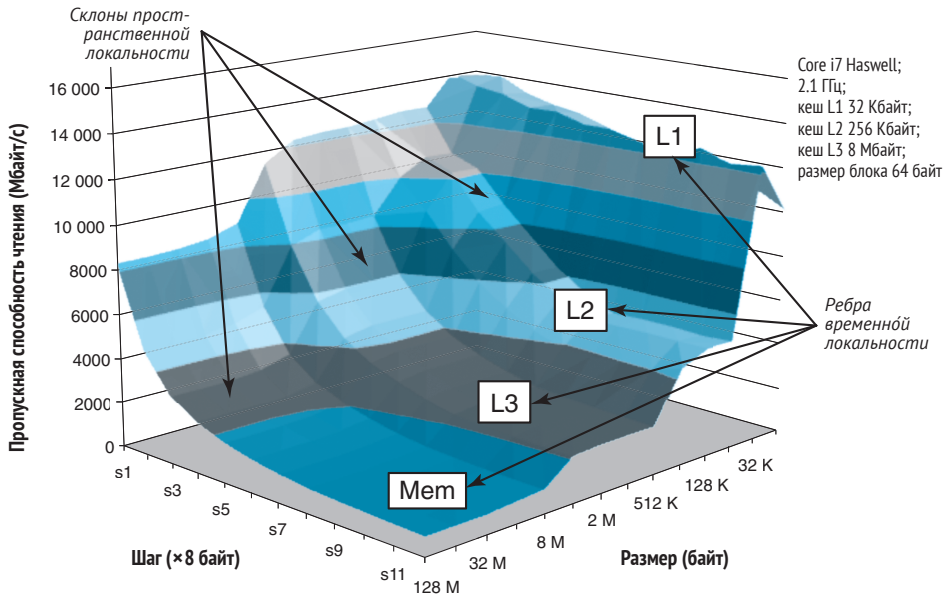
Аргументы `size` и `stride` функции `run` позволяют контролировать степень временной и пространственной локальности в последовательности операций чтения. Чем меньше значение `size`, тем меньше размер рабочего набора и тем лучше временная локальность. Чем меньше значение `stride`, тем лучше пространственная локальность. Если многократно вызывать функцию `run` с разными значениями `size` и `stride`, то можно восстановить двумерную функцию зависимости ширины полосы чтения от временной и пространственной локальностей, которую называют *горой памяти* (*memory mountain*) [112].

Каждый компьютер имеет свою уникальную гору памяти, характеризующую возможности системы памяти. Например, на рис. 6.29 показана гора памяти для системы Intel Core i7 Haswell. В этом примере значение `size` изменяется от 16 Кбайт до 128 Мбайт, а `stride` – от 1 до 12 элементов, где каждый элемент является 8-байтным значением типа `long int`.

Ландшафт горы памяти процессора Core i7 довольно разнообразен. Перпендикулярно оси размера простираются четыре *ребра*, соответствующих границам временной локальности, когда рабочий набор полностью входит в кеш L1, кеш L2, кеш L3 и основную память соответственно. Обратите внимание, что скорость чтения в верхней точке ребра кеша L1 (14 Гбайт/с) и в нижней точке ребра основной памяти (900 Мбайт/с) отличается на порядок.

К ребрам L2, L3 и основной памяти ведут склоны пространственной локальности, понижающиеся с увеличением шага. Обратите внимание, что даже когда рабочий набор

слишком велик, чтобы уместиться в каком-либо из кешей, самая высокая точка на ребре основной памяти в 8 раз выше самой нижней точки. То есть даже когда программа имеет плохую временную локальность, пространственная локальность все равно может оказывать положительное влияние на ситуацию.



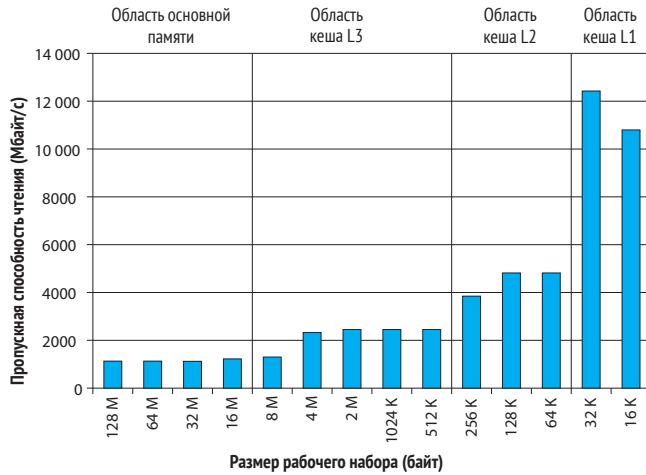
**Рис. 6.29.** Гора памяти. Показывает ширину полосы пропускания операций чтения как функцию временной и пространственной локальности

Особенно интересно выглядит почти горизонтальный профиль, простирающийся перпендикулярно оси шага для значений шага 1: пропускная способность сохраняется на уровне около 12 Гбайт/с, даже притом что размер рабочего набора превышает емкость кешей L1 и L2. Это, по-видимому, связано с аппаратным механизмом предварительной выборки в системе памяти Core i7, который автоматически определяет закономерность последовательного чтения с шагом 1 и пытается извлекать соответствующие блоки в кеш до того, как к ним произойдет обращение. Детали конкретного алгоритма предварительной выборки задокументированы, однако из профиля горы памяти ясно видно, что алгоритм лучше всего работает для небольших шагов – еще один довод в пользу последовательного доступа с шагом 1.

Если рассечь гору по определенному размеру шага и взглянуть на получившийся профиль, как показано на рис. 6.30, то можно отчетливо увидеть влияние размера кеша и временной локальности на производительность. Для размеров массива до 32 Кбайт включительно рабочий набор целиком умещается в кеше данных L1, соответственно, все операции чтения обслуживаются этим кешем с пропускной способностью около 12 Гбайт/с. Для размеров до 256 Кбайт рабочий набор целиком умещается в универсальном кеше L2, и для размеров до 8 Мбайт рабочий набор целиком умещается в универсальном кеше L3. Более крупные рабочие наборы обслуживаются главным образом из основной памяти.

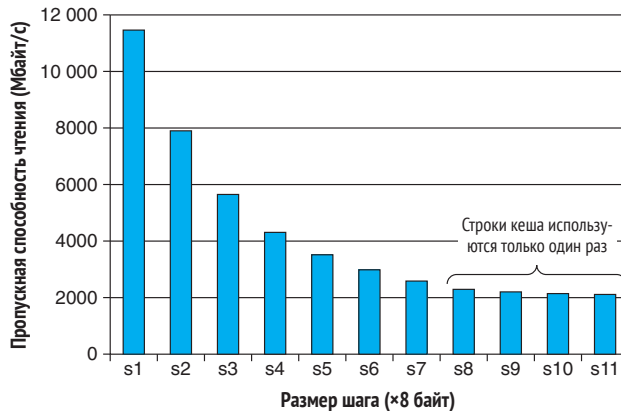
Интересно отметить падение пропускной способности на левых границах ребер кешей L2 и L3, где размеры рабочего набора равны размерам соответствующих кешей – 256 Кбайт и 8 Мбайт соответственно. Природа этих провалов не совсем понятна. Единственный способ выяснить причину – провести детальное моделирование работы

кеша, но вполне вероятно, что провал вызван конфликтами с другим кодом и строками данных.



**Рис. 6.30.** Ребра временной локальности в горе памяти. На графике показан профиль горы с рис. 6.29 вдоль линии, соответствующей размеру шага 8

Если рассечь гору в перпендикулярном направлении по определенному размеру рабочего набора, то можно получить некоторое представление о влиянии пространственной локальности на пропускную способность чтения. Например, на рис. 6.31 показан профиль горы для размера рабочего набора 4 Мбайт. Он проходит по гребню L3 на рис. 6.29 и соответствует случаю, когда рабочий набор полностью уместается в кеше L3, но слишком велик для кеша L2.



**Рис. 6.31.** Склон пространственной локальности. На графике показан профиль горы с рис. 6.29 вдоль линии, соответствующей размеру массива 4 Мбайт

Обратите внимание, как скорость чтения неуклонно снижается с увеличением шага от одного до восьми слов. В этой области горы промах чтения в L2 вызывает передачу блока из L3 в L2. За этим следует некоторое количество попаданий в L2, в зависимости от шага. С увеличением шага соотношение промахов и попаданий в L2 увеличивается. Поскольку промахи обслуживаются медленнее, чем попадания, пропускная способность чтения снижается. Как только шаг достигает восьми 8-байтных слов (в этой системе

совпадает с размером блока в 64 байта), при каждом чтении случается промах в L2 и возникает необходимость чтения данных из L3. В результате пропускная способность чтения для случаев с размером шага 8 слов и больше стабилизируется на уровне, определяемом скоростью переноса блоков из кеша L3 в кеш L2.

В завершение нашего обсуждения горы памяти можно отметить, что производительность системы памяти нельзя характеризовать некоторым постоянным числом. В действительности график зависимости производительности памяти от временной и пространственной локальностей напоминает горный ландшафт, и значения производительности в разных точках могут отличаться на порядок. Опытные программисты стараются структурировать свои программы так, чтобы их производительность пролегла вдоль высоких хребтов, а не долин. А для этого следует обеспечить временную локальность, чтобы часто используемые слова извлекались из кеша L1, и пространственную локальность, чтобы как можно больше слов извлекалось из одной и той же строки кеша L1.

#### Упражнение 6.21 (решение в конце главы)

Используя гору памяти на рис 6.29, оцените время в циклах синхронизации, необходимое для чтения 8-байтного слова из кеша данных L1.

### 6.6.2. Переупорядочение циклов для улучшения пространственной локальности

Рассмотрим задачу умножения пары матриц  $C = AB$  с размерами  $n \times n$  для случая  $n = 2$ .

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{21} \\ b_{21} & b_{22} \end{bmatrix},$$

где

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

Функция умножения матриц обычно реализуется с использованием трех вложенных циклов с индексами  $i$ ,  $j$  и  $k$ . Перестановкой циклов и некоторыми другими незначительными изменениями в коде можно создать шесть функционально эквивалентных версий умножения матриц (табл. 6.11). Каждая версия уникально идентифицируется порядком вложения циклов.

**Таблица 6.11.** Шесть версий процедур перемножения матриц. Каждая версия уникально идентифицируется порядком следования циклов

(a) Версия *ijk*

(b) Версия *jik*

```
code/mem/matmult/mm.c
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++) {
3         sum = 0.0;
4         for (k = 0; k < n; k++)
5             sum += A[i][k]*B[k][j];
6         C[i][j] += sum;
7     }
```

```
code/mem/matmult/mm.c
1 for (j = 0; j < n; j++)
2     for (i = 0; i < n; i++) {
3         sum = 0.0;
4         for (k = 0; k < n; k++)
5             sum += A[i][k]*B[k][j];
6         C[i][j] += sum;
7     }
```

(c) Версия *jki*

```

code/mem/matmult/mm.c
1 for (j = 0; j < n; j++)
2   for (k = 0; k < n; k++) {
3     r = B[k][j];
4     for (i = 0; i < n; i++)
5       C[i][j] += A[i][k]*r;
6   }
code/mem/matmult/mm.c

```

(d) Версия *kji*

```

code/mem/matmult/mm.c
1 for (k = 0; k < n; k++)
2   for (j = 0; j < n; j++) {
3     r = B[k][j];
4     for (i = 0; i < n; i++)
5       C[i][j] += A[i][k]*r;
6   }
code/mem/matmult/mm.c

```

(e) Версия *kij*

```

code/mem/matmult/mm.c
1 for (k = 0; k < n; k++)
2   for (i = 0; i < n; i++) {
3     r = A[i][k];
4     for (j = 0; j < n; j++)
5       C[i][j] += r*B[k][j];
6   }
code/mem/matmult/mm.c

```

(f) Версия *ikj*

```

code/mem/matmult/mm.c
1 for (i = 0; i < n; i++)
2   for (k = 0; k < n; k++) {
3     r = A[i][k];
4     for (j = 0; j < n; j++)
5       C[i][j] += r*B[k][j];
6   }
code/mem/matmult/mm.c

```

С внешней точки зрения эти шесть версий очень похожи. Так как сложение ассоциативно, все версии вычисляют идентичный результат<sup>1</sup>. Каждая версия выполняет  $O(n^3)$  операций и одинаковое количество операций сложения и умножения. Каждый из  $n^2$  элементов матриц  $A$  и  $B$  читается  $n$  раз. Каждый из  $n^2$  элементов  $C$  вычисляется как сумма  $n$  значений. Однако если проанализировать поведение итераций самого внутреннего цикла, то обнаружится, что имеют место различия в количестве обращений к памяти и локальности. Для целей дальнейшего анализа примем следующие допущения:

- все массивы – это массивы  $n \times n$  чисел типа `double`, где `sizeof(double) = 8`;
- существует только один кеш с размером блока 32 байт ( $B = 32$ );
- размер массива  $n$  настолько большой, что одна строка матрицы не умещается в кеше;
- компилятор сохраняет локальные переменные в регистрах, поэтому ссылки на локальные переменные внутри циклов не приводят к выполнению инструкций загрузки или сохранения.

В табл. 6.12 показаны общие результаты анализа внутренних циклов. Обратите внимание, что шесть версий объединяются в три эквивалентных класса, соответствующих парам матриц, к которым осуществляется доступ во внутреннем цикле. Например, версии *ijk* и *jik* являются членами класса  $AB$ , потому что они обращаются к массивам  $A$  и  $B$  (но не к  $C$ ) в самом внутреннем цикле. Для каждого класса мы подсчитали количество операций загрузки (чтения) и сохранения (записи) в каждой итерации внутреннего цикла, количество обращений к  $A$ ,  $B$  и  $C$ , которые приводят к промахам кеша в каждой итерации цикла, а также общее число промахов в каждой итерации.

<sup>1</sup> Как мы узнали в главе 2, сложение чисел с плавающей точкой коммутативно, но, по большому счету, не ассоциативно. Однако если в матрицах не смешиваются очень большие и очень маленькие значения, как это часто случается с матрицами физических свойства, то допущение ассоциативности вполне обосновано.

**Таблица 6.12.** Результаты анализа внутренних циклов в процедурах умножения матриц. Шесть версий делятся на три эквивалентных класса по парам массивов, к которым осуществляется доступ во внутреннем цикле

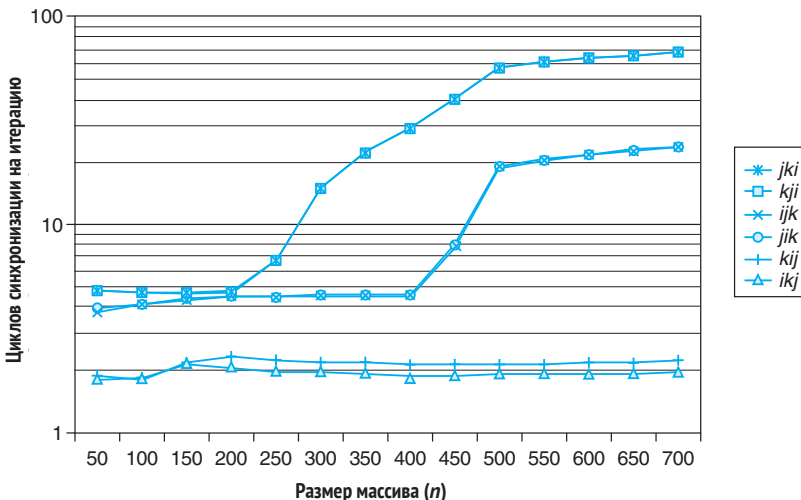
Класс процедур умножения матриц	На итерацию					
	Загрузок	Сохранений	Промашов А	Промашов В	Промашов С	Всего промашов
<i>ijk</i> & <i>jik</i> (AB)	2	0	0,25	1,00	0,00	1,25
<i>jki</i> & <i>kji</i> (AC)	2	1	1,00	0,00	1,00	2,00
<i>kij</i> & <i>ikj</i> (BC)	2	1	0,00	0,25	0,25	0,50

Внутренние циклы процедур класса AB (табл. 6.11 (а) и 6.11 (b)) просматривают строки массива А с шагом 1. Поскольку в каждом блоке кеша хранится четыре 8-байтных слова, частота промахов для А составляет 0,25 на итерацию. С другой стороны, внутренний цикл, просматривающий столбцы В, перебирает элементы с шагом  $n$ . Поскольку  $n$  – большое число, каждое обращение к массиву В приводит к промаху (всего 1,25 промаха на итерацию).

Внутренние циклы в процедурах класса AC (табл. 6.11 (с) и 6.11 (d)) демонстрируют определенные проблемы. В каждой итерации выполняются две операции загрузки и одна операция сохранения (в отличие от процедур класса AB, выполняющих 2 операции загрузки и ни одной операции сохранения). Внутренний цикл просматривает столбцы А и С с шагом  $n$ . В результате каждая операция загрузки сталкивается с промахом, что в итоге дает 2 промаха в каждой итерации. Обратите внимание, что перестановка циклов ухудшила пространственную локальность (по сравнению с процедурами класса AB).

Процедуры BC (табл. 6.11 (е) и 6.11 (f)) выглядят довольно интересно: с двумя операциями загрузки и одной операцией сохранения они, в отличие от процедур AB, выполняют на одну операцию с памятью больше. Однако, поскольку внутренний цикл в этих процедурах просматривает матрицы В и С по строкам с шагом 1, частота промахов для каждого массива составляет всего 0,25, что в сумме дает частоту промахов 0,50 на итерацию.

На рис. 6.32 показаны графики зависимости общей производительности (в циклах синхронизации на итерацию) от размеров матриц для разных версий функций умножения матриц в системе Core i7.



**Рис. 6.32.** Производительность функций умножения матриц в системе Core i7

Отметим несколько важных моментов на этом графике:

- для больших значений  $n$  самая быстродействующая версия работает в три раза быстрее самой медленной версии, несмотря на то что все они выполняют одинаковое количество арифметических операций с плавающей точкой;
- пары версий с одинаковым количеством обращений к памяти и частотой промахов на итерацию показывают примерно одинаковую производительность;
- две версии с худшими показателями в терминах количества обращений к памяти и частоты промахов выполняются значительно медленнее, чем остальные четыре версии, в которых выполняется меньше операций с памятью или возникает меньше промахов;
- частота промахов точнее отражает производительность, чем общее количество обращений к памяти. Например, процедуры класса  $BC$  с 0,5 промаха на итерацию работают намного быстрее, чем процедуры класса  $AB$  с 1,25 промаха, даже притом что подпрограммы класса  $BC$  выполняют больше обращений к памяти во внутреннем цикле (две операции загрузки и одна – сохранения), чем процедуры класса  $AB$  (две операции загрузки);
- для больших значений  $n$  производительность самой быстрой пары версий ( $kij$  и  $ikj$ ) постоянна. Несмотря на то что массив намного больше любого из кешей SRAM, аппаратный механизм предварительной выборки распознает закономерность последовательного доступа с шагом 1 и успевает извлекать соответствующие блоки в кеш до того, как к ним произойдет обращение во внутреннем цикле. Это потрясающее достижение инженеров Intel, разработавших такую систему памяти, еще больше стимулирует программистов писать программы с хорошей пространственной локальностью.

**Приложение в интернете MEM:BLOCKING.** Использование разбивки на блоки для улучшения временной локальности

Существует довольно интересный прием улучшения временной локальности вложенных циклов, называемый *разбивкой на блоки*. Суть его состоит в том, чтобы организовать структуры данных в программе в большие фрагменты, называемые *блоками*. (В этом контексте термином «блок» обозначается фрагмент данных на уровне приложения, а не блок кеша.) Программа структурирована таким образом, что загружает фрагмент в кеш L1, выполняет с этим фрагментом все необходимые операции чтения и записи, затем отбрасывает фрагмент, загружает следующий и т. д.

В отличие от простых преобразований циклов для улучшения пространственной локальности, разбивка на блоки затрудняет чтение и понимание кода. По данной причине этот прием лучше использовать для оптимизации компиляторов или часто выполняемых библиотечных подпрограмм. Разбивка на блоки не улучшает производительность умножения матриц на Core i7 из-за наличия сложного аппаратного механизма предварительной выборки. Однако этот метод стоит того, чтобы его изучали и знали, потому что он может помочь значительно увеличить производительность в системах без механизма предварительной выборки.

### 6.6.3. Использование локальности в программах

Как мы видели, система памяти структурирована в виде иерархии, на вершине которой находятся быстродействующие устройства небольшого размера, а внизу – медленные устройства большой емкости. Вследствие такой иерархической организации производительность памяти нельзя характеризовать некоторым постоянным числом. В действительности производительность зависит от локальности программы и может изменяться на порядки. Программы с хорошей локальностью получают большую часть



данных из быстрых кешей. Программы с плохой локальностью получают большую часть данных из сравнительно медленной основной памяти DRAM.

Программисты, понимающие природу иерархии памяти, могут использовать это понимание для создания более эффективных программ, независимо от конкретной организации системы памяти. В частности, мы можем порекомендовать:

- сосредоточиться на внутренних циклах, где выполняется большая часть вычислений и обращений к памяти;
- пытаться добиться максимальной пространственной локальности программ путем последовательного чтения данных в том порядке, в каком они хранятся в памяти;
- пытаться добиться максимальной временной локальности программ частым использованием данных после того, как они будут прочитаны из памяти.

## 6.7. Итоги

К основным технологиям хранения относятся: память с произвольным доступом (RAM), энергонезависимая память (ROM) и диски. Память с произвольным доступом имеет две основные формы. Статическая память RAM (SRAM) – более быстродействующая и дорогостоящая и используется как кеш. Динамическая память RAM (DRAM) – менее быстродействующая и более дешевая, используется в качестве основной памяти и графических буферов. Энергонезависимая память ROM (ее также называют постоянной памятью) сохраняет информацию даже при отключении питания и используется для хранения «вшитых» программ. Вращающиеся диски – энергонезависимые устройства, способные хранить огромные объемы данных, имеют самую низкую стоимость хранения одного бита информации, но время доступа к данным на них значительно больше времени доступа к DRAM. Твердотельные диски (SSD) на основе энергонезависимой флеш-памяти становятся все более привлекательной альтернативой вращающимся дискам для некоторых приложений.

В общем случае чем выше скорость работы устройства хранения, тем меньше его емкость и тем выше стоимость хранения одного бита. Соотношение цена/производительность разных технологий хранения меняется с разной скоростью. В частности, динамика уменьшения времени доступа к DRAM и дискам отстает от динамики увеличения тактовой частоты процессоров. Системы сокращают эти разрывы, организуя память в виде иерархии, в которой меньшие по объему и более быстродействующие устройства располагаются вверху, а более объемные и медленные – внизу. Так как грамотно написанные программы обладают хорошей локальностью, большая часть данных обслуживается на верхних уровнях иерархии, благодаря чему вся система памяти может работать со скоростью верхних уровней, имея стоимость и емкость нижних уровней.

Программисты могут значительно увеличить время выполнения программ, добиваясь хорошей временной и пространственной локальности. Здесь крайне важное значение имеет использование кеш-памяти на основе SRAM. Программы, выбирающие данные в основном из кеша L1, могут выполняться на порядок быстрее, чем программы, выбирающие данные из памяти.

## Библиографические заметки

Технологии хранения очень быстро совершенствуются. По нашему опыту, лучшими источниками технической информации являются веб-сайты производителей. Такие компании, как Micron, Toshiba и Samsung, предоставляют очень большие объемы технической информации об устройствах памяти. Сайты Seagate и Western Digital предоставляют такую же полезную информацию о дисках.

Подробную информацию о технологиях памяти можно найти в учебниках по схемотехнике и логическому проектированию [58, 89]. В *IEEE Spectrum* опубликована серия исследовательских статей о DRAM [55]. International Symposiums on Computer Architecture (ISCA) и High Performance Computer Architecture (HPCA) – самые известные форумы, посвященные характеристикам производительности памяти DRAM [28, 29, 18].

Уилкис (Wilkes) написал первую статью, посвященную кеш-памяти [117]. Смит (Smith) опубликовал классический исследовательский труд [104]. Пржибильский (Przybylski) написал хорошую книгу по проектированию кешей [86]. В работах Хеннеси (Hennesy) и Паттерсона (Patterson) также подробно обсуждаются вопросы проектирования кешей. Левинталь (Levinthal) написал исчерпывающее руководство по производительности для Intel Core i7 [70].

Стрикер (Strieker) ввел идею горы памяти как исчерпывающей характеристики системы памяти в [112] и предложил неформальный термин «гора памяти» в последующих выпусках работы. Разработчики компиляторов работают над улучшением локальности путем автоматического применения преобразований к коду, которые мы рассматривали в разделе 6.6 [22, 32, 66, 72, 79, 87, 119]. Картер (Carter) с коллегами предложил контроллер памяти с поддержкой кеша [17]. Другие исследователи разработали алгоритмы *без учета структуры кеша*, которые обеспечивают хорошую производительность без использования явных знаний организации базовой кеш-памяти [30, 38, 39, 9].

Существует масса литературы по построению и использованию дисковых накопителей. Многие исследователи ищут способы объединения отдельных дисков в более крупные, надежные и безопасные пулы памяти [20, 40, 41, 83, 121]. Другие ищут способы использования кешей и локальности для повышения эффективности доступа к дискам [12, 21]. Такие системы, как Exokernel, обеспечивают более эффективное управление диском и ресурсами памяти на пользовательском уровне [57]. Такие системы, как Andrew File System [78] и Coda [94], распространяют иерархию памяти на компьютерные сети и ноутбуки. Шиндлер (Schindler) и Гэнгер (Ganger) разработали интересный инструмент, который автоматически характеризует геометрию и производительность дисковых накопителей SCSI [95]. Некоторые исследователи изучают методы создания и использования твердотельных накопителей на основе флеш-памяти [8, 81].

## Домашние задания

### Упражнение 6.22 ♦♦

Представьте, что вам дано задание спроектировать вращающийся диск, в котором количество битов на дорожку постоянно. Вы знаете, что это количество битов определяется длиной окружности самой внутренней дорожки, которую можно принять равной окружности «дырки» в центре диска. То есть если дырку в центре диска сделать больше, тогда количество битов в каждой дорожке увеличится, однако общее количество дорожек уменьшится. Приняв за  $r$  радиус пластины, а за  $x \cdot r$  – радиус дырки, определите, какое значение  $x$  обеспечит максимальную емкость диска.

### Упражнение 6.23 ♦

Оцените среднее время (в мс) доступа к сектору на следующем диске:

Параметр	Значение
Скорость вращения	15 000 об/мин
Среднее время позиционирования ( $T_{avg\ seek}$ )	4 мс
Среднее количество секторов на дорожке	800

### Упражнение 6.24 ♦♦

Предположим, что файл размером 2 Мбайт, состоящий из 512-байтных логических блоков, хранится на диске со следующими характеристиками:

Параметр	Значение
Скорость вращения	15 000 об/мин
Среднее время позиционирования ( $T_{avg\ seek}$ )	4 мс
Среднее количество секторов на дорожке	1000
Поверхности	8
Размер сектора	512 байт

Для каждого из случаев, приведенных ниже, предположим, что программа читает логические блоки последовательно, один за другим, и время позиционирования головки на первом блоке равно  $T_{avg\ seek} + T_{avg\ Rotation}$ \*

1. *Лучший случай*: оцените оптимальное время (в мс) чтения файла при последовательном отображении логических блоков в секторы диска.
2. *Произвольный случай*: оцените время (в мс) чтения файла при произвольном (случайном) отображении логических блоков в секторы диска.

### Упражнение 6.25 ♦

В следующей таблице приводятся параметры нескольких разных кешей. Заполните недостающие значения в таблице. Напомню, что  $m$  – это размер адреса физической памяти в битах,  $C$  – размер кеша (количество байт данных),  $B$  – размер блока в байтах,  $E$  – ассоциативность,  $S$  – количество наборов в кеше,  $t$  – размер тега в битах,  $s$  – размер индекса набора в битах и  $b$  – размер смещения блока в битах.

Кеш	$m$	$C$	$B$	$E$	$S$	$t$	$s$	$b$
1	32	1024	4	4	_____	_____	_____	_____
2	32	1024	4	256	_____	_____	_____	_____
3	32	1024	8	1	_____	_____	_____	_____
4	32	1024	8	128	_____	_____	_____	_____
5	32	1024	32	1	_____	_____	_____	_____
6	32	1024	32	4	_____	_____	_____	_____

### Упражнение 6.26 ♦

В следующей таблице приводятся параметры нескольких разных кешей. Заполните недостающие значения в таблице. Напомню, что  $m$  – это размер адреса физической памяти в битах,  $C$  – размер кеша (количество байт данных),  $B$  – размер блока в байтах,  $E$  – ассоциативность,  $S$  – количество наборов в кеше,  $t$  – размер тега в битах,  $s$  – размер индекса набора в битах и  $b$  – размер смещения блока в битах.

Кеш	$m$	$C$	$B$	$E$	$S$	$t$	$s$	$b$
1	32	_____	8	1	_____	21	8	3
2	32	2048	_____	_____	128	23	7	2
3	32	1024	2	8	64	_____	_____	1
4	32	1024	_____	2	16	23	4	_____

Упражнение 6.27 ♦

Используя кеш, описанный в упражнении 6.12:

- 1. Перечислите все шестнадцатеричные адреса ячеек памяти, которые попадут в набор 1.
- 2. Перечислите все шестнадцатеричные адреса ячеек памяти, которые попадут в набор 6.

Упражнение 6.28 ♦♦

Используя кеш, описанный в упражнении 6.12:

- 1. Перечислите все шестнадцатеричные адреса ячеек памяти, которые попадут в набор 2.
- 2. Перечислите все шестнадцатеричные адреса ячеек памяти, которые попадут в набор 4.
- 3. Перечислите все шестнадцатеричные адреса ячеек памяти, которые попадут в набор 5.
- 4. Перечислите все шестнадцатеричные адреса ячеек памяти, которые попадут в набор 7.

Упражнение 6.29 ♦♦

Представьте, что у вас есть система со следующими свойствами:

- память имеет байтовую адресацию;
- доступ к памяти осуществляется к 1-байтным словам (не к 4-байтным);
- адреса имеют размер 12 бит;
- кеш – с 2-строчными ассоциативными наборами ( $E = 2$ ), размером блока 4 байта ( $B = 4$ ) и четырьмя наборами ( $S = 4$ ).

Вот содержимое кеша со всеми адресами, тегами и значениями, указанными в шестнадцатеричной форме:

Индекс набора	Тег	Достов.	Байт 0	Байт 1	Байт 2	Байт 3
0	00	1	40	41	42	43
	83	1	FE	97	CC	D0
1	00	1	44	45	46	47
	83	0	–	–	–	–
2	00	1	48	49	4A	4B
	40	0	–	–	–	–
3	FF	1	9A	C0	03	FF
	00	0	–	–	–	–

- 1. Следующая таблица иллюстрирует структуру адреса (одна ячейка – один бит). Покажите в этой таблице (поставив метки), какие биты будут использоваться для:
  - определения смещения в блоке (CO);
  - определения индекса набора (CI);
  - определения тега (CT).

12	11	10	9	8	7	6	5	4	3	2	1	0

2. Для каждого из следующих обращений к памяти укажите, произойдет ли попадание в кеш. Также укажите прочитанное значение, если его можно определить по информации о кеше.

Операция	Адрес	Попадание?	Прочитанное значение (или прочерк)
Чтение	0x834	_____	_____
Запись	0x836	_____	_____
Чтение	0xFFD	_____	_____

Упражнение 6.30 ♦

Представьте, что у вас есть система со следующими свойствами:

- память имеет байтовую адресацию;
- доступ к памяти осуществляется к 1-байтным словам (не к 4-байтным);
- адреса имеют размер 13 бит;
- кеш – с 4-строчными ассоциативными наборами ( $E = 4$ ), размером блока 4 байта ( $B = 4$ ) и восемью наборами ( $S = 8$ ).

Ниже приводится содержимое кеша со всеми адресами, тегами и значениями, указанными в шестнадцатеричной форме. В столбце «Индекс» указаны индексы всех четырехстрочных наборов. В столбце «Тег» – значения тегов строк. В столбце «V» – значения битов достоверности в строках. В столбце «Байты 0–3» – значения байтов в каждой строке от младшего (0) слева до старшего (3) справа.

Кеш с 4-строчными ассоциативными наборами

Индекс	Тег	V	Байты 0–3	Тег	V	Байты 0–3	Тег	V	Байты 0–3	Тег	V	Байты 0–3
0	F0	1	ED 32 0A A2	8A	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	0	25 44 6F 1A
1	BC	0	03 3E CD 38	A0	0	16 7B ED 5A	BC	1	8E 4C DF 18	E4	1	FB B7 12 02
2	BC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E
3	BE	0	2F 7E 3D A8	C0	1	27 95 A4 74	C4	0	07 11 6B D8	BC	0	C7 B7 AF C2
4	7E	1	32 21 1C 2C	8A	1	22 C2 DC 34	BC	1	BA DD 37 D8	DC	0	E7 A2 39 BA
5	98	0	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	BC	1	48 16 81 0A
6	38	0	5D 4D F7 DA	BC	1	69 C2 8C 74	8A	1	A8 CE 7F DA	38	1	FA 93 EB 48
7	8A	1	04 2A 32 6A	9E	0	B1 86 56 0E	CC	1	96 30 47 F2	BC	1	F8 1D 42 30

1. Определите размер этого кеша в байтах (C).
2. Следующая таблица иллюстрирует структуру адреса (одна ячейка – один бит). Покажите в этой таблице (поставив метки), какие биты будут использоваться для:
- определения смещения в блоке (CO);
  - определения индекса набора (CI);
  - определения тега (CT).

12	11	10	9	8	7	6	5	4	3	2	1	0

Упражнение 6.31 ♦♦

Предположим, что программа, выполняющаяся в системе с кешем из упражнения 6.30, обращается к 1-байтному слову по адресу 0x071A. Определите, к какому эле-

менту кеша произойдет обращение и какое значение будет возвращено из кеша. Если произойдет промах кеша, то поставьте прочерк в поле «Байт из кеша». *Подсказка:* обратите внимание на биты достоверности!

1. Структура адреса (одна ячейка – один бит):

12	11	10	9	8	7	6	5	4	3	2	1	0	

2. Обращение к ячейке памяти:

Параметр	Значение
Смещение в блоке (CO)	0x_____
Индекс набора (CI)	0x_____
Тег (CT)	0x_____
Попадание в кеш? (Д/Н)	_____
Байт из кеша	0x_____

### Упражнение 6.32 ♦♦

Решите задание в упражнении 6.31 для адреса 0x16E8.

1. Структура адреса (одна ячейка – один бит):

12	11	10	9	8	7	6	5	4	3	2	1	0	

2. Обращение к ячейке памяти:

Параметр	Значение
Смещение в блоке (CO)	0x_____
Индекс набора (CI)	0x_____
Тег (CT)	0x_____
Попадание в кеш? (Д/Н)	_____
Байт из кеша	0x_____

### Упражнение 6.33 ♦♦

Для кеша в упражнении 6.30 перечислите восемь адресов в памяти (в шестнадцатеричной форме), при обращении к которым будет выбираться набор 2.

### Упражнение 6.34 ♦♦

Взгляните на следующую процедуру транспонирования матрицы:

```

1 typedef int array[4][4];
2
3 void transpose2(array dst, array src)
4 {
5     int i, j;
6
7     for (i = 0; i < 4; i++) {
8         for (j = 0; j < 4; j++) {
9             dst[j][i] = src[i][j];
10        }
11    }
12 }
```

Предположим, что этот код выполняется на машине со следующими свойствами:

- `sizeof(int) = 4`;
  - массив `src` начинается с адреса 0, а массив `dst` – с адреса 64 (десятичный);
  - в системе имеется единственный кеш данных L1 с прямым отображением, сквозной записью и с записью с размещением; размер блока в строках равен 16 байт;
  - общий размер кеша – 32 байта, и изначально он пуст;
  - массивы `src` и `dst` – единственные источники операций чтения и записи соответственно.
1. Укажите для каждого значения `row` и `col`, вызывает ли обращение к `src[row][col]` и `dst[row][col]` попадание (h) или промах (m). Например, чтение из `src[0][0]` вызывает промах, как и запись в `dst[0][0]`.

массив dst					массив src				
	столбец 0	столбец 1	столбец 2	столбец 3		столбец 0	столбец 1	столбец 2	столбец 3
строка 0	m	_____	_____	_____	строка 0	m	_____	_____	_____
строка 1	_____	_____	_____	_____	строка 1	_____	_____	_____	_____
строка 2	_____	_____	_____	_____	строка 2	_____	_____	_____	_____
строка 3	_____	_____	_____	_____	строка 3	_____	_____	_____	_____

### Упражнение 6.35 ♦♦

Решите задание в упражнении 6.34 для кеша с общим размером 128 байт.

массив dst					массив src				
	столбец 0	столбец 1	столбец 2	столбец 3		столбец 0	столбец 1	столбец 2	столбец 3
строка 0	_____	_____	_____	_____	строка 0	_____	_____	_____	_____
строка 1	_____	_____	_____	_____	строка 1	_____	_____	_____	_____
строка 2	_____	_____	_____	_____	строка 2	_____	_____	_____	_____
строка 3	_____	_____	_____	_____	строка 3	_____	_____	_____	_____

### Упражнение 6.36 ♦♦

Это упражнение поможет вам проверить свою способность определять, насколько эффективно код на языке C использует кеш. Вам дается следующий код:

```

1 int x[2][128];
2 int i;
3 int sum = 0;
4
5 for (i = 0; i < 128; i++) {
6     sum += x[0][i] * x[1][i];
7 }
```

и определены следующие условия:

- `sizeof(int) = 4`;
- массив `x` начинается с адреса `0x0` и хранится в порядке по строкам;
- в каждом из перечисленных ниже случаев кеш изначально пуст;
- массив `x` – единственный источник обращений к памяти; все остальные переменные хранятся в регистрах.

Исходя из этих условий, оцените частоту промахов в следующих случаях:

- 1. Случай 1: размер кеша 512 байт; это кеш с прямым отображением и с блоками размером 16 байт. Какова будет частота промахов?
- 2. Случай 2: как изменится частота промахов, если размер кеша увеличить до 1024 байт?
- 3. Случай 3: теперь допустим, что размер кеша 512 байт; это ассоциативный кеш с 2-строчными наборами, использующий стратегию вытеснения LRU, и с блоками размером 16 байт. Какова будет частота промахов?
- 4. В случае 3 поможет ли увеличение размера кеша уменьшить частоту промахов? Почему?
- 5. В случае 3 поможет ли увеличение размера блока уменьшить частоту промахов? Почему?

Упражнение 6.37 ♦

Это еще одно упражнение, которое поможет вам проверить свою способность определять, насколько эффективно код на языке C использует кеш. Допустим, что вам дано задание оценить три функции суммирования из листинга 6.9, исходя из следующих условий:

- sizeof(int) = 4;
- общий размер кеша составляет 4 Кбайт; это кеш с прямым отображением и с блоками размером 16 байт;
- внутри двух циклов обращения к памяти происходят только при доступе к элементам массива; индексы циклов и значение sum хранятся в регистрах;
- массив находится в памяти, начиная с адреса 0x08000000.

Оцените частоту промахов кеша для двух случаев:  $N = 64$  и  $N = 60$ .

Функция	$N = 64$	$N = 60$
sumA	_____	_____
sumB	_____	_____
sumC	_____	_____

Листинг 6.9. Исходный код функций для упражнения 6.37

```
1 typedef int array_t[N][N];
2
3 int sumA(array_t a)
4 {
5     int i, j;
6     int sum = 0;
7     for (i = 0; i < N; i++)
8         for (j = 0; j < N; j++) {
9             sum += a[i][j];
10        }
11    return sum;
12 }
13
14 int sumB(array_t a)
15 {
16     int i, j;
17     int sum = 0;
18     for (j = 0; j < N; j++)
```



```

19         for (i = 0; i < N; i++) {
20             sum += a[i][j];
21         }
22     return sum;
23 }
24
25 int sumC(array_t a)
26 {
27     int i, j;
28     int sum = 0;
29     for (j = 0; j < N; j+=2)
30         for (i = 0; i < N; i+=2) {
31             sum += (a[i][j] + a[i+1][j]
32                   + a[i][j+1] + a[i+1][j+1]);
33         }
34     return sum;
35 }

```

### Упражнение 6.38 ♦

В компании ЗМ решили напечатать пояснительные заметки в желтых квадратах на белых листах бумаги. В процессе печати необходимо задать значения СМΥК (голубой (cyan), пурпурный (magenta), желтый (yellow), черный (black)) для каждой точки квадрата. Они наняли вас для оценки эффективности следующих алгоритмов на машине с 2048-байтным кешем данных с прямым отображением и 32-байтными блоками. Даны следующие определения:

```

1 struct point_color {
2     int c;
3     int m;
4     int y;
5     int k;
6 };
7
8 struct point_color square[16][16];
9 int i, j;

```

Допустим следующее:

- `sizeof(int) = 4`;
- `square` располагается в памяти, начиная с адреса 0;
- изначально кеш пуст;
- обращения к памяти происходят только при доступе к элементам массива `square`. Переменные `i` и `j` хранятся в регистрах.

Определите эффективность использования кеша следующим кодом:

```

1 for (i = 0; i < 16; i++){
2     for (j = 0; j < 16; j++) {
3         square[i][j].c = 0;
4         square[i][j].m = 0;
5         square[i][j].y = 1;
6         square[i][j].k = 0;
7     }
8 }

```

1. Определите общее количество операций записи.
2. Определите общее количество промахов записи в кеш.
3. Определите частоту промахов.

### Упражнение 6.39 ♦

Исходя из условий, перечисленных в упражнении 6.38, определите эффективность использования кеша следующим кодом:

```

1 for (i = 0; i < 16; i++){
2     for (j = 0; j < 16; j++) {
3         square[j][i].c = 0;
4         square[j][i].m = 0;
5         square[j][i].y = 1;
6         square[j][i].k = 0;
7     }
8 }
```

1. Определите общее количество операций записи.
2. Определите общее количество промахов записи в кеш.
3. Определите частоту промахов.

### Упражнение 6.40 ♦

Исходя из условий, перечисленных в упражнении 6.38, определите эффективность использования кеша следующим кодом:

```

1 for (i = 0; i < 16; i++) {
2     for (j = 0; j < 16; j++) {
3         square[i][j].y = 1;
4     }
5 }
6 for (i = 0; i < 16; i++) {
7     for (j = 0; j < 16; j++) {
8         square[i][j].c = 0;
9         square[i][j].m = 0;
10        square[i][j].k = 0;
11    }
12 }
```

1. Определите общее количество операций записи.
2. Определите общее количество промахов записи в кеш.
3. Определите частоту промахов.

### Упражнение 6.41 ♦♦

Представьте, что вы пишете новую трехмерную игру в надежде прославиться и неплохо заработать. В данный момент вы работаете над функцией очистки буфера дисплея, перед тем как нарисовать следующий кадр. Экран представлен массивом 640×480 пикселей. Машина, с которой вы работаете, имеет кеш размером 64 Кбайт с прямым отображением и с 4-байтными строками. В программе используются следующие структуры:

```

1 struct pixel {
2     char r;
3     char g;
4     char b;
5     char a;
6 };
7
8 struct pixel buffer[480][640];
9 int i, j;
10 char *cptr;
11 int *iptr;
```

Допустим следующее:

- `sizeof(char) = 1` и `sizeof(int) = 4`;
- `buffer` находится в памяти, начиная с адреса 0;
- кеш изначально пуст;
- обращения к памяти происходят только при доступе к элементам массива `buffer`. Переменные `i`, `j`, `cptr` и `iptr` и хранятся в регистрах.

Какой процент операций записи в следующем коде столкнется с промахами кеша?

```

1 for (j = 0; j < 640; j++) {
2     for (i = 0; i < 480; i++){
3         buffer[i][j].r = 0;
4         buffer[i][j].g = 0;
5         buffer[i][j].b = 0;
6         buffer[i][j].a = 0;
7     }
8 }
```

### Упражнение 6.42 ♦♦

Исходя из условий, перечисленных в упражнении 6.41, определите процент промахов кеша при записи в следующем коде:

```

1 char *cptr = (char *) buffer;
2 for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
3     *cptr = 0;
```

### Упражнение 6.43 ♦♦

Исходя из условий, перечисленных в упражнении 6.41, определите процент промахов кеша при записи в следующем коде:

```

1 int *iptr = (int *)buffer;
2 for (; iptr < ((int *)buffer + 640*480); iptr++)
3     *iptr = 0;
```

### Упражнение 6.44 ♦♦♦

Загрузите программу `mountain` с веб-сайта CS:APP и запустите ее в своей системе PC/Linux. Используя полученные результаты, оцените размеры кешей в своей системе.

### Упражнение 6.45 ♦♦♦♦

Для решения этого упражнения вам понадобится использовать идеи оптимизации приложений, интенсивно использующих память, изученные в главах 5 и 6. Ниже приводится процедура копирования и транспонирования элементов матрицы  $N \times N$  типа `int`. То есть для исходной матрицы  $S$  и матрицы назначения  $D$  необходимо скопировать каждый элемент  $s_{ij}$  в  $d_{ji}$ . Реализовать это можно в виде простого цикла:

```

1 void transpose(int *dst, int *src, int dim)
2 {
3     int i, j;
4
5     for (i = 0; i < dim; i++)
6         for (j = 0; j < dim; j++)
7             dst[j*dim + i] = src[i*dim + j];
8 }
```

Эта процедура получает три аргумента: указатели на матрицы `dst` и `src`, а также размер матриц  $N$  (`dim`). Ваша задача – разработать функцию транспонирования, выполняющуюся максимально быстро.

### Упражнение 6.46 ♦♦♦♦

Это упражнение является интересной вариацией упражнения 6.45. Рассмотрим задачу преобразования ориентированного графа  $g$  в его неориентированный аналог  $g'$ . Граф  $g'$  имеет ребро, связывающее вершины  $u$  и  $v$ , только если в графе  $g$  существует ребро, направленное от  $u$  к  $v$  или от  $v$  к  $u$ . Граф  $g$  представлен матрицей смежности  $G$ , как описывается далее. Если  $N$  – число вершин в  $g$ , то  $G$  – это матрица  $N \times N$ , содержащая в элементах значения 0 или 1. Предположим, что вершины в  $g$  обозначены как  $v_0, v_1, v_2, \dots, v_{N-1}$ . Тогда  $G[i][j] = 1$ , если существует ребро от  $v_i$  к  $v_j$ , и  $G[i][j] = 0$  в противном случае. Обратите внимание, что диагональные элементы матрицы  $G$  всегда равны 1, а матрица смежности неориентированного графа симметрична. Реализовать поставленную задачу можно в виде простого цикла:

```
1 void col_convert(int *G, int dim) {
2   int i, j;
3
4   for (i = 0; i < dim; i++)
5     for (j = 0; j < dim; j++)
6       G[j*dim + i] = G[j*dim + i] || G[i*dim + j];
7 }
```

Ваша задача – разработать функцию преобразования, выполняющуюся максимально быстро. Как и в предыдущем упражнении, вам понадобится использовать идеи оптимизации приложений, интенсивно применяющих память, изученные в главах 5 и 6.

## Решения упражнений

### Решение упражнения 6.1

Идея заключается в минимизации числа адресных битов путем минимизации характеристического отношения  $\max(r, c) / \min(r, c)$ . Другими словами, чем ближе массив к квадратной форме, тем меньше адресных битов.

Организация	$r$	$c$	$b_r$	$b_c$	$\max(b_r, b_c)$
16×1	4	4	2	2	2
16×4	4	4	2	2	2
128×8	16	8	4	3	4
512×4	32	16	5	4	5
1024×4	32	32	5	5	5

### Решение упражнения 6.2

Целью данного упражнения является закрепление понимания взаимосвязи между цилиндрами и дорожками. При наличии полного понимания упражнение решается легко и просто:

$$\begin{aligned} \text{Емкость} &= \frac{512 \text{ байт}}{\text{сектор}} = \frac{400}{\text{секторов}} = \frac{10\,000}{\text{дорожек}} = \frac{2}{\text{поверхности}} = \frac{2}{\text{пластины}} = \frac{2}{\text{диск}} \\ &= 8\,192\,000\,000 \text{ байт} \\ &= 8,192 \text{ Гбайт.} \end{aligned}$$

### Решение упражнения 6.3

Это упражнение решается простым применением формулы определения времени доступа к диску. Средняя задержка из-за вращения диска (в мс) составляет:

$$\begin{aligned}
 T_{avg\ rotation} &= 1/2 \times T_{max\ rotation} \\
 &= 1/2 \times (60\text{ с}/15\ 000\text{ об/мин}) \times 1000\text{ мс/с} \\
 &\approx 2\text{ мс}.
 \end{aligned}$$

Соответственно, среднее время доступа составит:

$$\begin{aligned}
 T_{access} &= T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \\
 &= 8\text{ мс} + 2\text{ мс} + 0,008\text{ мс} \\
 &\approx 10\text{ мс}.
 \end{aligned}$$

### Решение упражнения 6.4

Это упражнение поможет вам проверить свое понимание факторов, влияющих на производительность диска. Сначала нужно определить несколько основных свойств файла и диска. Файл состоит из 2000 логических блоков по 512 байт. Для диска с  $T_{avg\ seek} = 5\text{ мс}$ ,  $T_{max\ rotation} = 6\text{ мс}$  и  $T_{avg\ rotation} = 3\text{ мс}$ .

1. *Лучший случай:* в оптимальном случае блоки отображаются в смежные секторы одного и того же цилиндра, которые можно читать один за другим, не перемещая головку. Как только головка окажется над первым сектором, потребуется два полных оборота (1000 секторов на оборот) диска, чтобы прочитать все 2048 блоков. То есть общее время чтения файла составит

$$T_{avg\ seek} + T_{avg\ rotation} + 2 \times T_{max\ rotation} = 5 + 3 + 12 = 20\text{ мс}.$$

2. *Произвольный случай:* в этом случае блоки отображаются в секторы случайным образом и для чтения каждого из 2048 блоков потребуется  $T_{avg\ seek} + T_{avg\ rotation}$ . То есть общее время чтения файла составит

$$(T_{avg\ seek} + T_{avg\ rotation}) \times 2048 = 16\ 384\text{ мс (16 секунд!)}.$$

Теперь должно быть понятно, почему дефрагментация диска – хорошая идея!

### Решение упражнения 6.5

Цель этого простого упражнения – дать некоторое представление о возможностях твердотельных дисков. Напомним, что 1 Пбайт =  $10^9$  Мбайт. Прямой перевод единиц дает следующие прогнозируемые времена для каждого случая:

1. Худший случай для последовательной записи (470 Мбайт/с):  
 $(10^9 \times 128) \times (1/470) \times (1/(86\ 400 \times 365)) \approx 8\text{ лет}.$
2. Худший случай для произвольной записи (303 Мбайт/с):  
 $(10^9 \times 128) \times (1/303) \times (1/(86\ 400 \times 365)) \approx 13\text{ лет}.$
3. Средний случай (20 Гбайт/день):  
 $(10^9 \times 128) \times (1/20\ 000) \times (1/365) \approx 17\ 535\text{ лет}.$

Таким образом, даже если твердотельный диск работает непрерывно, он должен прослужить не менее 8 лет, что больше ожидаемого срока службы большинства компьютеров.

### Решение упражнения 6.6

За 10-летний период с 2005 по 2015 год цена на вращающиеся диски упала в 166 раз, то есть цена снижается примерно в 2 раза каждые 18 месяцев или около того. Если предположить, что эта тенденция сохранится, то стоимость петабайтного диска, стоившего около 30 000 долларов в 2015 году, упадет ниже 500 долларов после примерно семи таких 2-кратных снижений стоимости. Поскольку они происходят каждые 18 месяцев, то можно ожидать, что петабайтное хранилище будет доступно за 500 долларов примерно к 2025 году.

## Решение упражнения 6.7

Чтобы получить шаблон обращений с шагом 1, необходимо переставить местами циклы так, чтобы правые индексы менялись чаще всего.

```

1 int sumarray3d(int a[N][N][N])
2 {
3     int i, j, k, sum = 0;
4
5     for (k = 0; k < N; k++) {
6         for (i = 0; i < N; i++) {
7             for (j = 0; j < N; j++) {
8                 sum += a[k][i][j];
9             }
10        }
11    }
12    return sum;
13 }
```

Важно понимать, почему именно такая перестановка циклов дает шаблон обращений с шагом 1.

## Решение упражнения 6.8

Ключом к решению этого упражнения являются четкое представление, как массив располагается в памяти, и анализ закономерностей обращений к нему. Функция `clear1` выполняет обход массива, следуя закономерности обращений с шагом 1, и, следовательно, имеет лучшую пространственную локальность. Функция `clear2` просматривает каждую из  $N$  структур по порядку, что само по себе неплохо, но в пределах каждой структуры она «прыгает», нарушая закономерность обращений с шагом 1, читая элементы структур со смещениями в порядке: 0, 12, 4, 16, 8, 20. Поэтому `clear2` имеет пространственную локальность хуже, чем `clear1`. Функция `clear3` «прыгает» не только в пределах каждой структуры, но также между структурами. Поэтому `clear3` имеет самую худшую пространственную локальность.

## Решение упражнения 6.9

Решение заключается в простой подстановке различных параметров кеша в табл. 6.8. Задача может показаться не самой интеллектуальной, но для истинного понимания особенностей работы кеша сначала необходимо понять, как его организация влияет на формирование разделов в последовательности адресных битов.

Кеш	$m$	$C$	$B$	$E$	$S$	$t$	$s$	$b$
1	32	1024	4	1	256	22	8	2
2	32	1024	8	4	32	24	5	3
3	32	1024	32	32	1	27	0	5

## Решение упражнения 6.10

Добавление дополнительных элементов в массив устраняет конфликтные промахи. Благодаря этому доля попаданий в кеш составит три четверти от всех обращений к памяти.

## Решение упражнения 6.11

Иногда анализ причин, почему та или иная идея себя не оправдывает, помогает понять, почему альтернативная идея может оказаться удачной. В данном случае неудачной идеей является индексирование кеша битами из старших разрядов, а не из середины.

1. При индексировании битами из старших разрядов массив будет разделен на смежные фрагменты, состоящие из  $2^t$  блоков, где  $t$  – число битов тега. В результате первые  $2^t$  смежных блоков массива отобразятся в набор 0, следующие  $2^t$  блоков – в набор 1 и т. д.
2. Емкость кеша с прямым отображением, где  $(S, E, B, m) = (512, 1, 32, 32)$ , составит 512 блоков по 32 байта с  $t = 18$  бит тега в каждой строке. Таким образом, первые  $2^{18}$  блоков в массиве отобразятся в набор 0, следующие  $2^{18}$  блоков – в набор 1. Поскольку массив состоит всего из  $(4096 \times 4) / 32 = 512$  блоков, все блоки массива отобразятся в множество 0. То есть в любой момент времени кеш будет хранить не более одного блока массива, даже притом что массив достаточно мал, чтобы уместиться в кеше целиком. Очевидно, что индексирование наборов битами из старших разрядов снижает эффективность использования кеша.

### Решение упражнения 6.12

Два младших бита – это смещение в блоке (CO), следующие три бита – индекс набора (CI); оставшиеся биты играют роль тега (СТ):

СТ	СТ	СТ	СТ	СТ	СТ	СТ	СТ	CI	CI	CI	CO	CO
12	11	10	9	8	7	6	5	4	3	2	1	0

### Решение упражнения 6.13

Адрес: 0x0E34

1. Формат адреса (одна ячейка – один бит):

СТ	СТ	СТ	СТ	СТ	СТ	СТ	СТ	CI	CI	CI	CO	CO
0	1	1	1	0	0	0	1	1	0	1	0	0
12	11	10	9	8	7	6	5	4	3	2	1	0

2. Обращение к ячейке памяти:

Параметр	Значение
Смещение в блоке (CO)	0x0
Индекс набора (CI)	0x5
Тег (СТ)	0x71
Попадание в кеш? (Д/Н)	Д
Байт из кеша	0xB

### Решение упражнения 6.14

Адрес: 0x0DD5

1. Формат адреса (одна ячейка – один бит):

СТ	СТ	СТ	СТ	СТ	СТ	СТ	СТ	CI	CI	CI	CO	CO
0	1	1	0	1	1	1	0	1	0	1	0	1
12	11	10	9	8	7	6	5	4	3	2	1	0

2. Обращение к ячейке памяти:

Параметр	Значение
Смещение в блоке (CO)	0x1
Индекс набора (CI)	0x5
Тег (СТ)	0x6E
Попадание в кеш? (Д/Н)	Н
Байт из кеша	–

Решение упражнения 6.15

Адрес: 0x1FE4

1. Формат адреса (одна ячейка – один бит):

СТ	СТ	СТ	СТ	СТ	СТ	СТ	СТ	CI	CI	CI	CO	CO
1	1	1	1	1	1	1	1	0	0	1	0	0
12	11	10	9	8	7	6	5	4	3	2	1	0

2. Обращение к ячейке памяти:

Параметр	Значение
Смещение в блоке (CO)	0x1
Индекс набора (CI)	0x5
Тег (СТ)	0x6E
Попадание в кеш? (Д/Н)	Н
Байт из кеша	–

Решение упражнения 6.16

Это упражнение предлагает решить задачу, противоположную задачам в упражнениях 6.12–6.15, и требует двигаться в обратном направлении – от содержимого кеша к адресам, которые отобразятся в конкретный набор. В данном случае набор 3 содержит одну действительную строку с тегом 0x32. Поскольку в наборе имеется только одна действительная строка, попадание в кеш будет иметь место для четырех адресов. Вот эти адреса в двоичном виде: 0 0110 0100 11xx. То есть в шестнадцатеричном виде они будут выглядеть так: 0x064C, 0x064D, 0x064E и 0x064F.

Решение упражнения 6.17

1. Ключом к решению этого упражнения является схема на рис. 6.33. Обратите внимание, что каждая строка кеша содержит только одну строку массива, кеш достаточно велик, чтобы вместить один массив, и для всех  $i$  соответствующие строки в массивах src и dst отображаются в одну и ту же строку кеша. Так как кеш слишком мал, чтобы вместить сразу два массива, обращения к одному массиву вытесняют строки, хранящие данные из другого массива. Так, операция записи в `dst[0][0]` вытеснит строку, загруженную при чтении `src[0][0]`. Поэтому при чтении следующего элемента `src[0][1]` произойдет промах.

массив dst			массив src		
столбец 0		столбец 1	столбец 0		столбец 1
строка 0	m	m	строка 0	m	m
строка 1	m	m	строка 1	m	h

2. При увеличении объема кеша до 32 байт он оказывается достаточно большим для хранения обоих массивов. Поэтому промахи будут случаться только при первоначальных обращениях к массиву, когда кеш еще «холодный».

массив dst			массив src		
столбец 0		столбец 1	столбец 0		столбец 1
строка 0	m	h	строка 0	m	h
строка 1	m	h	строка 1	m	h

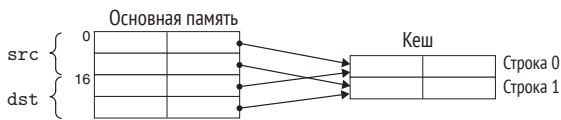


Рис. 6.33. Схема для упражнения 6.17



### Решение упражнения 6.18

Каждая 16-байтная строка кеша содержит две соседние структуры `algae_position`. Каждый цикл перебирает эти структуры в порядке их размещения в памяти, читая каждый раз один целочисленный элемент. Поэтому в каждом цикле будет наблюдаться чередование промахов и попаданий: промах, попадание, промах, попадание и т. д. Обратите внимание, что для данного упражнения можно предсказать частоту промахов фактически и без перечисления общего числа чтений и промахов.

1. Общее количество чтений 512.
2. Общее количество чтений с промахами кеша 256.
3. Частота промахов  $256 / 512 = 50 \%$ .

### Решение упражнения 6.19

В данном случае кеш может хранить только половину массива. Поэтому обход второй половины массива по столбцам вытесняет строки, загруженные во время обхода первой половины. Например, чтение первого элемента `grid[8][0]` вытеснит строку, загруженную при чтении элементов, начиная с `grid[0][0]`. Эта строка содержит также `grid[0][1]`. Поэтому при обходе следующего столбца обращение к первому элементу `grid[0][1]` приведет к промаху.

1. Общее количество чтений 512.
2. Общее количество чтений, приводящих к промахам кеша, 256.
3. Частота промахов  $256 / 512 = 50 \%$ .
4. Если бы кеш был в два раза больше, то в него уместился бы весь массив `grid`, и тогда промахи случались бы только при первоначальных обращениях, когда кеш еще «холодный», а частота промахов составила бы 25 %.

### Решение упражнения 6.20

Цикл следует шаблону обращений с шагом 1, поэтому промахи случаются только при первоначальных обращениях, когда кеш еще «холодный».

1. Общее количество чтений 512.
2. Общее количество чтений, приводящих к промахам кеша, 128.
3. Частота промахов  $128 / 512 = 25 \%$ .
4. Никакое увеличение размера кеша не изменит частоту промахов, потому что «холодные» промахи неизбежны.

### Решение упражнения 6.21

Устойчивая пропускная способность при больших шагах в L1 составляет около 12 000 Мбайт/с, тактовая частота равна 2100 МГц, а отдельные операции чтения извлекают 8-байтные значения `long`. Таким образом, из этого графика вытекает, что для доступа к слову в L1 на данной машине требуется  $2100 / 12\,000 \times 8 = 1,4 \approx 1,5$  цикла, что примерно в 2,5 раза быстрее, чем предполагает номинальная задержка доступа к L1 в 4 цикла. Это ускорение объясняется параллельным выполнением нескольких операций загрузки, которое обеспечивает развертывание цикла  $4 \times 4$ .

## Часть II

# Выполнение программ в системе

**М**ы продолжим наше исследование компьютерных систем, перейдя к более подробному изучению системного программного обеспечения, с помощью которого создаются и выполняются прикладные программы. Редактор связей объединяет различные части наших программ в единый файл, который можно загрузить в память и запустить на выполнение. Современные операционные системы, взаимодействуя с аппаратным обеспечением, создают иллюзию, что каждая программа получает исключительный доступ к процессору и оперативной памяти, тогда как в действительности в любой момент времени в системе выполняется несколько различных программ.

В первой части этой книги вы получили полное представление о взаимодействиях программ с аппаратным обеспечением. Вторая часть книги расширит ваши познания и даст вам правильное понимание взаимодействий ваших программ с операционной системой. Вы узнаете, как пользоваться услугами, предоставляемыми операционной системой, такими как командные оболочки Unix и средства динамического распределения памяти.

# Глава 7

## Связывание

- 7.1. Драйверы компиляторов.
- 7.2. Статическое связывание.
- 7.3. Объектные файлы.
- 7.4. Перемещаемые объектные файлы.
- 7.5. Идентификаторы и таблицы имен.
- 7.6. Разрешение ссылок.
- 7.7. Перемещение.
- 7.8. Выполняемые объектные файлы.
- 7.9. Загрузка выполняемых объектных файлов.
- 7.10. Динамическое связывание с разделяемыми библиотеками.
- 7.11. Загрузка и связывание с разделяемыми библиотеками из приложений.
- 7.12. Перемещаемый программный код.
- 7.13. Подмена библиотечных функций.
- 7.14. Инструменты для управления объектными файлами.
- 7.15. Итоги.

Библиографические заметки.

Домашние задания.

Решения упражнений.

**С**вязывание (*компоновка*) – это процесс сборки и объединения различных частей программного кода и данных в единый файл, который можно загрузить в память и запустить на выполнение. Связывание может выполняться *во время компиляции* – после трансляции исходного кода в машинный; *во время загрузки* – сразу после загрузки программы в память *загрузчиком*; и даже *во время выполнения*. В ранних компьютерных системах связывание выполнялось вручную. В современных системах это делается автоматически – специальными программами, которые называются *редакторами связей*, или *компоновщиками*.

Компоновщики играют ключевую роль в разработке программного обеспечения, потому что обеспечивают возможность *раздельной компиляции*. Они избавляют от необходимости создавать большие приложения в одном монолитном файле с исходным кодом и позволяют расчленить его на множество меньших по размеру модулей, которые могут изменяться и компилироваться раздельно. После изменения любых модулей их можно скомпилировать и снова скомпоновать приложение без повторной компиляции остальных файлов.

Ошибки во время компоновки случаются редко, и у студентов не возникает серьезных проблем при разработке маленьких программ во вводных курсах по программированию, поэтому давайте определим, что дает знание и понимание особенностей связывания программ:

- *понимание особенностей работы компоновщиков помогает при создании больших программ.* Программисты, работающие над большими программами, часто сталкиваются с ошибками времени компоновки, вызванными отсутствием модулей, библиотек или несовместимостью версий этих модулей и библиотек. Если не знать, как редактор связей разрешает ссылки на библиотеки, то такие ошибки будут вас ставить в тупик;
- *понимание особенностей работы компоновщиков помогает избежать опасных ошибок программирования.* Решения, которые принимают редакторы связей в Linux при разрешении ссылок, могут незаметно для вас воздействовать на корректность программ. Программы, неправильно определяющие различные глобальные переменные, по умолчанию не вызывают предупреждений компоновщика, но получившийся в результате выполняемый код может производить необъяснимые действия, и такие ошибки чрезвычайно трудно отлаживать. Мы покажем вам, как это происходит и как этого можно избежать;
- *понимание особенностей работы компоновщиков помогает усвоить правила областей видимости в языке,* например понять разницу между глобальными и локальными переменными, смысл объявления переменных или функций статическими;
- *понимание особенностей работы компоновщиков помогает освоить другие важные понятия.* Выполняемые объектные файлы, сформированные редакторами связей, играют важнейшую роль в работе таких системных функций, как загрузка и запуск программ, управление виртуальной памятью, страничная организация памяти и отображение памяти;
- *понимание особенностей работы компоновщиков дает возможность пользоваться разделяемыми библиотеками.* В течение многих лет компоновка считалась простой и неинтересной операцией. Однако вместе с распространением разделяемых библиотек и динамического связывания в современных операционных системах компоновка превратилась в сложный творческий процесс, предоставляющий опытным программистам богатые возможности. Например, многие программные продукты используют разделяемые библиотеки для расширения возможностей двоичных прикладных программ во время выполнения. Кроме того, многие веб-серверы широко используют разделяемые библиотеки для обслуживания динамически меняющегося содержимого.

В этой главе мы подробно рассмотрим все аспекты компоновки, от традиционно статического до динамического связывания с разделяемыми библиотеками во время загрузки и выполнения. Мы опишем работу стандартных механизмов на реальных примерах и покажем, в каких ситуациях проблемы компоновки могут влиять на корректную работу программ. Чтобы объяснения оставались конкретными и понятными, мы будем формулировать наши рассуждения в контексте системы Linux для архитектуры x86-64 с использованием стандартного формата объектных файлов ELF-64 (далее мы будем называть его просто ELF). Однако важно понимать, что основные идеи компоновки универсальны и не зависят от операционной системы, архитектуры набора команд (ISA) или формата представления объектных файлов. Детали могут меняться, но идеи остаются неизменными.

## 7.1. Драйверы компиляторов

Рассмотрим программу в листинге 7.1. Она будет служить нам простым практическим примером на протяжении всей главы, на котором мы будем демонстрировать особенности работы компоновщиков.

**Листинг 7.1. Пример программы 1.** Программа состоит из двух файлов с исходным кодом: `main.c` и `sum.c`. Функция `main` инициализирует массив целых чисел, а затем вызывает функцию `sum`, чтобы подсчитать сумму элементов массива

(a) `main.c`

```
code/link/main.c
1 int sum(int *a, int n);
2
3 int array[2] = {1, 2};
4
5 int main()
6 {
7     int val = sum(array, 2);
8     return val;
9 }
```

(b) `sum.c`

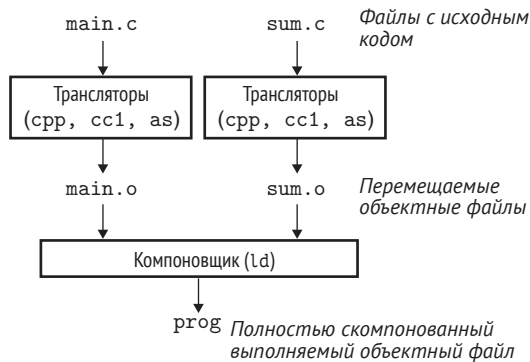
```
code/link/sum.c
1 int sum(int *a, int n)
2 {
3     int i, s = 0;
4
5     for (i = 0; i < n; i++) {
6         s += a[i];
7     }
8     return s;
9 }
```

Большинство систем компиляции предоставляют *драйвер компилятора*, который вызывает препроцессор, компилятор, ассемблер и компоновщик по мере необходимости. Например, собрать программу из примера выше с помощью системы компиляции GNU можно, вызвав драйвер GCC следующей командой:

```
linux> gcc -Og -o prog main.c sum.c
```

На рис. 7.1 показаны действия, выполняемые драйвером по мере преобразования исходного кода программы в выполняемый объектный файл. (Чтобы увидеть выполнение этих шагов во всех деталях, запустите `gcc` с параметром `-v`.) Драйвер сначала запускает препроцессор языка C (`cpp`)<sup>1</sup>, который преобразует файл `main.c` с исходным кодом в файл с промежуточным кодом `main.i`:

```
cpp [другие аргументы] main.c /tmp/main.i
```



**Рис. 7.1. Статическое связывание.** Компоновщик объединяет перемещаемые объектные файлы и формирует из них выполняемый объектный файл `prog`

<sup>1</sup> В некоторых версиях GCC препроцессор встроен непосредственно в драйвер.

Далее драйвер вызывает компилятор языка C (cc1), который транслирует промежуточный исходный код из файла `main.i` в код на языке ассемблера и сохраняет его в файле `main.s`:

```
cc1 /tmp/main.i -Og [другие аргументы] -o /tmp/main.s
```

Затем драйвер вызывает ассемблер (as), который транслирует код на языке ассемблера из файла `main.s` в двоичный *перемещаемый объектный код* и сохраняет его в файле `main.o`:

```
as [другие аргументы] -o /tmp/main.o /tmp/main.s
```

Тот же процесс повторяется для файла `sum.c`, в результате чего получается файл `sum.o`. Наконец, драйвер вызывает компоновщика `ld`, который объединяет `main.o` и `sum.o` с другими обязательными системными объектными файлами и создает двоичный выполняемый объектный файл `prog`:

```
ld -o prog [системные объектные файлы и аргументы] /tmp/main.o /tmp/sum.o
```

Чтобы запустить получившуюся программу, нужно ввести ее имя в командной оболочке Linux:

```
linux> ./prog
```

Командная оболочка обратится к *загрузчику* в операционной системе, который скопирует код и данные из выполняемого файла `prog` в память и затем передаст управление в начальную точку программы.

## 7.2. Статическое связывание

Статический компоновщик, такой как программа LD, принимает параметры и группу файлов с перемещаемым объектным кодом и генерирует полностью связанный выполняемый файл, который можно загрузить и запустить. Файлы с перемещаемым объектным кодом содержат различные секции программного кода и данных, каждая из которых представляет собой непрерывную последовательность байтов. Инструкции находятся в одной секции, инициализированные глобальные переменные – в другой, а неинициализированные переменные – в третьей.

Сборка выполняемого файла компоновщиком выполняется в два этапа.

### Этап 1

*Разрешение ссылок.* В объектных файлах содержатся определения *имен* и ссылки на них, где каждое имя соответствует функции, глобальной переменной или *статической переменной* (т. е. любой переменной, объявленной с атрибутом `static`). Цель разрешения ссылок – связать каждую *ссылку* с соответствующим *определением*.

### Этап 2

*Перемещение.* Компиляторы и ассемблеры генерируют секции программного кода и данных, начиная с нулевого адреса. Компоновщик *перемещает* эти секции, связывая каждое определение имени с адресом в памяти и изменяя все ссылки на эти имена так, чтобы они указывали на соответствующий адрес в памяти. Компоновщик выполняет эти перемещения, слепо следуя подробным инструкциям, сгенерированным ассемблером, которые называются *записями перемещения*.

В последующих разделах эти этапы описываются более подробно. Когда вы перейдете к ним, то имейте в виду некоторые основные факты, связанные с компоновщиком.



ми: объектные файлы – это просто совокупности блоков байтов. Одни блоки содержат программный код, другие – данные, а третьи – структуры данных, которые управляют компоновщиком и загрузчиком. Компоновщик объединяет (связывает) блоки, определяет для этих блоков адреса и модифицирует различные ссылки внутри блоков данных и программного кода. Компоновщики имеют минимальное представление о целевой машине, потому что вся основная работа уже проделана компиляторами и ассемблерами, сгенерировавшими объектные файлы.

### 7.3. Объектные файлы

Объектные файлы могут иметь одну из трех форм:

- *файлы с перемещаемым объектным кодом.* Содержат двоичный код и данные в форме, позволяющей объединить их с другими файлами с перемещаемым объектным кодом, чтобы получить выполняемый объектный код;
- *файлы с выполняемым объектным кодом.* Содержат двоичный код и данные, которые можно загрузить непосредственно в память и запустить на выполнение;
- *файлы с разделяемым объектным кодом.* Особый вид перемещаемого объектного кода, который можно загрузить в память и динамически связать с другими модулями во время загрузки или во время выполнения.

Компиляторы и ассемблеры генерируют перемещаемый объектный код (включая разделяемый объектный код). Компоновщики генерируют выполняемый объектный код. Технически *объектный код* – это последовательность байтов, а *объектный файл* – это объектный код, хранимый в файле на диске. Однако мы будем использовать эти термины как взаимозаменяемые.

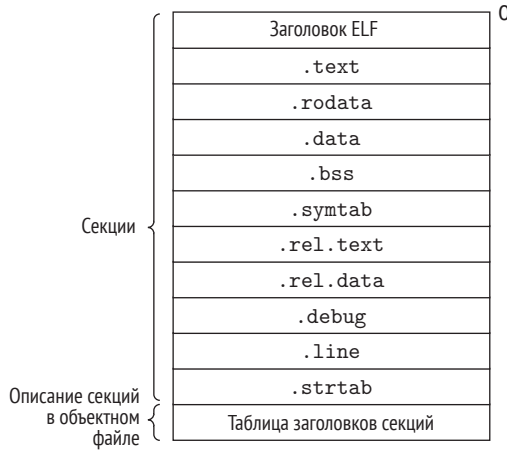
В разных системах используются разные *форматы объектных файлов*. В первых версиях Unix, созданных в Bell Labs, использовался формат a.out. (И по сей день выполняемые файлы называют a.out.) В Windows используется формат переносимых выполняемых файлов (Portable Executable, PE). В Mac OS-X используется формат Mach-O. Современные системы Linux и Unix для архитектуры x86-64 используют *формат выполняемых и компонуемых модулей* (Executable and Linkable Format, ELF). Далее мы сосредоточимся исключительно на формате ELF, однако базовые принципы не зависят от конкретного формата.

### 7.4. Перемещаемые объектные файлы

На рис. 7.2 показан формат типичного перемещаемого объектного файла ELF. *Заголовок ELF* начинается с 16-байтной последовательности, описывающей размер слова и порядок следования байтов в системе, где был сгенерирован этот файл. Остальная часть заголовка ELF содержит информацию, которая позволяет компоновщику анализировать и интерпретировать объектный файл, в том числе размер заголовка ELF, тип объектного файла (перемещаемый, выполняемый или разделяемый), вид аппаратной архитектуры (например, x86-64), смещение таблицы заголовков секций в файле, размер и количество записей в таблице заголовков секций. Адреса и размеры различных секций описаны в *таблице заголовков секций*, которая содержит элементы фиксированного размера с описанием каждой секции в объектном файле.

Между заголовком ELF и таблицей заголовков секций находятся сами секции. Типичный перемещаемый объектный файл ELF содержит следующие секции:

- *text* – машинный код скомпилированной программы;
- *rodata* – данные, доступные только для чтения, такие как строки формата в вызовах printf и таблицах переходов инструкций switch;



**Рис. 7.2.** Типичный перемещаемый объектный файл ELF

- `.data` – *инициализированные* глобальные переменные. Локальные переменные размещаются в стеке и поэтому не перечисляются ни в секции `.data`, ни в секции `.bss`;
- `.bss` – *неинициализированные* глобальные переменные. Фактически эта секция не занимает места в объектном файле, она просто служит «заполнителем места». Форматы объектных файлов различают инициализированные и неинициализированные переменные для экономии памяти: неинициализированные переменные не занимают места в объектном файле на диске. Во время выполнения для них выделяется память, которая предварительно заполняется нулевыми байтами;
- `.symtab` – *таблица имен* (symbol table) с информацией о функциях и глобальных переменных в программе, на которые имеются ссылки. Некоторые программисты ошибочно считают, что для создания таблицы имен программа должна компилироваться с параметром `-g`. Но на самом деле каждый перемещаемый объектный файл имеет таблицу имен в секции `.symtab` (если, конечно, программист явно не удалил ее командой STRIP). Однако, в отличие от таблицы идентификаторов в компиляторе, таблица имен `.symtab` не содержит информации о локальных переменных;
- `.rel.text` – список адресов в секции `.text`, которые должны быть изменены в процессе компоновки этого объектного файла с другими. В общем случае любая инструкция, вызывающая внешнюю функцию или ссылающаяся на глобальную переменную, должна быть модифицирована. С другой стороны, инструкции, вызывающие локальные функции, не должны модифицироваться. Имейте в виду, что информация о перемещении не требуется в выполняемых объектных файлах и обычно опускается, если пользователь явно не сообщит компоновщику включить ее;
- `.rel.data` – информация о перемещении для каждой глобальной переменной, на которую имеется ссылка или которая определяется в данном модуле. В общем случае каждая инициализированная глобальная переменная, начальное значение которой является адресом глобальной переменной или внешней функции, должна быть модифицирована;
- `.debug` – таблица имен для отладки с информацией о локальных переменных и определениях типов в программе, глобальных переменных, на которые имеют-



ся ссылки или которые определяются в программе, а также о файле с исходным кодом. Эта секция добавляется в объектный файл, только если драйвер компилятора был вызван с параметром `-g`;

`.line` – таблица соответствий между номерами строк в исходном коде и машинными инструкциями в секции `.text`. Эта секция добавляется в объектный файл, только если драйвер компилятора был вызван с параметром `-g`;

`.strtab` – таблица строк для таблиц имен в секциях `.symtab` и `.debug` и для секции имен в заголовках секций. Таблица строк представляет собой последовательность символьных строк с нулевым символом в конце.

#### Почему секция с неинициализированными данными называется `.bss`?

Имя `.bss` для секции с неинициализированными данными широко распространено. Первоначально (приблизительно в 1957 году) это была директива ассемблера IBM 704 «block started by symbol» (начало блока памяти), и это обозначение так и сохранилось. Для простоты запоминания различий между секциями `.data` и `.bss` представляйте «bss» как сокращение от «Better Save Space!» (экономь память).

## 7.5. Идентификаторы и таблицы имен

Каждый перемещаемый объектный модуль *m* имеет таблицу имен с информацией обо всех именах, которые определяются или на которые имеются ссылки в *m*. С точки зрения компоновщика имеются три вида имен:

- *глобальные имена* (global symbol), которые определены в модуле *m* и на которые могут ссылаться другие модули. Глобальные имена соответствуют *нестатическим* функциям и глобальным переменным;
- глобальные имена, на которые ссылается модуль *m*, но определяемые в других модулях. Такие имена называются *внешними* (external) и соответствуют функциям и переменным, объявленным в других модулях;
- *локальные имена* (local symbol), которые определены и используются исключительно внутри модуля *m*. Эти имена соответствуют статическим функциям и глобальным переменным, объявленным с атрибутом `static`. Они доступны из любой точки внутри модуля *m*, но недоступны из других.

Важно понимать, что локальные имена – это не локальные переменные в программе. Таблица имен в `.symtab` не содержит имен локальных нестатических переменных программы. Последние размещаются в стеке во время выполнения и не представляют интереса для компоновщика.

Интересно отметить, что локальные переменные в функциях, объявленные с атрибутом `static`, размещаются не в стеке. Для каждой такой переменной компилятор выделяет место в `.data` или `.bss` и создает в таблице имен уникальное локальное имя для компоновщика. Например, представьте, что в одном и том же модуле есть пара функций, определяющих статическую локальную переменную *x*:

```

1 int f()
2 {
3     static int x = 0;
4     return x;
5 }
6
```

```

7 int g()
8 {
9     static int x = 1;
10    return x;
11 }

```

В этом случае компилятор создаст пару разных локальных имен, которые подставит в ассемблерный код. Например, для локальной статической переменной в функции `f` он может создать имя `x.1`, и для локальной статической переменной в функции `g` – имя `x.2`.

В таблицах имен, созданных ассемблерами, используются имена, переданные компилятором в ассемблерный файл `.s`. Таблица имен ELF находится в секции `.symtab`. Она содержит массив записей. В листинге 7.2 показан формат каждой такой записи.

**Листинг 7.2.** Структура записей в таблице имен ELF. Поля `type` и `binding` имеют размеры по 4 бита каждый

```

code/link/elfstructs.c
1 typedef struct {
2     int name;           /* Смещение в таблице строк */
3     char type:4;        /* Функция или данные (4 бита) */
4     binding:4;         /* Локальное или глобальное имя (4 бита) */
5     char reserved;     /* Не используется */
6     short section;     /* Индекс заголовка секции */
7     long value;        /* Смещение в секции или абсолютный адрес */
8     long size;         /* Размер объекта в байтах */
9 } Elf64_Symbol;
code/link/elfstructs.c

```

Поле `name` хранит смещение (в байтах) в таблице строк, где находится строка с именем переменной, `value` хранит адрес переменной. Для перемещаемых модулей поле `value` определяет смещение от начала секции, где определен данный объект. Для выполняемых объектных файлов `value` – это абсолютный адрес времени выполнения. Поле `size` хранит размер (в байтах) объекта, `type` определяет тип объекта: `data` или `function`. Таблица имен может также содержать элементы, описывающие отдельные секции и путь к файлу с исходным кодом программы. Поле `binding` определяет вид имени – глобальное или локальное.

### Соккрытие имен переменных и функций

Чтобы скрыть объявления переменных и функций в модулях, программисты на C используют атрибут `static`, подобно тому, как в Java и C++ использовали бы объявления `public` и `private`. В языке C роль модулей играют файлы с исходным кодом. Каждая глобальная переменная или функция, объявленная с атрибутом `static`, становится приватной для этого модуля. Точно так же каждая глобальная переменная или функция, объявленная без атрибута `static`, становится общедоступной и может использоваться другими модулями. Хорошей практикой считается защищать переменные и функции с помощью атрибута `static` везде, где это возможно.

Каждое имя связано с некоторой секцией в объектном файле, индекс которой в таблице заголовков секций указывается в поле `section`. Имеются три специальные секции, не имеющие соответствующих записей в таблице заголовков секций: `ABS` – с перемещаемыми именами; `UNDEF` – с неопределенными именами (на которые имеются ссылки в данном объектном модуле, но определяемые в другом модуле); `COMMON` – с неи-

инициализированными и пока неразмещенными в памяти объектами данных. Для имен в секции COMMON поле value задает требования к выравниванию, а size определяет минимальный размер. Обратите внимание, что эти специальные секции присутствуют только в перемещаемых объектных файлах – их нет в выполняемых объектных файлах.

Секции COMMON и .bss имеют тонкое различие. Современные версии GCC помещают имена в секции COMMON и .bss в перемещаемых объектных файлах, используя следующее соглашение:

COMMON	Неинициализированные глобальные переменные
.bss	Неинициализированные статические переменные, и глобальные или статические переменные, инициализированные нулевыми значениями

Причина такого, казалось бы, непонятного разделения обусловлена способом разрешения ссылок в компоновщике, о котором рассказывается в разделе 7.6.

Программа GNU READELF – удобный инструмент для просмотра содержимого объектных файлов. Например, вот последние три записи в таблице имен из перемещаемого объектного файла main.o, полученного при компиляции программы из листинга 7.1. Первые восемь записей, которые здесь не показаны, – это локальные имена, которые компоновщик использует для внутренних целей.

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
8:	0000000000000000	24	FUNC	GLOBAL	DEFAULT	1	main
9:	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	3	array
10:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	sum

В этом примере можно видеть запись, соответствующую определению глобального имени main, 24-байтной функции, находящейся в секции .text с нулевым смещением (поле value). За ней следует запись, соответствующая определению глобального имени array, 4-байтного объекта, находящегося в секции .data с нулевым смещением. И последняя запись соответствует ссылке на внешнее имя sum. Программа READELF идентифицирует секции по целочисленным индексам. Ndx=1 соответствует секции .text, а Ndx=3 – секции .data.

**Листинг 7.3.** Пример программы для упражнения 7.1

<p>(a) m.c</p> <p><a href="#">code/link/m.c</a></p> <pre>1 void swap(); 2 3 int buf[2] = {1, 2}; 4 5 int main() 6 { 7     swap(); 8     return 0; 9 }</pre> <p><a href="#">code/link/m.c</a></p>	<p>(b) swap.c</p> <p><a href="#">code/link/swap.c</a></p> <pre>1 extern int buf[]; 2 3 int *bufp0 = &amp;buf[0]; 4 int *bufp1; 5 6 void swap() 7 { 8     int temp; 9 10    bufp1 = &amp;buf[1]; 11    temp = *bufp0; 12    *bufp0 = *bufp1; 13    *bufp1 = temp; 14 }</pre> <p><a href="#">code/link/swap.c</a></p>
--	---

Имя	Имеется запись в .symtab?	Тип имени	Модуль, где определено	Секция
buf	_____	_____	_____	_____
bufp0	_____	_____	_____	_____
bufp1	_____	_____	_____	_____
swap	_____	_____	_____	_____
temp	_____	_____	_____	_____

### Упражнение 7.1 (решение в конце главы)

В этом упражнении используются модули `m.o` и `swap.o` из листинга 7.3. Для каждого имени, которое определяется или на которое имеется ссылка в `swap.o`, укажите, будет ли для него создана запись в таблице имен в секции `.symtab` в модуле `swap.o`. Если да, то укажите модуль, в котором определяется имя (`swap.o` или `m.o`), тип имени (локальное, глобальное или внешнее) и секцию (`.text`, `.data`, `.bss` или `COMMON`) в этом модуле, где оно находится.

## 7.6. Разрешение ссылок

Компоновщик разрешает ссылки на имена, связывая каждую ссылку с конкретным определением имени в таблицах имен перемещаемых объектных файлов. Разрешение ссылок на локальные имена, которые определены в том же модуле, где находится сама ссылка, выполняется просто. Компилятор требует, чтобы каждое имя было определено в модуле только один раз. Также компилятор гарантирует уникальность имен локальных статических переменных, которые получит компоновщик.

Разрешение ссылок на глобальные имена – однако, более сложная задача. Встретив имя (переменной или функции), которое не определено в текущем модуле, компилятор предполагает, что это имя определено в некотором другом модуле, генерирует запись в таблицы имен и оставляет его для дальнейшей обработки компоновщиком. Если ни в одном из указанных входных модулей компоновщик не найдет определение имени, на которое имеется ссылка, он выведет сообщение об ошибке (часто загадочное) и завершится. Например, попробуем скомпилировать и скомпоновать в Linux следующий исходный код:

```

1 void foo(void);
2
3 int main() {
4     foo();
5     return 0;
6 }
```

Компилятор успешно выполнит свою часть работы, но компоновщик сообщит об ошибке, не сумев разрешить ссылку на `foo`:

```

linux> gcc -Wall -Og -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function 'main':
/tmp/ccSz5uti.o(.text+0x7): undefined reference to 'foo'
```

Разрешение ссылок на глобальные имена осложняется тем, что одно и то же имя может быть определено в нескольких объектных модулях. В таком случае компоновщик

должен сообщить об ошибке или выбрать одно из определений и отвергнуть остальные. В Linux использован подход, основанный на тесном взаимодействии компилятора, ассемблера и компоновщика, вследствие чего при безалаберном отношении к программированию могут возникать «необъяснимые» ошибки.

### Преобразование имен в языках C++ и Java специально для компоновщика

Оба языка, C++ и Java, поддерживают перегрузку методов, то есть позволяют определять методы с одинаковыми именами, но с разными списками параметров. Как компоновщик отличает такие перегруженные функции? Дело в том, что механизм поддержки перегруженных функций в компиляторах языков C++ и Java преобразует имена перегруженных методов с учетом комбинации его параметров и создает уникальные имена специально для компоновщика.

Интересно отметить, что языки C++ и Java используют совместимые схемы преобразования имен. Преобразованное имя класса состоит из количества символов в имени, за которым следует оригинальное имя. Например, имя класса Foo будет преобразовано в `3Foo`. Преобразованное имя метода включает оригинальное имя, за которым следует два символа подчеркивания (`__`), затем преобразованное имя класса и далее однобуквенные коды, представляющие типы параметров. Например, имя `Foo::bar(int, long)` будет преобразовано в `bar__3Fooil`. Похожие схемы используются для преобразования имен глобальных переменных и шаблонов.

## 7.6.1. Как компоновщик разрешает ссылки на повторяющиеся имена

На входе компоновщик получает набор перемещаемых объектных модулей. Каждый из этих модулей определяет набор имен – локальных (видимых только внутри модуля, где они определены) и глобальных (видимых другим модулям). Что случится, если в нескольких модулях определить одинаковые глобальные имена? Далее описывается решение, реализованное в системах компиляции Linux.

Во время компиляции компилятор передает ассемблеру каждое глобальное имя как *строго* или как *слабо* определенное, а ассемблер неявно кодирует эту информацию в таблице имен перемещаемого объектного файла. Имена функций и инициализированных глобальных переменных считаются строго определенными, а неинициализированные глобальные переменные – слабо определенными.

Основываясь на этих понятиях строго и слабо определенных имен, компоновщики в Linux используют следующие правила для обработки повторяющихся имен:

1. Наличие нескольких строго определенных имен недопустимо.
2. При наличии одного строго определенного и нескольких слабо определенных имен выбирается строго определенное имя.
3. При наличии нескольких слабо определенных имен выбирается любое из слабо определенных имен.

Предположим, что нам нужно скомпилировать и скомпоновать следующие два модуля:

```
1 /* foo1.c */
2 int main()
3 {
4     return 0;
5 }
```

```

1 /* bar1.c */
2 int main()
3 {
4     return 0;
5 }

```

В данном случае компоновщик сгенерирует сообщение об ошибке, потому что строго определенное имя `main` встречается несколько раз (правило 1):

```

linux> gcc foo1.c bar1.c
/tmp/ccq2Uxnd.o: In function 'main':
bar1.c:(.text+0x0): multiple definition of 'main'

```

Точно так же компоновщик сгенерирует сообщение об ошибке для следующих модулей, потому что строго определенное имя `x` определено дважды (правило 1):

```

1 /* foo2.c */
2 int x = 15213;
3
4 int main()
5 {
6     return 0;
7 }

```

```

1 /* bar2.c */
2 int x = 15213;
3
4 void f()
5 {
6 }

```

Однако если в одном модуле объявить переменную `x`, но не инициализировать ее, то компоновщик выберет имя, строго определенное в другом модуле (правило 2):

```

1 /* foo3.c */
2 #include <stdio.h>
3 void f(void);
4
5 int x = 15213;
6
7 int main()
8 {
9     f();
10    printf("x = %d\n", x);
11    return 0;
12 }

```

```

1 /* bar3.c */
2 int x;
3
4 void f()
5 {
6     x = 15212;
7 }

```

Во время выполнения функция `f` заменит значение 15213 в переменной `x` на 15212, что может оказаться неприятной неожиданностью для автора функции `main`. Обратите внимание, что компоновщик обычно не выводит никаких сообщений, обнаружив несколько разных определений `x`:

```
linux> gcc -o foobar3 foo3.c bar3.c
linux> ./foobar3
x = 15212
```

То же может произойти при наличии двух слабых определений `x` (правило 3):

```
1 /* foo4.c */
2 #include <stdio.h>
3 void f(void);
4
5 int x;
6
7 int main()
8 {
9     x = 15213;
10    f();
11    printf("x = %d\n", x);
12    return 0;
13 }

1 /* bar4.c */
2 int x;
3
4 void f()
5 {
6     x = 15212;
7 }
```

Применение правил 2 и 3 может привести к появлению некоторых коварных ошибок во время выполнения, малопонятных неопытному программисту, особенно если повторные определения объявляют имена с другими типами. Рассмотрим следующий пример, где в одном модуле переменная `x` определена с типом `int`, а в другом с типом `double`:

```
1 /* foo5.c */
2 #include <stdio.h>
3 void f(void);
4
5 int y = 15212;
6 int x = 15213;
7
8 int main()
9 {
10    f();
11    printf("x = 0x%x y = 0x%x \n",
12           x, y);
13    return 0;
14 }

1 /* bar5.c */
2 double x;
3
4 void f()
5 {
6     x = -0.0;
7 }
```

На машине `x86-64/Linux` для переменной типа `double` отводится 8 байт, а для переменной типа `int` – 4 байта. В нашей системе переменная `x` хранится в памяти по адресу `0x601020`, а переменная `y` – по адресу `0x601024`. Таким образом, операция присваивания

$x = -0.0$  в строке 6 в модуле `bar5.c` изменит содержимое ячеек памяти, занимаемых переменными  $x$  и  $y$  (строки 5 и 6 в модуле `foo5.c`)!

```
linux> gcc -Wall -Og -o foobar5 foo5.c bar5.c
/usr/bin/ld: Warning: alignment 4 of symbol 'x' in /tmp/ccLUFK5g.o
is smaller than 8 in /tmp/ccbTlcb9.o
linux> ./foobar5
x = 0x0 y = 0x80000000
```

Это тонкая и опасная ошибка, особенно потому, что вызывает только вывод предупреждения компоновщика и проявляется позже – во время выполнения программы, далеко от того места, где возникла. В больших системах с сотнями модулей ошибки такого рода чрезвычайно трудно находить и устранять, особенно потому, что многие программисты не знают, как работают компоновщики, и игнорируют предупреждения компилятора. Если появляется сомнение, вызовите компоновщик с ключами, такими как `-fno-common`, чтобы потребовать генерировать ошибку при встрече многократных определений глобальных имен. Или используйте параметр `-Werror`, чтобы преобразовать любые предупреждения в ошибки.

В разделе 7.5 мы видели, как компилятор связывает имена с секциями `COMMON` и `.bss`, используя, казалось бы, произвольное соглашение. В действительности это соглашение обусловлено тем, что в некоторых случаях компоновщик позволяет определять одинаковые глобальные имена сразу в нескольких модулях. Когда компилятор транслирует какой-то модуль и встречает слабо определенное имя, скажем  $x$ , он не знает, определяют ли другие модули имя  $x$ , и потому не может предсказать, какой из нескольких экземпляров  $x$  выберет компоновщик. Поэтому компилятор перекладывает принятие решения на компоновщика, связывая  $x$  с секцией `COMMON`. С другой стороны, если переменная  $x$  инициализируется нулевым значением, то это строго определенное имя (и, согласно правилу 2, должно быть уникальным), поэтому компилятор может уверенно связать его с секцией `.bss`. Точно так же статические имена уникальны по своей природе, поэтому компилятор может уверенно связать их с секцией `.data` или `.bss`.

### Упражнение 7.2 (решение в конце главы)

В этом упражнении запись  $\text{REF}(x.i) \rightarrow \text{DEF}(x.k)$  означает, что компоновщик связывает произвольную ссылку на имя  $x$  в модуле  $i$  с определением  $x$  в модуле  $k$ . Для каждого последующего примера используйте эту нотацию, чтобы показать, как редактор связей разрешил бы ссылки на многократно определенное имя в каждом модуле. Если вы предполагаете, что на этапе компоновки возникнет ошибка (правило 1), то впишите слово «ERROR». Если вы считаете, что компоновщик произвольно выберет одно из определений (правило 3), то впишите слово «UNKNOWN».

1.

<pre>/* Модуль 1 */ int main() { }</pre>	<pre>/* Модуль 2 */ int main; int p2() { }</pre>
<pre>(a) REF(main.1) → DEF( _____ . _____ ) (b) REF(main.2) → DEF( _____ . _____ )</pre>	

2.

<pre>/* Модуль 1 */ int main() { }</pre>	<pre>/* Модуль 2 */ int main = 1; int p2() { }</pre>
--	--



```

(a) REF(main.1) → DEF( _____ . _____ )
(b) REF(main.2) → DEF( _____ . _____ )

3.
/* Модуль 1 */           /* Модуль 2 */
int x;                   double x = 1.0;
int main()               int p2()
{                         {
}                         }

(a) REF(main.1) → DEF( _____ . _____ )
(b) REF(main.2) → DEF( _____ . _____ )

```

## 7.6.2. Связывание со статическими библиотеками

До сих пор мы предполагали, что компоновщик получает набор перемещаемых объектных файлов и связывает их вместе в выходной выполняемый файл. Практически все системы компиляции обеспечивают механизм, позволяющий упаковывать взаимосвязанные объектные модули в единый файл, называемый *статической библиотекой*, который можно затем передать компоновщику. Когда компоновщик связывает выходной выполняемый файл, он копирует из библиотеки только те объектные модули, на которые имеются ссылки из прикладной программы.

Почему системы поддерживают понятие библиотек? Рассмотрим стандарт ISO C99, определяющий широкий набор функций стандартного ввода/вывода, операций над строками и целочисленных математических функций, таких как `atoi`, `printf`, `scanf`, `strcpy` и `rand`. Они доступны каждой программе на языке C в библиотеке `libc.a`. ISO C99 определяет также обширный набор математических функций для операций с числами с плавающей точкой, таких как `sin`, `cos` и `sqrt`, в библиотеке `libm.a`.

Рассмотрим различные подходы, которые могли бы использовать разработчики компиляторов, чтобы дать пользователям доступ к этим функциям, не прибегая к использованию статических библиотек. Первый возможный подход: сконструировать компилятор, распознающий вызовы стандартных функций и добавляющий в программу соответствующий код. В языке Pascal, который предусматривает наличие небольшого набора стандартных функций, принят именно такой подход, но он неприемлем для языка C ввиду большого количества функций, определяемых стандартом языка. Выбор этого подхода существенно усложнил бы компилятор и потребовал изменить версию компилятора при добавлении новых, удалении старых или изменении существующих функций. Прикладным программистам, однако, такой подход был бы весьма удобен, потому что стандартные функции постоянно были бы доступны.

Другой подход: поместить все стандартные функции языка C в единственный перемещаемый объектный модуль, скажем `libc.o`, который прикладные программисты могли бы связывать со своими выполняемыми модулями:

```
linux> gcc main.c /usr/lib/libc.o
```

Преимущество этого подхода в том, что он отделяет реализацию стандартных функций от реализации компилятора и все еще достаточно удобен для программистов. Однако он имеет пару больших недостатков. Во-первых, каждый выполняемый файл будет содержать копии всех стандартных функций, что чрезвычайно расточительно с точки зрения использования дискового пространства. (В нашей системе `libc.a` занимает около 5 Мбайт и `libm.a` еще около 2 Мбайт.) Хуже того, каждая выполняющаяся программа тоже будет содержать копии всех стандартных функций в памяти, что было бы чрезвычайно расточительно, с точки зрения использования памяти. Во-вторых, любое изменение в любой стандартной функции, сколь бы незначительным оно ни было, потребует

от разработчиков библиотеки повторной компиляции всех файлов с исходным кодом, а это довольно продолжительная операция, которая усложнила бы разработку и сопровождение стандартных функций.

Мы могли бы ослабить эти проблемы, создав для каждой стандартной функции отдельный перемещаемый файл и сохранив эти файлы в известном каталоге. Однако этот подход потребовал бы от прикладных программистов явно компоновать свои программы с соответствующими объектными модулями, а это довольно трудоемкий процесс, при выполнении которого легко ошибиться:

```
linux> gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...
```

Для устранения неудобств и недостатков этих подходов была разработана концепция статической библиотеки. Взаимосвязанные функции можно собрать в отдельные объектные модули и затем упаковать в единственный файл статической библиотеки. Впоследствии прикладные программы смогут получить доступ к любым библиотечным функциям, для чего достаточно указать в командной строке единственное имя файла. Например, программу, использующую функции из стандартной и математической библиотек языка C, можно скомпилировать и скомпоновать с помощью такой команды:

```
linux> gcc main.c /usr/lib/libm.a /usr/lib/libc.a
```

Во время связывания компоновщик копирует только те объектные модули, на которые имеются ссылки в программе, что уменьшает размер выполняемого файла на диске и в памяти. С другой стороны, от прикладного программиста требуется включить имена лишь нескольких файлов библиотек. (Фактически драйверы компиляторов C всегда передают `libc.a` компоновщику, поэтому ссылка на `libc.a`, о которой говорилось выше, не нужна.)

В Linux статические библиотеки хранятся на диске в файлах, имеющих определенный формат, который называют *архивным*. Архив – это коллекция перемещаемых объектных файлов с заголовком, описывающим размер и адрес каждого элемента объектного файла. Имена архивных файлов обозначаются с использованием расширения `.a`. Чтобы сделать обсуждение библиотек более конкретным, предположим, что у нас есть пара процедур для обработки векторов, представленные в листинге 7.4. Каждая процедура определена в своем объектном модуле, выполняет операцию с парой векторов и сохраняет результат в выходном векторе. Дополнительно каждая процедура запоминает, сколько раз она была вызвана, увеличивая счетчик в глобальной переменной. (Это пригодится нам, когда мы перейдем обсуждению идеи позиционно-независимого кода в разделе 7.12.)

**Листинг 7.4.** Объектные файлы, составляющие библиотеку `libvector`

(a) `addvec.o`

```
code/link/addvec.c
1 int addcnt = 0;
2
3 void addvec(int *x, int *y,
4 int *z, int n)
5 {
6     int i;
7
8     addcnt++;
9
10    for (i = 0; i < n; i++)
11        z[i] = x[i] + y[i];
12 }
```

(b) `multvec.o`

```
code/link/multvec.c
1 int multcnt = 0;
2
3 void multvec(int *x, int *y,
4 int *z, int n)
5 {
6     int i;
7
8     multcnt++;
9
10    for (i = 0; i < n; i++)
11        z[i] = x[i] * y[i];
12 }
```

Создать статическую библиотеку из этих функций можно с помощью инструмента AR:

```
linux> gcc -c addvec.c multvec.c
linux> ar rcs libvector.a addvec.o multvec.o
```

Чтобы использовать эту библиотеку, можно написать приложение, как, например, `main2.c` в листинге 7.5, вызывающее библиотечную функцию `addvec` (заголовочный файл `vector.h` определяет прототипы функций из `libvector.a`).

**Листинг 7.5. Пример программы 2.** Эта программа вызывает функцию из библиотеки `libvector`

```
1 #include <stdio.h>
2 #include "vector.h"
3
4 int x[2] = {1, 2};
5 int y[2] = {3, 4};
6 int z[2];
7
8 int main()
9 {
10     addvec(x, y, z, 2);
11     printf("z = [%d %d]\n", z[0], z[1]);
12     return 0;
13 }
```

`ccode/link/main2.c`

`ccode/link/main2.c`

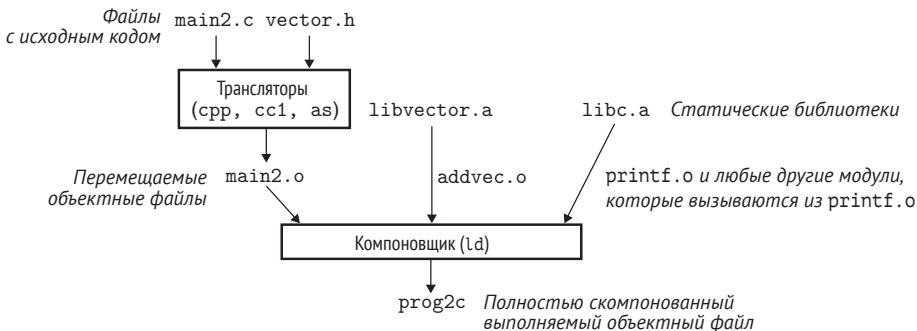
Чтобы собрать выполняемый файл, нужно скомпилировать и связать входные файлы `main2.o` и `libvector.a`:

```
linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o ./libvector.a
```

или так:

```
linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o -L. -lvector
```

На рис. 7.3 приводится обобщенная схема сборки программы. Параметр `-static` сообщает драйверу компилятора, что компоновщик должен собрать полностью связанный выполняемый объектный файл, который можно загрузить в память и запустить на выполнение без всякой дополнительной обработки во время загрузки. Аргумент `-lvector` – это более краткая форма записи `libvector.a`, а параметр `-L.` сообщает компоновщику, что тот должен искать `libvector.a` в текущем каталоге.



**Рис. 7.3.** Связывание со статическими библиотеками

Обработывая объектные файлы, компоновщик обнаруживает, что в `main2.o` имеется ссылка на имя `addvec`, определенное в `addvec.o`, поэтому он копирует `addvec.o` в выполняемый файл. Так как эта программа не ссылается на имена, определенные в `multvec.o`, компоновщик *не* копирует этот модуль в выполняемый файл. Кроме того, компоновщик копирует в выполняемый файл модуль `printf.o` из `libc.a` и еще несколько других модулей.

### 7.6.3. Как компоновщики разрешают ссылки на статические библиотеки

Статические библиотеки – полезные и необходимые инструменты, но они также являются источником недопонимания, если программист не имеет полного представления, как действует компоновщик Linux при разрешении внешних ссылок. При разрешении ссылок компоновщик просматривает перемещаемые объектные файлы и архивы слева направо, в той же последовательности, в какой они указаны в командной строке драйвера компилятора. (Драйвер автоматически преобразует любые файлы `.c`, указанные в командной строке, в файлы `.o`.) В процессе поиска имен компоновщик поддерживает набор  $E$  перемещаемых объектных файлов, которые будут объединены в выполняемый файл, набор  $U$  еще не найденных имен (на которые есть ссылки, но которые еще не определены) и набор имен  $D$ , которые были определены в предыдущих входных файлах. Первоначально  $E$ ,  $U$  и  $D$  – пустые.

- Для каждого входного файла  $f$  в командной строке компоновщик определяет, является ли  $f$  объектным файлом или архивом. Если  $f$  – объектный файл, то компоновщик добавляет его в набор  $E$ , корректирует  $U$  и  $D$ , чтобы отразить определения имени и ссылки в  $f$ , и переходит к следующему входному файлу.
- Если  $f$  – это архив, то компоновщик пытается отыскать в нем имена из набора неразрешенных ссылок  $U$ . Если какой-то член архива, например  $m$ , определяет имя, присутствующее в  $U$ , то  $m$  будет добавлен в  $E$  и компоновщик скорректирует наборы  $U$  и  $D$  так, чтобы отразить найденные определения имен в  $m$  и разрешенные ссылки на них. Этот процесс повторяется для всех членов архива, пока не будет достигнута точка, когда  $U$  и  $D$  перестают изменяться. После этого все члены объектных файлов, отсутствующие в  $E$ , просто отбрасываются, и компоновщик переходит к следующему входному файлу.
- Если после обработки входных файлов в наборе  $U$  еще останутся неразрешенные ссылки, то компоновщик выведет сообщение об ошибке и завершится, иначе – объединит и переместит объектные файлы в  $E$ , чтобы собрать выходной выполняемый файл.

К сожалению, этот алгоритм может приводить к некоторым странным ошибкам времени компоновки из-за того, что порядок следования библиотек и объектных файлов в командной строке имеет существенное значение. Если библиотека, где определено некоторое имя, появляется в командной строке перед объектным файлом, который ссылается на это имя, то ссылка не будет разрешена и связывание завершится ошибкой. Например, взгляните на следующий пример:

```
linux> gcc -static ./libvector.a main2.c
/tmp/cc9XH6Rp.o: In function 'main':
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to 'addvec'
```

Что здесь не так? После обработки `libvector.a` набор  $U$  останется пустым, поэтому в  $E$  не будет добавлено никаких объектных модулей из `libvector.a`. Как следствие ссылка на `addvec` не будет разрешена, и компоновщик выведет сообщение об ошибке и завершится.

Общее правило для библиотек – размещать их в конце командной строки. Если члены различных библиотек не зависят друг от друга, в том смысле, что если одна библиотека не содержит ссылок на имена, определенные в другой библиотеке, то эти библиотеки можно поместить в конец командной строки в любом порядке. Напротив, если библиотеки зависят друг от друга, то они должны быть указаны в командной строке в таком порядке, чтобы ссылки на имена в одном архиве предшествовали их определениям в другом. Например, предположим, что `foo.c` вызывает функции из `libx.a` и `libz.a`, а те вызывают функции из `liby.a`. В таком случае `libx.a` и `libz.a` должны предшествовать `liby.a` в командной строке:

```
linux> gcc foo.c libx.a libz.a liby.a
```

Библиотеки могут повторяться в командной строке, если это необходимо для удовлетворения циклических зависимостей. Например, если `foo.c` вызывает функцию из библиотеки `libx.a`, которая вызывает функцию из `liby.a`, а та вызывает функцию из `libx.a`, то `libx.a` необходимо повторно указать в командной строке:

```
linux> gcc foo.c libx.a liby.a libx.a
```

Как вариант можно объединить `libx.a` и `liby.a` в один архив.

### Упражнение 7.3 (решение в конце главы)

Пусть `a` и `b` обозначают объектные модули или статические библиотеки в текущем каталоге, и пусть `a → b` обозначает зависимость `a` от `b`, то есть `b` определяет имя, на которое ссылается `a`. Для каждого из следующих сценариев укажите минимальную командную строку (с наименьшим количеством объектных файлов и библиотек), которая позволит статическому компоновщику разрешить все ссылки.

1. `p.o → libx.a`
2. `p.o → libx.a → liby.a`
3. `p.o → libx.a → liby.a` и `liby.a → libx.a → p.o`

## 7.7. Перемещение

По завершении этапа разрешения ссылок каждая ссылка в программном коде будет связана с точно одним определением имени (элементом в таблице имен в одном из объектных модулей). К этому моменту компоновщик будет знать точные размеры секций программного кода и данных и готов приступить к этапу перемещения, в ходе которого объединит входные модули и назначит адрес каждому имени. Перемещение выполняется в два шага.

1. *Перемещение секций и определений имен.* На этом шаге компоновщик объединяет все секции одного типа в новую составную секцию того же типа. Например, все секции `.data` из входных модулей объединяются в единую секцию `.data` в выходном выполняемом объектном файле. Затем компоновщик назначает адреса времени выполнения новым объединенным секциям, включающим секции, определяемые входными модулями, и каждому имени, определенному во входных модулях. По завершении этого шага каждая (машинная) инструкция и каждая глобальная переменная в программе получают свой уникальный адрес.
2. *Перемещение ссылок на имена внутри секций.* На этом шаге компоновщик изменяет все ссылки на имена в программном коде и в секции данных так, чтобы они указывали на корректные адреса времени выполнения. Для этого компо-

новичок использует структуры данных в перемещаемых объектных модулях, так называемые записи перемещения, которые описываются далее.

### 7.7.1. Записи перемещения

Когда ассемблер генерирует объектный модуль, он не знает, где код и данные этого модуля будут находиться в памяти во время выполнения. Он также не знает адресов внешних функций или глобальных переменных, на которые ссылается данный модуль. Поэтому всякий раз, когда, столкнувшись со ссылкой на объект, чье местоположение неизвестно, ассемблер генерирует запись перемещения, чтобы сообщить компоновщику, как следует изменить ссылку при объединении объектных файлов в выполняемый. Записи перемещения для программного кода помещаются в секцию `.rel.text`; записи перемещения для инициализированных данных – в `.rel.data`.

В листинге 7.6 показан формат записи перемещения ELF. Поле `offset` – это смещение ссылки внутри секции, которую нужно изменить. Поле `symbol` определяет имя, на которое должна указывать ссылка. Поле `type` сообщает компоновщику, как следует изменить новую ссылку. Поле `addend` – константа со знаком, которая используется при перемещениях некоторых видов для корректировки значения измененной ссылки.

**Листинг 7.6.** Запись перемещения в формате ELF. Каждая запись определяет ссылку, которую необходимо переместить, и как получить измененную ссылку

```

1 typedef struct {
2     long offset; /* Смещение перемещаемой ссылки */
3     long type:32, /* Тип перемещения*/
4     symbol:32; /* Индекс в таблице имен */
5     long addend; /* Постоянная часть выражения перемещения */
6 } Elf64_Rela;

```

*ccode/link/elfstructs.c*

*ccode/link/elfstructs.c*

Формат ELF определяет 32 разных вида перемещений, порой весьма загадочных. Мы остановимся только на двух самых главных типах.

1. `R_X86_64_PC32`. Перемещает ссылку, использующую 32-битный адрес относительно счетчика инструкций (PC). Как рассказывалось в разделе 3.6.4, относительный адрес определяет смещение относительно текущего значения счетчика инструкций. Когда процессор выполняет инструкцию, использующую относительную адресацию, он формирует *эффективный адрес* (например, адрес процедуры в инструкции `call`), прибавляя 32-битное значение, закодированное в инструкции, к текущему значению счетчика инструкций, которое всегда является адресом следующей инструкции в памяти.
2. `R_X86_64_32`. Перемещает ссылку, использующую абсолютный 32-битный адрес. При использовании абсолютной адресации процессор использует 32-битное значение, закодированное в инструкции, непосредственно как эффективный адрес, без всяких модификаций.

Эти два типа перемещений поддерживают *малая модель кода* x86-64, в которой предполагается, что общий размер кода и данных в исполняемом объектном файле занимает меньше 2 Гбайт и к ним можно обращаться, используя 32-битные относительные адреса. Малая модель кода применяется в GCC по умолчанию. Программы с размером более 2 Гбайт можно скомпилировать с использованием флагов `-mmodel=medium` (*средняя модель кода*) и `-mmodel=large` (*большая модель кода*), но мы не будем их обсуждать.

## 7.7.2. Перемещение ссылок

В листинге 7.7 показан псевдокод, демонстрирующий алгоритм перемещения ссылок, используемый компоновщиком. Строки 1 и 2 организуют перебор каждой секции *s* и каждой записи перемещения *r* в секциях. Для конкретности предположим, что каждая секция *s* – это массив байтов, а каждая запись перемещения *r* – структура типа `Elf64_Rela` из листинга 7.6. Предположим также, что к тому времени, когда начнет выполняться алгоритм, компоновщик уже выбрал адреса времени выполнения для каждой секции (`ADDR(s)`) и каждого имени (`ADDR(r.symbol)`). В строке 3 вычисляется адрес в массиве *s* 4-байтных ссылок, которые должны быть перемещены. В случае относительной адресации перемещение будет выполнено в строках 5–9. В случае абсолютной адресации перемещение будет выполнено в строках 11–13.

**Листинг 7.7.** Алгоритм перемещения

```

1 foreach section s {
2   foreach relocation entry r {
3     refptr = s + r.offset; /* указатель на перемещаемую ссылку */
4
5     /* Перемещение ссылок с относительной адресацией */
6     if (r.type == R_X86_64_PC32) {
7       refaddr = ADDR(s) + r.offset; /* ссылка на адрес времени выполнения */
8       *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9     }
10
11    /* Перемещение ссылок с абсолютной адресацией */
12    if (r.type == R_X86_64_32)
13      *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14  }
15 }
```

Давайте посмотрим, как компоновщик использует этот алгоритм для перемещения ссылок в нашем примере программы в листинге 7.1. В листинге 7.8 показан дизассемблированный код из `main.o`, сгенерированный инструментом `GNU OBJDUMP` (`objdump -dx main.o`).

**Листинг 7.8.** Записи перемещения из `main.o`. Исходный код на C приводится в листинге 7.1

```

cocode/link/main-relo.d
1 0000000000000000 <main>:
2 0: 48 83 ec 08      sub $0x8,%rsp
3 4: be 02 00 00 00   mov $0x2,%esi
4 9: bf 00 00 00 00   mov $0x0,%edi      %edi = &array
5                          a: R_X86_64_32 array   Запись перемещения
6 e: e8 00 00 00 00   callq 13 <main+0x13> sum()
7                          f: R_X86_64_PC32 sum-0x4 Запись перемещения
8 13: 48 83 c4 08      add $0x8,%rsp
9 17: c3 retq

cocode/link/main-relo.d
```

Функция `main` ссылается на два глобальных имени, `array` и `sum`. Для каждой ссылки ассемблер сгенерировал запись перемещения, которая отображается в следующей строке<sup>2</sup>. Записи перемещения сообщают компоновщику, что ссылку на `sum` следует пере-

<sup>2</sup> Напомню, что записи перемещения и инструкции фактически хранятся в разных секциях объектного файла. Инструмент `OBJDUMP` отображает их вместе для удобства.



местить с использованием 32-битного относительного адреса, а ссылку на array – с использованием 32-битного абсолютного адреса. В следующих двух разделах подробно рассказывается, как компоновщик перемещает эти ссылки.

## Перемещение относительных ссылок

В строке 6 в листинге 7.8 функция main вызывает функцию sum, которая определена в модуле sum.o. Инструкция call находится со смещением 0xe от начала раздела и состоит из 1-байтного кода операции 0xe8, за которым следуют 4 байта относительной ссылки на вызываемую функцию sum.

Соответствующая запись перемещения r состоит из четырех полей:

```
r.offset = 0xf
r.symbol = sum
r.type   = R_X86_64_PC32
r.addend = -4
```

Согласно этим полям, компоновщик должен изменить 32-битную относительную ссылку, начиная со смещения 0xf, чтобы во время выполнения она указывала на процедуру sum. Теперь предположим, что компоновщик определил, что

```
ADDR(s) = ADDR(.text) = 0x4004d0
```

и

```
ADDR(r.symbol) = ADDR(sum) = 0x4004e8
```

Применяя алгоритм из листинга 7.8, компоновщик сначала вычисляет адрес времени выполнения данной ссылки (строка 7):

```
refaddr = ADDR(s) + r.offset
         = 0x4004d0 + 0xf
         = 0x4004df
```

Затем скорректируем саму ссылку так, чтобы она указывала на подпрограмму sum во время выполнения (строка 8):

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
          = (unsigned) (0x4004e8      + (-4)      - 0x4004df)
          = (unsigned) (0x5)
```

В получившемся выполняемом объектном файле инструкция call приобретает следующий переместимый вид:

```
4004de: e8 05 00 00 00    callq 4004e8 <sum>      sum()
```

Во время выполнения инструкция call будет находиться по адресу 0x4004de. Когда процессор приступит к ее выполнению, в счетчике инструкций PC будет храниться значение 0x4004e3 – адрес инструкции, следующей непосредственно за инструкцией call. Чтобы выполнить инструкцию вызова, процессор предпримет следующие действия:

1. Втолкнет PC в стек.
2.  $PC \leftarrow PC + 0x5 = 0x4004e3 + 0x5 = 0x4004e8$ .

То есть следующей будет выполнена первая инструкция в процедуре sum, что нам и нужно!

## Перемещение абсолютных ссылок

Перемещение абсолютных ссылок несложно. Например, в строке 4 в листинге 7.8 инструкция mov копирует адрес array (32-битное непосредственное значение) в ре-



гистр %edi. Инструкция mov начинается со смещения 0x9 от начала секции и состоит из 1-байтного кода операции 0xbf, за которым следуют 4 байта абсолютной ссылки на array.

Соответствующая запись перемещения r состоит из четырех полей:

```
r.offset = 0xa
r.symbol = array
r.type   = R_X86_64_32
r.addend = 0
```

Согласно этим полям, компоновщик должен изменить абсолютную ссылку, начиная со смещения 0xa, чтобы во время выполнения она указывала на первый байт array. Теперь предположим, что компоновщик определил, что

```
ADDR(r.symbol) = ADDR(array) = 0x601018
```

Выполнив строку 13 в листинге 7.7, компоновщик скорректирует ссылку:

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend)
          = (unsigned) (0x601018 + 0)
          = (unsigned) (0x601018)
```

В получившемся выполняемом объектном файле ссылка приобретает следующий переместимый вид:

```
4004d9: bf 18 10 60 00    mov     $0x601018,%edi    %edi = &array
```

В конечном итоге получатся секции .text и .data, как показано в листинге 7.9. Во время загрузки программы загрузчик скопирует байты из этих разделов непосредственно в память и запустит инструкции на выполнение без каких-либо дополнительных изменений.

**Листинг 7.9.** Секции .text и .data в выполняемом файле prog после перемещения. Исходный код на C показан в листинге 7.1

(a) Секция .text после перемещения

```
1  0000000004004d0 <main>:
2  4004d0: 48 83 ec 08      sub     $0x8,%rsp
3  4004d4: be 02 00 00 00   mov     $0x2,%esi
4  4004d9: bf 18 10 60 00   mov     $0x601018,%edi    %edi = &array
5  4004de: e8 05 00 00 00   callq   4004e8 <sum>      sum()
6  4004e3: 48 83 c4 08      add     $0x8,%rsp
7  4004e7: c3 retq

8  0000000004004e8 <sum>:
9  4004e8: b8 00 00 00 00   mov     $0x0,%eax
10 4004ed: ba 00 00 00 00   mov     $0x0,%edx
11 4004f2: eb 09           jmp     4004fd <sum+0x15>
12 4004f4: 48 63 ca        movslq  %edx,%rcx
13 4004f7: 03 04 8f        add     (%rdi,%rcx,4),%eax
14 4004fa: 83 c2 01        add     $0x1,%edx
15 4004fd: 39 f2          cmp     %esi,%edx
16 4004ff: 7c f3          jl      4004f4 <sum+0xc>
17 400501: f3 c3          repz retq
```

(b) Секция .data после перемещения

```
1  000000000601018 <array>:
2  601018: 01 00 00 00 02 00 00 00
```

**Упражнение 7.4 (решение в конце главы)**

Исследуйте перемещенный код в листинге 7.9 (а) и ответьте на следующие вопросы:

1. Определите адрес (шестнадцатеричный) ссылки на `sum` в строке 5.
2. Определите значение (шестнадцатеричное) ссылки на `sum` в строке 5.

**Упражнение 7.5 (решение в конце главы)**

Пусть для вызова функции `swap` в объектном файле `m.o` (листинг 7.3)

```
9: e8 00 00 00 00    callq e <main+0xe>    swap()
```

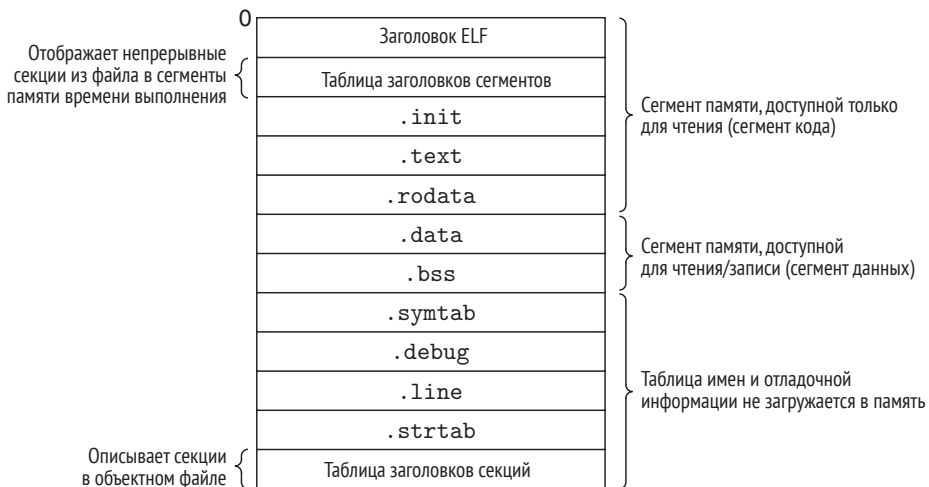
имеется следующая запись перемещения:

```
r.offset = 0xa
r.symbol = swap
r.type   = R_X86_64_PC32
r.addend = -4
```

Предположим, что компоновщик перемещает секцию `.text` из `m.o` в адрес `0x4004d0` и секцию `.text` из `swap.o` в адрес `0x4004e8`. Какое значение получит ссылка на `swap` в инструкции `callq` после перемещения?

## 7.8. Выполняемые объектные файлы

Мы уже видели, как компоновщик объединяет несколько объектных модулей в единый выполняемый объектный файл. Наша программа на языке C, которая начинала свое существование как простая совокупность текстовых файлов, была преобразована в единственный двоичный файл, содержащий всю информацию, необходимую для загрузки программы в память и ее выполнения. На рис. 7.4 приводится обобщенная картина организации типичного выполняемого файла ELF со всей заключенной в нем информацией.



**Рис. 7.4.** Типичный выполняемый объектный файл ELF

Формат выполняемого объектного файла напоминает формат перемещаемого объектного файла. Заголовок ELF описывает общий формат файла. Он также определяет *точку входа* в программу – адрес инструкции, с которой начинается выполнение программы. Секции `.text`, `.rodata` и `.data` подобны аналогичным секциям перемещаемого объектного файла, за исключением того, что они уже перемещены в адреса памяти времени выполнения. Секция `.init` определяет небольшую функцию с именем `_init`, которая вызывается кодом инициализации программы. Поскольку выполняемый файл является *полностью связанным*, он не нуждается в секциях `.rel`.

Формат выполняемых файлов ELF разработан так, чтобы упростить их загрузку в память и смежные фрагменты выполняемого файла отображались в смежные сегменты памяти. Это отображение описывается в *таблице заголовков сегментов*. В листинге 7.10 показана часть таблицы заголовков сегментов из выполняемого файла нашего примера программы `prog`, как ее выводит утилита `OBJDUMP`.

**Листинг 7.10.** Таблица заголовков сегментов из примера выполняемого файла `prog`.

Здесь `off` означает смещение (`offset`) от начала объектного файла; `vaddr/paddr` – адрес в памяти; `align` – требование к выравниванию; `filesz` – размер сегмента в объектном файле; `memsz` – размер сегмента в памяти; `flags` – разрешения времени выполнения

`ccode/link/prog-exe.d`

Сегмент кода, доступен только для чтения

```
1 LOAD off 0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
2     filesz 0x0000000000000069c memsz 0x0000000000000069c flags r-x
```

Сегмент данных, доступен для чтения и записи

```
3 LOAD off 0x0000000000000df8 vaddr 0x0000000000600df8 paddr 0x0000000000600df8 align 2**21
4     filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw
```

`ccode/link/prog-exe.d`

Из таблицы заголовков сегментов мы видим, что два сегмента памяти будут инициализированы содержимым выполняемого объектного файла. Строки 1 и 2 сообщают, что первый сегмент (*сегмент кода*) имеет разрешения на чтение и выполнение, начинается в памяти с адреса `0x400000`, имеет общий размер `0x69c` байт и будет инициализирован первыми `0x69c` байтами из выполняемого объектного файла, которые включают заголовок ELF, таблицу заголовков сегментов и секции `.init`, `.text` и `.rodata`.

Строки 3 и 4 сообщают, что второй сегмент (*сегмент данных*) имеет разрешения на чтение и запись, начинается в памяти с адреса `0x600df8`, занимает `0x230` байт и будет инициализирован `0x228` байтами из секции `.data` в объектном файле. Остальные 8 байт в этом сегменте соответствуют данным в секции `.bss` и будут инициализированы нулями во время выполнения.

Для любого сегмента `s` компоновщик должен выбрать такой начальный адрес `vaddr`, что

$$\text{vaddr} \bmod \text{align} = \text{off} \bmod \text{align}$$

где `off` – смещение первой секции в объектном файле, соответствующей сегменту, а `align` – выравнивание, указанное в заголовке программы ( $2^{21} = 0x200000$ ). Например, для сегмента данных в листинге 7.10:

$$\text{vaddr} \bmod \text{align} = 0x600df8 \bmod 0x200000 = 0xdf8$$

и

$$\text{off} \bmod \text{align} = 0xdf8 \bmod 0x200000 = 0xdf8$$

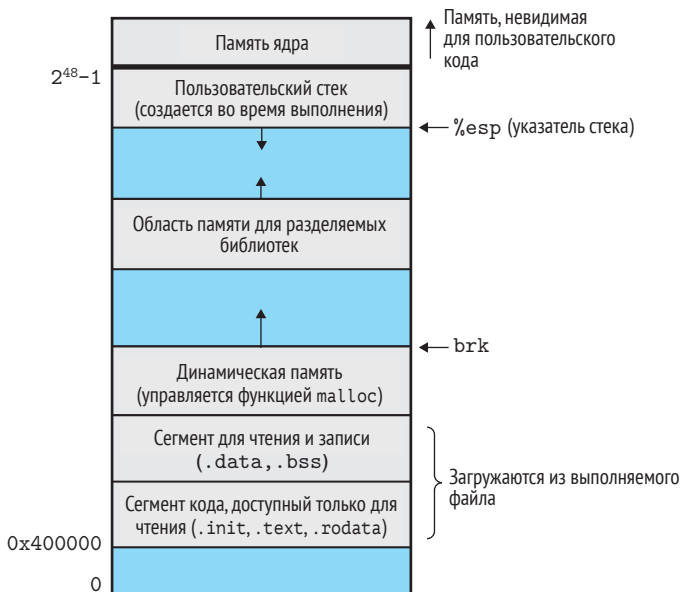
Это требование к выравниванию является оптимизацией, позволяющей эффективно загружать сегменты объектного файла в память во время выполнения программы. Причина такого поведения связана с организацией виртуальной памяти в виде больших непрерывных фрагментов с размерами, кратными степени двойки. Подробнее о виртуальной памяти рассказывается в главе 9.

## 7.9. Загрузка выполняемых объектных файлов

Чтобы запустить выполняемый объектный файл `prog`, достаточно ввести в командной строке его имя:

```
linux> ./prog
```

Поскольку имя `prog` не соответствует никакой встроенной команде, то командная оболочка предполагает, что `prog` – это выполняемый объектный файл, и передает его имя загрузчику операционной системы. Любая программа в Linux может запустить загрузчик, вызвав функцию `execve`, которую мы подробно обсудим в разделе 8.4.6. Загрузчик копирует код и данные из выполняемого объектного файла на диске в память и затем запускает программу, передав управление ее первой инструкции в *точке входа*. Этот процесс копирования программы в память и последующего выполнения называется *загрузкой*.



**Рис. 7.5.** Образ памяти времени выполнения в Linux с архитектурой x86-64. Разрывы, обусловленные требованиями к выравниванию и рандомизацией, здесь не показаны

Каждая программа в Linux имеет образ памяти времени выполнения, подобный изображенному на рис. 7.5. В системах Linux с архитектурой x86-64 сегмент программного кода всегда начинается с адреса `0x400000`. За ним следует сегмент данных. *Динамическая память* (куча) следует за сегментом данных и растет вверх по мере обращений к библиотечной функции `malloc` (подробнее функцию `malloc` и динамическую память мы обсудим в разделе 9.9). Далее следует область, зарезервированная для разделяемых модулей. Пользовательский стек начинается с наибольшего допустимого адреса ( $2^{48} - 1$ ) и

растет вниз (в сторону меньших адресов памяти). Область выше стека, начиная с адреса  $2^{48}$ , зарезервирована для кода и данных резидентной части операционной системы, известной как *ядро*.

Для простоты сегменты кучи, данных и кода изображены на рис. 7.5 как примыкающие друг к другу, а вершина стека размещена по наибольшему адресу, допустимому в пространстве пользователя. Но в действительности между сегментами кода и данных имеется разрыв, обусловленный требованиями к выравниванию сегмента `.data` (раздел 7.8). Кроме того, компоновщик использует рандомизацию адресного пространства (раздел 3.10.4), выбирая адреса времени выполнения для стека, разделяемых библиотек и кучи. Несмотря на то что расположение этих областей меняется при каждом запуске программы, их относительное положение одинаково.

После вызова загрузчик создает образ памяти, показанной на рис. 7.5. Руководствуясь таблицей заголовков сегментов в выполняемом файле, он копирует его сегменты программного кода и данных в память и затем передает управление в точку входа, которая всегда совпадает с адресом функции `_start`. Эта функция определяется в системном объектном файле `crt1.o` и одинакова для всех программ на языке C. Функция `_start` вызывает *системную функцию запуска* `__libc_start_main`, которая определена в `libc.so`. Он инициализирует среду выполнения, вызывает пользовательскую функцию `main`, обрабатывает возвращаемое ею значение и, при необходимости, возвращает управление ядру.

#### Как на самом деле работает загрузчик

Наше описание загрузки концептуально корректно, но в нем опущены многие детали. Чтобы понять, как в действительности происходит загрузка, необходимо знать и понимать такие концепции, как *процессы*, *виртуальная память* и *отображение памяти*, которые мы еще не обсуждали. Позже, когда мы познакомимся с этими понятиями в главах 8 и 9, мы вновь вернемся к вопросу загрузки и постепенно откроем эту тайну.

Для тех, кому не терпится, вот краткий обзор фактической работы загрузчика. Каждая программа в системе Linux выполняется в контексте процесса со своим собственным виртуальным адресным пространством. Когда командная оболочка запускает программу, родительский процесс запускает дочерний процесс – точную копию самого себя. Дочерний запускает загрузчик обращением к системному вызову `execve`. Загрузчик удаляет сегменты виртуальной памяти дочернего процесса и создает новый набор сегментов для программного кода, данных, кучи и стека. Новые сегменты стека и кучи инициализируются нулевыми байтами. Новые сегменты кода и данных инициализируются значением содержимого выполняемого файла путем отображения страниц виртуального адресного пространства во фрагменты выполняемого файла. Затем загрузчик передает управление в точку `_start`, что в конечном итоге приводит к вызову процедуры `main` приложения. Кроме некоторой информации из заголовка, никакие другие данные не копируются с диска в память. Копирование откладывается до того момента, когда процессор обратится к отображенной виртуальной странице, и в этот момент операционная система автоматически загрузит страницу с диска в память, используя свой механизм подкачки страниц.

## 7.10. Динамическое связывание с разделяемыми библиотеками

Статические библиотеки, которые мы изучали в разделе 7.6.2, решают многие проблемы, связанные с созданием больших коллекций функций для прикладных программ. Однако статические библиотеки все же имеют некоторые существенные недостат-

ки – они, как и любое другое программное обеспечение, должны поддерживаться и обновляться. Если прикладной программист захочет использовать самую последнюю версию библиотеки, он должен иметь возможность узнать, какие библиотеки изменились, и затем заново скомпоновать свои программы, использующие измененные библиотеки.

Еще одна проблема в том, что почти каждая программа на С использует стандартные функции ввода/вывода, такие как `printf` и `scanf`. Во время выполнения программный код этих функций будет дублироваться в сегменте `.text` каждого выполняющегося процесса. В типичной системе выполняются сотни процессов, поэтому дублирование кода стандартных функций может приводить к существенным затратам такого дефицитного ресурса, как память. (Интересное свойство памяти – она *всегда* будет оставаться дефицитным ресурсом, сколько бы ее ни было в системе. Это свойство также присуще дисковому пространству и мусорным ведрам на кухнях.)

*Разделяемые библиотеки* (совместно используемые библиотеки) – одна из недавних новинок, созданная с целью избавиться от недостатков статических библиотек. Разделяемая библиотека – это объектный модуль, который во время выполнения может быть загружен в произвольный адрес памяти и связан с выполняющейся программой. Этот процесс известен как *динамическое связывание* и выполняется программой, называемой *динамическим компоновщиком*. Разделяемые библиотеки также называют *разделяемыми объектами* (shared objects) и в системах Linux обычно обозначают расширением файла `.so`. В операционных системах Microsoft разделяемые библиотеки тоже широко используются и называются библиотеками DLL (Dynamic Link Libraries – динамически связываемые библиотеки).

Разделяемые библиотеки могут «разделяться» (совместно использоваться) двумя способами. Во-первых, в любой файловой системе имеется точно один файл `.so` определенной библиотеки. Программный код и данные в этом файле совместно используются всеми выполняемыми объектными файлами, которые ссылаются на данную библиотеку, в отличие от содержимого статических библиотек, которое копируется и встраивается в выполняемые файлы, ссылающиеся на них. Во-вторых, единственная копия секции `.text` разделяемой библиотеки в памяти может совместно использоваться разными выполняющимися процессами. Мы рассмотрим этот вопрос более подробно после знакомства с виртуальной памятью в главе 9.

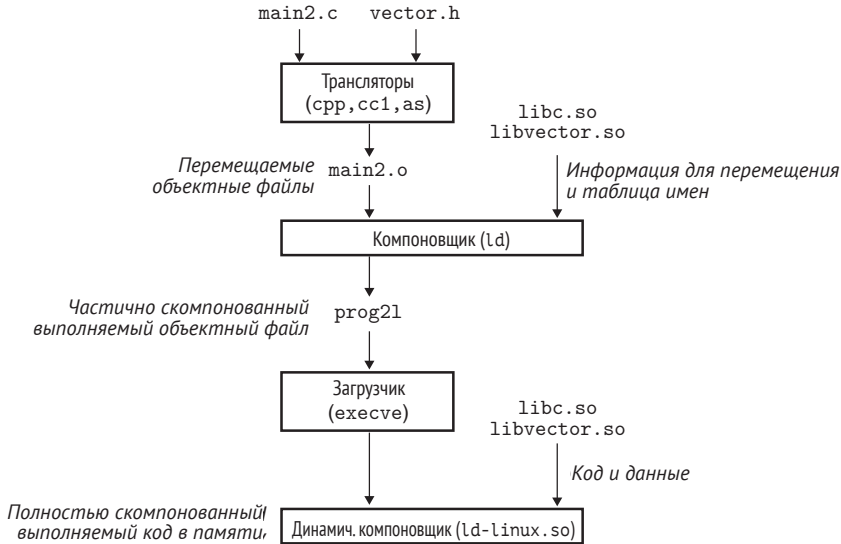
На рис. 7.6 приводится обобщенная схема процесса динамического связывания для примера программы из листинга 7.5. Чтобы построить разделяемую библиотеку `libvector.so` функций для выполнения операций с векторами из листинга 7.4, мы должны передать драйверу компилятора специальную директиву для компоновщика:

```
linux> gcc -shared -fPIC -o libvector.so addvec.c multvec.c
```

Флаг `-fPIC` предписывает компилятору сгенерировать *позиционно-независимый код* (Position-Independent Code, PIC). Флаг `-shared` предписывает компоновщику создать разделяемый объектный файл. После создания библиотеки ее можно связать с нашей программой из листинга на рис. 7.5:

```
linux> gcc -o prog21 main2.c ./libvector.so
```

Эта команда создаст выполняемый объектный файл `prog21` в форме, позволяющей скомпоновать его с `libvector.so` во время выполнения. Основная идея – настроить одну часть связей статически, в процессе создания выполняемого файла, а другую – после загрузки программы. Важно отметить, что в этот момент никакие секции – кода или данных – из `libvector.so` не копируются в выполняемый файл `prog21`. Компоновщик копирует лишь некоторую информацию из таблицы перемещений и таблицы имен, которая позволит разрешить ссылки на код и данные в `libvector.so` во время загрузки.



**Рис. 7.6.** Динамическое связывание с разделяемыми библиотеками

Когда загрузчик загружает и затем запускает выполняемый модуль `prog2l`, он загружает частично скомпонованный выполняемый файл `prog2l`, используя методы, описанные в разделе 7.9. Затем замечает, что `prog2l` содержит секцию `.interp`, содержащую путь к динамическому компоновщику, который сам является разделяемым объектом (например, `ld-linux.so` в системах Linux). Вместо того чтобы передать управление приложению, как это обычно делается, загрузчик загружает и запускает динамический компоновщик, который затем завершает задачу компоновки, выполняя следующие действия:

- перемещение кода и данных из `libc.so` в некоторый сегмент памяти;
- перемещение кода и данных из `libvector.so` в другой сегмент памяти;
- перемещение каждой ссылки в `prog2l` на имена, определенные в `libc.so` и `libvector.so`.

Наконец, динамический компоновщик передает управление приложению. С этого момента местоположение разделяемых библиотек фиксируется и не изменяется в течение всего времени выполнения программы.

## 7.11. Загрузка и связывание с разделяемыми библиотеками из приложений

Вплоть до этого момента мы рассматривали сценарий, когда динамический загрузчик загружает и связывает разделяемые библиотеки в процессе загрузки приложения, непосредственно перед его выполнением. Однако приложение может также обратиться к динамическому загрузчику, чтобы загрузить и задействовать произвольную разделяемую библиотеку, уже после запуска.

Динамическое связывание – мощный и полезный инструмент. Вот только некоторые примеры из практики:

- *распространение программного обеспечения.* Разработчики приложений для Microsoft Windows часто используют разделяемые библиотеки для распростра-

нения обновлений своих программ. Они генерируют новую копию разделяемой библиотеки, после чего пользователи могут загрузить и установить ее взамен текущей версии. После этого при следующем запуске приложение автоматически загрузит и будет использовать новую версию разделяемой библиотеки;

- *построение высокопроизводительных веб-серверов.* Многие веб-серверы управляют динамическим контентом, таким как персонифицированные веб-страницы, учетные записи и рекламные объявления. Ранние веб-серверы генерировали динамический контент с помощью функций `fork` и `execve`, используя дочерний процесс для выполнения «программы CGI». Но современные высокопроизводительные веб-серверы используют намного более эффективный и совершенный подход, основанный на динамическом связывании.

Идея состоит в том, чтобы упаковывать каждую функцию, генерирующую динамический контент, в разделяемую библиотеку. Получив запрос от веб-браузера, сервер динамически загрузит соответствующую функцию и вызовет ее напрямую, без запуска дочернего процесса с помощью `fork` и `execve`. Функция кешируется в адресном пространстве сервера, поэтому последующие аналогичные запросы будут обрабатываться намного быстрее, простым вызовом этой функции. Это может оказать существенное влияние на эффективность работы сайта. Кроме того, можно модифицировать существующие функции и добавлять новые прямо во время выполнения, не останавливая сервер.

Системы класса Linux предлагают простой интерфейс к динамическому компоновщику, который позволяет прикладным программам загружать и связывать разделяемые библиотеки во время выполнения.

```
#include <dlfcn.h>
```

```
void *dlopen(const char *filename, int flag);
```

Возвращает указатель на дескриптор в случае успеха  
и NULL в случае ошибки

Функция `dlopen` загружает и связывает разделяемую библиотеку `filename`. Внешние имена в `filename` разрешаются с использованием библиотек, ранее открытых с флагом `RTLD_GLOBAL`. Если текущий выполняемый модуль скомпилирован с флагом `-rdynamic`, то его глобальные имена также доступны для разрешения. Аргумент `flag` должен включать флаг `RTLD_NOW`, сообщающий компоновщику, что тот должен разрешать ссылки на внешние имена непосредственно, или `RTLD_LAZY`, сообщающий, что компоновщик должен отложить разрешение ссылок до момента обращения к данной библиотеке. Любой из этих флагов можно объединить логической операцией ИЛИ вместе с флагом `RTLD_GLOBAL`.

```
#include <dlfcn.h>
```

```
void *dlsym(void *handle, char *symbol);
```

Возвращает указатель на `symbol` в случае успеха  
и NULL в случае ошибки

Функция `dlsym` принимает аргумент `handle` с дескриптором предварительно открытой разделяемой библиотеки и идентификатор `symbol` и возвращает адрес, соответствующий этому идентификатору, если тот существует, или NULL в противном случае.



```
#include <dlfcn.h>
```

```
int dlclose (void *handle);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Функция `dlclose` выгружает разделяемую библиотеку, если она нигде больше не используется.

```
#include <dlfcn.h>
```

```
const char *dlerror(void);
```

Возвращает сообщение об ошибке, если предыдущий вызов `dlopen`, `dlsym` или `dlclose` потерпел неудачу, и `NULL`, если предыдущие вызовы перечисленных функций завершились успехом

Функция `dlerror` возвращает строку, описывающую самую последнюю ошибку, возникшую при вызове `dlopen`, `dlsym` или `dlclose`, или `NULL`, если ошибки не было.

В листинге 7.11 показано, как можно использовать этот интерфейс для динамического связывания нашей разделяемой библиотеки `libvector.so` с приложением во время выполнения и затем вызвать процедуру `addvec`. Чтобы скомпилировать программу, GCC следует вызывать, как показано ниже:

```
linux> gcc -rdynamic -o prog2r dll.c -ldl
```

**Листинг 7.11. Пример программы 3.** Динамическая загрузка и связывание разделяемой библиотеки `libvector.so` во время выполнения

*ccode/link/dll.c*

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <dlfcn.h>
4
5 int x[2] = {1, 2};
6 int y[2] = {3, 4};
7 int z[2];
8
9 int main()
10 {
11     void *handle;
12     void (*addvec)(int *, int *, int *, int);
13     char *error;
14
15     /* Динамически загрузить разделяемую библиотеку с функцией addvec() */
16     handle = dlopen("./libvector.so", RTLD_LAZY);
17     if (!handle) {
18         fprintf(stderr, "%s\n", dlerror());
19         exit(1);
20     }
21
22     /* Получить указатель на функцию addvec() */
23     addvec = dlsym(handle, "addvec");
24     if ((error = dlerror()) != NULL) {
25         fprintf(stderr, "%s\n", error);
```

```

26     exit(1);
27 }
28
29 /* Теперь можно вызвать addvec(), как самую обычную функцию */
30 addvec(x, y, z, 2);
31 printf("z = [%d %d]\n", z[0], z[1]);
32
33 /* Выгрузить разделяемую библиотеку */
34 if (dlclose(handle) < 0) {
35     fprintf(stderr, "%s\n", dlerror());
36     exit(1);
37 }
38 return 0;
39 }

```

*ccode/link/dll.c*

### Разделяемые библиотеки и интерфейс языка Java

Язык Java определяет стандарт соглашения вызовов двоичных функций *Java Native Interface (JNI)*. Этот стандарт допускает вызов функций, написанных на языках C и C++ из программ на Java. Основная идея JNI состоит в том, чтобы скомпилировать функцию на языке C (например, *foo*) в разделяемую библиотеку (например, *foo.so*). Когда программа на Java попытается вызвать функцию *foo*, интерпретатор Java вызовет интерфейс *dlopen* (или подобный ему), чтобы динамически загрузить и связать *foo.so*, а затем вызвать *foo*.

## 7.12. Перемещаемый программный код

Основное назначение разделяемых библиотек – позволить нескольким процессам совместно использовать один и тот же библиотечный код и таким образом сэкономить дорогостоящий ресурс памяти. Но как несколько процессов могут вместе использовать единственную копию программного кода? Одно из возможных решений – назначить каждой разделяемой библиотеке определенный участок адресного пространства, а затем потребовать, чтобы загрузчик всегда загружал разделяемую библиотеку по этому адресу. Такое решение кажется довольно простым, но в действительности порождает серьезные проблемы. Подобное использование адресного пространства очень неэффективно, потому что некоторые его области будут зарезервированы для возможного использования, даже если процесс не использует библиотеку. Кроме того, таким механизмом трудно управлять. Мы не можем гарантировать, что никакие области не будут перекрываться. После каждого изменения библиотеки придется удостовериться, что она все еще умещается в пределы назначенной ей области памяти, иначе придется подыскать для нее новый участок. А если мы создадим новую библиотеку, то придется найти место и для нее. Через какое-то время мы получим сотни библиотек и их версий и будет очень трудно предотвратить фрагментацию адресного пространства на множество маленьких и непригодных для использования «дырок». Хуже того, в разных системах библиотеки придется закреплять за разными областями памяти, что вызовет еще большую головную боль.

Чтобы избежать подобных проблем, современные системы компилируют библиотечный код так, чтобы его можно было загрузить по любому адресу без модификации с использованием компоновщика. При таком подходе мы получаем единственную копию сегмента разделяемого кода, который может использоваться неограниченным количеством процессов. (При этом, конечно же, каждый процесс получит свою копию сегмента данных, доступных для чтения/записи.)

Программный код, который можно загружать без перемещения, называется *позиционно-независимым* (Position-Independent Code, PIC). В системе компиляции GNU такой код генерируется при компиляции с параметром `-fpic`. Разделяемые библиотеки всегда должны компилироваться с этим параметром.

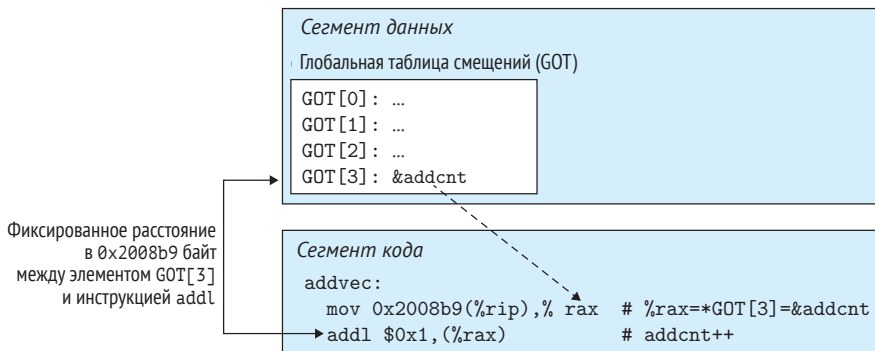
В системах x86-64 ссылки на идентификаторы в том же выполняемом объектном модуле не требуют специальной подготовки, чтобы получить позиционно-независимый код. При компиляции этих ссылок уже используется относительная адресация и перемещение выполняется статическим компоновщиком при сборке объектного файла. Но ссылки на внешние процедуры и глобальные переменные, определяемые в разделяемой библиотеке, требуют дополнительной обработки с применением некоторых специальных приемов, о которых рассказывается далее.

### Позиционно-независимые ссылки на данные

Компиляторы генерируют позиционно-независимые ссылки на глобальные переменные, используя следующий интересный факт: независимо от того, куда в памяти загружается объектный модуль (в том числе и разделяемые), сегмент данных всегда будет следовать сразу за сегментом кода. Соответственно, *расстояние* от любой инструкции в сегменте кода до любой переменной в сегменте данных во время выполнения остается постоянным и не зависит от абсолютных адресов начала сегментов кода и данных.

Чтобы получить позиционно-независимые ссылки на глобальные переменные, компилятор создает в начале сегмента данных *глобальную таблицу смещений* (Global Offset Table, GOT). Таблица GOT содержит запись для каждого глобального объекта данных (процедуру или глобальную переменную), на который имеется ссылка в объектном модуле. Компилятор также генерирует запись перемещения для каждого элемента таблицы GOT. Во время загрузки динамический компоновщик преобразует каждую запись в GOT так, чтобы она содержала абсолютный адрес соответствующего объекта. Каждый объектный модуль, ссылающийся на глобальные данные, имеет свою таблицу GOT.

На рис. 7.7 показана таблица GOT для нашего примера разделяемой библиотеки `libvector.so`. Подпрограмма `addvec` получает адрес глобальной переменной `addcnt` из `GOT[3]`, а затем увеличивает значение `addcnt` в памяти. Ключевая идея заключается в том, что относительное смещение ссылки на `GOT[3]` является постоянным во время выполнения.



**Рис. 7.7.** Использование таблицы GOT для ссылки на глобальную переменную. Подпрограмма `addvec` в `libvector.so` применяет косвенную ссылку на `addcnt` через таблицу GOT библиотеки `libvector.so`

Поскольку `addcnt` определяется модулем `libvector.so`, компилятор мог бы использовать постоянное расстояние между сегментами кода и данных, создав прямую относительную ссылку на `addcnt` и добавив для компоновщика задание на перемещение

при сборке разделяемого модуля. Однако если бы `addcnt` была определена другим разделяемым модулем, то потребовался бы косвенный доступ через GOT. В этом случае компилятор решил использовать наиболее универсальное решение – адресацию через GOT – для всех ссылок.

### Позиционно-независимые ссылки на функции

Предположим, что программа вызывает функцию, которая определена в разделяемой библиотеке. Компилятор не может предсказать, по какому адресу окажется функция во время выполнения, потому что разделяемый модуль может быть загружен в любое место. Решить эту проблему можно созданием записи перемещения для ссылки, которую динамический компоновщик мог бы разрешить при загрузке программы. Однако такой подход не может считаться позиционно-независимым, так как требует от компоновщика изменить сегмент кода вызывающего модуля. Системы компиляции GNU решают эту проблему с помощью любопытного метода, получившего название *ленивое связывание*, в соответствии с которым связывание каждой процедуры откладывается до *первого* ее вызова.

Причина откладывания связывания заключается в том, что типичная прикладная программа вызывает лишь несколько из сотен или тысяч функций, экспортируемых такими универсальными библиотеками, как `libc.so`. Откладывая разрешение адреса функции до ее фактического вызова, динамический компоновщик может избежать сотен или тысяч ненужных перемещений во время загрузки. При первом вызове функции, конечно, возникают накладные расходы, но для всех последующих вызовов требуется только одна инструкция и одна ссылка на память для извлечения адреса.

Отложенное связывание реализуется с использованием компактного, но немного сложного взаимодействия между двумя структурами данных: глобальной таблицей смещений GOT и *таблицей связывания процедур* (Procedure Linkage Table, PLT). Если объектный модуль вызывает какие-либо функции, находящиеся в разделяемых библиотеках, то он имеет свои таблицы GOT и PLT. GOT является частью сегмента данных, а PLT – частью сегмента кода.

На рис. 7.8 показано, как используются таблицы PLT и GOT для разрешения адреса функции во время выполнения. Для начала рассмотрим содержимое каждой из этих таблиц.

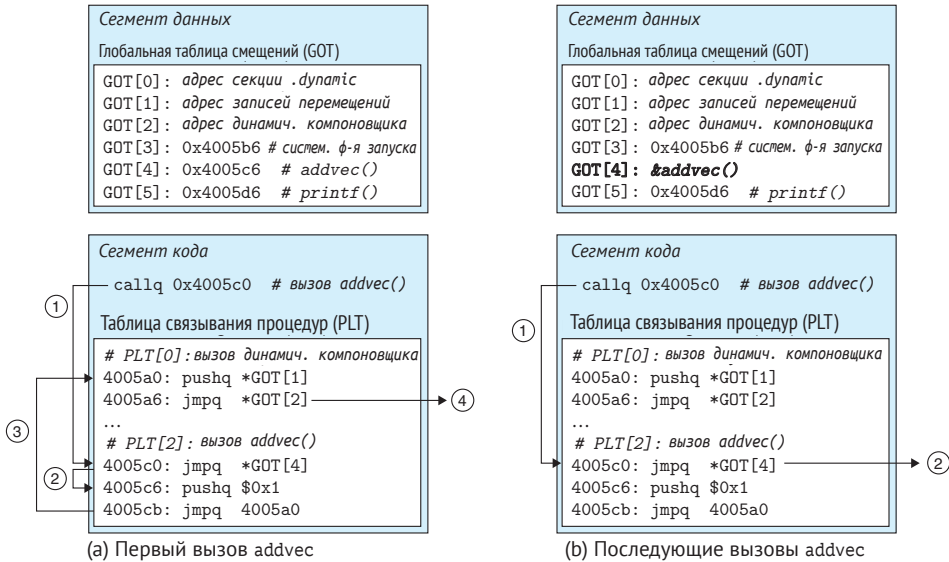
#### Таблица связывания процедур (PLT)

PLT – это массив 16-байтных записей. `PLT[0]` – специальная запись, вызывающая динамический компоновщик. Для каждой функции в разделяемой библиотеке, которая вызывается выполняемым файлом, имеется своя запись в таблице PLT. Каждая запись отвечает за вызов определенной функции. `PLT[1]` (здесь не показана) вызывает системную функцию запуска (`__libc_start_main`), которая инициализирует окружение выполнения, в свою очередь вызывает функцию `main` и обрабатывает возвращаемое ею значение. Записи, начинающиеся с `PLT[2]`, соответствуют библиотечным функциям, вызываемым пользовательским кодом. В нашем примере `PLT[2]` вызывает `addvec`, а `PLT[3]` (здесь не показана) вызывает `printf`.

#### Таблица глобальных смещений (GOT)

Как мы уже видели, таблица GOT – это массив 8-байтных записей с адресами. При использовании в паре с PLT записи `GOT[0]` и `GOT[1]` содержат информацию, которая помогает динамическому компоновщику разрешать адреса функций. `GOT[2]` – это точка входа в динамический компоновщик в модуле `ld-linux.so`. Все остальные записи соответствуют вызываемым функциям, адреса которых необходимо разрешить во время выполнения. Каждой из них соответствует своя запись в PLT.

Например, GOT[4] и PLT[2] соответствуют функции `addvec`. Первоначально каждая запись в GOT указывает на вторую инструкцию в соответствующей записи PLT.



**Рис. 7.8.** Использование таблиц PLT и GOT для вызова внешних функций. Динамический компоновщик разрешает адрес `addvec` при первом вызове

На рис. 7.8 (а) показано, как механизм отложенного разрешения использует таблицы GOT и PLT для определения адреса функции `addvec` при первом вызове:

- Шаг 1.** Вместо прямого вызова `addvec` программа вызывает запись PLT[2], которая соответствует функции `addvec`.
- Шаг 2.** Первая инструкция в записи PLT выполняет косвенный переход через GOT[4]. Поскольку каждая запись в GOT первоначально ссылается на вторую инструкцию в соответствующей записи PLT, косвенный переход просто передает управление следующей инструкции в PLT[2].
- Шаг 3.** После помещения идентификационного номера `addvec` (0x1) в стек PLT[2] переходит к PLT[0].
- Шаг 4.** PLT[0] передает аргумент динамическому компоновщику косвенно через GOT[1], а затем косвенно переходит к динамическому компоновщику через GOT[2]. Динамический компоновщик извлекает две записи из стека для определения местоположения `addvec` во время выполнения, записывает полученный адрес в GOT[4] и передает управление `addvec`.

На рис. 7.8 (б) показано, как происходят все последующие вызовы `addvec`:

- Шаг 1.** Управление все так же передается записи PLT[2].
- Шаг 2.** Однако на этот раз косвенный переход через GOT[4] передает управление непосредственно `addvec`.

## 7.13. Подмена библиотечных функций

Компоновщики Linux поддерживают мощную методику *подмены библиотечных функций* (library interpositioning), которая позволяет перехватывать вызовы библиотечных

функций и вместо них выполнять свой код. Используя методику подмены, можно, например, подсчитать, сколько раз вызывалась конкретная библиотечная функция, проверить ее аргументы и результат и даже заменить ее совершенно другой реализацией.

Вот основная идея: для некоторой *целевой функции*, вызов которой нужно перехватить, создается функция-обертка с прототипом, идентичным прототипу целевой функции. Затем, используя определенный механизм подмены, вы обманываете систему, заставляя ее вызывать функцию-обертку вместо целевой функции. Функция-обертка обычно выполняет свою логику, а затем вызывает целевую функцию и передает возвращаемое ею значение обратно вызывающему коду.

Подмена может происходить во время компиляции, компоновки или выполнения, когда программа загружается и выполняется. Для изучения этих трех механизмов мы используем пример программы в листинге 7.12 (а). Она вызывает функции `malloc` и `free` из стандартной библиотеки C (`libc.so`). Вызов `malloc` выделяет из кучи блок размером 32 байта и возвращает указатель на этот блок. Вызов `free` возвращает блок обратно в кучу для использования последующими вызовами `malloc`. Наша цель – использовать механизм подмены для отслеживания вызовов `malloc` и `free` во время работы программы.

### 7.13.1. Подмена во время компиляции

В листинге 7.12 показано, как реализовать подмену на этапе компиляции с помощью препроцессора C. Каждая функция-обертка в `mymalloc.c` (листинг 7.12 (с)) вызывает целевую функцию, печатает трассировку и возвращает результат. Локальный заголовочный файл `malloc.h` (рис. 7.12 (b)) сообщает препроцессору, что тот должен заменить все вызовы целевой функции вызовом ее обертки. Вот как скомпилировать и скомпоновать программу:

```
linux> gcc -DCOMPILETIME -c mymalloc.c
linux> gcc -l -o intc int.c mymalloc.o
```

Подмена обеспечивается параметром `-I.`, аргумент (точка) которого требует от препроцессора C искать заголовочные файлы (в том числе и `malloc.h`) сначала в текущем каталоге, а затем в обычных системных каталогах. Обратите внимание, что функции-обертки в `mymalloc.c` компилируются со стандартным заголовочным файлом `malloc.h`.

Эта программа выведет следующее:

```
linux> ./intc
malloc(32)=0x9ee010
free(0x9ee010)
```

#### Листинг 7.12. Подмена во время компиляции с помощью препроцессора C

(а) Пример программы `int.c`

```
1 #include <stdio.h>
2 #include <malloc.h>
3
4 int main()
5 {
6     int *p = malloc(32);
7     free(p);
8     return(0);
9 }
```

<code/link/interpose/int.c>

<code/link/interpose/int.c>

(b) Локальный файл `malloc.h`*code/link/interpose/malloc.h*

```

1 #define malloc(size) mymalloc(size)
2 #define free(ptr) myfree(ptr)
3
4 void *mymalloc(size_t size);
5 void myfree(void *ptr);

```

*code/link/interpose/malloc.h*(c) Функции-обертки в `mymalloc.c`*code/link/interpose/mymalloc.c*

```

1 #ifdef COMPILETIME
2 #include <stdio.h>
3 #include <malloc.h>
4
5 /* Функция-обертка для malloc */
6 void *mymalloc(size_t size)
7 {
8     void *ptr = malloc(size);
9     printf("malloc(%d)=%p\n",
10          (int)size, ptr);
11     return ptr;
12 }
13
14 /* Функция-обертка для free */
15 void myfree(void *ptr)
16 {
17     free(ptr);
18     printf("free(%p)\n", ptr);
19 }
20 #endif

```

*code/link/interpose/mymalloc.c*

### 7.13.2. Подмена во время компоновки

Статический компоновщик Linux поддерживает подмену во время компоновки, предлагая для этого флаг `--wrap f`. Этот флаг требует от компоновщика разрешать ссылки на имя `f` как на `__wrap_f` (два символа подчеркивания в начале) и ссылки на имя `__real_f` (два символа подчеркивания в начале) как на `f`. В листинге 7.13 показаны обертки для нашего примера программы.

Вот как скомпилировать исходные файлы в перемещаемые объектные файлы:

```

linux> gcc -DLINKTIME -c mymalloc.c
linux> gcc -c int.c

```

**Листинг 7.13.** Подмена во время компоновки с использованием флага `--wrap`

*code/link/interpose/mymalloc.c*

```

1 #ifdef LINKTIME
2 #include <stdio.h>
3
4 void *__real_malloc(size_t size);
5 void __real_free(void *ptr);
6
7 /* Функция-обертка для malloc */

```

```

8 void *__wrap_malloc(size_t size)
9 {
10     void *ptr = __real_malloc(size); /* Вызов malloc из libc */
11     printf("malloc(%d) = %p\n", (int)size, ptr);
12     return ptr;
13 }
14
15 /* Функция-обертка для free */
16 void __wrap_free(void *ptr)
17 {
18     __real_free(ptr); /* Вызов free из libc */
19     printf("free(%p)\n", ptr);
20 }
21 #endif

```

*code/link/interpose/mymalloc.c*

А вот как скомпоновать объектные файлы в выполняемый модуль:

```
linux> gcc -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.o mymalloc.o
```

Параметр `-Wl, option` передает свой аргумент `option` компоновщику. Запятые в `option` заменяются пробелами. То есть параметр `-Wl,--wrap,malloc` передаст `--wrap malloc` компоновщику, аналогично сработает параметр `-Wl,--wrap,free`.

Эта программа выведет следующее:

```

linux> ./intl
malloc(32) = 0x18cf010
free(0x18cf010)

```

### 7.13.3. Подмена во время выполнения

Чтобы выполнить подмену во время компиляции, необходимо иметь доступ к файлам с исходным кодом программы. Для подмены во время компоновки необходимо иметь доступ к перемещаемым объектным файлам. Однако существует механизм подмены во время выполнения, которому достаточно иметь доступ только к выполняемому объектному файлу. Этот удивительный механизм основан на переменной окружения `LD_PRELOAD` динамического компоновщика.

Если в переменной окружения `LD_PRELOAD` задать список путей к разделяемым библиотекам (через пробелы или двоеточие), то при загрузке и выполнении программы динамический компоновщик (`LD-LINUX.SO`) сначала будет искать библиотеки в каталогах, перечисленных в `LD_PRELOAD`, и только потом в других стандартных каталогах. С помощью этого механизма можно перехватить вызов любой функции в любой библиотеке, включая `libc.so`, во время загрузки и в ходе выполнения любого выполняемого файла.

В листинге 7.14 показаны функции-обертки для `malloc` и `free`. Вызов `dlsym` в каждой из них возвращает указатель на целевую функцию в `libc`. Затем обертка вызывает целевую функцию, выводит свою информацию и возвращает значение, полученное от целевой функции.

**Листинг 7.14.** Подмена во время выполнения с помощью `LD_PRELOAD`

*code/link/interpose/mymalloc.c*

```

1 #ifdef RUNTIME
2 #define _GNU_SOURCE
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <dlfcn.h>

```



```

6
7 /* Функция-обертка для malloc */
8 void *malloc(size_t size)
9 {
10     void *(*mallocp)(size_t size);
11     char *error;
12
13     mallocp = dlsym(RTLD_NEXT, "malloc"); /* Получить адрес malloc в libc */
14     if ((error = dlerror()) != NULL) {
15         fputs(error, stderr);
16         exit(1);
17     }
18     char *ptr = mallocp(size); /* Вызов malloc из libc */
19     printf("malloc(%d) = %p\n", (int)size, ptr);
20     return ptr;
21 }
22
23 /* Функция-обертка free */
24 void free(void *ptr)
25 {
26     void (*freep)(void *) = NULL;
27     char *error;
28
29     if (!ptr)
30         return;
31
32     freep = dlsym(RTLD_NEXT, "free"); /* Получить адрес free в libc */
33     if ((error = dlerror()) != NULL) {
34         fputs(error, stderr);
35         exit(1);
36     }
37     freep(ptr); /* Вызвать free из libc */
38     printf("free(%p)\n", ptr);
39 }
40 #endif

```

*code/link/interpose/mymalloc.c*

Вот как собрать разделяемую библиотеку, содержащую функции-обертки:

```
linux> gcc -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
```

Следующая команда скомпилирует основную программу:

```
linux> gcc -o intr int.c
```

Вот как запустить программу в командной оболочке bash<sup>3</sup>:

```
linux> LD_PRELOAD="./mymalloc.so" ./intr
malloc(32) = 0x1bf7010
free(0x1bf7010)
```

А так – в csh или в tcsh:

```
linux> (setenv LD_PRELOAD "./mymalloc.so"; ./intr; unsetenv LD_PRELOAD)
malloc(32) = 0x2157010
free(0x2157010)
```

Обратите внимание, что с помощью LD\_PRELOAD можно подменить вызовы библиотечных функций из любых программ!

<sup>3</sup> Если вы не знаете, какая командная оболочка используется у вас, введите команду `printenv SHELL`.

```

linux> LD_PRELOAD= "/mymalloc.so" /usr/bin/uptime
malloc(568) = 0x21bb010
free(0x21bb010)
malloc(15) = 0x21bb010
malloc(568) = 0x21bb030
malloc(2255) = 0x21bb270
free(0x21bb030)
malloc(20) = 0x21bb030
malloc(20) = 0x21bb050
malloc(20) = 0x21bb070
malloc(20) = 0x21bb090
malloc(20) = 0x21bb0b0
malloc(384) = 0x21bb0d0
20:47:36 up 85 days, 6:04, 1 user, load average: 0.10, 0.04, 0.05

```

## 7.14. Инструменты управления объектными файлами

В системе Linux имеется множество утилит, которые помогут вам исследовать объектные файлы и управлять ими. В частности, особенно полезным будет пакет GNU *binutils*, имеющийся в любой системе Linux.

- AR – создает статические библиотеки, вставляет, удаляет, перечисляет и извлекает элементы.
- STRINGS – выводит все печатаемые строки, имеющиеся в объектном файле.
- STRIP – удаляет таблицу имен из объектного файла.
- NM – перечисляет имена, определенные в таблице имен объектного файла.
- SIZE – перечисляет имена и размеры секций в объектном файле.
- READ ELF – показывает полную структуру объектного файла, включая информацию в заголовке ELF. Обладает возможностями утилит SIZE и NM.
- OBJDUMP – основа всех инструментов для работы с двоичными файлами. Может отобразить всю информацию об объектном файле. Чаще всего используется для дизассемблирования двоичного кода в секции `.text`.

Системы Linux также предоставляют программу LDD для управления разделяемыми библиотеками.

- LDD – перечисляет разделяемые библиотеки, необходимые выполняемому файлу во время выполнения.

## 7.15. Итоги

Компоновку можно выполнить во время компиляции с помощью статического компоновщика и во время загрузки с помощью динамического компоновщика. Компоновщики работают с двоичными файлами, которые называют объектными. Объектные файлы бывают трех видов: перемещаемые, выполняемые и разделяемые. Перемещаемые объектные файлы собираются с помощью статических компоновщиков в выполняемый объектный файл, который может быть загружен в память и запущен на выполнение. Разделяемые объектные файлы (разделяемые библиотеки) связываются и загружаются с помощью динамических загрузчиков во время выполнения – либо неявно, когда вызывающая программа загружается и начинает выполняться, либо явно, когда программа вызывает функции из библиотеки `dlopen`.

Компоновщики решают две основные задачи – разрешают ссылки, связывая каждое глобальное имя в объектном файле с уникальным определением, и выполняют переме-

шение, определяя для каждого имени окончательный адрес в памяти и соответственно модифицируя ссылки на них.

Статические компоновщики вызываются с помощью диспетчера компилятора, такого как GCC. Они связывают несколько перемещаемых объектных файлов в один выполняемый объектный файл. В различных объектных файлах может определяться одно и то же глобальное имя, и правила, используемые компоновщиками для разрешения ссылок на различные определения, могут привести к коварным ошибкам в пользовательских программах.

Несколько объектных файлов могут быть связаны в одну статическую библиотеку. Библиотеки используются редакторами связей для разрешения ссылок на имена в других объектных модулях. Последовательный просмотр слева направо, который используют многие редакторы связей для разрешения ссылок на имя, является еще одним источником труднообъяснимых ошибок.

Загрузчики отображают содержимое выполняемых файлов в память и запускают их. Компоновщики могут также создавать частично скомпонованные выполняемые объектные файлы с неразрешенными ссылками на процедуры и данные, определяемые в разделяемой библиотеке. Во время загрузки загрузчик отображает частично скомпонованный выполняемый файл в память и затем передает управление динамическому компоновщику, который завершает задачу связывания, загружая разделяемую библиотеку и перемещая ссылки в программе.

Разделяемые библиотеки, скомпилированные в позиционно-независимый код, могут загружаться в любое место адресного пространства и совместно использоваться несколькими процессами. Кроме того, приложения могут использовать динамический компоновщик во время выполнения, чтобы загрузить и связать функции и данные в разделяемых библиотеках.

## Библиографические заметки

Вопросы компоновки плохо освещены в компьютерной литературе. Компоновка охватывает такие области знаний, как компиляция, компьютерная архитектура и операционные системы, поэтому для ее понимания требуется знать особенности генерирования объектного кода, программирования на машинном языке, реализации программ и виртуальной памяти. Компоновка не вписывается ни в одну из обычных специальностей в области компьютерных систем, и потому типичные вопросы, возникающие в этой сфере, недостаточно хорошо освещены в литературе. Впрочем, монография Левина (Levine) [69] предоставляет неплохое общее введение в предмет. Исходные спецификации ELF и DWARF для архитектуры IA32 (описывающие содержимое секций `.debug` и `.line`) описаны в [54]. Расширения формата ELF для архитектуры x86-64 описаны в [36]. Описание прикладного двоичного интерфейса (Application Binary Interface, ABI) для архитектуры x86-64 с соглашениями по компиляции, компоновке и выполнению программ для архитектуры x86-64 и правилами перемещения и оформления позиционно-независимого кода можно найти в [77].

## Домашние задания

### Упражнение 7.6 ♦

Исследуйте модуль `m.o` в листинге 7.3 и следующую версию функции `swap`, которая запоминает, сколько раз она вызывалась:

```
1 extern int buf[];
2
3 int *bufp0 = &buf[0];
4 static int *bufp1;
```

```
5
6 static void incr()
7 {
8     static int count=0;
9
10    count++;
11 }
12
13 void swap()
14 {
15     int temp;
16
17     incr();
18     bufp1 = &buf[1];
19     temp = *bufp0;
20     *bufp0 = *bufp1;
21     *bufp1 = temp;
22 }
```

Для каждого имени, которое определено и на которое есть ссылка в swap.o, укажите, будет ли создана соответствующая запись в таблице имен в секции .symtab в модуле swap.o. Если да, то укажите модуль, в котором определено это имя (swap.o или m.o), тип имени (локальное, глобальное или внешнее), а также секцию (.text, .data или .bss), в которой это имя находится.

Имя	Имеется запись в .symtab в модуле swap.o?	Тип имени	Модуль, где определено	Секция
buf	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
bufp0	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
bufp1	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
swap	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
temp	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

Упражнение 7.7 ♦

Не изменяя никаких имен переменных, модифицируйте bar5.c в разделе 7.6.1 так, чтобы foo5.c выводила корректные значения x и y (т. е. шестнадцатеричные представления целых чисел 15213 и 15212).

Упражнение 7.8 ♦

Пусть REF(x.i) → DEF(x.k) обозначает тот факт, что компоновщик связывает некоторую ссылку на имя x в модуле i с определением x в модуле k. Используйте эту нотацию для каждого примера, чтобы указать, как компоновщик разрешит ссылки на неоднократно определенное имя в каждом модуле. Если на этапе компоновки возникнет ошибка (правило 1), впишите «ERROR». Если компоновщик произвольно выберет одно из возможных определений (правило 3), впишите «UNKNOWN».

```
1.
    /* Модуль 1 */
    int main()
    {
    }

    /* Модуль 2 */
    static int main=1;
    int p2()
    {
    }
```

(a) REF(main.1) → DEF( \_\_\_\_\_ . \_\_\_\_\_ )  
 (b) REF(main.2) → DEF( \_\_\_\_\_ . \_\_\_\_\_ )

2.

/* Модуль 1 */	/* Модуль 2 */
int x;	double x;
int main()	int p2()
{	{
}	}

(a) REF(x.1) → DEF( \_\_\_\_\_ . \_\_\_\_\_ )  
 (b) REF(x.2) → DEF( \_\_\_\_\_ . \_\_\_\_\_ )

3.

/* Модуль 1 */	/* Модуль 2 */
int x=1;	double x=1.0;
int main()	int p2()
{	{
}	}

(a) REF(x.1) → DEF( \_\_\_\_\_ . \_\_\_\_\_ )  
 (b) REF(x.2) → DEF( \_\_\_\_\_ . \_\_\_\_\_ )

### Упражнение 7.9 ♦

Взгляните на следующую программу, состоящую из двух объектных модулей:

```

1 /* foo6.c */
2 void p2(void);
3
4 int main()
5 {
6     p2();
7     return 0;
8 }

1 /* bar6.c */
2 #include <stdio.h>
3
4 char main;
5
6 void p2()
7 {
8     printf("0x%x\n", main);
9 }
```

Если эта программа будет скомпилирована и запущена в системе Linux с архитектурой x86-64, она выведет строку "0x48\n" и нормально завершит работу, даже при том что функция p2 нигде не инициализирует переменную main. Сможете ли вы объяснить такое поведение?

### Упражнение 7.10 ♦♦

Пусть а и b обозначают объектные модули или статические библиотеки в текущем каталоге, и пусть  $a \rightarrow b$  обозначает зависимость а от b, в том смысле, что b определяет имя, на которое имеется ссылка в а. Для каждого из следующих сценариев приведите минимальную команду (с наименьшим количеством объектных файлов и библиотек), которая позволит статическому компоновщику разрешить все ссылки на имена:

1.  $p.o \rightarrow libx.a \rightarrow p.o$
2.  $p.o \rightarrow libx.a \rightarrow liby.a$  и  $liby.a \rightarrow libx.a$
3.  $p.o \rightarrow libx.a \rightarrow liby.a \rightarrow libz.a$  и  $liby.a \rightarrow libx.a \rightarrow libz.a$

### Упражнение 7.11 ♦♦

Из заголовка сегмента в листинге 7.10 видно, что этот сегмент данных занимает в памяти 0x230 байт. Однако только первые 0x228 байт принадлежат секциям выполняемого файла. Чем вызвано это несоответствие?

### Упражнение 7.12 ♦♦

Вызов функции `swap` в объектном файле `m.o` выглядит так (упражнение 7.6):

```
9: e8 00 00 00 00      callq e <main+0xe>      swap()
```

Для этого вызова имеется следующая запись перемещения:

```
r.offset = 0xa
r.symbol = swap
r.type   = R_X86_64_PC32
r.addend = -4
```

1. Если предположить, что компоновщик перемещает `.text` в модуле `m.o` по адресу 0x4004e0 и `swap` – по адресу 0x4004f8, какое значение получит ссылка на `swap` в инструкции `callq`?
2. Если предположить, что компоновщик перемещает `.text` в модуле `m.o` по адресу 0x4004d0 и `swap` – по адресу 0x400500, какое значение получит ссылка на `swap` в инструкции `callq`?

### Упражнение 7.13 ♦♦

Это упражнение поможет вам приобрести дополнительный опыт использования различных средств управления объектными файлами.

1. Сколько объектных файлов содержится в библиотеках `libc.a` и `libm.a` в вашей системе?
2. Будет ли отличаться машинный код, скомпилированный командой `gcc -Og`, от машинного кода, скомпилированного командой `gcc -Og -g`?
3. Какие разделяемые библиотеки использует драйвер GCC в вашей системе?

## Решения упражнений

### Решение упражнения 7.1

Цель этого упражнения – помочь вам понять связь между именами внутри компоновщика и идентификаторами переменных и функций в программном коде на C. Обратите внимание, что для локальной переменной `temp` в исходном коде на C компилятор не создает запись в таблице имен.

Имя	Имеется запись в <code>.symtab</code> ?	Тип имени	Модуль, где определено	Секция
<code>buf</code>	Да	внешнее	<code>m.o</code>	<code>.data</code>
<code>bufp0</code>	Да	глобальное	<code>swap.o</code>	<code>.data</code>
<code>bufp1</code>	Да	глобальное	<code>swap.o</code>	COMMON
<code>swap</code>	Да	глобальное	<code>swap.o</code>	<code>.text</code>
<code>temp</code>	Да	–	–	–

## Решение упражнения 7.2

Цель этого упражнения – помочь вам проверить свое понимание правил, которыми пользуется компоновщик в Unix при разрешении ссылок на глобальные имена, определяемые в более чем одном модуле. Знание и понимание этих правил поможет вам избежать некоторых неприятных ошибок при программировании.

1. Компоновщик выберет строго определенное имя из модуля 1, оставив в стороне слабо определенное имя в модуле 2 (правило 2):
  - (a)  $\text{REF}(\text{main.1}) \rightarrow \text{DEF}(\text{main.1})$
  - (b)  $\text{REF}(\text{main.2}) \rightarrow \text{DEF}(\text{main.1})$
2. Здесь следует вписать слово «ERROR», потому что оба модуля содержат строгие определения одного и того же имени `main` (правило 1).
3. Компоновщик выберет строго определенное имя из модуля 2, оставив в стороне слабо определенное имя в модуле 1 (правило 2):
  - (a)  $\text{REF}(x.1) \rightarrow \text{DEF}(x.2)$
  - (b)  $\text{REF}(x.2) \rightarrow \text{DEF}(x.2)$

## Решение упражнения 7.3

Перечисление статических библиотек в командной строке в неправильном порядке является распространенным источником ошибок времени компоновки, которые ставят в тупик многих программистов. Однако если разобраться с тем, как компоновщик использует статические библиотеки для разрешения ссылок, все станет на свои места. Это небольшое упражнение поможет вам проверить ваше понимание данной идеи:

1. `linux> gcc p.o libx.a`
2. `linux> gcc p.o libx.a liby.a`
3. `linux> gcc p.o libx.a liby.a libx.a`

## Решение упражнения 7.4

В этом упражнении предлагалось проанализировать листинг 7.9 (а). Его цель – дать вам некоторую практику чтения листингов дизассемблера и помочь сложить представление об относительной адресации.

1. Ссылка в строке 5 окажется по адресу `0x4004df`.
2. Ссылка в строке 5 получит шестнадцатеричное значение `0x5`. Напомню, что в дизассемблированном листинге значения ссылок выводятся с использованием обратного (little-endian) порядка следования байтов.

## Решение упражнения 7.5

Это упражнение поможет вам проверить свое понимание, как компоновщик производит перемещение ссылок при использовании относительной адресации. Вам было дано:

```
ADDR(s) = ADDR(.text) = 0x4004d0
```

и

```
ADDR(r.symbol) = ADDR(swap) = 0x4004e8
```

Согласно алгоритму в листинге 7.7, компоновщик сначала вычислит адрес ссылки:

```
refaddr = ADDR(s) + r.offset
         = 0x4004d0 + 0xa
         = 0x4004da
```

а затем обновит ссылку:

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
          = (unsigned) (0x4004e8      + (-4)      - 0x4004da)
          = (unsigned) (0xa)
```

В результате относительная ссылка на swap получит значение 0xa:

```
4004d9: e8 0a 00 00 00      callq 4004e8 <swap>
```



## Управление исключениями

- 8.1. Исключения.
- 8.2. Процессы.
- 8.3. Системные вызовы и обработка ошибок.
- 8.4. Управление процессами.
- 8.5. Сигналы.
- 8.6. Нелокальные переходы.
- 8.7. Инструменты управления процессами.
- 8.8. Итоги.

Библиографические заметки.

Домашние задания.

Решения упражнений.

С момента включения компьютера и до момента его выключения счетчик инструкций процессора будет принимать последовательность значений

$$a_0, a_1, \dots, a_{n-1},$$

где каждое значение  $a_k$  – это адрес, соответствующий некоторой инструкции  $I_k$ . Каждый переход из  $a_k$  в  $a_{k+1}$  называется *передачей управления*. Последовательность таких передач управления называется *поток управления* процессором.

Простейший поток управления представляет собой «линейную» последовательность, где обе инструкции,  $I_k$  и  $I_{k+1}$ , находятся в памяти друг за другом. Как правило, неожиданные изменения направления течения этого линейного потока, когда  $I_{k+1}$  находится не рядом с  $I_k$ , вызываются уже знакомыми вам инструкциями, такими как инструкции перехода, вызова подпрограмм и возврата. Подобные инструкции совершенно необходимы, потому что позволяют программам реагировать на изменения в их внутреннем состоянии, хранящемся в программных переменных.

Но системы должны также реагировать на изменения в состоянии системы, которые определяются не только внутренними программными переменными и не обязательно связаны с выполнением самой программы. Например, аппаратный таймер срабатывает через регулярные интервалы времени, и его сигналы должны обрабатываться. Пакеты поступают в сетевой адаптер и должны сохраняться в памяти. Программа запрашивает данные с диска и приостанавливается до тех пор, пока не получит уведомление о готовности данных. Родительский процесс запускает дочерние процессы и должен быть уведомлен, когда его дочерние процессы завершат выполнение.

Современные системы реагируют на такие ситуации, выполняя скачкообразные переходы и тем самым меняя направление потока управления. Такие скачкообразные переходы называются *передачей управления по исключению* (Exceptional Control Flow, ECF). Передача управления по исключению (ECF) может происходить на любом уровне компьютерной системы. Например, на аппаратном уровне события, зафиксированные аппаратными средствами, вызывают передачу управления обработчикам исключительных ситуаций. На уровне операционной системы ядро передает управление из одного пользовательского процесса в другой путем переключения контекста. На прикладном уровне один процесс может отправить *сигнал* другому процессу, который передаст управление своему обработчику сигнала. Отдельная программа может реагировать на ошибки в обход механизмов управления стеком и выполнять нелокальные переходы по произвольным адресам в другие функции.

Можно привести множество веских доводов, почему программист должен знать и понимать, как происходит передача управления по исключению:

- *понимание сути передачи управления по исключению поможет разобраться с важными понятиями организации систем.* Передача управления по исключению – это базовый механизм, используемый операционными системами для реализации ввода/вывода, процессов и виртуальной памяти. Прежде чем приступить к изучению этих важных понятий, следует четко знать и понимать суть передачи управления по исключению;
- *понимание сути передачи управления по исключению поможет разобраться в особенностях взаимодействий приложений с операционной системой.* Приложения посылают запросы службам операционной системы, используя разнообразность передачи управления по исключению, известную как *системное прерывание* (trap) или *системный вызов*. Например, запись данных на диск, чтение данных из сети, запуск нового процесса и завершение текущего – все эти операции выполняются путем обращения к системным вызовам. Понимание базовых механизмов, лежащих в основе системных вызовов, поможет вам разобраться, как такие службы могут использоваться приложениями;
- *понимание сути передачи управления по исключению поможет вам писать новые интересные прикладные программы.* Операционная система дает возможность прикладным программам использовать эффективные механизмы передачи управления по исключению для создания новых процессов, ожидания завершения процессов, уведомления других процессов об исключительных ситуациях, а также для обнаружения этих событий и реакции на них. Разобравшись в механизмах передачи управления по исключению, вы сможете применять их при написании таких интересных программ, как командные оболочки Unix и веб-серверы;
- *понимание сути передачи управления по исключению поможет вам освоить механизмы управления конкурентным выполнением.* Передача управления по исключению – это базовый механизм реализации конкурентного выполнения в компьютерных системах. Вот некоторые примеры применения конкуренции: обработчик исключений, прерывающий выполнение прикладной программы; процессы и потоки, выполнение которых перекрывается во времени; обработчик сигнала, прерывающий выполнение прикладной программы. Понимание передачи управления по исключению – это первый шаг к пониманию конкурентного выполнения. Мы еще вернемся к нему в главе 12;
- *понимание сути передачи управления по исключению поможет вам разобраться, как работают программные исключения.* Такие языки, как C++ и Java, обеспечивают доступ к механизмам программных исключений через инструкции try,

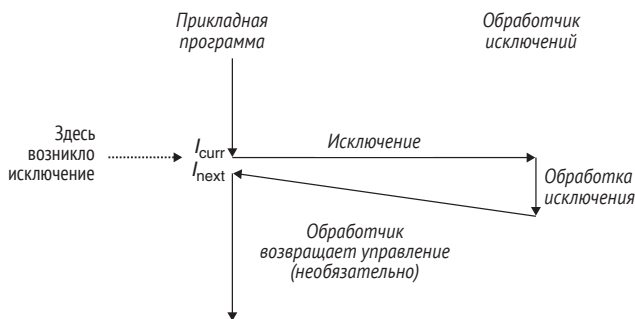
catch и throw. Программные исключения позволяют программам выполнять *нелокальные* переходы (т. е. передавать управление в обход обычных механизмов управления стеком), чтобы среагировать на возникшую ошибку. Нелокальные переходы – это форма передачи управления по исключению прикладного уровня, и в языке С они реализуются функциями `setjmp` и `longjmp`. Изучение этих низкоуровневых функций поможет вам понять, как можно реализовать высокоуровневые программные исключения.

До сих пор мы в основном интересовались взаимодействиями между приложениями и аппаратными средствами. В этой главе мы совершим крутой поворот и приступим к изучению вопросов взаимодействий приложений с операционной системой. Интересно, что все эти взаимодействия тесно связаны с передачей управления по исключению. Мы опишем различные формы передачи управления по исключению, существующие на разных уровнях компьютерной системы, и начнем с исключений, лежащих на стыке аппаратных средств и операционной системы. Мы также обсудим системные вызовы, которые являются исключениями, предоставляющие приложениям точки входа в операционную систему. Затем перейдем на более высокий уровень абстракции и опишем процессы и сигналы, лежащие на стыке прикладных программ и операционной системы. А в заключение обсудим нелокальные переходы – одну из форм передачи управления по исключению прикладного уровня.

## 8.1. Исключения

Исключения – это одна из форм передачи управления по исключению. Их особенность состоит в том, что частично они реализованы аппаратно, а частично – средствами операционной системы. Поскольку реализация связана с аппаратными средствами, ее тонкости могут отличаться в разных системах, но основная идея остается неизменной для всех систем. В этом разделе мы постараемся составить общее представление об исключениях и их обработке, а также сорвать покров таинственности и помочь разобраться в некоторых трудно объяснимых аспектах современных компьютерных систем.

*Исключение* – это скачкообразный переход в потоке управления в ответ на некоторые изменения в состоянии процессора. Основная идея исключения показана на рис. 8.1.



**Рис. 8.1. Анатомия исключения.** Изменение состояния процессора (событие) вызывает передачу управления (исключение) из прикладной программы обработчику исключения.

Завершив работу, обработчик либо возвращает управление прерванной программе, либо останавливает ее

На этом рисунке процессор выполняет некоторую инструкцию  $I_{curr}$ , когда происходит изменение его *состояния*. Это состояние кодируется набором различных битов и сигналов внутри процессора. Изменение состояния называется *событием*.

### Аппаратные и программные исключения

Программисты на C++ и Java должны иметь в виду, что термин «исключение» используется также для обозначения ECF-механизмов, поддерживаемых в языках C++ и Java с помощью операторов `catch`, `throw` и `try`. Если быть предельно точными, то следовало бы провести четкую грань между аппаратными и программными исключениями, но обычно это излишне, потому что смысл и так ясен из контекста.

Событие может быть непосредственно связано с текущей выполняемой инструкцией. Например, может произойти сбой из-за обращения к несуществующей странице виртуальной памяти в результате арифметического переполнения или при попытке выполнить деление на ноль. С другой стороны, событие может быть не связано с текущей инструкцией. Например, оно может быть вызвано срабатыванием системного таймера или завершением обработки запроса на ввод/вывод.

В любом случае, когда процессор обнаруживает, что произошло событие, он использует таблицу переходов, которая называется *таблицей исключений*, или *таблицей векторов прерываний*, чтобы выполнить косвенный вызов (исключение) некоторой подпрограммы в операционной системе (*обработчика исключений*), специально предназначенной для обработки этого конкретного события. Когда обработчик исключений завершит работу, то в зависимости от вида события, вызвавшего исключение, он выполняет одно из трех действий:

1. Возвращает управление текущей инструкции  $I_{curr}$ , выполнявшейся в тот момент, когда возникло событие.
2. Возвращает управление инструкции  $I_{next}$ , которая выполнялась бы следующей, если бы не возникло исключение.
3. Аварийно завершает выполнение прерванной программы.
4. Более подробно об этих действиях мы поговорим в разделе 8.1.2.

#### 8.1.1. Обработка исключений

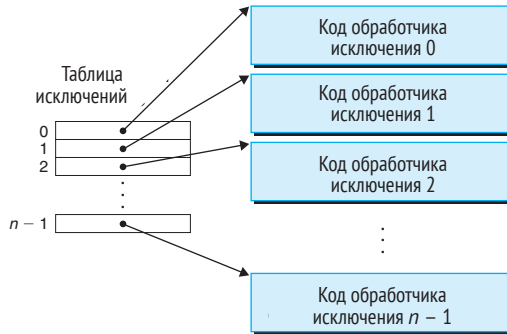
Разобраться в работе исключений не всегда просто, потому что их обработка предполагает тесное взаимодействие аппаратных и программных средств. Нетрудно запутаться в том, какой компонент какую задачу выполняет. Рассмотрим более подробно, как распределена работа между аппаратными и программными средствами.

Каждому типу возможного исключения присваивается уникальный неотрицательный целочисленный *номер исключения*. Некоторые из этих номеров присваиваются разработчиками процессора. Другие – разработчиками *ядра* операционной системы (части операционной системы, постоянно находящейся в памяти). Примерами исключений первого вида являются деление на ноль, ошибка обращения к несуществующей странице, нарушение прав доступа к памяти, точки останова и арифметические переполнения. Примерами исключений второго вида могут служить системные вызовы и сигналы от внешних устройств ввода/вывода.

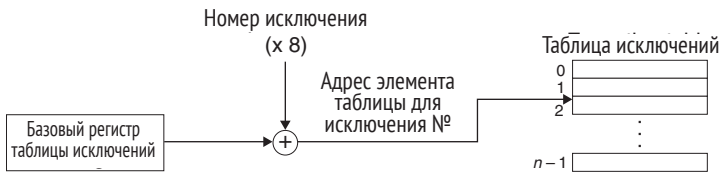
Во время загрузки системы (когда компьютер перезагружается или включается) операционная система создает и инициализирует таблицу переходов, называемую *таблицей исключений*, так чтобы  $k$ -й элемент содержал адрес обработчика для исключения с номером  $k$ . Формат таблицы исключений показан на рис. 8.2.

Если во время выполнения (время, когда система выполняет некоторую программу) процессор обнаружит некоторое событие, то он определяет соответствующий номер  $k$  исключения и затем, используя  $k$ -й элемент таблицы исключений, выполняет косвенный вызов соответствующего обработчика исключения. На рис. 8.3 показано,

как процессор использует таблицу исключений, чтобы получить адрес обработчика. Номер исключения – это индекс в таблице исключений, начальный адрес которой хранится в специальном регистре процессора, который называют *базовым регистром таблицы исключений*.



**Рис. 8.2.** Таблица исключений. Таблица исключений – это таблица переходов, в которой запись  $k$  содержит адрес обработчика исключения  $k$



**Рис. 8.3.** Вычисление адреса обработчика исключений. Номер исключения служит индексом в таблице исключений

Передача управления обработчику исключения похожа на вызов процедуры, но с некоторыми важными отличиями:

- так же как в случае вызова обычной подпрограммы, перед передачей управления обработчику процессор запоминает на стеке адрес возврата. Однако, в зависимости от класса исключения, адресом возврата будет адрес текущей инструкции (которая выполнялась в момент, когда произошло событие) или адрес следующей инструкции (которая выполнялась бы после текущей инструкции, если бы событие не произошло);
- также процессор запоминает на стеке некоторые атрибуты своего состояния, которые понадобятся для возобновления выполнения прерванной программы после того, как обработчик вернет управление. Например, система x86-64 запоминает на стеке содержимое регистра EFLAGS, хранящего, кроме всего прочего, текущее состояние флагов;
- если управление передается из прикладной программы в ядро, то все эти элементы запоминаются в стеке ядра, а не прикладной программы;
- обработчики исключений выполняются в *привилегированном режиме* (см. раздел 8.2.4) и, соответственно, имеют полный доступ ко всем системным ресурсам.

После того как оборудование сгенерирует исключение, всю остальную работу выполняет программное обеспечение – обработчик исключительных ситуаций. Когда обработчик обработает событие, он может вернуть управление прерванной программе, выполнив специальную инструкцию «возврата из прерывания», которая снимет со сте-

ка сохраненное состояние и запишет его в управляющие регистры и регистры данных процессора и, если исключение прервало работу прикладной программы, восстановит состояние выполнения в *пространстве пользователя* (раздел 8.2.4). После этого управление возвращается прерванной программе.

### 8.1.2. Классы исключений

Исключения можно разделить на четыре класса: *аппаратные прерывания, системные прерывания, сбой и аварийное завершение*. В табл. 8.1 перечислены обобщенные признаки этих классов.

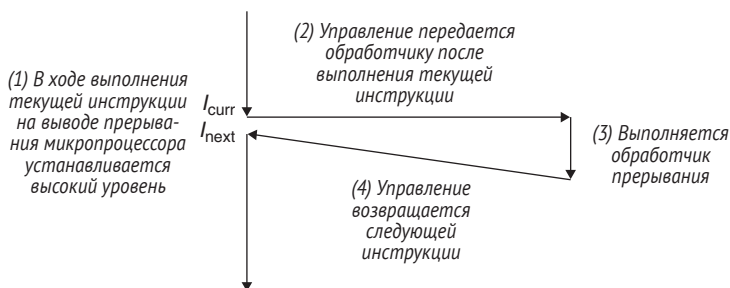
**Таблица 8.1.** Классы исключений. Асинхронные исключения возникают в результате событий в устройствах ввода/вывода – внешних по отношению к процессору. Синхронные исключения возникают как прямой результат выполнения инструкции

Класс	Причина	Синхронное/ Асинхронное	Поведение при возврате
Аппаратное прерывание	Сигнал от устройства ввода/вывода	Асинхронное	Всегда возвращает управление следующей инструкции
Системное прерывание	Предусмотренное исключение	Синхронное	Всегда возвращает управление следующей инструкции
Сбой	Потенциально исправимая ошибка	Синхронное	Управление может вернуться текущей инструкции
Аварийное завершение	Фатальная ошибка	Синхронное	Никогда не возвращает управление

### Аппаратные прерывания

*Аппаратные прерывания* (interrupt) возникают *асинхронно* при получении сигналов от устройств ввода/вывода, внешних по отношению к процессору. Они считаются асинхронными в том смысле, что не вызваны выполнением какой-то конкретной инструкции. Обработчики исключений, относящихся к классу аппаратных прерываний, часто называются *обработчиками прерываний*.

На рис. 8.4 показана обобщенная схема обработки прерывания. Устройства ввода/вывода, такие как сетевые адаптеры, контроллеры дисков и микросхемы таймеров, инициируют прерывания, посылая сигнал на определенный вывод микропроцессора и выставляя номер исключения на системную шину, идентифицирующую устройство, вызвавшее прерывание.



**Рис. 8.4.** Обработка аппаратных прерываний. Обработчик возвращает управление следующей инструкции в потоке управления прикладной программы

Если, завершив выполнение текущей инструкции, процессор обнаружит высокий уровень на выводе прерывания, то тогда он получит с системной шины номер исклю-

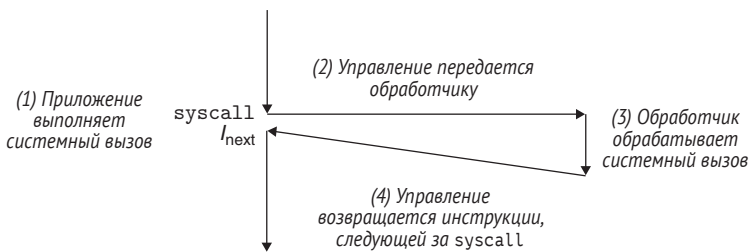
чения и вызовет соответствующий обработчик прерывания. Когда обработчик выполнит инструкцию возврата, управление будет передано следующей инструкции (т. е. инструкции, которая в потоке управления была бы выполнена после текущей, если бы прерывание не возникло). В результате программа продолжает выполняться, как если бы ничего не случилось.

Исключения остальных классов (системные прерывания, сбои и аварийное завершение) возникают синхронно, в результате выполнения текущей инструкции. Такие инструкции мы будем называть *сбойными инструкциями*.

## Системные прерывания

*Системные прерывания* – это *преднамеренные* исключения, возникающие в результате выполнения инструкции. Подобно обработчикам аппаратных прерываний, обработчики системных прерываний возвращают управление следующей инструкции. Наиболее важной задачей системных прерываний является поддержка процедурного интерфейса между прикладными программами и ядром, так называемого интерфейса *системных вызовов*.

Прикладные программы часто запрашивают услуги у ядра, такие как чтение файла (read), запуск нового процесса (fork), загрузка новой программы (execve) или завершение текущего процесса (exit). Для организации управляемого доступа к таким услугам процессоры предоставляют специальную инструкцию `syscall`, которая может выполняться прикладными программами, если потребуется послать запрос службе с номером  $n$ . В результате выполнения инструкции `syscall` возникает системное прерывание и вызывается обработчик исключений, который анализирует аргумент и вызывает соответствующую подпрограмму ядра. На рис. 8.5 показана обобщенная схема обработки системного вызова.



**Рис. 8.5.** Обработка системных вызовов. Обработчик системного вызова возвращает управление следующей инструкции в потоке управления прикладной программы

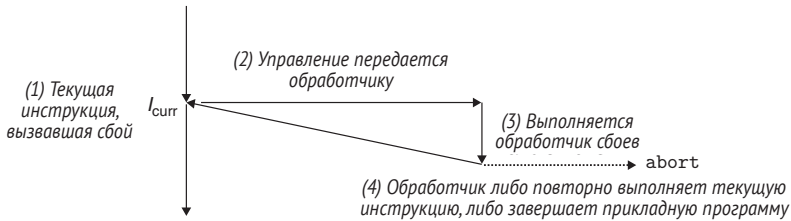
С точки зрения программиста, системный вызов похож на вызов обычной функции. Однако в действительности реализации системных вызовов и вызовов обычных функций сильно отличаются. Обычные функции выполняются в *пользовательском режиме*, ограничивающем набор команд, допустимых для выполнения, и эти функции используют тот же стек, что и вызывающая функция. Системный вызов выполняется в *привилегированном режиме*, допускающем выполнение любых команд, и использует стек ядра. Более подробно пользовательский и привилегированный режимы обсуждаются в разделе 8.2.4.

## Сбои

*Сбои* (fault) являются следствием ошибочных состояний, которые, возможно, можно исправить. Когда возникает сбой, процессор передает управление обработчику ошибок. Если обработчик в состоянии исправить ошибку, то возвращает управление инструк-



ции, вызвавшей сбой, выполняя ее повторно. В противном случае управление передается ядру – подпрограмме `abort`, которая завершает работу прикладной программы, вызвавшей сбой. На рис. 8.6 представлена обобщенная схема обработки сбоев.

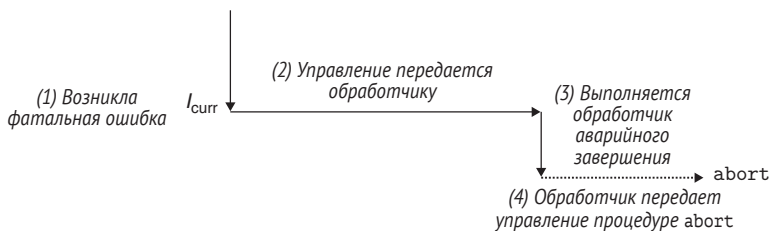


**Рис. 8.6.** Обработка сбоев. В зависимости от возможности устранить ошибку обработчик либо повторно выполняет инструкцию, вызвавшую сбой, либо завершает программу

Классический пример сбоя – исключение при обращении к несуществующей странице виртуальной памяти, когда инструкция ссылается на виртуальный адрес, соответствующий физической странице, которая отсутствует в данный момент в памяти и которую следует загрузить с диска. Как будет показано в главе 9, страница – это непрерывный блок (обычно 4 Кбайт) виртуальной памяти. Обработчик ошибки обращения к страницам загружает нужную страницу с диска и затем возвращает управление инструкции, вызвавшей данный сбой. Когда эта инструкция выполнится повторно, нужная физическая страница уже будет находиться в оперативной памяти, и эта инструкция сможет выполниться безаварийно.

### Аварийное завершение

Аварийное завершение (`abort`) является результатом фатальной ошибки. Обычно это аппаратные ошибки, такие как ошибки четности, возникающие при потере значений отдельных битов в DRAM или SRAM. Обработчик аварийного завершения никогда не возвращает управление прикладной программе. Как показано на рис. 8.7, обработчик передает управление подпрограмме `abort`, которая завершает работу прикладной программы.



**Рис. 8.7.** Обработка аварийного завершения. Обработчик передает управление специальной подпрограмме ядра, которая завершает прикладную программу

### 8.1.3. Исключения в системах Linux/x86-64

Чтобы сделать наши рассуждения более конкретными, рассмотрим некоторые исключения, определяемые системами x86-64. Такие системы могут иметь до 256 различных видов исключений [50]. Номера в интервале от 0 до 31 соответствуют исключениям, определяемым архитектурой Intel, и потому идентичны для любых систем x86-64. Номера в интервале от 32 до 255 соответствуют аппаратным и системным прерываниям, которые определяются операционной системой. В табл. 8.2 представлены некоторые примеры.



Таблица 8.2. Примеры исключений в системах x86-64

Номер исключения	Описание	Класс исключения
0	Ошибка деления	Сбой
13	Общее нарушение защиты	Сбой
14	Сбой страницы	Сбой
18	Сбой аппаратного контроля	Аварийное завершение
32–255	Исключения, определяемые операционной системой	Аппаратное или системное прерывание

### О терминологии

Терминология в области классов исключений до сих пор не устоялась и в разных системах может отличаться. В спецификациях архитектурного набора инструкций (ISA) процессоров часто различаются асинхронные «прерывания» и синхронные «исключения», и пока нет никакого обобщающего термина, обозначающего эти очень схожие понятия. Чтобы постоянно не прибегать к использованию словосочетаний «исключения и прерывания» и «исключения или прерывания», мы будем использовать «исключение» как обобщающий термин и при этом различать асинхронные исключения (аппаратные прерывания) и синхронные исключения (системные прерывания, сбои и аварийные завершения), но только там, где это будет иметь смысл. Как уже отмечалось, основные положения остаются неизменными для любой системы, но следует иметь в виду, что некоторые разработчики в своих руководствах используют слово «исключение», обозначая им изменения в потоке управления, вызванные синхронными событиями.

## Сбои и аварийные завершения в Linux/x86-64

### Ошибка деления (исключение 0)

Возникает, когда приложение делает попытку выполнить деление на ноль или когда результат деления слишком велик для операнда-приемника. Linux не пытается исправлять ошибки деления, предпочитая просто прервать программу. Командные оболочки Linux обычно сообщают об ошибках деления как «Ошибка операции с плавающей точкой».

### Общее нарушение защиты (исключение 13)

Печально известное общее нарушение защиты возникает по разным причинам, но чаще всего потому, что программа пытается обратиться к несуществующей области виртуальной памяти или записать данные в сегмент кода, доступный только для чтения. Linux не пытается исправлять такие сбои. Командные оболочки Linux обычно сообщают об ошибках общего нарушения защиты как «Сбой сегментации».

### Сбой страницы (исключение 14)

Это пример исключения, после обработки которого инструкция, вызвавшая ошибку, будет выполнена повторно. Обработчик отобразит недостающую страницу виртуальной памяти, находящуюся на диске, в физическую память и передаст управление инструкции, вызвавшей сбой, для повторного выполнения. В главе 9 мы подробно рассмотрим, как работает этот механизм.

### Ошибка аппаратного контроля (исключение 18)

Возникает в результате фатальной аппаратной ошибки и обнаруживается в процессе выполнения инструкции, вызвавшей ее. Обработчик ошибок аппаратного контроля никогда не возвращает управление прикладной программе.

## Системные вызовы в Linux/x86-64

Linux поддерживает сотни системных вызовов, которые прикладные программы могут использовать, чтобы запросить у ядра такие услуги, как чтение файла, запись в файл или запуск нового процесса. В табл. 8.3 перечислены некоторые наиболее часто используемые системные вызовы Linux. Каждому системному вызову соответствует уникальное целое число, определяющее смещение в таблице переходов в ядре. (Обратите внимание, что это другая таблица, отличная от таблицы исключений.)

Программы на C могут напрямую обратиться к любому системному вызову с помощью функции `syscall`. Однако на практике этот прием используется редко. Стандартная библиотека языка C предоставляет набор более удобных функций-обертки для большинства системных вызовов. Функции-обертки упаковывают аргументы, посылают прерывание ядру с помощью соответствующей инструкции системного вызова, а затем передают результат вызывающей программе. В этой книге мы будем ссылаться на системные вызовы и связанные с ними функции-обертки как на *функции системного уровня*.

**Таблица 8.3.** Примеры наиболее часто используемых системных вызовов в системах Linux x86-64

Номер	Имя	Описание	Номер	Имя	Описание
0	read	Читает из файла	33	pause	Приостановка процесса до получения сигнала
1	write	Записывает в файл	37	alarm	Планирует передачу сигнала тревоги
2	open	Открывает файл	39	getpid	Возвращает числовой идентификатор процесса
3	close	Закрывает файл	57	fork	Создает процесс
4	stat	Возвращает информацию о файле	59	execve	Запускает программу
9	mmap	Отображает страницу памяти в файл	60	_exit	Завершает процесс
12	brk	Изменяет размер «кучи»	61	wait4	Ждет завершения процесса
32	dup2	Копирует дескриптор файла	62	kill	Посылает сигнал процессу

Обращение к системным вызовам в системах x86-64 осуществляется с помощью специальной инструкции `syscall`. Вам наверняка будет интересно узнать, как использовать эту инструкцию для прямого обращения к системным вызовам Linux. Все аргументы всех системных вызовов в Linux передаются через регистры общего назначения, а не через стек. По соглашению в регистре `%rax` передается номер системного вызова, а шесть аргументов – в регистрах `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` и `%r9`. Первый аргумент передается в `%rdi`, второй – в `%rsi` и т. д. При возврате из системного вызова содержимое регистров `%rcx` и `%r11` уничтожается, а `%rax` содержит возвращаемое значение. Отрицательное возвращаемое значение в интервале между `-4095` и `-1` указывает на ошибку, соответствующую отрицательному значению `errno`.

Например, рассмотрим следующую версию программы `hello`, написанную с использованием функции системного уровня `write` (раздел 10.4) вместо `printf`:

```
1 int main()
2 {
3     write(1, "hello, world\n", 13);
4     _exit(0);
5 }
```

Первый аргумент `write` требует осуществить вывод в `stdout`. Вторым аргументом – это последовательность байтов для вывода, а третий – количество байтов.

В листинге 8.1 показана версия `hello` на языке ассемблера, использующая инструкцию `syscall` для непосредственного обращения к системным вызовам `write` и `exit`. Строки 9–13 вызывают функцию `write`. В строке 9 номер системного вызова `write` записывается в регистр `%rax`, а в строках 10–12 задаются аргументы. Затем в строке 13 выполняется инструкция `syscall`, которая производит системный вызов. Точно так же строки 14–16 выполняют обращение к системному вызову `_exit`.

**Листинг 8.1.** Реализация программы `hello` с непосредственным использованием инструкции `syscall`

*code/ecf/hello-asm64.sa*

```
1 .section .data
2 string:
3     .ascii "hello, world\n"
4 string_end:
5     .equ len, string_end - string
6 .section .text
7 .globl main
8 main:
9     Сначала следует вызов write(1, "hello, world\n", 13)
10    movq $1, %rax        write -- это системный вызов с номером 1
11    movq $1, %rdi        arg1: stdout, имеющий дескриптор 1
12    movq $string, %rsi    arg2: строка "hello world"
13    movq $len, %rdx       arg3: длина строки
14    syscall              Выполнить системный вызов
15    Затем следует вызов _exit(0)
16    movq $60, %rax       _exit -- это системный вызов с номером 60
17    movq $0, %rdi        arg1: код завершения 0
18    syscall              Выполнить системный вызов
```

*code/ecf/hello-asm64.sa*

## 8.2. Процессы

Исключения – это один из тех «кирпичиков», которые используются для воплощения и развития одной из самых успешных идей в информатике – концепции *процесса*.

Когда современная вычислительная система запускает программу, она создает иллюзию, что эта программа – единственная выполняющаяся в текущий момент. С точки зрения программы все выглядит так, будто она монопольно распоряжается процессором и памятью, и процессор выполняет инструкции в программе одну за другой, без всяких прерываний. Наконец, создается иллюзия, что кроме программного кода и данных этой программы в памяти системы нет других объектов. Такая иллюзия создается благодаря концепции процесса.

Традиционно процесс определяется как *экземпляр выполняющейся программы*. Каждая программа в системе выполняется в контексте некоторого процесса. Контекст опре-

деляется некоторой структурой и позволяет программе выполняться правильно. Эта структура включает программный код и данные, хранящиеся в памяти, стек программы, содержимое ее регистров общего назначения и счетчика инструкций, переменные окружения и набор дескрипторов открытых файлов.

Каждый раз, когда пользователь запускает программу, вводя в командной строке имя выполняемого объектного файла, командная оболочка создает новый процесс и запускает выполняемый объектный файл в контексте этого нового процесса. Прикладные программы тоже могут создавать новые процессы и в их контекстах запускать свой программный код или другие приложения.

Подробное обсуждение особенностей реализации процессов в операционных системах выходит за рамки нашего обсуждения. Поэтому мы сосредоточим свое внимание на следующих ключевых абстракциях, предлагаемых концепцией процессов:

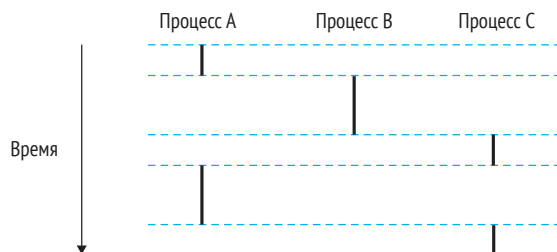
- автономный логический поток управления, создающий иллюзию монопольного использования процессора программой;
- изолированное адресное пространство, создающее иллюзию монопольного использования памяти программой.

Рассмотрим эти абстракции более подробно.

### 8.2.1. Логический поток управления

Процесс создает иллюзию, что каждая программа монопольно использует процессор, даже если в системе одновременно выполняется несколько программ. Если бы мы использовали отладчик в пошаговом режиме, мы могли бы наблюдать последовательность значений счетчика инструкций (PC), соответствующую исключительно инструкциям, содержащимся в выполняемом объектном файле программы или в разделяемых объектах, связанных с нашей программой динамически во время выполнения. Эта последовательность значений счетчика называется *логическим потоком управления*, или просто *логическим потоком*.

Рассмотрим систему, в которой выполняются три процесса, как показано на рис. 8.8. Единый физический поток управления процессора делится на три логических потока, по одному для каждого процесса. Каждая вертикальная линия представляет часть логического потока для процесса. В данном примере в течение некоторого времени выполняется процесс А. Затем процесс В, который выполняется до своего завершения. Потом в течение некоторого времени выполняется процесс С. Далее следует процесс А, который выполняется до своего завершения. Наконец, процесс С получает возможность выполниться до своего завершения.



**Рис. 8.8.** Логические потоки управления. Процессы создают для каждой программы иллюзию, что она монопольно распоряжается процессором. Каждая вертикальная линия представляет логический поток управления для процесса

Обратите внимание, что, как показано на рис. 8.8, процессы сменяют друг друга на процессоре. Каждый процесс выполняет часть своего потока и затем *вытесняется*

(временно приостанавливается), чтобы дать возможность поработать другим процессам. Программе, выполняющейся в контексте одного из этих процессов, кажется, что она монопольно использует процессор. Единственный способ доказать обратное – измерить время, затраченное каждой инструкцией. В этом случае можно заметить, что процессор периодически как бы останавливается между выполнением некоторых инструкций. Однако после каждой такой «остановки» процессор возобновляет выполнение программы без всяких изменений в содержимом памяти программы или регистров.

### 8.2.2. Конкурентные потоки управления

Логические потоки в компьютерных системах принимают множество разных форм. Обработчики исключений, процессы, обработчики сигналов, потоки и процессы Java – все это примеры логических потоков.

Логические потоки, пересекающиеся во времени с другими логическими потоками, называются *конкурентными*, и мы говорим, что эти два потока *выполняются конкурентно*. Выражаясь точнее, потоки *X* и *Y* выполняются конкурентно тогда и только тогда, когда выполнение потока *X* начинается после начала выполнения потока *Y* и до его завершения или когда выполнение потока *Y* начинается после начала выполнения потока *X* и до его окончания. Например, на рис. 8.8 процессы *A* и *B* выполняются конкурентно, так же процессы *A* и *C*. С другой стороны, процессы *B* и *C* не являются конкурентными, потому что последняя инструкция в *B* выполняется перед первой инструкцией в *C*.

В общем случае конкурентное выполнение нескольких потоков называют *конкуренцией*. Идея чередования процессов также известна как *многозадачность*. Каждый период времени, в течение которого процесс выполняет часть своего потока управления, называется *квантом времени*. Поэтому многозадачность иногда называют также *разделением времени*. Например, на рис. 8.8 поток процесса *A* выполняется в течение двух квантов времени.

Примечательно, что идея конкурентных потоков не зависит от количества ядер в процессоре или компьютеров, на которых выполняются потоки. Если два потока перекрываются во времени, то они считаются конкурентными, даже если выполняются на одном и том же процессоре. Однако иногда бывает полезно идентифицировать подмножество конкурентных потоков, известных как *параллельные потоки*. Если два потока выполняются одновременно на разных процессорных ядрах или компьютерах, то их называют *параллельными потоками*, то есть они *выполняются параллельно*.

#### Упражнение 8.1 (решение в конце главы)

Вот три процесса со следующими моментами начала и завершения:

Процесс	Время начала	Время завершения
A	0	2
B	1	4
C	3	5

Для каждой пары процессов укажите, выполняются они конкурентно (Да) или нет (Нет):

Пара процессов	Конкурентны?
AB	_____
AC	_____
BC	_____

### 8.2.3. Изолированное адресное пространство

Помимо иллюзии монопольного распоряжения процессором, концепция процессов создает также иллюзию, что каждая программа монопольно распоряжается *адресным пространством* системы. На машине с  $n$ -разрядными адресами адресное пространство насчитывает  $2^n$  возможных адресов,  $0, 1, \dots, 2^n - 1$ . Процесс предоставляет каждой программе свое собственное *изолированное адресное пространство*. Это пространство называется изолированным в том смысле, что ни к какому байту в этом конкретном пространстве невозможно обратиться (прочитать или записать) из другого процесса.

Содержимое памяти, связанной с каждым изолированным адресным пространством, будет отличаться, но в общем случае все адресные пространства имеют одинаковую структуру. Например, на рис. 8.9 представлена структура организации адресного пространства для процессов в Linux.

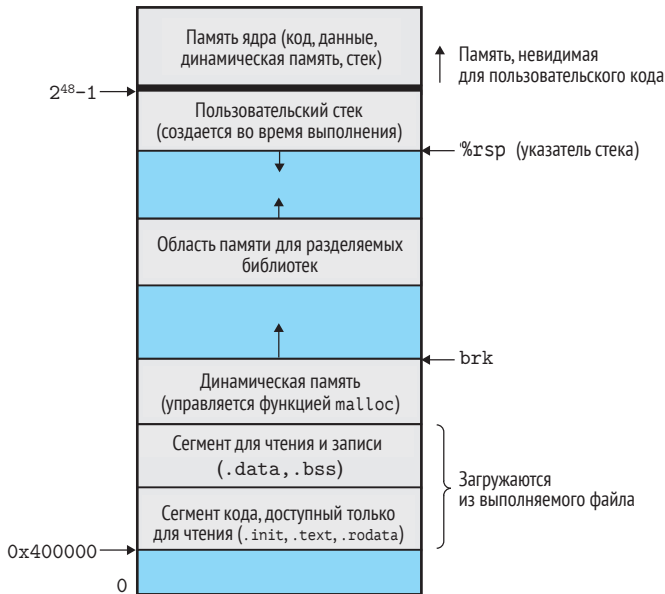


Рис. 8.9. Адресное пространство процесса

Нижняя часть адресного пространства зарезервирована для прикладных программ с обычными сегментами кода, данных и стека. Сегмент кода всегда начинается с адреса  $0x400000$ . Верхняя часть зарезервирована для ядра системы (резидентной части операционной системы, постоянно находящейся в памяти). Эта часть адресного пространства содержит программный код, данные и стек, которые ядро использует для выполнения инструкций от имени процесса (например, когда прикладная программа выполняет системный вызов).

### 8.2.4. Пользовательский и привилегированный режимы

Для поддержки модели изолированных процессов процессор должен обеспечить механизм ограничения набора инструкций, доступных приложениям, а также ограничения доступа к части адресного пространства.

Обычно такую возможность процессоры обеспечивают, используя *бит режима* в управляющем регистре, определяющий привилегии, которыми обладает процесс в настоящее время. Когда бит режима установлен, процесс выполняется в привилегиро-

ванном режиме (иногда его называют *режимом супервизора*). Процесс, выполняющийся в привилегированном режиме, может выполнить любую инструкцию и обратиться к любому адресу в памяти системы.

Если бит режима не установлен, то процесс выполняется в *пользовательском режиме*. В этом режиме процесс не может выполнять *привилегированные инструкции*, такие как остановка процессора, изменение бита режима или запуск операции ввода/вывода. Он также не может ссылаться на программный код или данные в адресном пространстве ядра. Любая такая попытка приводит к фатальной ошибке защиты. Для выполнения подобных операций пользовательские программы должны посылать запросы ядру через интерфейс системных вызовов.

Процесс, выполняющий прикладной программный код, изначально запускается в пользовательском режиме. Единственный способ сменить режим процесса на привилегированный – вызвать исключение, такое как прерывание, сбой или системный вызов. В случае исключения управление будет передано обработчику, и процессор изменит режим с пользовательского на привилегированный. Обработчик выполняется в привилегированном режиме. Когда управление возвращается приложению, процессор меняет режим с привилегированного на пользовательский.

Linux предоставляет хитроумный механизм – файловую систему `/proc`, – позволяющий процессам, действующим в пользовательском режиме, обращаться к содержимому структур данных ядра. Файловая система `/proc` открывает доступ ко многим структурам данных ядра посредством иерархии текстовых файлов, доступных пользовательским программам для чтения. Например, используя файловую систему `/proc`, можно получить справку о характеристиках системы, таких как тип процессора (`/proc/cpuinfo`), или о сегментах памяти, используемых конкретным процессом (`/proc/<id процесса>/maps`). В версии 2.6 ядра Linux появилась файловая система `/sys`, экспортирующая дополнительную низкоуровневую информацию о системных шинах и устройствах.

### 8.2.5. Переключение контекста

Ядро операционной системы реализует многозадачность, используя высокоуровневую форму передачи управления по исключению, известную как *переключение контекста*. Механизм переключения контекста основывается на низкоуровневом механизме исключений, который мы обсудили в разделе 8.1.

Для каждого процесса ядро поддерживает *контекст* – структуру с информацией о состоянии, которую ядро должно восстановить при возобновлении прерванного процесса. Эта структура включает, например, регистры общего назначения, регистры с плавающей точкой, счетчик инструкций, пользовательский стек, регистры флагов, стек ядра и различные структуры ядра, такие как *таблицу страниц*, определяющую структуру адресного пространства; *таблицу процессов* с информацией о текущем процессе, *таблицу файлов* с информацией о файлах, открытых процессом.

В определенные моменты ядро может решить прервать текущий процесс и возобновить выполнение другого процесса, прерванного ранее. Такое поведение называется *планированием* и осуществляется благодаря программному коду в ядре, который называют *планировщиком*. Когда ядро выбирает новый процесс для запуска на выполнение, мы говорим, что ядро *планирует* этот процесс. После того как ядро запланирует новый процесс, оно прерывает текущий процесс и передает управление новому процессу, используя механизм переключения контекста, который сохранит контекст текущего процесса, восстановит сохраненный контекст возобновляемого процесса и передаст управление этому процессу.

Переключение контекста может произойти, когда ядро выполняет системный вызов от имени пользователя. Если системный вызов должен заблокировать процесс, пока не наступит некоторое событие, то ядро может приостановить текущий процесс и дать по-



работать другому процессу. Например, если системный вызов `read` должен выполнить операцию с диском, то ядро может переключить контекст и возобновить другой процесс вместо ожидания, пока данные будут прочитаны с диска. Еще один пример: системный вызов `sleep`, который явно приостанавливает процесс. В общем случае, даже если системный вызов может выполняться без блокировки прикладного процесса, ядро все равно может решить переключить контекст.

Переключение контекста может также произойти в результате прерывания. Например, некоторые системы поддерживают периодические прерывания от таймера, обычно каждые 10 мс. Каждый раз, когда поступает прерывание от таймера, ядро может решить, что текущий процесс выполняется достаточно долго, и переключиться на другой процесс.

На рис. 8.10 показан пример переключения контекста между парой процессов *A* и *B*. В этом примере процесс *A* продолжает выполняться в пользовательском режиме, пока не выполнит системное прерывание, обратившись к системному вызову `read`. Обработчик системного прерывания в ядре отправит контроллеру диска запрос на пересылку данных методом DMA и предпримет меры, чтобы контроллер диска сгенерировал прерывание по окончании передачи данных с диска в память.

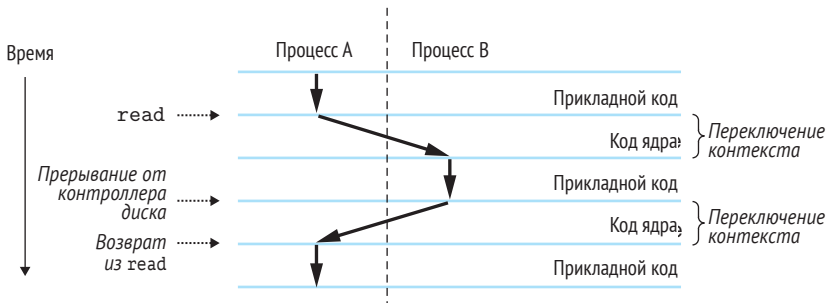


Рис. 8.10. Порядок переключения контекста процесса

Контроллеру диска требуется довольно много времени (порядка десятков миллисекунд), чтобы прочитать данные, поэтому, чтобы не терять время понапрасну, ядро выполнит переключение контекста и передаст управление процессу *B*. Имейте в виду, что перед переключением ядро выполняет инструкции в пользовательском режиме от имени процесса *A* (то есть в системе не существует такой вещи, как процесс ядра). В течение первой фазы переключения ядро выполняет инструкции в привилегированном режиме от имени процесса *A*. Затем в некоторый момент оно начинает выполнять инструкции (все еще в привилегированном режиме) от имени процесса *B*, и после того как переключение будет завершено, ядро начнет выполнять инструкции в пользовательском режиме от имени процесса *B*.

Затем процесс *B* будет выполняться некоторое время в пользовательском режиме, пока диск не сгенерирует прерывание, чтобы сообщить о завершении передачи данных с диска в память. В этот момент ядро решает, что процесс *B* выполнялся уже достаточно долго, и производит переключение контекста на процесс *A*, возвращая управление инструкции, следующей непосредственно за системным вызовом `read` в процессе *A*. После этого процесс *A* продолжает выполняться, пока не произойдет следующее исключение и т. д.

## 8.3. Системные вызовы и обработка ошибок

Когда в системных функциях Unix возникают ошибки, они обычно возвращают `-1` и записывают в глобальную целочисленную переменную `errno` номер ошибки, сообщающий причину. Программисты *всегда* должны проверять наличие ошибок, но, к сожалению,



нию, не выполняют такие проверки, потому что из-за этого программный код разбухнет и становится трудночитаемым. Например, вот как можно проверить наличие ошибки при обращении к системному вызову `fork`:

```
1 if ((pid = fork()) < 0) {
2     fprintf(stderr, "fork error: %s\n", strerror(errno));
3     exit(0);
4 }
```

Функция `strerror` возвращает строку с описанием ошибки, соответствующей конкретному значению `errno`. Этот программный код можно немного упростить, определив следующую функцию вывода сообщения об ошибке:

```
1 void unix_error(char *msg) /* Вывод сообщений об ошибках в стиле Unix */
2 {
3     fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4     exit(0);
5 }
```

При наличии этой функции вызов `fork` можно уместить в две строки вместо четырех:

```
1 if ((pid = fork()) < 0)
2     unix_error("fork error");
```

Этот программный код можно упростить еще больше, используя *обертки с поддержкой обработки ошибок*. Для заданной базовой функции `foo` можно определить функцию-обертку `Foo` с идентичным набором параметров, в имени которой первый символ – прописная буква. Обертка вызывает основную функцию, проверяет наличие ошибки и завершает работу программы, если имеются какие-либо проблемы. Например, вот как выглядит обертка с обработкой ошибок для системного вызова `fork`:

```
1 pid_t Fork(void)
2 {
3     pid_t pid;
4
5     if ((pid = fork()) < 0)
6         unix_error("Fork error");
7     return pid;
8 }
```

С этой оберткой вызов функции `fork` сокращается до одной компактной строки:

```
1 pid = Fork();
```

Мы будем использовать такие обертки на протяжении всей оставшейся части книги. Это позволит нам сохранить примеры программ компактными и не создавать впечатления, что игнорировать ошибки допустимо. Имейте в виду, что при обсуждении системных функций мы всегда будем ссылаться на них, указывая их имена в нижнем регистре, но не имена соответствующих им обертки.

Обсуждение вопросов обработки ошибок в Unix, а также обертки с обработкой ошибок, используемые в этой книге, вы найдете в приложении А. Обертки определены в файле `csapp.c`, а их прототипы – в заголовочном файле `csapp.h`. Они доступны на веб-сайте CS:APP.

## 8.4. Управление процессами

В Unix имеется несколько системных вызовов для управления процессами из программ на языке C. В этом разделе описаны наиболее важные функции, а также даны примеры их использования.

### 8.4.1. Получение идентификатора процесса

Каждый процесс имеет уникальный положительный целочисленный (отличный от нуля) идентификатор (Process Identifier, PID). Функция `getpid` возвращает PID вызывающего ее процесса. Функция `getppid` возвращает PID родительского процесса (создавшего вызывающий процесс).

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

Возвращает PID вызывающего процесса или его родителя

Функции `getpid` и `getppid` возвращают целочисленное значение типа `pid_t`, который в системах Linux определен в `types.h` как `int`.

### 8.4.2. Создание и завершение процессов

С точки зрения программиста процесс находится в одном из трех состояний:

- *выполнение* – когда процесс выполняется на процессоре или ожидает наступления события, когда он будет наконец запланирован ядром;
- *приостановка* – процесс *приостановлен* и не будет запланирован на выполнение в ближайшем будущем. Процесс приостанавливается при получении одного из сигналов – `SIGSTOP`, `SIGTSTP`, `SIGTTIN` или `SIGTTOU` – и остается в таком состоянии, пока не будет получен сигнал `SIGCONT`, после чего продолжит выполняться с точки, где был приостановлен (*сигналы* – это разновидность программных прерываний, которая подробно описывается в разделе 8.5);
- *завершение* – завершенный процесс – это процесс, остановившийся навсегда. Процесс переходит в состояние завершения по одной из трех причин: при получении сигнала, который по умолчанию завершает выполнение процесса; при возврате из подпрограммы `main` или при вызове функции `exit`.

```
#include <stdlib.h>

void exit(int status);
```

Эта функция не возвращает управление

Функция `exit` завершает выполнение процесса с *кодом завершения*, указанным в параметре `status`. (Другой способ вернуть определенный код завершения – вернуть его из функции `main` инструкцией `return`.)

*Родительский процесс* создает новый выполняющийся дочерний процесс вызовом функции `fork`.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Возвращает 0 в дочернем процессе, PID дочернего процесса в родителе и -1 в случае ошибки

Вновь созданный дочерний процесс почти, но не совсем, идентичен своему родителю. Дочерний процесс становится идентичной (и самостоятельной) копией родителя в виртуальном адресном пространстве и имеет свои сегменты кода, данных, динамической памяти, разделяемых библиотек и стека. Дочерний процесс получает также идентичные копии всех дескрипторов файлов, открытых родителем, а это означает, что он может читать и писать во все файлы, открытые в родительском процессе до момента вызова `fork`. Наиболее важное отличие между родителем и потомком – они имеют разные PID.

Функция `fork` интересна тем, что вызывается один раз, но возвращает значение дважды: один раз в вызывающем (родительском) процессе и один раз во вновь созданном дочернем процессе. Родительскому процессу `fork` возвращает PID дочернего процесса, а дочернему – значение 0. Поскольку PID дочернего процесса всегда отличен от нуля, возвращаемое значение однозначно указывает, кому оно предназначено: родителю или потомку.

В листинге 8.2 показан простой пример родительского процесса, который вызывает `fork`, чтобы создать дочерний процесс. Когда `fork` возвращает управление (строка 6), переменная `x` имеет значение 1 в обоих процессах, родительском и дочернем. Дочерний процесс увеличивает и выводит значение своей копии `x` (строка 8). Родительский процесс, напротив, уменьшает значение и выводит значение своей копии `x` (строка 13).

**Листинг 8.2.** Использование функции `fork` для создания нового процесса

```

1 int main()
2 {
3     pid_t pid;
4     int x = 1;
5
6     pid = Fork();
7     if (pid == 0) { /* Дочерний процесс */
8         printf("child : x=%d\n", ++x);
9         exit(0);
10    }
11
12    /* Родительский процесс */
13    printf("parent: x=%d\n", --x);
14    exit(0);
15 }
```

Если запустить эту программу в системе Unix, она выведет следующее:

```

linux> ./fork
parent: x=0
child : x=2
```

В этом простом примере можно отметить некоторые интересные особенности.

*Вызывается однажды, возвращается дважды.*

Функция `fork` вызывается один раз родителем, но возвращает управление дважды, один раз в родительском процессе и один раз во вновь созданном дочернем процессе. Этот аспект не вызывает путаницы, когда родитель создает только одного потомка. Но в программах, где `fork` вызывается многократно, легко запутаться, поэтому разбираться в них нужно очень внимательно.

*Конкурентное выполнение.*

Родительский и дочерний процессы – это разные процессы, выполняющиеся конкурентно. Ядро системы может произвольно чередовать выполнение ин-

струкций в их логических потоках управления. Когда мы попробовали запустить программу из листинга 8.2 в нашей системе, родительский процесс первым выполнил вызов `printf`, а потом то же самое сделал дочерний процесс. Однако в другой системе возможен обратный порядок. В общем случае вы не должны полагаться на какой-то определенный порядок выполнения различных процессов.

#### Идентичные, но отдельные адресные пространства.

Если бы можно было остановить родительский и дочерний процессы сразу после возврата из функции `fork`, то мы увидели бы, что адресные пространства этих процессов идентичны. Оба процесса будут иметь одинаковые стеки, одинаковые значения локальных переменных, одинаковые значения глобальных переменных, а также одинаковый программный код. Мы уже видели в нашем примере, что локальная переменная `x` имеет значение 1 в обоих процессах, когда функция `fork` возвращает управление (строка 6). Однако, поскольку родительский и дочерний процессы – это разные процессы, каждый имеет свое собственное изолированное адресное пространство. Любые последующие изменения переменной `x` в родительском или дочернем процессе не отражаются на содержимом этой же переменной в другом процессе. Вот почему родительский и дочерний процессы выводят разные значения.

#### Общие файлы.

Обратите внимание, что после запуска программы из примера в листинге 8.2 оба процесса выводят результаты на один и тот же экран. Это возможно, потому что дочерний процесс наследует все файлы, открытые родителем. Когда родительский процесс вызывает `fork`, в нем открыт файл `stdout`, соответствующий экрану. Дочерний процесс наследует этот файл и потому тоже осуществляет вывод на экран.

Только начиная осваивать функцию `fork`, часто полезно нарисовать *граф процессов*, иллюстрирующий порядок выполнения инструкций в программе. Каждая вершина *a* соответствует выполнению оператора программы. Направленное ребро  $a \rightarrow b$  указывает, что оператор *a* «выполняется до» оператора *b*. Ребра можно подписывать дополнительной информацией, такой как текущее значение переменной. Вершины, соответствующие вызовам `printf`, можно снабдить результатом, который выводит `printf`. Каждый граф начинается с вершины, соответствующей родительскому процессу, вызывающему `main`. У этой вершины нет входящих ребер и ровно одно исходящее ребро. Последовательность вершин для каждого процесса заканчивается вершиной, соответствующей вызову `exit`. Эта вершина имеет одно входящее ребро и не имеет исходящих ребер.

Например, на рис. 8.11 показан граф процессов для примера программы в листинге 8.2. Первоначально родительский процесс присваивает переменной `x` значение 1. Затем вызывает функцию `fork`, которая создает дочерний процесс, выполняющийся конкурентно с родительским процессом в своем собственном адресном пространстве.

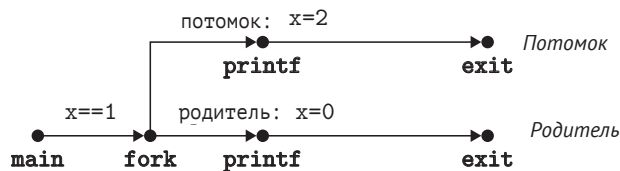


Рис. 8.11. Граф процессов для примера программы в листинге 8.2

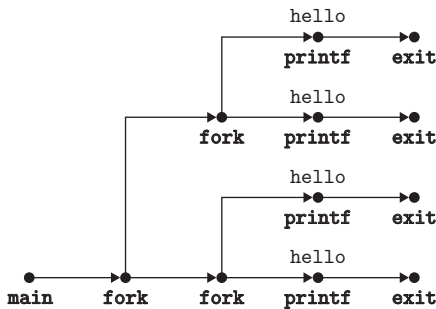
Для программы, выполняющейся на одном процессоре, любая *топологическая сортировка* вершин в графе процессов представляет допустимый общий порядок вы-

полнения операторов в программе. Вот простой способ понять суть топологической сортировки: на основе некоторой перестановки вершин в графе процессов нарисуйте последовательность вершин на одной линии, слева направо, а затем добавьте направленные ребра. Перестановка является топологической сортировкой тогда и только тогда, когда каждое ребро на рисунке направлено слева направо. Таким образом, в нашем примере программы в листинге 8.2 операторы `printf` в родительском и дочернем процессах могут следовать в любом порядке, потому что каждый порядок соответствует некоторой топологической сортировке вершин графа.

Граф процесса может особенно пригодиться для понимания программ с вложенными вызовами `fork`. Например, в листинге 8.3 показана программа с двумя вызовами `fork`. Соответствующий граф процесса (рис. 8.12) помогает увидеть, что эта программа создает четыре процесса, которые вызывают `printf` и могут выполняться в любом порядке.

### Листинг 8.3. Пример вложенных вызовов `fork`

```
1 int main()
2 {
3     Fork();
4     Fork();
5     printf("hello\n");
6     exit(0);
7 }
```



**Рис. 8.12.** Граф процессов, порождаемый вложенными вызовами функции `fork`

### Упражнение 8.2 (решение в конце главы)

Взгляните на следующую программу:

*code/ecf/forkprob0.c*

```
1 int main()
2 {
3     int x = 1;
4
5     if (Fork() == 0)
6         printf("p1: x=%d\n", ++x);
7     printf("p2: x=%d\n", --x);
8     exit(0);
9 }
```

*code/ecf/forkprob0.c*

1. Что выведет дочерний процесс?
2. Что выведет родительский процесс?

### 8.4.3. Утилизация дочерних процессов

Когда процесс завершается по той или иной причине, ядро не удаляет его немедленно. Процесс остается в состоянии завершения, пока не будет *утилизирован* родителем. После того как родитель утилизирует завершившийся дочерний процесс, ядро передаст родителю код завершения дочернего процесса, а затем удалит завершённый процесс, и в этот момент тот прекратит свое существование. Завершившийся, но еще не утилизированный процесс называется *зомби*.

Если родительский процесс завершится, не утилизировав свои дочерние процессы-зомби, ядро переподчинит их процессу `init`, который выполнит утилизацию. Процесс `init` имеет PID 1 и создается ядром во время запуска системы. Такие долгоживущие программы, как командные оболочки или серверы, всегда должны утилизировать дочерние процессы-зомби. Зомби продолжают потреблять память системы, даже притом, что они не выполняются.

Процесс может дожидаться завершения или остановки своих дочерних процессов вызовом функции `waitpid`.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *statusp, int options);

        Возвращает PID дочернего процесса в случае успеха, 0
        (при вызове с параметром WNOHANG) и -1 в случае ошибки
```

Функция `waitpid` – довольно сложная. По умолчанию (когда в параметре `options` передается 0) `waitpid` приостанавливает выполнение вызвавшего ее процесса, пока не завершится дочерний процесс в его *наборе ожидания*. Если процесс, имеющийся в наборе ожидания, уже завершился к моменту вызова, то `waitpid` вернет управление немедленно. В обоих случаях `waitpid` возвращает PID завершившегося дочернего процесса и утилизирует его. После утилизации дочернего процесса уберет из системы любые следы, напоминающие о нем.

#### Почему завершённые дочерние процессы называют «зомби»?

Слово «зомби» пришло из западноафриканского фольклора и обозначает «ходячий мертвец» – существо, наполовину живое, а наполовину мертвое. Процесс-зомби имеет некоторое сходство с такими «ходячими мертвецами» в том смысле, что даже после завершения («смерти») ядро продолжает хранить информацию о его состоянии, пока он не будет утилизирован родителем.

### Определение членства в наборе ожидания

Членство в наборе ожидания определяется с помощью аргумента `pid`:

- если `pid > 0`, то в набор ожидания входит единственный дочерний процесс с идентификатором, равным значению `pid`;
- если `pid = -1`, то в набор ожидания входят все дочерние процессы.

Функция `waitpid` поддерживает также другие виды наборов ожидания, включая группы процессов Unix, которые мы здесь не будем рассматривать.

## Изменение поведения по умолчанию

Поведение по умолчанию можно изменить, передавая в параметре `options` разные комбинации значений констант `WNOHANG`, `WUNTRACED` и `WCONTINUED`:

`WNOHANG` – требует вернуть управление немедленно (с возвращаемым значением 0), если ни один из дочерних процессов в наборе ожидания еще не завершился. По умолчанию `waitpid` приостанавливает вызывающий процесс до завершения дочернего процесса; этот параметр может пригодиться в случаях, когда программа должна продолжать выполнять полезную работу, ожидая завершения дочернего процесса;

`WUNTRACED` – требует приостановить вызывающий процесс, пока не завершится или не остановится один из процессов в наборе ожидания. Возвращает PID завершившегося или остановившегося дочернего процесса. По умолчанию `waitpid` возвращает управление только при завершении какого-либо из потомков; этот параметр может пригодиться в случаях, когда требуется проверить наличие завершившегося *или* остановившегося дочернего процесса;

`WCONTINUED` – требует приостановить вызывающий процесс, пока не завершится один из процессов в наборе ожидания или пока не возобновится остановившийся процесс (сигналом `SIGCONT`). (Подробнее о сигналах рассказывается в разделе 8.5.)

Параметры можно комбинировать с помощью поразрядной операции ИЛИ. Например:

`WNOHANG | WUNTRACED` – требует приостановить вызывающий процесс, пока не завершится или не остановится дочерний процесс в наборе ожидания, и затем вернуть PID завершившегося или приостановившегося процесса.

## Проверка кода завершения утилизированного дочернего процесса

Если в аргументе `statusp` передать непустой указатель, то `waitpid` запишет по указанному адресу код завершения дочернего процесса. В заголовочном файле `wait.h` имеется несколько макроопределений, которые могут помочь в интерпретации значения кода завершения:

`WIFEXITED(status)` – возвращает истинное значение, если дочерний процесс завершился нормально, т. е. вызовом `exit` или выполнив оператор `return`;

`WEXITSTATUS(status)` – возвращает код завершения дочернего процесса. Этот код определен, только если `WIFEXITED()` возвращает истинное значение;

`WIFSIGNALED(status)` – возвращает истинное значение, если дочерний процесс завершился по сигналу, который не был им перехвачен;

`WTERMSIG(status)` – возвращает номер сигнала, ставший причиной завершения дочернего процесса. Этот код определен, только если `WIFSIGNALED()` возвращает истинное значение;

`WIFSTOPPED(status)` – возвращает истинное значение, если дочерний процесс в настоящее время остановлен;

`WSTOPSIG(status)` – возвращает номер сигнала, ставший причиной остановки дочернего процесса. Этот код определен, только если `WIFSTOPPED()` возвращает истинное значение;

`WIFCONTINUED(status)` – возвращает истинное значение, если дочерний процесс был возобновлен сигналом `SIGCONT`.

## Коды ошибок

Если вызывающий процесс не имеет дочерних процессов, то `waitpid` вернет `-1` и запишет в переменную `errno` значение `ECHILD`. Если функция `waitpid` была прервана сигналом, то она вернет `-1` и значение `EINTR` в `errno`.

### Упражнение 8.3 (решение в конце главы)

Перечислите все, что выводит следующая программа:

*code/ecf/waitprob0.c*

```
1 int main()
2 {
3     if (Fork() == 0) {
4         printf("a"); fflush(stdout);
5     }
6     else {
7         printf("b"); fflush(stdout);
8         waitpid(-1, NULL, 0);
9     }
10    printf("c"); fflush(stdout);
11    exit(0);
12 }
```

*code/ecf/waitprob0.c*

## Функция wait

Функция `wait` – это упрощенная версия `waitpid`.

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait(int *statusp);
```

Возвращает PID дочернего процесса в случае успеха или `-1` в случае ошибки

Вызов `wait(&status)` эквивалентен вызову `waitpid(-1, &status, 0)`.

## Примеры использования waitpid

Функция `waitpid` довольно сложна, поэтому будет полезно рассмотреть несколько примеров. В листинге 8.4 показана программа, использующая функцию `waitpid` для ожидания завершения всех своих  $N$  потомков в произвольном порядке. В строке 11 родительский процесс создает  $N$  потомков, а в строке 12 каждый потомок завершается с уникальным кодом.

### Листинг 8.4. Использование функции waitpid для утилизации зомби

*code/ecf/waitpid1.c*

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6     int status, i;
```



```

7   pid_t pid;
8
9   /* Родитель создает N потомков */
10  for (i = 0; i < N; i++)
11      if ((pid = Fork()) == 0) /* Потомок */
12          exit(100+i);
13
14  /* Родитель утилизирует дочерние процессы без определенного порядка */
15  while ((pid = waitpid(-1, &status, 0)) > 0) {
16      if (WIFEXITED(status))
17          printf("child %d terminated normally with exit status=%d\n",
18                pid, WEXITSTATUS(status));
19      else
20          printf("child %d terminated abnormally\n", pid);
21  }
22
23  /* Нормальное завершение возможно, только если не осталось потомков */
24  if (errno != ECHILD)
25      unix_error("waitpid error");
26
27  exit(0);
28 }

```

*code/ecf/waitpid1.c*

Прежде чем двинуться дальше, проверьте себя, что понимаете, почему строка 12 выполняется всеми дочерними процессами, но не выполняется родителем.

В строке 15 родитель ждет завершения всех дочерних процессов, используя `waitpid` в условном выражении цикла `while`. Поскольку в первом аргументе передается значение `-1`, вызов `waitpid` приостановит родительский процесс до момента, когда завершится любой из дочерних процессов. Когда очередной дочерний процесс завершается, `waitpid` возвращает ненулевой идентификатор PID этого процесса. Строка 16 проверяет код завершения дочернего процесса. Если он завершился нормально – в данном случае вызовом функции `exit`, – то родитель извлекает код завершения и выводит его в `stdout`.

После утилизации всех дочерних процессов очередной вызов `waitpid` возвращает `-1` и устанавливает в `errno` значение `ECHILD`. Строка 24 проверяет, нормально ли завершилась функция `waitpid`, и если нет, то выводит сообщение об ошибке. Когда мы запустили эту программу в нашей системе Linux, она вывела следующее:

```

linux> ./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101

```

Обратите внимание, что программа утилизирует дочерние процессы в произвольном порядке. Порядок утилизации является свойством этой конкретной системы. В другой системе или даже когда программа будет запущена в этой же системе в другой раз, два дочерних процесса могут утилизироваться в противоположном порядке. Это пример *недетерминированного* поведения, которое может затруднить рассуждения о конкуренции. Любой из двух возможных исходов одинаково вероятен, и как программист вы никогда не должны полагаться на то, что один из исходов будет иметь место всегда, каким бы маловероятным ни казался другой исход. Единственное верное предположение: все возможные исходы равновероятны.

В листинге 8.5 показано простое изменение, устраняющее этот недетерминизм, благодаря которому утилизация дочерних процессов происходит в том же порядке, в каком они создавались родителем. В строке 11 родитель сохраняет идентификаторы PID своих дочерних процессов по порядку, а затем ожидает завершения каждого из них в том же порядке, вызывая `waitpid` с соответствующим PID в первом аргументе.

**Листинг 8.5.** Использование waitpid для утилизации потомков-зомби в порядке их создания*code/ecf/waitpid2.c*

```

1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6     int status, i;
7     pid_t pid[N], retpid;
8
9     /* Родитель создает N потомков */
10    for (i = 0; i < N; i++)
11        if ((pid[i] = Fork()) == 0) /* Потомок */
12            exit(100+i);
13
14    /* Родитель утилизирует дочерние процессы в порядке их создания */
15    i = 0;
16    while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
17        if (WIFEXITED(status))
18            printf("child %d terminated normally with exit status=%d\n",
19                retpid, WEXITSTATUS(status));
20        else
21            printf("child %d terminated abnormally\n", retpid);
22    }
23
24    /* Нормальное завершение возможно, только если не осталось потомков */
25    if (errno != ECHILD)
26        unix_error("waitpid error");
27
28    exit(0);
29 }

```

*code/ecf/waitpid2.c***Упражнение 8.4 (решение в конце главы)**

Взгляните на следующую программу:

```

1 int main()
2 {
3     int status;
4     pid_t pid;
5
6     printf("Hello\n");
7     pid = Fork();
8     printf("%d\n", !pid);
9     if (pid != 0) {
10         if (waitpid(-1, &status, 0) > 0) {
11             if (WIFEXITED(status) != 0)
12                 printf("%d\n", WEXITSTATUS(status));
13         }
14     }
15     printf("Bye\n");
16     exit(2);
17 }

```

1. Сколько строк выведет эта программа?
2. Укажите один из возможных порядков вывода строк.

**Константы, связанные с функциями Unix**

Такие константы, как `WNOHANG` и `WUNTRACED`, определяются в системных заголовочных файлах. Например, `WNOHANG` и `WUNTRACED` определяются (косвенно) в заголовочном файле `wait.h`:

```
/* Биты в третьем аргументе 'waitpid'. */
#define WNOHANG    1 /* Не блокировать в ожидании. */
#define WUNTRACED  2 /* Сообщить состояние приостановленного потомка. */
```

Чтобы получить возможность использовать эти константы, нужно подключить заголовочный файл `wait.h`:

```
#include <sys/wait.h>
```

На странице справочного руководства для каждой функции Unix перечислены заголовочные файлы, которые следует подключать перед использованием этой функции. Кроме того, для проверки кодов ошибок, таких как `ECHILD` и `EINTR`, необходимо подключить `errno.h`. Чтобы упростить наши примеры, мы подключаем один заголовочный файл с именем `csapp.h`, который подключает заголовочные файлы с определениями для всех функций, используемых в книге. Заголовочный файл `csapp.h` доступен на веб-сайте CS:APP.

**8.4.4. Приостановка процессов**

Функция `sleep` приостанавливает процесс на указанный период времени.

```
#include <unistd.h>
```

```
unsigned int sleep(unsigned int secs);
```

Возвращает количество секунд до окончания приостановки

Функция `sleep` возвращает ноль, если заданное время истекло, иначе – количество секунд, оставшихся до окончания приостановки. Последнее возможно, если функция `sleep` вернула управление преждевременно. Такое может случиться, если во время приостановки процесс получил сигнал. Сигналы мы будем рассматривать в разделе 8.5.

Еще одна полезная функция – функция `pause`, которая приостанавливает вызывающий процесс до получения им сигнала.

```
#include <unistd.h>
```

```
int pause(void);
```

Всегда возвращает -1

**Упражнение 8.5 (решение в конце главы)**

Напишите функцию-обертку вокруг `sleep` с именем `snooze`, имеющую следующий интерфейс:

```
unsigned int snooze(unsigned int secs);
```

Функция `snooze` должна действовать в точности как функция `sleep` и дополнительно выводить сообщение, дающее представление о том, как долго на самом деле процесс был приостановлен:

Процесс простаивал 4 секунды из 5.

### 8.4.5. Загрузка и запуск программ

Функция `execve` загружает и запускает новую программу в контексте текущего процесса.

```
#include <unistd.h>
```

```
int execve(const char *filename, const char *argv[],
           const char *envp[]);
```

Не возвращает управления в случае успеха; иначе возвращает `-1`

Функция `execve` загружает и запускает выполняемый объектный файл `filename` со списком параметров `argv` и списком переменных окружения `envp`. Возвращает управление в вызывающую программу, только если произошла ошибка, например не был найден файл с указанным именем. То есть в отличие от функции `fork`, которая вызывается один раз и возвращает управление дважды, `execve` вызывается один раз и никогда не возвращает управления.

Список параметров должен иметь структуру, изображенную на рис. 8.13. В параметре `argv` должен передаваться массив указателей на строки, завершающийся нулевым элементом. По соглашению, `argv[0]` – это имя выполняемого объектного файла. Список переменных окружения представлен аналогичной структурой, изображенной на рис. 8.14. В параметре `envp` должен передаваться массив указателей на строки с определениями переменных окружения, завершающийся нулевым элементом. Каждая строка-определение в этом массиве должна включать пару *имя=значение* в формате *имя=значение*.

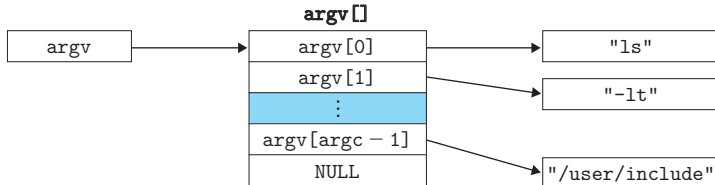


Рис. 8.13. Организация списка аргументов

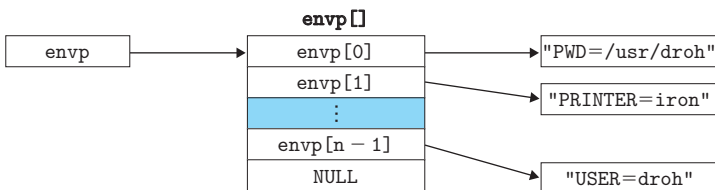


Рис. 8.14. Организация списка переменных окружения

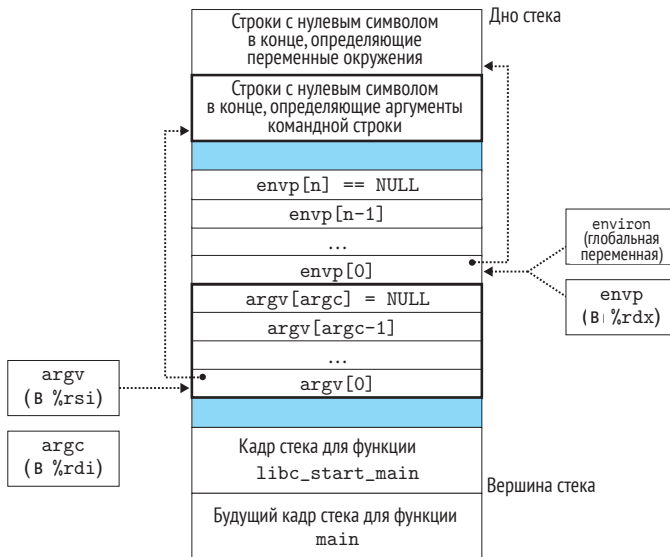
После того как `execve` загрузит `filename`, она вызывает процедуру запуска, описанную в разделе 7.9. Процедура запуска настраивает стек и передает управление функции `main` в новой программе, прототип которой имеет форму

```
int main(int argc, char **argv, char **envp);
```

или, что то же самое:

```
int main(int argc, char *argv[], char *envp[]);
```

Когда `main` начнет выполнение в системе Linux, пользовательский стек будет организован, как показано на рис. 8.15. Пройдемся от дна стека (самый старший адрес) к его вершине (самый младший адрес). Первыми в стеке располагаются строки аргументов и определений переменных окружения, которые хранятся в стеке последовательно, одна за другой, без каких-либо промежутков. Далее следует массив указателей, завершающийся нулевым элементом, ссылающимся на строку в стеке с определением переменных окружения. Глобальная переменная `environ` ссылается на первый из этих указателей, `environ[0]`. За массивом указателей на определения переменных окружения следует массив указателей, завершающийся нулевым элементом. Каждый из указателей ссылается на строку с аргументами в стеке. На вершине стека располагается кадр системной функции запуска `libc_start_main` (раздел 7.9).



**Рис. 8.15.** Типичная организация пользовательского стека в момент запуска новой программы

Функция `main` имеет три аргумента, которые передаются ей в регистрах, в соответствии с соглашениями для архитектуры x86-64:

- 1) `argc` – количество непустых указателей в массиве `argv[]`;
- 2) `argv` – указатель на первый элемент в `argv[]`;
- 3) `envp` – указатель на первый элемент в `envp[]`.

Linux предоставляет несколько функций для работы с массивом определений переменных окружения:

```
#include <stdlib.h>
```

```
char *getenv(const char *name);
```

Возвращает указатель на переменную окружения с именем `name`, если определена, или `NULL`.

Функция `getenv` просматривает массив со строковыми определениями переменных окружения вида `name=значение`. Если искомая переменная найдена, возвращается указатель на *значение*, иначе возвращается `NULL`.

```
#include <stdlib.h>
```

```
int setenv(const char *name, const char *newvalue, int overwrite);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

```
void unsetenv(const char *name);
```

Ничего не возвращает

Если массив с переменными окружения содержит строку в формате `name=старое_значение`, то `unsetenv` удаляет ее, а `setenv` заменяет старое значение новым `newvalue`, но только если в параметре `overwrite` передано ненулевое значение. Если переменная с именем `name` не существует, то `setenv` добавляет в массив элемент `name=newvalue`.

#### Упражнение 8.6 (решение в конце главы)

Напишите программу, называющуюся `myecho`, которая выводит аргументы командной строки и переменные окружения. Например:

```
linux> ./myecho arg1 arg2
```

```
Command-line arguments:
```

```
  argv[ 0]: myecho
```

```
  argv[ 1]: arg1
```

```
  argv[ 2]: arg2
```

```
Environment variables:
```

```
  envp[ 0]: PWD=/usr0/droh/ics/code/ecf
```

```
  envp[ 1]: TERM=emacs
```

```
  ...
```

```
  envp[25]: USER=droh
```

```
  envp[26]: SHELL=/usr/local/bin/tcsh
```

```
  envp[27]: HOME=/usr0/droh
```

#### Программы и процессы

Сейчас самое время сделать паузу и выяснить, насколько хорошо вы понимаете разницу между программой и процессом. Программа – это совокупность программного кода и данных; программы могут существовать в виде объектных модулей на диске или как сегменты в адресном пространстве. Процесс – это конкретный экземпляр программы в стадии выполнения; программа всегда выполняется в контексте некоторого процесса. Понимание этих различий важно, если вы хотите разобраться в работе функций `fork` и `execve`. Функция `fork` запускает ту же самую программу в новом дочернем процессе – полную копию родителя. Функция `execve` загружает и запускает новую программу в контексте текущего процесса. Несмотря на то что она затирает адресное пространство текущего процесса, эта функция не порождает нового процесса. Новая программа по-прежнему получает PID дочернего процесса и наследует все дескрипторы файлов, которые были открыты к моменту вызова функции `execve`.

### 8.4.6. Запуск программ с помощью функций `fork` и `execve`

Такие программы, как командные оболочки Unix и веб-серверы, интенсивно используют функции `fork` и `execve`. *Командная оболочка* – это прикладная интерактивная про-

грамма, которая запускает другие от имени пользователя. Самой первой командной оболочкой была программа `sh`, за которой последовали такие ее варианты, как `csh`, `tcsh`, `ksh` и `bash`. Командная оболочка выполняет последовательность шагов *чтения/обработки*, после чего завершает свою работу. На шаге чтения она читает команду, введенную пользователем. На шаге обработки анализирует команду и выполняет программы от имени пользователя.

В листинге 8.6 показана функция `main` простейшей командной оболочки. Она выводит приглашение к вводу, ожидает, пока пользователь введет свою команду, после чего обрабатывает ее.

**Листинг 8.6.** Функция `main` программы простой командной оболочки

*code/ecf/shellex.c*

```

1 #include "csapp.h"
2 #define MAXARGS 128
3
4 /* Прототипы функций */
5 void eval(char *cmdline);
6 int parseline(char *buf, char **argv);
7 int builtin_command(char **argv);
8
9 int main()
10 {
11     char cmdline[MAXLINE]; /* Команда */
12
13     while (1) {
14         /* Читать */
15         printf("> ");
16         fgets(cmdline, MAXLINE, stdin);
17         if (feof(stdin))
18             exit(0);
19
20         /* Обработать */
21         eval(cmdline);
22     }
23 }
```

*code/ecf/shellex.c*

В листинге 8.7 представлен код, обрабатывающий введенную команду. Сначала он вызывает функцию `parseline` (листинг 8.8), которая анализирует аргументы, перечисленные через пробел, и формирует вектор `argv`, который в конечном счете будет передан функции `execve`. Предполагается, что первый аргумент содержит либо имя встроенной команды командной оболочки, которая будет интерпретироваться непосредственно, либо имя выполняемого объектного файла, который будет загружен и запущен в контексте нового дочернего процесса.

**Листинг 8.7.** Функция `eval` выполняет введенную команду

*code/ecf/shellex.c*

```

1 /* eval -- обрабатывает введенную команду */
2 void eval(char *cmdline)
3 {
4     char *argv[MAXARGS]; /* Список аргументов для execve() */
5     char buf[MAXLINE];    /* Хранит измененную команду */
6     int bg;               /* Режим запуска задания: bg или fg? */
```

```

7     pid_t pid;                /* Идентификатор процесса */
8
9     strcpy(buf, cmdline);
10    bg = parseline(buf, argv);
11    if (argv[0] == NULL)
12        return; /* Игнорировать пустые строки */
13
14    if (!builtin_command(argv)) {
15        if ((pid = Fork()) == 0) { /* Запустить задание в дочернем процессе */
16            if (execve(argv[0], argv, environ) < 0) {
17                printf("%s: Command not found.\n", argv[0]);
18                exit(0);
19            }
20        }
21
22        /* Родитель должен дождаться завершения потомка, */
23        /* запущенного на переднем плане */
24        if (!bg) {
25            int status;
26            if (waitpid(pid, &status, 0) < 0)
27                unix_error("waitfg: waitpid error");
28        }
29        else
30            printf("%d %s", pid, cmdline);
31    }
32    return;
33 }
34
35 /* Если первый аргумент -- встроенная команда, выполнить ее и вернуть true */
36 int builtin_command(char **argv)
37 {
38     if (!strcmp(argv[0], "quit")) /* команда quit */
39         exit(0);
40     if (!strcmp(argv[0], "&"))    /* Пропустить одиночный символ & */
41         return 1;
42     return 0;                    /* Невстроенная команда */
43 }

```

*code/ecf/shellx.c***Листинг 8.8.** *parseline* анализирует строку, полученную командной оболочкой

```

1 /* parseline -- анализирует команду и конструирует массив argv */
2 int parseline(char *buf, char **argv)
3 {
4     char *delim; /* Указывает на первый разделительный пробел */
5     int argc;    /* Количество аргументов */
6     int bg;      /* Фоновое задание? */
7
8     buf[strlen(buf)-1] = ' '; /* Заменить конечный '\n' пробелом */
9     while (*buf && (*buf == ' ')) /* Пропустить начальные пробелы */
10         buf++;
11
12     /* Сконструировать список argv */
13     argc = 0;
14     while ((delim = strchr(buf, ' '))) {

```

*code/ecf/shellx.c*



```

15     argv[argc++] = buf;
16     *delim = '\\0';
17     buf = delim + 1;
18     while (*buf && (*buf == ' ')) /* Пропустить пробелы */
19         buf++;
20 }
21 argv[argc] = NULL;
22
23 if (argc == 0) /* Пропустить пустую строку */
24     return 1;
25
26 /* Задание должно запускаться в фоновом режиме? */
27 if ((bg = (*argv[argc-1] == '&')) != 0)
28     argv[--argc] = NULL;
29
30 return bg;
31 }

```

*code/ecf/shellex.c*

Если последним аргументом является символ '&', то `parseline` возвращает 1, показывая, что программа должна запускаться в *фоновом режиме* (командная оболочка не будет ждать ее завершения). Иначе возвращается 0, показывающий, что программа должна запускаться *на переднем плане* (командная оболочка будет ждать ее завершения).

После анализа команды `eval` вызывает функцию `builtin_command`, которая проверяет, является ли первый аргумент встроенной командой. Если да, то она интерпретирует команду непосредственно и возвращает 1. В противном случае возвращается 0. Наша простейшая командная оболочка имеет только одну встроенную команду, `quit`, которая завершает работу оболочки. Настоящие командные оболочки имеют довольно много встроенных команд, например `pwd`, `jobs` и `fg`.

Если `builtin_command` вернула 0, то командная оболочка создает дочерний процесс и запускает в его контексте новую программу. Если пользователь указал, что программа должна запускаться в фоновом режиме, то командная оболочка возвращается в начало цикла и ждет ввода следующей команды. Иначе оболочка вызывает функцию `waitpid`, чтобы дождаться завершения данного задания. Когда задание завершится, командная оболочка переходит в начало следующей итерации.

Имейте в виду, что это простейшая командная оболочка – он не утилизирует запущенные ею фоновые дочерние процессы. Для исправления этого недостатка необходимо использовать сигналы, которые описываются в следующем разделе.

## 8.5. Сигналы

К настоящему моменту, изучая потоки управления с исключениями, вы познакомились с особенностями взаимодействий аппаратного и программного обеспечения, обеспечивающих базовый механизм низкоуровневых исключений. Вы также узнали, как операционная система использует исключения для поддержки высокоуровневой формы передачи управления по исключению, известной как переключение контекста. В этом разделе вы познакомитесь с высокоуровневой формой программных исключений (сигналами), которая дает процессам и ядру возможность прерывать другие процессы.

*Сигнал* – это короткое сообщение, уведомляющее процесс о наступлении некоторого события. Например, в табл. 8.4 перечислены 30 различных сигналов, поддерживаемых системами Linux.

**Таблица 8.4.** Сигналы Linux. *Примечания:* (а) вывод дампа памяти означает запись образа процесса с сегментами кода и данных на диск; (b) этот сигнал нельзя ни перехватить, ни проигнорировать (см. `man 7 signal`. Данные Linux Foundation)

Номер	Имя	Действие по умолчанию	Соответствующее событие
1	SIGHUP	Завершить	Потеря связи с терминалом
2	SIGINT	Завершить	Прерывание с клавиатуры
3	SIGQUIT	Завершить	Команда завершения с клавиатуры
4	SIGILL	Завершить	Недопустимая инструкция
5	SIGTRAP	Завершить и вывести дамп памяти <sup>а</sup>	Системное прерывание трассировки
6	SIGABRT	Завершить и вывести дамп памяти <sup>а</sup>	Сигнал аварийного завершения от функции <code>abort</code>
7	SIGBUS	Завершить	Ошибка шины
8	SIGFPE	Завершить и вывести дамп памяти <sup>а</sup>	Исключение в операции с плавающей точкой
9	SIGKILL	Завершить <sup>б</sup>	Прекратить выполнение программы
10	SIGUSR1	Завершить	Пользовательский сигнал 1
11	SIGSEGV	Завершить и вывести дамп памяти <sup>а</sup>	Ссылка на недопустимый адрес памяти (ошибка сегментации)
12	SIGUSR2	Завершить	Пользовательский сигнал 2
13	SIGPIPE	Завершить	Запись в программный канал при отсутствии получателя
14	SIGALRM	Завершить	Сигнал таймера от функции <code>alarm</code>
15	SIGTERM	Завершить	Программный сигнал завершения
16	SIGSTKFLT	Завершить	Ошибка стека в сопроцессоре
17	SIGCHLD	Игнорировать	Дочерний процесс приостановился или завершился
18	SIGCONT	Игнорировать	Возобновление процесса после приостановки
19	SIGSTOP	Приостановить до сигнала SIGCONT <sup>б</sup>	Сигнал останова не от терминала
20	SIGTSTP	Приостановить до сигнала SIGCONT	Сигнал останова от терминала
21	SIGTTIN	Приостановить до сигнала SIGCONT	Фоновый процесс читает из терминала
22	SIGTTOU	Приостановить до сигнала SIGCONT	Фоновый процесс пишет в терминал
23	SIGURG	Игнорировать	Срочное событие в сокете
24	SIGXCPU	Завершить	Исчерпан лимит использования процессора
25	SIGXFSZ	Завершить	Превышен предельный размер файла
26	SIGVTALRM	Завершить	Сигнал от виртуального таймера
27	SIGPROF	Завершить	Сигнал от таймера профилирования
28	SIGWINCH	Игнорировать	Размер окна изменился
29	SIGIO	Завершить	Ввод/вывод теперь возможен по дескриптору
30	SIGPWR	Завершить	Сбой в системе электропитания

Каждый тип сигнала соответствует некоторому событию в системе. Аппаратные исключения обрабатываются обработчиками исключений в ядре и обычно невидимы для пользовательских процессов. Сигналы обеспечивают механизм уведомления о возник-

ших исключениях. Например, если процесс предпринимает попытку деления на ноль, то ядро посылает ему сигнал SIGFPE (сигнал 8). Если процесс выполняет недопустимую инструкцию, то ядро посылает ему сигнал SIGILL (сигнал 4). Если процесс пытается сослаться на недопустимый адрес памяти, то ядро посылает ему сигнал SIGSEGV (сигнал 11). Другие сигналы соответствуют высокоуровневым программным событиям в ядре или в иных прикладных процессах. Например, если запустить программу на переднем плане в терминале и затем нажать комбинацию клавиш **Ctrl+C**, то ядро пошлет сигнал SIGINT (сигнал 2) всем процессам в группе переднего плана. Процесс может принудительно завершить другой процесс, пошлав ему сигнал SIGKILL (сигнал 9). Когда дочерний процесс завершается или приостанавливается, ядро посылает родительскому процессу сигнал SIGCHLD (сигнал 17).

### 8.5.1. Терминология сигналов

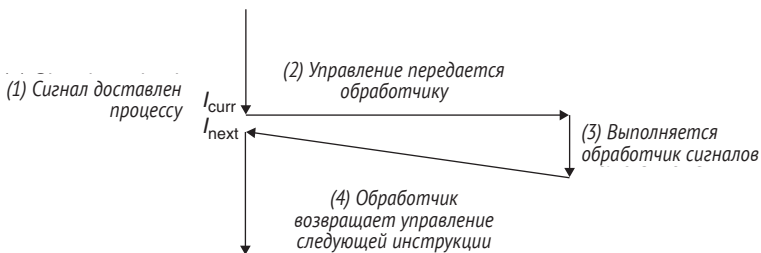
Передача сигнала процессу происходит в два этапа.

#### Посылка сигнала.

Ядро *посылает (доставляет)* сигнал процессу-получателю путем изменения некоторого состояния в его контексте. Сигнал может отправляться по одной из двух причин: (1) в системе наступило некоторое событие, например ошибка деления на ноль или завершение дочернего процесса; (2) процесс вызвал функцию kill (обсуждается в следующем разделе), чтобы явно запросить ядро послать сигнал процессу-получателю. Процесс может послать сигнал самому себе.

#### Получение сигнала.

Процесс-получатель *принимает* сигнал в тот момент, когда ядро принудительно возобновляет его выполнение, дав ему возможность отреагировать на сигнал. Процесс может проигнорировать сигнал, завершиться или *перехватить* (обработать) сигнал, вызвав свою функцию *обработки сигналов*. На рис. 8.16 показана обобщенная схема обработки перехваченного сигнала.



**Рис. 8.16.** Обработка сигнала. Доставка сигнала инициирует передачу управления функции-обработчику. После завершения обработчик возвращает управление прерванной программе

Посланный, но пока не полученный сигнал называется *ожидającym сигналом*. В каждый момент времени у любого процесса может иметься не более одного ожидающего сигнала одного типа. Если для процесса имеется ожидающий сигнал типа  $k$ , то никакие последующие сигналы этого же типа, отправленные этому процессу, *не будут поставлены в очередь*; они просто теряются. Процесс может выборочно *блокировать* доставку некоторых сигналов. Если доставка сигнала заблокирована, он может отправляться, но не будет доставлен процессу, пока тот не разблокирует этот сигнал.

Ожидающий сигнал доставляется не больше одного раза. Для каждого процесса ядро поддерживает набор ожидающих сигналов в векторе битов pending и набор заблокиро-

ванных сигналов в векторе битов `blocked`<sup>1</sup>. Ядро устанавливает бит  $k$  в `pending` всякий раз, когда процессу посылается сигнал типа  $k$ , и сбрасывает бит  $k$  в `pending` после доставки сигнала.

## 8.5.2. Посылка сигналов

В системах Unix имеется несколько механизмов посылки сигналов процессам. Все эти механизмы основываются на понятии *группы процессов*.

### Группы процессов

Каждый процесс принадлежит одной *группе процессов*, которая идентифицируется положительным целочисленным идентификатором группы процессов. Функция `getpgrp` возвращает идентификатор группы, которой принадлежит текущий процесс.

```
#include <unistd.h>
```

```
pid_t getpgrp(void);
```

Возвращает идентификатор группы процессов, которой принадлежит вызывающий процесс

По умолчанию дочерний процесс принадлежит той же группе, что и родитель. Процесс может сменить группу для себя или другого процесса вызовом функции `setpgid`:

```
#include <unistd.h>
```

```
int setpgid(pid_t pid, pid_t pgid);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Функция `setpgid` переносит процесс `pid` в группу `pgid`. Если в `pid` передать ноль, то используется PID текущего процесса. Если в `pgid` передать ноль, то в качестве идентификатора группы процессов будет использоваться `pid`. Например, если с PID=15213 вызовет

```
setpgid(0, 0);
```

то будет создана новая группа процессов с идентификатором 15213 и в нее добавится процесс 15213.

### Посылка сигналов с помощью программы `/bin/kill`

Программа `/bin/kill` посылает произвольный сигнал другому процессу. Например, команда

```
linux> /bin/kill -9 15213
```

пошлет сигнал 9 (SIGKILL) процессу 15213. Если в качестве PID передать отрицательное число, то сигнал будет отправлен всем процессам в группе с номером, равным абсолютному значению PID. Например, команда

```
linux> /bin/kill -9 -15213
```

пошлет сигнал 9 (SIGKILL) всем процессам в группе с идентификатором 15213. Обратите внимание, что в команде используется полный путь `/bin/kill`, потому что некоторые командные оболочки Unix имеют свою встроенную команду `kill`.

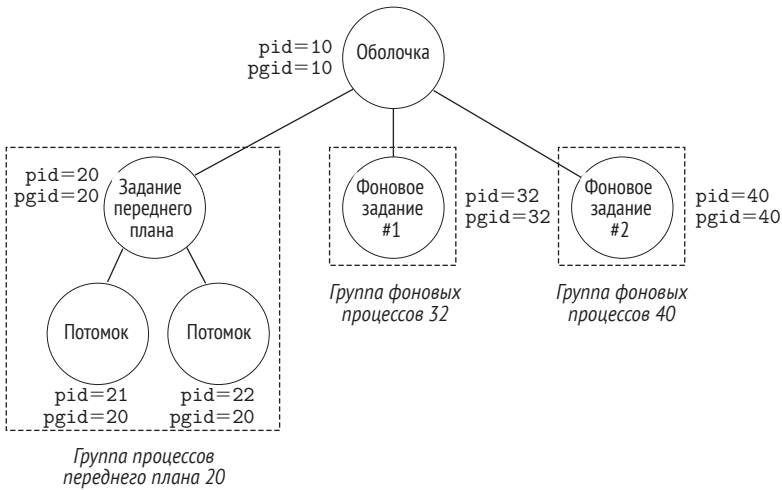
<sup>1</sup> Также известном как *маска сигналов*.

## Посылка сигналов с клавиатуры

Командные оболочки Unix используют абстракцию задания (job) для представления процессов, создаваемых в процессе обработки команды. В любой момент времени имеется, самое большее, одно задание переднего плана и, возможно, несколько фоновых заданий. Например, команда

```
linux> ls | sort
```

создаст задание переднего плана, включающее два процесса, связанных каналом Unix: в одном из них выполняется программа `ls`, а в другом – программа `sort`. Командная оболочка создает для каждого задания отдельную группу процессов. Обычно в роли идентификатора группы процессов выступает идентификатор одного из родительских процессов в этом задании. Например, на рис. 8.17 показана командная оболочка с одним заданием переднего плана и двумя фоновыми заданиями. PID родительского процесса в задании переднего плана равен 20, и идентификатор группы процессов тоже равен 20. Родительский процесс создал два дочерних процесса, тоже принадлежащих группе с идентификатором 20.



**Рис. 8.17.** Группы процессов переднего плана и фоновых процессов

Если на клавиатуре нажать комбинацию **Ctrl+C**, то ядро пошлет сигнал SIGINT всем процессам в группе переднего плана. По умолчанию доставка этого сигнала приводит к завершению задания переднего плана. Аналогично, если нажать комбинацию **Ctrl+Z**, ядро пошлет сигнал SIGTSTP всем процессам в группе переднего плана. По умолчанию доставка этого сигнала приводит к приостановке задания переднего плана.

## Посылка сигналов с помощью функции kill

Процессы могут посылать сигналы другим процессам (в том числе и себе) вызовом функции `kill`:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Если значение `pid` больше нуля, то функция `kill` посылает процессу `pid` сигнал с номером `sig`. Если `pid` равен нулю, то `kill` посылает сигнал `sig` всем процессам в группе, которой принадлежит вызывающий процесс. Если `pid` меньше нуля, то `kill` посылает сигнал `sig` всем процессам в группе `abs(pid)`. В листинге 8.9 показан пример программы родительского процесса, который использует функцию `kill` для отправки сигнала `SIGKILL` своему дочернему процессу.

**Листинг 8.9.** Использование функции `kill` для отправки сигнала дочернему процессу

*code/ecf/kill.c*

```

1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6
7     /* Дочерний процесс приостанавливается до получения сигнала SIGKILL,
8        после чего завершается */
9     if ((pid = Fork()) == 0) {
10         Pause(); /* Ждать доставки сигнала */
11         printf("control should never reach here!\n");
12         exit(0);
13     }
14
15     /* Родитель посылает сигнал SIGKILL потомку */
16     Kill(pid, SIGKILL);
17     exit(0);
18 }
```

*code/ecf/kill.c*

## Посылка сигналов с помощью функции `alarm`

Вызовом функции `alarm` процесс может послать себе сигнал `SIGALRM`.

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned int secs);
```

Возвращает время, оставшееся до получения ранее запланированного сигнала `SIGALRM`, или 0, если такой сигнал еще не планировался

Функция `alarm` передает ядру запрос послать вызывающему процессу сигнал `SIGALRM` через `secs` секунд. Если в параметре `secs` передать ноль, то передача сигнала `SIGALRM` не будет запланирована. В любом случае вызов `alarm` отменяет все ожидающие сигналы `SIGALRM` и возвращает количество секунд, оставшееся до того момента, когда должен быть доставлен ожидающий сигнал `SIGALRM` (если бы этот вызов `alarm` не отменил его), или 0, если такой сигнал еще не планировался.

### 8.5.3. Получение сигналов

Когда ядро возвращает управление процессу *p* (например, из системного вызова или завершив переключение контекста), оно проверяет набор разблокированных ожидающих сигналов (`pending & ~blocked`) для *p*. Если этот набор пуст (обычная ситуация), то управление передается следующей инструкции ( $I_{next}$ ) в логическом потоке управления *p*. Но если набор не пустой, то ядро выбирает некоторый сигнал *k* из этого набора (обычно с меньшим значением *k*) и *доставляет* сигнал *k* процессу *p*. В ответ на

доставленный сигнал процесс выполняет некоторое ответное *действие*, по завершении которого управление передается следующей инструкции ( $I_{\text{next}}$ ) в логическом потоке управления  $p$ . Для каждого типа сигналов предусматривается определенное действие по умолчанию из перечисленных ниже:

- процесс завершается;
- процесс завершается с выводом содержимого памяти в файл;
- процесс приостанавливается до момента, когда ему будет доставлен сигнал SIGCONT;
- процесс игнорирует сигнал.

В табл. 8.4 перечислены стандартные действия, связанные с каждым типом сигнала. Например, сигнал SIGKILL по умолчанию завершает процесс-получатель. С другой стороны, сигнал SIGCHLD по умолчанию игнорируется. Процесс может изменить свою реакцию на сигнал, назначив свой обработчик с помощью функции `signal`. Исключениями являются только сигналы SIGSTOP и SIGKILL, реакцию по умолчанию на которые изменить нельзя.

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Возвращает указатель на предыдущий обработчик в случае успеха, SIG\_ERR в случае ошибки (а также записывает это значение в переменную `errno`)

Функция `signal` предполагает три варианта действий:

- если в `handler` передать SIG\_IGN, то процесс будет игнорировать сигналы типа `signum`;
- если в `handler` передать SIG\_DFL, то процесс будет выполнять действие по умолчанию для сигналов типа `signum`;
- в остальных случаях `handler` интерпретируется как адрес *функции-обработчика сигнала*, которая должна вызываться всякий раз, когда процесс будет получать сигнал типа `signum`. Изменение поведения по умолчанию передачей адреса функции-обработчика в вызов `signal` называется *установкой обработчика*. Вызов обработчика называется *перехватом сигнала*. Выполнение обработчика называется *обработкой сигнала*.

Когда процесс перехватывает сигнал типа  $k$ , вызывается его обработчик сигнала  $k$  с единственным целочисленным аргументом – значением  $k$ . Благодаря этому можно использовать один и тот же обработчик для обработки разных типов сигналов.

Когда обработчик выполняет оператор `return`, управление (обычно) передается назад той же инструкции в потоке управления, на которой процесс был прерван сигналом. Мы говорим «обычно», потому что в некоторых системах прерванный системный вызов немедленно возвращает ошибку.

В листинге 8.10 показана программа, перехватывающая сигнал SIGINT, посылаемый командной оболочкой всякий раз, когда пользователь нажимает комбинацию **Ctrl+C**. По умолчанию сигнал SIGINT завершает процесс. В этом примере программа изменяет стандартное поведение – она перехватывает сигнал, выводит сообщение и только потом завершается.

**Листинг 8.10.** Программа, перехватывающая сигнал SIGINT*code/ecf/sigint.c*

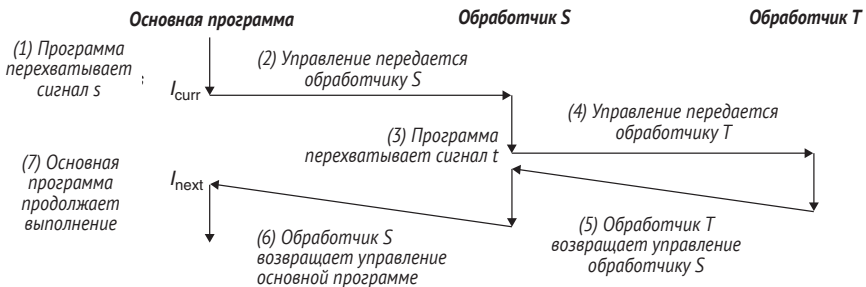
```

1 #include "csapp.h"
2
3 void sigint_handler(int sig) /* обработчик SIGINT */
4 {
5     printf("Caught SIGINT!\n");
6     exit(0);
7 }
8
9 int main()
10 {
11     /* Установить обработчик сигнала SIGINT */
12     if (signal(SIGINT, sigint_handler) == SIG_ERR)
13         unix_error("signal error");
14
15     Pause(); /* Ждать получения сигнала */
16
17     return 0;
18 }

```

*code/ecf/sigint.c*

Обработчики сигналов могут прерываться другими обработчиками, как показано на рис. 8.18. В этом примере основная программа перехватывает сигнал  $s$ , прерывающий выполнение основной программы, и передает управление обработчику  $S$ . Пока обработчик  $S$  выполняется, программа перехватывает другой сигнал  $t \neq s$ , прерывая выполнение обработчика  $S$  и передавая управление обработчику  $T$ . Когда  $T$  возвращает управление, выполнение обработчика  $S$  возобновляется. Наконец, когда обработчик  $S$  завершается, управление передается основной программе и она возобновляет выполнение с того места, где была прервана.

**Рис. 8.18.** Обработчик может прерываться другими сигналами**Упражнение 8.7 (решение в конце главы)**

Напишите программу `snooze`, которая принимает из командной строки единственный аргумент, вызывает функцию `snooze` из упражнения 8.5 с этим аргументом и завершается. Напишите эту программу так, чтобы пользователь мог прервать функцию `snooze`, нажав комбинацию **Ctrl+C**.

```

linux> ./snooze 5
CTRL+C
Slept for 3 of 5 secs.
linux>

```

Пользователь нажимает Ctrl+C через 3 секунды



### 8.5.4. Блокировка и разблокировка сигналов

Linux поддерживает механизмы явной и неявной блокировки сигналов.

*Механизм неявной блокировки.*

По умолчанию ядро блокирует любые ожидающие сигналы того типа, который в данный момент обрабатывается обработчиком. Например, взгляните на рис. 8.18 и представьте, что программа перехватила сигнал *s* и в настоящее время выполняет обработчик *S*. Если процессу будет отправлен другой сигнал *s*, то *s* превратится в ожидающий сигнал и не будет доставлен приложению, пока не завершится обработчик *S*.

*Механизм явной блокировки.*

Приложения могут явно блокировать и разблокировать выбранные сигналы, используя `sigprocmask` и другие вспомогательные функции.

```
#include <signal.h>
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

Возвращают 0 в случае успеха, -1 в случае ошибки

```
int sigismember(const sigset_t *set, int signum);
```

Возвращает 1, если входит в указанное множество *set*,  
0 – если нет, -1 в случае ошибки

Функция `sigprocmask` изменяет набор заблокированных в настоящее время сигналов (вектор битов `blocked` описан в разделе 8.5.1). Конкретное поведение зависит от значения параметра `how`:

`SIG_BLOCK` – добавляет сигналы, перечисленные в `set`,  
в `blocked` (`blocked = blocked | set`);

`SIG_UNBLOCK` – удаляет сигналы, перечисленные в `set`,  
из `blocked` (`blocked = blocked & ~set`);

`SIG_SETMASK` – `blocked = set`.

Если `oldset` не `NULL`, то предыдущее значение вектора битов `blocked` сохраняется в `oldset`.

Наборами сигналов, такими как `set`, можно управлять с помощью следующих функций:

`sigemptyset` инициализирует набор `set`, очищая его;  
`sigfillset` добавляет все поддерживаемые сигналы в `set`;  
`sigaddset` добавляет `signum` в `set`;  
`sigdelset` удаляет `signum` из `set`;  
`sigismember` возвращает 1, если `signum` входит в множество `set`, и 0, если нет.

В листинге 8.11 показано, как можно использовать `sigprocmask` для временной блокировки сигналов `SIGINT`.

**Листинг 8.11.** Временная блокировка сигнала

```

1 sigset_t mask, prev_mask;
2
3 Sigemptyset(&mask);
4 Sigaddset(&mask, SIGINT);
5
6 /* Заблокировать SIGINT и сохранить прежний набор заблокированных сигналов */
7 Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
8
9 ...
10 // Этот код не будет прерываться сигналом SIGINT
11 ...
12 /* Восстановить прежний набор заблокированных сигналов, разблокировать SIGINT */
13 Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
14
15
16

```

### 8.5.5. Обработка сигналов

Обработка сигналов – один из самых сложных аспектов системного программирования в Linux. Обработчики имеют несколько атрибутов, затрудняющих их анализ: (1) обработчики работают конкурентно с основной программой и вместе используют одни и те же глобальные переменные, вследствие чего могут мешать основной программе и другим обработчикам; (2) правила доставки сигналов часто противоречат здравому смыслу; (3) разные системы могут иметь разную семантику обработки сигналов.

В этом разделе мы рассмотрим эти проблемы и дадим некоторые рекомендации по разработке надежных, правильных и переносимых обработчиков сигналов.

#### Надежная обработка сигналов

Сложность обработчиков сигналов обусловлена тем, что они могут работать одновременно с основной программой и друг с другом, как было показано на рис. 8.18. Если обработчик и основная программа одновременно обращаются к одной и той же глобальной структуре данных, то результаты могут быть самыми непредсказуемыми и часто фатальными.

Мы подробно рассмотрим конкурентное программирование в главе 12. А здесь наша цель – дать вам некоторые консервативные рекомендации по разработке обработчиков, которые можно без опаски запускать одновременно. Игнорируя эти рекомендации, вы рискуете внести малозаметные ошибки, с которыми ваша программа большую часть времени будет работать верно, но иногда терпеть неудачи, завершаясь непредсказуемым и неповторимым образом. Эти ошибки ужасно сложно отлаживать. Мы вас предупредили, а предупрежден – значит вооружен!

*Правило 0. Обработчики должны быть максимально простыми.*

Лучший способ избежать проблем – сделать обработчики как можно меньше и проще. Например, обработчик может просто установить глобальный флаг и немедленно вернуть управление, а всю обработку может выполнять основная программа, периодически проверяя (и сбрасывая) флаг.

*Правило 1. Вызывайте в своих обработчиках только функции, безопасные в асинхронном окружении.*

Функцию, безопасную в асинхронном окружении, или просто безопасную функцию, можно без опаски вызывать из обработчика сигнала, потому что она или

*реентерабельная* (например, обращается только к локальным переменным; раздел 12.7.2), или не может быть прервана обработчиком сигнала. В табл. 8.4 перечислены системные функции, безопасность которых Linux гарантирует. Обратите внимание, что в этом списке нет многих популярных функций, таких как `printf`, `sprintf`, `malloc` и `exit`.

Единственный безопасный способ что-то вывести в функции-обработчике сигнала – использовать функцию `write` (раздел 10.1). Функции `printf` или `sprintf` небезопасны в асинхронном окружении. Чтобы обойти это досадное ограничение, мы разработали несколько безопасных функций, включив их в пакет SIO (Safe I/O – безопасный ввод/вывод), которые можно использовать для вывода простых сообщений из обработчиков сигналов.

**Таблица 8.4.** Функции, безопасные в асинхронном окружении

<code>_Exit</code>	<code>fexecve</code>	<code>poll</code>	<code>sigqueue</code>
<code>_exit</code>	<code>fork</code>	<code>posix_trace_event</code>	<code>sigset</code>
<code>abort</code>	<code>fstat</code>	<code>pselect</code>	<code>sigsuspend</code>
<code>accept</code>	<code>fstatat</code>	<code>raise</code>	<code>sleep</code>
<code>access</code>	<code>fsync</code>	<code>read</code>	<code>socketmark</code>
<code>aio_error</code>	<code>ftruncate</code>	<code>readlink</code>	<code>socket</code>
<code>aio_return</code>	<code>futimens</code>	<code>readlinkat</code>	<code>socketpair</code>
<code>aio_suspend</code>	<code>getegid</code>	<code>recv</code>	<code>stat</code>
<code>alarm</code>	<code>geteuid</code>	<code>recvfrom</code>	<code>symlink</code>
<code>bind</code>	<code>getgid</code>	<code>recvmsg</code>	<code>symlinkat</code>
<code>cfgetispeed</code>	<code>getgroups</code>	<code>rename</code>	<code>tcdrain</code>
<code>cfgetospeed</code>	<code>getpeername</code>	<code>renameat</code>	<code>tcflow</code>
<code>cfsetispeed</code>	<code>getpgrp</code>	<code>rmdir</code>	<code>tcflush</code>
<code>cfsetospeed</code>	<code>getpid</code>	<code>select</code>	<code>tcgetattr</code>
<code>chdir</code>	<code>getppid</code>	<code>sem_post</code>	<code>tcgetpgrp</code>
<code>chmod</code>	<code>getsockname</code>	<code>send</code>	<code>tcsendbreak</code>
<code>chown</code>	<code>getsockopt</code>	<code>sendmsg</code>	<code>tcsetattr</code>
<code>clock_gettime</code>	<code>getuid</code>	<code>sendto</code>	<code>tcsetpgrp</code>
<code>close</code>	<code>kill</code>	<code>setgid</code>	<code>time</code>
<code>connect</code>	<code>link</code>	<code>setpgid</code>	<code>timer_getoverrun</code>
<code>creat</code>	<code>linkat</code>	<code>setsid</code>	<code>timer_gettime</code>
<code>dup</code>	<code>listen</code>	<code>setsockopt</code>	<code>timer_settime</code>
<code>dup2</code>	<code>lseek</code>	<code>setuid</code>	<code>times</code>
<code>execl</code>	<code>lstat</code>	<code>shutdown</code>	<code>umask</code>
<code>execle</code>	<code>mkdir</code>	<code>sigaction</code>	<code>uname</code>
<code>execv</code>	<code>mkdirat</code>	<code>sigaddset</code>	<code>unlink</code>
<code>execve</code>	<code>mkfifo</code>	<code>sigdelset</code>	<code>unlinkat</code>
<code>faccessat</code>	<code>mkfifoat</code>	<code>sigemptyset</code>	<code>utime</code>
<code>fchmod</code>	<code>mknod</code>	<code>sigfillset</code>	<code>utimensat</code>
<code>fchmodat</code>	<code>mknodat</code>	<code>sigismember</code>	<code>utimes</code>
<code>fchown</code>	<code>open</code>	<code>signal</code>	<code>wait</code>
<code>fchownat</code>	<code>openat</code>	<code>sigpause</code>	<code>waitpid</code>
<code>fcntl</code>	<code>pause</code>	<code>sigpending</code>	<code>write</code>
<code>fdatasync</code>	<code>pipe</code>	<code>sigprocmask</code>	

Функции `sio_putl` и `sio_puts` выводят в стандартный вывод длинное целое (`long`) и строку соответственно. Функция `sio_error` выводит сообщение об ошибке и завершает работу.

```
#include "csapp.h"
```

```
ssize_t sio_putl(long v);
ssize_t sio_puts(char s[]);
```

Возвращает количество переданных байтов  
в случае успеха, `-1` в случае ошибки

```
void sio_error(char s[]);
```

Ничего не возвращает

В листинге 8.12 показана реализация пакета SIO, в котором используются две приватные реентерабельные функции из `csapp.c`. Функция `sio_strlen` в строке 3 возвращает длину строки `s`. Функция `sio_ltoa` в строке 10, основанная на функции `ltoa` из [61], преобразует `v` в строковое представление `s` по основанию `b`. Функция `_exit` в строке 17 – это асинхронно-безопасный вариант `exit`.

**Листинг 8.12.** Пакет SIO с функциями для использования в обработчиках сигналов

```
code/src/csapp.c
1 ssize_t sio_puts(char s[]) /* Вывод строки */
2 {
3     return write(STDOUT_FILENO, s, sio_strlen(s));
4 }
5
6 ssize_t sio_putl(long v) /* Вывод длинного целого */
7 {
8     char s[128];
9
10    sio_ltoa(v, s, 10); /* Основана на ltoa() Кернигана и Ритчи */
11    return sio_puts(s);
12 }
13
14 void sio_error(char s[]) /* Вывод сообщения об ошибке и выход */
15 {
16     sio_puts(s);
17     _exit(1);
18 }
```

code/src/csapp.c

В листинге 8.13 показана безопасная версия обработчика сигнала SIGINT из листинга 8.10.

**Листинг 8.13.** Безопасная версия обработчика сигнала SIGINT из листинга 8.10

```
code/ecf/sigintsafe.c
1 #include "csapp.h"
2
3 void sigint_handler(int sig) /* Безопасный обработчик SIGINT */
4 {
5     Sio_puts("Caught SIGINT!\n"); /* Безопасный вывод */
6 }
```

```

6  _exit(0); /* Безопасный выход */
7 }

```

*code/ecf/sigintsafe.c*

*Правило 2. Сохраняйте и восстанавливайте errno.*

Многие функции Linux, безопасные в асинхронном окружении, устанавливают переменную `errno`, когда завершаются с ошибкой. Вызов таких функций внутри обработчика может помешать другим частям программы, использующим `errno`.

Обходное решение – сохранить `errno` в локальной переменной при входе в обработчик и восстановить ее перед возвратом из обработчика. Обратите внимание, что это необходимо только в случае возврата из обработчика. Если обработчик завершает процесс, вызывая `_exit`, то в этом нет необходимости.

*Правило 3. Ограничивайте доступ к общим глобальным структурам данных, блокируя все сигналы.*

Если обработчик использует глобальную структуру данных совместно с основной программой или с другими обработчиками, то обработчики и основная программа должны временно заблокировать все сигналы перед обращением к этой структуре данных. Необходимость следования этому правилу объясняется тем, что для доступа к структуре данных *d* из основной программы обычно требуется выполнить последовательность инструкций. Если эта последовательность прерывается обработчиком, который обращается к *d*, то к тому времени, когда запустится обработчик, структура *d* может оказаться в несогласованном состоянии, что может привести к непредсказуемым результатам. Временная блокировка сигналов перед доступом к *d* гарантирует, что обработчик не прервет критическую последовательность инструкций.

*Правило 4. Объявляйте глобальные переменные со спецификатором volatile.*

Представьте обработчик и функцию `main`, которые вместе используют глобальную переменную *g*. Обработчик обновляет *g*, а `main` периодически читает ее. Оптимизирующему компилятору может показаться, что значение *g* никогда не меняется в `main` и будет вполне безопасно использовать копию *g*, кешированную в регистре, чтобы ускорить ссылки на нее. В этом случае функция `main` никогда не увидит значения, записываемые обработчиком.

Вы можете потребовать от компилятора не кешировать переменную, объявив ее со спецификатором `volatile`. Например:

```
volatile int g;
```

Спецификатор `volatile` вынуждает компилятор читать значение *g* из памяти всякий раз, когда переменная упоминается в коде. В общем случае, как и при использовании любой общей структуры данных, доступ к глобальным переменным должен защищаться временной блокировкой сигналов.

*Правило 5. Объявляйте флаги с типом sig\_atomic\_t.*

В одной из наших рекомендаций мы советовали сохранять обработчики максимально простыми и фиксировать факт получения сигнала в глобальном флаге. Основная программа может периодически проверять флаг, реагировать на сигнал и очищать флаг. Для организации подобных флагов язык C предоставляет целочисленный тип данных `sig_atomic_t`, операции чтения и записи с которым выполняются атомарно (то есть не могут быть прерваны), потому что реализуются единственной инструкцией, например:

```
volatile sig_atomic_t flag;
```

Поскольку операции с такими флагами нельзя прервать, переменные типа `sig_atomic_t` можно без опаски читать и изменять без временной блокировки сигналов. Обратите внимание, что гарантия атомарности распространяется только на отдельные операции чтения и записи. Это не относится к таким составным операциям, как `flag++` или `flag = flag + 10`, для реализации которых может потребоваться несколько инструкций.

Имейте в виду, что все представленные рекомендации весьма консервативны, в том смысле, что не всегда строго необходимы. Например, если обработчик не может изменить `errno`, то нет нужды сохранять и восстанавливать `errno`. Или если есть абсолютная уверенность, что ни один вызов `printf` в основной программе не может быть прерван обработчиком, то можете без опаски вызывать `printf` из обработчика. То же относится и к глобальным структурам данных. Однако достичь абсолютной уверенности часто бывает очень трудно. Поэтому мы рекомендуем придерживаться консервативного подхода и следовать рекомендациям, максимально упрощая обработчики, используя безопасные функции, сохраняя и восстанавливая `errno`, защищая доступ к общим структурам данных и используя `volatile` и `sig_atomic_t`.

### Корректная обработка сигналов

Один из малопонятных аспектов сигналов связан с тем, что ожидающие сигналы не ставятся в очередь. Поскольку битовый вектор `pending` содержит ровно один бит для каждого типа сигнала, для любого процесса не может быть больше одного ожидающего сигнала любого конкретного типа. Поэтому если процессу-получателю отправить два сигнала типа *k* в тот момент, когда процесс заблокировал доставку сигналов этого типа, потому что, например, в данный момент обрабатывает ранее полученный сигнал *k*, то второй сигнал будет потерян; он не ставится в очередь, потому что нет никакой очереди. Наличие ожидающего сигнала просто указывает на то, что поступил *по крайней мере* один сигнал.

Чтобы понять, как это влияет на корректность, рассмотрим простое приложение, похожее на настоящие программы, такие как командные оболочки и веб-серверы. Суть программы проста: родительский процесс создает несколько дочерних процессов, которые некоторое время работают независимо, а затем завершаются. Родитель должен утилизировать потомков, чтобы не оставить зомби в системе. Но при этом родитель должен выполнять еще какую-то работу, пока потомки занимаются своим делом. Поэтому мы решили организовать утилизацию дочерних процессов с помощью обработчика сигнала `SIGCHLD` вместо явного ожидания их завершения. (Как вы наверняка помните, ядро отправляет сигнал `SIGCHLD` родителю всякий раз, когда один из его потомков завершает работу или приостанавливается.)

В листинге 8.14 показана наша первая попытка. Родитель устанавливает обработчик `SIGCHLD`, а затем создает три дочерних процесса. Пока потомки выполняются, родитель ждет ввода пользователя с терминала и обрабатывает его. Обработка моделируется бесконечным циклом. Когда любой из потомков завершается, ядро уведомляет родителя, посылая сигнал `SIGCHLD`. Родитель перехватывает `SIGCHLD`, утилизирует один дочерний процесс, выполняет некоторую дополнительную работу по очистке (смоделированную инструкцией `sleep`) и возвращает управление основной программе.

**Листинг 8.14. signal1.** Эта программа работает некорректно, потому что предполагает, что сигналы ставятся в очередь

*code/ecf/signal1.c*

```
1 /* ВНИМАНИЕ: этот код содержит ошибку! */
2
3 void handler1(int sig)
```

```
4 {
5     int olderrno = errno;
6
7     if ((waitpid(-1, NULL, 0)) < 0)
8         sio_error("waitpid error");
9     Sio_puts("Handler reaped child\n");
10    Sleep(1);
11    errno = olderrno;
12 }
13
14 int main()
15 {
16     int i, n;
17     char buf[MAXBUF];
18
19     if (signal(SIGCHLD, handler1) == SIG_ERR)
20         unix_error("signal error");
21
22     /* Родитель запускает дочерние процессы */
23     for (i = 0; i < 3; i++) {
24         if (Fork() == 0) {
25             printf("Hello from child %d\n", (int)getpid());
26             exit(0);
27         }
28     }
29
30     /* Родитель ждет ввода с терминала и обрабатывает его */
31     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
32         unix_error("read");
33
34     printf("Parent processing input\n");
35     while (1)
36         ;
37
38     exit(0);
39 }
```

*code/ecf/signal1.c*

Программа `signal1` в листинге 8.14 кажется довольно простой. Однако, запустив ее в своей системе Linux, мы получили следующий вывод:

```
linux> ./signal1
Hello from child 14073
Hello from child 14074
Hello from child 14075
Handler reaped child
Handler reaped child
CR
Parent processing input
```

Из этого вывода видно, что родителю было отправлено три сигнала `SIGCHLD`, но получены были только два из них, поэтому родитель утилизировал только двух потомков. Если приостановить родительский процесс, то можно увидеть, что дочерний процесс с `PID=14075` действительно не утилизировался и остался в системе как зомби (обозначается строкой `<defunct>` в выводе команды `ps`):

```
Ctrl+Z
Suspended
linux> ps t
```

```

PID TTY      STAT   TIME COMMAND
.
.
.
14072 pts/3    T       0:02 ./signal1
14075 pts/3    Z       0:00 [signal1] <defunct>
14076 pts/3    R+      0:00 ps t

```

Почему это произошло? Дело в том, что наш код не учитывает тот факт, что сигналы не ставятся в очередь. Вот что случилось на самом деле: первый сигнал был доставлен и получен родителем. Пока обработчик обрабатывал первый сигнал, процессу был доставлен второй сигнал и добавлен в набор ожидающих сигналов. Однако, поскольку получение сигналов SIGCHLD блокируется обработчиком SIGCHLD, второй сигнал был задержан. Вскоре после этого, пока обработчик все еще обрабатывал первый сигнал, поступил третий сигнал. Поскольку ожидающий сигнал SIGCHLD уже имеется в наборе ожидающих сигналов, этот третий сигнал SIGCHLD был просто отброшен. Некоторое время спустя после завершения обработчика ядро заметило наличие ожидающего сигнала SIGCHLD и заставило родителя получить этот сигнал. Родитель перехватил сигнал и вызвал обработчика второй раз. К тому моменту, когда обработчик закончит обработку второго сигнала, ожидающих сигналов SIGCHLD не останется, и они больше никогда не поступят, потому что вся информация о третьем сигнале SIGCHLD потеряна. *Главный вывод из этого примера: сигналы нельзя использовать для подсчета событий в других процессах.*

Попробуем исправить эту проблему, а для этого вспомним, что наличие ожидающего сигнала означает только то, что по меньшей мере один сигнал был доставлен с момента последнего получения сигнала этого типа. То есть мы должны изменить обработчик SIGCHLD так, чтобы при каждом вызове он утилизировал как можно больше дочерних процессов-зомби. В листинге 8.15 показан измененный обработчик SIGCHLD.

**Листинг 8.15. signal2.** Улучшенная версия обработчика в листинге 8.14, которая корректно учитывает тот факт, что сигналы не ставятся в очередь

```

code/ecf/signal2.c

1 void handler2(int sig)
2 {
3     int olderrno = errno;
4
5     while (waitpid(-1, NULL, WNOHANG) > 0) {
6         Sio_puts("Handler reaped child\n");
7     }
8     if (errno != ECHILD)
9         Sio_error("waitpid error");
10    Sleep(1);
11    errno = olderrno;
12 }

code/ecf/signal2.c

```

Когда мы запустили эту версию в своей системе Linux, она корректно утилизировала все завершившиеся дочерние процессы:

```

linux> ./signal2
Hello from child 15237
Hello from child 15238
Hello from child 15239
Handler reaped child
Handler reaped child

```



Handler reaped child  
CR  
Parent processing input

### Упражнение 8.8 (решение в конце главы)

Что выведет следующая программа?

*code/ecf/signalprob0.c*

```

1 volatile long counter = 2;
2
3 void handler1(int sig)
4 {
5     sigset_t mask, prev_mask;
6
7     Sigfillset(&mask);
8     Sigprocmask(SIG_BLOCK, &mask, &prev_mask); /* Блокировать сигналы */
9     Sio_putl(--counter);
10    Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Восстановить маску сигналов */
11
12    _exit(0);
13 }
14
15 int main()
16 {
17     pid_t pid;
18     sigset_t mask, prev_mask;
19
20     printf("%ld", counter);
21     fflush(stdout);
22
23     signal(SIGUSR1, handler1);
24     if ((pid = Fork()) == 0) {
25         while(1) {};
26     }
27     Kill(pid, SIGUSR1);
28     Waitpid(-1, NULL, 0);
29
30     Sigfillset(&mask);
31     Sigprocmask(SIG_BLOCK, &mask, &prev_mask); /* Блокировать сигналы */
32     printf("%ld", ++counter);
33     Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Восстановить */
34                                           /* маску сигналов */
35     exit(0);
36 }

```

*code/ecf/signalprob0.c*

## Переносимая обработка сигналов

Еще одна удручающая особенность обработки сигналов в Unix – в разных системах обработка сигналов имеет разную семантику. Например:

- *в разных системах функция signal имеет разную семантику.* В некоторых старых системах Unix действие по умолчанию для сигнала  $k$  восстанавливается после того, как сигнал  $k$  будет перехвачен обработчиком. В этих системах обработчик должен явно переустанавливать себя, вызывая `signal` при каждом запуске;

- *системные вызовы могут прерываться.* Системные вызовы, такие как `read`, `wait` и `assert`, способные заблокировать процесс на длительный период времени, называются *медленными системными вызовами*. В некоторых старых версиях Unix медленные системные вызовы, которые прерываются при поступлении сигнала, не возобновляются, когда обработчик сигнала возвращает управление, и вместо возобновления системного вызова управление немедленно возвращается прикладной программе с признаком ошибки и значением `EINTR` в переменной `errno`. В этих системах программисты должны включать код, перезапускающий прерванные системные вызовы.

Для решения этих проблем стандарт POSIX определяет функцию `sigaction`, позволяющую точно указывать семантику обработки сигналов при установке обработчика.

```
#include <signal.h>

int sigaction(int signum, struct sigaction *act,
              struct sigaction *oldact);

        Возвращает 0 в случае успеха, -1 в случае ошибки
```

Пользоваться функцией `sigaction` неудобно, потому что она требует заполнения сложной структуры. Более простой подход, первоначально предложенный Ричардом У. Стивенсом (W. Richard Stevens) [110], заключается в определении функции-обертки `Signal`, которая вызывает `sigaction`. В листинге 8.16 показано определение функции `Signal`, которая вызывается так же, как функция `signal`.

**Листинг 8.16. Signal.** Функция-обертка для `sigaction`, поддерживающая переносимую обработку сигналов в Posix-совместимых системах

```
code/src/csapp.c

1 handler_t *Signal(int signum, handler_t *handler)
2 {
3     struct sigaction action, old_action;
4
5     action.sa_handler = handler;
6     sigemptyset(&action.sa_mask); /* Заблокировать сигналы с типом
7                                   текущего обрабатываемого сигнала */
8     action.sa_flags = SA_RESTART; /* Перезапустить системный вызов,
9                                   если возможно */
10    if (sigaction(signum, &action, &old_action) < 0)
11        unix_error("Signal error");
12    return (old_action.sa_handler);
13 }
```

code/src/csapp.c

Функция-обертка `Signal` устанавливает обработчик сигнала со следующей семантикой обработки:

- блокируются только сигналы того типа, который в данный момент обрабатывается;
- как и во всех реализациях сигналов, сигналы не ставятся в очередь;
- прерванные системные вызовы автоматически перезапускаются, когда это возможно;
- после установки обработчик сигнала остается установленным, пока не будет вызвана функция `Signal` со значением `SIG_IGN` или `SIG_DFL` в аргументе `handler`.

Далее мы будем использовать функцию-обертку `Signal` во всем нашем коде.

### 8.5.6. Синхронизация потоков во избежание неприятных ошибок конкурентного выполнения

Проблема программирования конкурентных потоков, которые используют одни и те же области памяти, преследует не одно поколение программистов. В общем случае количество возможных чередований потоков выполнения увеличивается экспоненциально с увеличением количества инструкций. Одни из этих чередований генерируют правильные ответы, а другие – нет. Главная проблема в том, чтобы каким-то образом *синхронизировать* конкурирующие потоки и обеспечить наибольший набор возможных чередований, дающих правильный ответ.

Конкурентное программирование – глубокая и важная проблема, которую мы подробно обсудим в главе 12. Однако уже сейчас мы можем использовать то, что узнали о потоках управления с исключениями в этой главе, чтобы вы могли получить представление об интересных интеллектуальных проблемах, связанных с конкуренцией. Например, рассмотрим программу в листинге 8.17, в которой показана структура типичной командной оболочки Unix. Родитель следит за своими потомками, используя глобальный список заданий, в котором каждый элемент соответствует одному заданию. Функции `addjob` и `deletejob` добавляют и удаляют записи из списка заданий.

После создания нового дочернего процесса родитель добавляет его в список заданий. Когда родитель утилизирует завершившийся дочерний процесс (зомби) в обработке сигнала `SIGCHLD`, он удаляет соответствующий элемент из списка заданий.

На первый взгляд этот код кажется корректным, но, к сожалению, возможна такая последовательность событий:

1. Родитель вызывает функцию `fork`, и ядро планирует запуск только что созданного потомка вместо родителя.
2. Прежде чем управление вернется родителю, дочерний процесс завершается и становится зомби, а ядро посылает родителю сигнал `SIGCHLD`.
3. Когда родитель снова будет готов к выполнению, ядро замечает ожидающий `SIGCHLD` и доставляет его процессу, запуская обработчик сигнала в родителе.
4. Обработчик сигнала утилизирует завершившийся дочерний процесс и вызывает `deletejob`, который ничего не делает, потому что родитель еще не добавил соответствующий элемент в список.
5. По завершении обработчика ядро передает управление родительскому процессу, который возвращается из `fork` и ошибочно добавляет (несуществующий) дочерний процесс в список заданий, вызывая `addjob`.

Таким образом, в некоторых чередованиях основной программы родителя и потоков обработки сигналов функция `deletejob` может быть вызвана раньше `addjob`. Это приводит к добавлению ошибочной записи в список заданий, соответствующей заданию, которого больше не существует, и эта запись никогда не будет удалена. С другой стороны, в некоторых чередованиях события происходят в правильном порядке. Например, если ядро запланировало выполнение родительского процесса при возврате из вызова `fork` раньше дочернего, то родительский процесс правильно добавит дочерний процесс в список заданий до того, как тот завершится, а обработчик сигнала удалит задание из списка.

Этот пример классической ошибки синхронизации, известной как *гонка*. В этом случае гонка имеет место между вызовом `addjob` в процедуре `main` и вызовом `deletejob` в обработчике. Если `addjob` выигрывает гонку, то общий результат получается правильным. Если нет, то результат получается неверным. Такие ошибки чрезвычайно трудно отлаживать, потому что часто невозможно протестировать каждое чередование. Вы можете запустить код миллиард раз и не столкнуться с ошибкой, но следующий же запуск приведет к чередованию, в котором возникнет ошибка.

**Листинг 8.17.** Программа командной оболочки с трудноуловимой ошибкой синхронизации. Если потомок завершится раньше, чем родитель вернется из вызова `fork`, то функции `addjob` и `deletejob` будут вызваны в неправильном порядке

*code/ecf/procmask1.c*

```

1 /* ВНИМАНИЕ: Эта программа содержит ошибку! */
2 void handler(int sig)
3 {
4     int olderrno = errno;
5     sigset_t mask_all, prev_all;
6     pid_t pid;
7
8     Sigfillset(&mask_all);
9     /* Утилизировать зомби потомка */
10    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
11        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
12        deletejob(pid); /* Удалить потомка из списка заданий */
13        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
14    }
15    if (errno != ECHILD)
16        Sio_error("waitpid error");
17    errno = olderrno;
18 }
19
20 int main(int argc, char **argv)
21 {
22     int pid;
23     sigset_t mask_all, prev_all;
24
25     Sigfillset(&mask_all);
26     Signal(SIGCHLD, handler);
27     initjobs(); /* Инициализировать список заданий */
28
29     while (1) {
30         if ((pid = Fork()) == 0) { /* Дочерний процесс */
31             Execve("/bin/date", argv, NULL);
32         }
33         Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Родительский процесс */
34         addjob(pid); /* Добавить потомка в список заданий */
35         Sigprocmask(SIG_SETMASK, &prev_all, NULL);
36     }
37     exit(0);
38 }

```

*code/ecf/procmask1.c*

В листинге 8.18 показано одно из решений, устраняющих гонку из программы в листинге 8.17. Блокируя сигналы `SIGCHLD` перед вызовом `fork` и разблокируя их только после вызова `addjob`, можно гарантировать, что потомок будет утилизироваться только после добавления в список заданий. Обратите внимание, что потомки наследуют маску сигналов от своих родителей, поэтому нужно быть осторожными и не забыть разблокировать сигнал `SIGCHLD` в потомке перед вызовом `execve`.

**Листинг 8.18.** Использование `sigprocmask` для синхронизации процессов.

В этом примере родитель гарантирует, что `addjob` выполнится раньше, чем `deletejob`

*code/ecf/procmask2.c*

```

1 void handler(int sig)
2 {

```

```

3  int olderrno = errno;
4  sigset_t mask_all, prev_all;
5  pid_t pid;
6  Sigfillset(&mask_all);
7  while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
8      /* Утилизировать зомби потомка */
9      Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
10     deletejob(pid); /* Удалить потомка из списка заданий */
11     Sigprocmask(SIG_SETMASK, &prev_all, NULL);
12 }
13 if (errno != ECHILD)
14     Sio_error("waitpid error");
15 errno = olderrno;
16 }
17
18 int main(int argc, char **argv)
19 {
20     int pid;
21     sigset_t mask_all, mask_one, prev_one;
22
23     Sigfillset(&mask_all);
24     Sigemptyset(&mask_one);
25     Sigaddset(&mask_one, SIGCHLD);
26     Signal(SIGCHLD, handler);
27     initjobs(); /* Инициализировать список заданий */
28
29     while (1) {
30         Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Заблокировать SIGCHLD */
31         if ((pid = Fork()) == 0) { /* Дочерний процесс */
32             Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Разблокировать SIGCHLD */
33             Execve("/bin/date", argv, NULL);
34         }
35         Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Родительский процесс */
36         addjob(pid); /* Добавить потомка в список заданий */
37         Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Разблокировать SIGCHLD */
38     }
39     exit(0);
40 }

```

*code/ecf/procmask2.c*

### 8.5.7. Явное ожидание сигналов

Иногда основная программа должна явно дожидаться запуска определенного обработчика сигнала. Например, когда командная оболочка Linux создает задание переднего плана, она должна дождаться его завершения и обработки обработчиком сигнала SIGCHLD, прежде чем принять следующую пользовательскую команду.

В листинге 8.19 показана основная идея такого подхода. Родитель устанавливает обработчики для сигналов SIGINT и SIGCHLD, а затем входит в бесконечный цикл. Он блокирует SIGCHLD, чтобы избежать гонки между родителем и потомком, о которой мы говорили в разделе 8.5.6. После создания потомка родитель сбрасывает `pid` в ноль, разблокирует SIGCHLD и ожидает в цикле, пока `pid` не станет ненулевым. После завершения дочернего процесса обработчик утилизирует его и присваивает ненулевой PID глобальной переменной `pid`. После этого цикл ожидания завершается, и родитель выполняет дополнительную работу перед началом следующей итерации.

**Листинг 8.19.** Ожидание сигнала в цикле. Это корректный код, но цикл напрасно расходует ресурсы процессора

*code/ecf/waitforsignal.c*

```

1 #include "csapp.h"
2
3 volatile sig_atomic_t pid;
4
5 void sigchld_handler(int s)
6 {
7     int olderrno = errno;
8     pid = waitpid(-1, NULL, 0);
9     errno = olderrno;
10 }
11
12 void sigint_handler(int s)
13 {
14 }
15
16 int main(int argc, char **argv)
17 {
18     sigset_t mask, prev;
19
20     Signal(SIGCHLD, sigchld_handler);
21     Signal(SIGINT, sigint_handler);
22     Sigemptyset(&mask);
23     Sigaddset(&mask, SIGCHLD);
24
25     while (1) {
26         Sigprocmask(SIG_BLOCK, &mask, &prev); /* Заблокировать SIGCHLD */
27         if (Fork() == 0) /* Потомок */
28             exit(0);
29
30         /* Родитель */
31         pid = 0;
32         Sigprocmask(SIG_SETMASK, &prev, NULL); /* Разблокировать SIGCHLD */
33
34         /* Ждать сигнала SIGCHLD (напрасно расходуя ресурсы) */
35         while (!pid)
36             ;
37
38         /* Выполнить некоторую работу после получения SIGCHLD */
39         printf(".");
40     }
41     exit(0);
42 }

```

*code/ecf/waitforsignal.c*

В целом это верное решение, но цикл ожидания понапрасну расходует ресурсы процессора, и у нас может возникнуть соблазн исправить этот недостаток, вставив вызов `pause` в тело цикла:

```

while (!pid) /* Гонка! */
    pause();

```

Обратите внимание, что нам все еще нужен цикл, потому что ожидание в `pause` может быть прервано получением одного или нескольких сигналов `SIGINT`. Однако этот код имеет серьезную проблему с состоянием гонки: если сигнал `SIGCHLD` будет получен после проверки `while`, но до вызова `pause`, то `pause` заблокирует процесс навечно.

Другой вариант – заменить функцию `pause` функцией `sleep`:

```
while (!pid) /* Слишком медленно! */
    sleep(1);
```

Это корректный код, но он слишком медленный. Если сигнал будет получен после `while`, но до вызова `sleep`, то программа будет вынуждена ждать (относительно) долгое время, прежде чем сможет снова проверить условие завершения цикла. Использование функции `sleep` с более высоким разрешением, например в несколько наносекунд, тоже неприемлемо, потому что не существует хорошего правила для определения интервала ожидания. Если сделать его слишком коротким, то цикл станет слишком расточительным. Если сделать его слишком долгим, то программа будет работать слишком медленно.

Правильное решение – использовать `sigsuspend`.

```
#include <signal.h>
```

```
int sigsuspend(const sigset_t *mask);
```

Возвращает -1

Функция `sigsuspend` временно заменяет текущий набор заблокированных сигналов маской `mask`, а затем приостанавливает процесс до получения сигнала, действие по умолчанию которого заключается в запуске обработчика или в завершении процесса. Если по умолчанию процесс должен завершиться, то он завершается без возврата из `sigsuspend`. Если по умолчанию должен запуститься обработчик, то `sigsuspend` возвращается после возврата обработчика, восстанавливая заблокированный набор в состояние, имевшее место на момент вызова `sigsuspend`.

Функция `sigsuspend` эквивалентна атомарной (непрерываемой) версии следующего кода:

```
1 sigprocmask(SIG_SETMASK, &mask, &prev);
2 pause();
3 sigprocmask(SIG_SETMASK, &prev, NULL);
```

Свойство атомарности гарантирует, что вызовы `sigprocmask` (строка 1) и `pause` (строка 2) будут выполнены последовательно без перерыва. Это устраняет потенциальную гонку, когда сигнал принимается после вызова `sigprocmask` и до вызова `pause`.

В листинге 8.20 показано, как можно использовать `sigsuspend` для замены цикла ожидания в листинге 8.19. Перед каждым вызовом `sigsuspend` программа блокирует `SIGCHLD`. Функция `sigsuspend` временно разблокирует его, а затем приостанавливается, пока родитель не получит сигнал. Перед возвратом `sigsuspend` восстанавливает прежний набор заблокированных сигналов и тем самым вновь блокирует `SIGCHLD`. Если родитель получил `SIGINT`, то проверка цикла завершится успехом, и следующая итерация снова вызовет `sigsuspend`. Если родитель получил `SIGCHLD`, то проверка цикла завершится неудачей и произойдет выход из цикла. В этот момент `SIGCHLD` заблокирован, поэтому нужно дополнительно разблокировать его. Этот прием может пригодиться в настоящей командной оболочке с фоновыми заданиями, которые необходимо утилизировать.

**Листинг 8.20.** Ожидание сигнала с использованием `sigsuspend`

*code/ecf/sigsuspend.c*

```
1 #include "csapp.h"
2
3 volatile sig_atomic_t pid;
4
5 void sigchld_handler(int s)
```

```

6 {
7     int olderrno = errno;
8     pid = Waitpid(-1, NULL, 0);
9     errno = olderrno;
10 }
11
12 void sigint_handler(int s)
13 {
14 }
15
16 int main(int argc, char **argv)
17 {
18     sigset_t mask, prev;
19
20     Signal(SIGCHLD, sigchld_handler);
21     Signal(SIGINT, sigint_handler);
22     Sigemptyset(&mask);
23     Sigaddset(&mask, SIGCHLD);
24
25     while (1) {
26         Sigprocmask(SIG_BLOCK, &mask, &prev); /* Заблокировать SIGCHLD */
27         if (Fork() == 0) /* Потомок */
28             exit(0);
29
30         /* Ждать сигнала SIGCHLD */
31         pid = 0;
32         while (!pid)
33             sigsuspend(&prev);
34
35         /* Если необходимо, разблокировать SIGCHLD */
36         Sigprocmask(SIG_SETMASK, &prev, NULL);
37
38         /* Выполнить некоторую работу после получения SIGCHLD */
39         printf(".");
40     }
41     exit(0);
42 }

```

*code/ecf/sigsuspend.c*

Версия с `sigsuspend` менее расточительна, чем исходный цикл ожидания, позволяет избежать гонки, вызванной функцией `pause`, и более эффективна, чем версия с функцией `sleep`.

## 8.6. Нелокальные переходы

Язык С предоставляет пользователям возможность передачи управления по исключению, которая называется *нелокальные переходы*. Механизм нелокальных переходов позволяет передать управление непосредственно из одной функции в другую без необходимости выполнять стандартную последовательность действий «вызов и возврат». Нелокальные переходы реализованы в виде функций `setjmp` и `longjmp`.

```
#include <setjmp.h>
```

```
int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
```

Возвращает: 0 из `setjmp`, ненулевое значение из `longjmp`



Функция `setjmp` сохраняет текущее *окружение вызова* в буфере `env` для последующего использования в `longjmp` и возвращает 0. Окружение вызова включает счетчик инструкций, указатель стека и регистры общего назначения. По некоторым тонким причинам, описание которых выходит за рамки нашего обсуждения, значение, возвращаемое функцией `setjmp`, не должно присваиваться переменной:

```
rc = setjmp(env); /* Неправильно! */
```

Однако его можно использовать для проверки в операторе `switch` или в условном выражении [62].

```
#include <setjmp.h>
```

```
void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
```

Никогда не возвращается

Функция `longjmp` восстанавливает окружение вызова из буфера `env` и затем инициирует возврат из самого последнего вызова `setjmp`, который инициализировал `env`. После этого `setjmp` возвращает ненулевое значение `retval`.

На первых порах взаимодействия между `setjmp` и `longjmp` могут показаться запутанными и непонятными. Функция `setjmp` вызывается один раз, но возвращает управление *несколько раз*: один раз, когда `setjmp` вызывается для сохранения окружения вызова в буфере `env`, и по одному разу для каждого вызова `longjmp`. Функция `longjmp`, напротив, вызывается один раз, но никогда не возвращает управления.

Нелокальные переходы используются для непосредственного возврата из глубоко вложенных вызовов функций, обычно после обнаружения некоторой ошибки. Если ошибка обнаруживается в глубоко вложенном вызове функции, то с помощью нелокального перехода можно вернуться сразу в обработчик ошибок вместо утомительного раскручивания стека вызовов.

В листинге 8.21 показан пример, как это реализовать. Функция `main` сначала вызывает `setjmp`, чтобы сохранить текущее окружение вызова, а затем вызывает функцию `foo`, которая в свою очередь вызывает функцию `bar`. Если в `foo` или в `bar` возникнет ошибка, то они вернут управление непосредственно из `setjmp` через вызов `longjmp`. Ненулевое значение, возвращаемое из `setjmp`, сообщает тип ошибки, который можно расшифровать и обработать в одном месте в коде.

Функция `longjmp`, позволяющая пропускать все промежуточные вызовы, может иметь непредвиденные последствия. Например, если в промежуточных вызовах выделялась память, которую предполагалось освободить в конце, то код, освобождающий память, не выполнится, что приведет к утечке памяти.

**Листинг 8.21.** Пример нелокального перехода. В этом примере показан порядок использования нелокальных переходов для восстановления после ошибок в глубоко вложенных функциях без необходимости раскручивать весь стек

*code/ecf/setjmp.c*

```
1 #include "csapp.h"
2
3 jmp_buf buf;
4
5 int error1 = 0;
6 int error2 = 1;
7
```

```

8 void foo(void), bar(void);
9
10 int main()
11 {
12     switch(setjmp(buf)) {
13         case 0:
14             foo();
15             break;
16         case 1:
17             printf("Detected an error1 condition in foo\n");
18             break;
19         case 2:
20             printf("Detected an error2 condition in foo\n");
21             break;
22         default:
23             printf("Unknown error condition in foo\n");
24     }
25     exit(0);
26 }
27
28 /* Глубоко вложенная функция foo */
29 void foo(void)
30 {
31     if (error1)
32         longjmp(buf, 1);
33     bar();
34 }
35
36 void bar(void)
37 {
38     if (error2)
39         longjmp(buf, 2);
40 }

```

*code/ecf/setjmp.c*

Другое важное приложение нелокальных переходов – выход из обработчика сигнала по заданному адресу в программе вместо возврата к прерванной команде. В листинге 8.22 показана простая программа, иллюстрирующая этот стандартный прием. Программа использует сигналы и нелокальные переходы, позволяющие производить теплый рестарт всякий раз, когда пользователь нажимает комбинацию **Ctrl+C**. Функции `sigsetjmp` и `siglongjmp` – это версии функций `setjmp` и `longjmp`, которые могут использоваться обработчиками сигналов.

**Листинг 8.22.** Программа, использующая нелокальные переходы для перезапуска себя, когда пользователь нажимает комбинацию **Ctrl+C**

*code/ecf/restart.c*

```

1 #include "csapp.h"
2
3 sigjmp_buf buf;
4
5 void handler(int sig)
6 {
7     siglongjmp(buf, 1);
8 }
9

```

```

10 int main()
11 {
12     if (!sigsetjmp(buf, 1)) {
13         Signal(SIGINT, handler);
14         Sio_puts("starting\n");
15     }
16     else
17         Sio_puts("restarting\n");
18
19     while(1) {
20         Sleep(1);
21         Sio_puts("processing...\n");
22     }
23     exit(0); /* Управление никогда не достигает этой точки */
24 }

```

*code/ecf/restart.c*

Самый первый вызов функции `sigsetjmp` сохраняет окружение вызова и контекст сигнала (включая векторы ожидающих и заблокированных сигналов). Затем функция `main` входит в бесконечный цикл. Если пользователь нажмет комбинацию **Ctrl+C**, то ядро отправит сигнал `SIGINT` процессу, а тот перехватит его. Вместо того простого возврата в прерванный цикл обработчик сигнала выполняет нелокальный переход обратно в начало функции `main`. Когда мы запустили эту программу в своей системе, то получили следующий вывод:

```

linux> ./restart
starting
processing...
processing...
Ctrl+C
restarting
processing...
Ctrl+C
restarting
processing...

```

В этой программе есть еще пара интересных аспектов. Во-первых, чтобы избежать гонки, обработчик следует устанавливать после вызова `sigsetjmp`. Иначе есть риск, что обработчик запустится до первоначального вызова `sigsetjmp`, который установит окружение вызова для `siglongjmp`. Во-вторых, обратите внимание, что функции `sigsetjmp` и `siglongjmp` отсутствуют в списке функций, безопасных в асинхронном окружении в табл. 8.4. Причина в том, что `siglongjmp` может переходить в произвольный код, поэтому нужно быть осторожными и в любом коде, доступном из `siglongjmp`, вызывать только безопасные функции. В нашем примере мы вызываем безопасные функции `sio_puts` и `sleep`. Небезопасная функция `exit` недоступна.

### Программные исключения в C++ и Java

Механизмы исключений, предоставляемые языками C++ и Java, – это более высокоуровневые и структурированные версии функций `setjmp` и `longjmp` в языке C. Ветвь `catch` внутри оператора `try` можно рассматривать как некий аналог функции `setjmp`. Точно так же можно провести параллели между оператором `throw` и функцией `longjmp`.

## 8.7. Инструменты управления процессами

Системы Linux предоставляют множество удобных инструментов управления процессами:

STRACE. Выводит трассировку каждого системного вызова, произведенного программой, и ее дочерними процессами. Это увлекательнейшее времяпрепровождение для любопытных студентов. Скомпилируйте вашу программу с параметром `-static`, чтобы получить более чистую трассировку без вывода информации о системных вызовах, выполненных разделяемыми библиотеками;

PS. Выводит список процессов (включая зомби), присутствующих в настоящее время в системе;

TOP. Выводит информацию об использовании ресурсов текущими процессами;

PMAP. Отображает раскладку памяти процесса;

/proc. Виртуальная файловая система, открывающая доступ к большому количеству структур данных в ядре в текстовом формате, доступная для чтения пользовательским программам. Например, введите команду `cat /proc/loadavg`, чтобы увидеть текущую среднюю нагрузку на вашу систему Linux.

## 8.8. Итоги

Передача управления по исключению (Exceptional Control Flow, ECF) может происходить на всех уровнях компьютерной системы.

На аппаратном уровне исключения – это непредвиденные переходы в потоке управления, которые вызваны событиями в процессоре. Поток управления передается программному обработчику, который выполняет некоторые действия и возвращает управление прерванному потоку.

Всего имеется четыре типа исключений: аппаратные прерывания, ошибки, аварийные завершения и системные прерывания. Аппаратные прерывания происходят асинхронно, если внешнее устройство ввода/вывода, например микросхема таймера или контроллер диска, подает сигнал прерывания на вывод процессора. Управление возвращается инструкции, следующей за инструкцией, при выполнении которой возникло прерывание. Ошибки и аварийные завершения происходят синхронно, как результат выполнения инструкции. Обработчики ошибок возобновляют выполнение команды, вызвавшей ошибку, тогда как обработчики аварийного завершения никогда не возвращают управление прерванному потоку. Наконец, системные прерывания подобны вызовам функций и используются для реализации системных вызовов, обеспечивающих приложения управляемыми точками входа в программный код операционной системы.

На уровне операционной системы ядро использует передачу управления по прерываниям для реализации фундаментальной концепции процесса. Процессы предоставляют приложениям две важные абстракции: (1) логических потоков управления, которые создают иллюзию, что каждая программа монопольно использует процессор, и (2) изолированные адресные пространства, создающие иллюзию, что каждая программа монопольно использует оперативную память.

Используя интерфейс с операционной системой, приложения могут создавать дочерние процессы, ждать их остановки или завершения, запускать новые программы, а также перехватывать сигналы, посылаемые другими процессами. Семантика обработки сигналов – это довольно тонкая материя, которая может изменяться от системы к системе. Тем не менее в POSIX-совместимых системах имеются механизмы, дающие возможность программам четко определять ожидаемую семантику обработки сигнала.

Наконец, на прикладном уровне программы на С могут использовать нелокальные переходы, позволяющие передавать управление непосредственно из одной функции в другую в обход стандартного механизма управления стеком.

## Библиографические заметки

Важным справочником по всем аспектам программирования в среде Linux является книга Керриска (Kerrisk) [62]. Спецификация Intel ISA содержит подробное описание исключений и прерываний в процессорах Intel [50]. Дополнительную информацию об исключениях, процессах и сигналах можно найти в книгах об операционных системах [102, 106, 113]. Классический труд У. Ричарда Стивенса (W. Richard Stevens) [111], несмотря на его давность, продолжает оставаться ценным и очень подробным руководством по работе с процессами и сигналами из прикладных программ. Удивительно ясное описание ядра Linux, включая детали реализации процессов и сигналов, можно найти в книге Бовета (Bovet) и Цезати (Cesati) [11].

## Домашние задания

### Упражнение 8.9 ♦

Представьте, что имеется четыре процесса. В следующей таблице приводятся время-на их запуска и завершения:

Процесс	Время запуска	Время завершения
A	5	7
B	2	4
C	3	6
D	1	8

Для каждой пары процессов укажите, выполняются они конкурентно (Да) или нет (Нет):

Пара процессов	Конкурентны?
AB	_____
AC	_____
AD	_____
BC	_____
BD	_____
CD	_____

### Упражнение 8.10 ♦

В этой главе мы ввели некоторые функции с необычным поведением при вызове и возврате: `setjmp`, `longjmp`, `execve` и `fork`. Сопоставьте каждой из этих функций одно из соответствующих ей поведений:

1. Вызывается один раз, возвращает управление дважды.
2. Вызывается один раз, но никогда не возвращает управления.
3. Вызывается один раз, возвращает управление один или несколько раз.

### Упражнение 8.11 ♦

Сколько раз эта программа выведет строку «hello»?

*code/ecf/forkprob1.c*

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int i;
6
7     for (i = 0; i < 2; i++)
8         Fork();
9     printf("hello\n");
10    exit(0);
11 }

```

*code/ecf/forkprob1.c*

### Упражнение 8.12 ♦

Сколько раз эта программа выведет строку «hello»?

*code/ecf/forkprob4.c*

```

1 #include "csapp.h"
2
3 void doit()
4 {
5     Fork();
6     Fork();
7     printf("hello\n");
8     return;
9 }
10
11 int main()
12 {
13     doit();
14     printf("hello\n");
15     exit(0);
16 }

```

*code/ecf/forkprob4.c*

### Упражнение 8.13 ♦

Приведите один из возможных выводов следующей программы:

*code/ecf/forkprob3.c*

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int x = 3;
6
7     if (Fork() != 0)
8         printf("x=%d\n", ++x);
9
10    printf("x=%d\n", --x);
11    exit(0);
12 }

```

*code/ecf/forkprob3.c*

### Упражнение 8.14 ♦

Сколько раз эта программа выведет строку «hello»?

*code/ecf/forkprob5.c*

```

1 #include "csapp.h"

```

```
2
3 void doit()
4 {
5     if (Fork() == 0) {
6         Fork();
7         printf("hello\n");
8         exit(0);
9     }
10    return;
11 }
12
13 int main()
14 {
15     doit();
16     printf("hello\n");
17     exit(0);
18 }
```

---

*code/ecf/forkprob5.c*

### Упражнение 8.15. ♦

Сколько раз эта программа выведет строку «hello»?

---

*code/ecf/forkprob6.c*

```
1 #include "csapp.h"
2
3 void doit()
4 {
5     if (Fork() == 0) {
6         Fork();
7         printf("hello\n");
8         return;
9     }
10    return;
11 }
12
13 int main()
14 {
15     doit();
16     printf("hello\n");
17     exit(0);
18 }
```

---

*code/ecf/forkprob6.c*

### Упражнение 8.16 ♦

Что выведет следующая программа?

---

*code/ecf/forkprob7.c*

```
1 #include "csapp.h"
2 int counter = 1;
3
4 int main()
5 {
6     if (fork() == 0) {
7         counter--;
8         exit(0);
9     }
10    else {
11        Wait(NULL);
12        printf("counter = %d\n", ++counter);
13    }
```

```

13     }
14     exit(0);
15 }

```

[code/ecf/forkprob7.c](#)

### Упражнение 8.17 ♦

Перечислите все возможные строки, которые выводит программа из упражнения 8.4.

### Упражнение 8.18 ♦♦

Взгляните на следующую программу:

```

1 #include "csapp.h"
2
3 void end(void)
4 {
5     printf("2"); fflush(stdout);
6 }
7
8 int main()
9 {
10     if (Fork() == 0)
11         atexit(end);
12     if (Fork() == 0) {
13         printf("0"); fflush(stdout);
14     }
15     else {
16         printf("1"); fflush(stdout);
17     }
18     exit(0);
19 }

```

[code/ecf/forkprob2.c](#)

[code/ecf/forkprob2.c](#)

Определите, который из следующих результатов может иметь место. *Обратите внимание:* функция `atexit` принимает указатель на функцию и добавляет его в список функций (изначально пустой), которые будут вызываться при вызове функции `exit`.

1. 112002
2. 211020
3. 102120
4. 122001
5. 100212

### Упражнение 8.19 ♦♦

Сколько строк выведет следующая функция? Дайте ответ как функцию от  $n$ . Предположим, что  $n \geq 1$ .

```

1 void foo(int n)
2 {
3     int i;
4
5     for (i = 0; i < n; i++)
6         Fork();
7     printf("hello\n");
8     exit(0);
9 }

```

[code/ecf/forkprob8.c](#)

[code/ecf/forkprob8.c](#)



### Упражнение 8.20 ♦♦

Используя `execve`, напишите программу `mys`, действующую подобно программе `/bin/ls`. Ваша программа должна принимать те же самые аргументы командной строки, интерпретировать те же переменные окружения и производить идентичный выход.

Программа `ls` получает ширину экрана из переменной окружения `COLUMNS`. Если `COLUMNS` не установлена, то `ls` предполагает, что экран имеет 80 столбцов в ширину. Таким образом, путем записи в `COLUMNS` значения меньше 80 можно управлять выводом:

```
linux> setenv COLUMNS 40
linux> ./mys
.
. // Вывод занимает 40 столбцов в ширину
.

linux> unsetenv COLUMNS
linux> ./mys
.
. // Теперь вывод занимает 80 столбцов в ширину
.
```

### Упражнение 8.21 ♦♦

Определите возможные варианты вывода следующей программы.

*code/ecf/waitprob3.c*

```
1 int main()
2 {
3     if (fork() == 0) {
4         printf("a"); fflush(stdout);
5         exit(0);
6     }
7     else {
8         printf("b"); fflush(stdout);
9         waitpid(-1, NULL, 0);
10    }
11    printf("c"); fflush(stdout);
12    exit(0);
13 }
```

*code/ecf/waitprob3.c*

### Упражнение 8.22 ♦♦♦

Напишите свою версию функции `system`:

```
int mysystem(char *command);
```

Она должна выполнять команду `command`, вызывая `/bin/sh -c command`, и по завершении выполнения команды возвращать управление. Если команда `command` завершается как обычно (вызовом функции `exit` или выполнением оператора `return`), то `mysystem` должна вернуть код завершения `command`. Например, если `command` завершается вызовом `exit(8)`, то `mysystem` должна вернуть значение 8. Иначе, если `command` завершается ненормально, то `mysystem` должна вернуть значение, возвращаемое командной оболочкой.

### Упражнение 8.23 ♦♦

Один из ваших коллег задумал использовать сигналы, чтобы реализовать в родительском процессе подсчет событий, возникающих в дочернем процессе. Идея в том, чтобы потомок уведомлял родителя о каждом событии, посылая сигнал, обработчик которого

в родительском процессе будет увеличивать глобальную переменную `counter`, и по завершении потомка выводить ее значение. Он написал программу, показанную в листинге 8.23, но когда запустил ее, то обнаружил, что родительский процесс всегда выводит значение 2, несмотря на то что дочерний процесс послал пять сигналов. Озадаченный, он пришел к вам за помощью. Сможете ли вы объяснить, в чем состоит его ошибка?

**Листинг 8.23.** Программа подсчета событий для упражнения 8.23

*code/ecf/counterprob.c*

```

1 #include "csapp.h"
2
3 int counter = 0;
4
5 void handler(int sig)
6 {
7     counter++;
8     sleep(1); /* Выполнить некоторую работу в обработке */
9     return;
10 }
11
12 int main()
13 {
14     int i;
15
16     Signal(SIGUSR2, handler);
17
18     if (Fork() == 0) { /* Потомок */
19         for (i = 0; i < 5; i++) {
20             Kill(getppid(), SIGUSR2);
21             printf("sent SIGUSR2 to parent\n");
22         }
23         exit(0);
24     }
25
26     Wait(NULL);
27     printf("counter=%d\n", counter);
28     exit(0);
29 }
```

*code/ecf/counterprob.c*

### Упражнение 8.24 ♦♦♦

Измените программу в листинге 8.4 так, чтобы она соответствовала следующим двум условиям:

1. Каждый дочерний процесс завершается ненормально, попытавшись выполнить запись в сегмент кода, доступный только для чтения.
2. Родительский процесс производит вывод, идентичный следующему (за исключением PID):

```

child 12255 terminated by signal 11: Segmentation fault
child 12254 terminated by signal 11: Segmentation fault
```

*Подсказка:* прочитайте страницу справочного руководства `man 3 psignal`.

### Упражнение 8.25 ♦♦♦

Напишите версию функции `fgets`, называющуюся `tfgets`, которая ожидает ввода в течение 5 секунд. Функция `tfgets` должна принимать те же параметры, что и `fgets`. Если

пользователь ничего не ввел в течение 5 секунд, то `tfgets` должна вернуть `NULL`, иначе – указатель на введенную строку.

### Упражнение 8.26 ♦♦♦♦

Взяв за основу листинг 8.15, напишите программу командной оболочки с поддержкой управления заданиями. Ваша командная оболочка должна обладать следующими свойствами:

- команда, вводимая пользователем, включает имя и (возможно) аргументы, перечисленные через пробелы. Если пользователь ввел имя встроенной команды, то командная оболочка должна сразу же выполнить ее и ждать ввода следующей команды. Иначе она должна предположить, что имя определяет выполняемый файл, который следует загрузить и выполнить в контексте дочернего процесса (задания). Идентификатор группы процессов для задания должен совпадать с PID дочернего процесса;
- каждое задание идентифицируется идентификатором процесса (PID) или идентификатором задания (JID) – небольшим положительным целым числом, назначенным командной оболочкой. Идентификаторы JID обозначаются в командной строке префиксом `%`. Например, `%5` обозначает JID 5, а 5 – PID 5;
- если команда заканчивается амперсандом (`&`), то командная оболочка должна запустить задание в фоновом режиме, иначе задание должно выполняться на переднем плане;
- нажатие комбинации **Ctrl+C** (**Ctrl+Z**) вынуждает ядро послать сигнал `SIGINT` (`SIGTSTP`) командной оболочке. В ответ на это она должна послать тот же сигнал каждому процессу в группе процессов переднего плана<sup>2</sup>;
- встроенная команда `jobs` должна выводить список всех фоновых заданий;
- встроенная команда `bg job` должна возобновлять приостановленное задание `job`, посылая ему сигнал `SIGCONT` и переводя его в фоновый режим. В аргументе `job` может передаваться PID или JID;
- встроенная команда `fg job` должна возобновлять приостановленное задание `job`, посылая ему сигнал `SIGCONT` и переводя его в выполнение на переднем плане;
- командная оболочка должна утилизировать дочерние процессы после их завершения. Если любое задание завершается из-за того, что был получен сигнал, который не перехватывается им, то командная оболочка должна вывести в терминал сообщение с PID задания и описанием сигнала.

В листинге 8.24 показан пример сеанса работы командной оболочки.

#### Листинг 8.24. Пример сеанса работы командной оболочки для упражнения 8.26

```
linux> ./shell                Запуск программы командной оболочки
>bogus
bogus: Command not found.    Execve не нашла выполняемый файл
>foo 10
Job 5035 terminated by signal: Interrupt    Пользователь нажал Ctrl+C
>foo 100 &
[1] 5036 foo 100 &
```

<sup>2</sup> Обратите внимание, что это упрощенный вариант работы реальных командных оболочек. В реальных оболочках ядро реагирует на комбинацию **Ctrl+C** (**Ctrl+Z**), посылая `SIGINT` (`SIGTSTP`) непосредственно каждому процессу в терминальной группе процессов переднего плана. Оболочка управляет членством в этой группе, используя функцию `tcsetpgrp`, и атрибутами терминала, используя функцию `tcsetattr`, обсуждение которых выходит за рамки этой книги. Подробности ищите в [62].

```

>foo 200 &
[2] 5037 foo 200 &
>jobs
[1] 5036 Running   foo 100 &
[2] 5037 Running   foo 200 &
>fg %1
Job [1] 5036 stopped by signal: Stopped      Пользователь нажал Ctrl+Z
>jobs
[1] 5036 Stopped   foo 100 &
[2] 5037 Running   foo 200 &
>bg 5035
5035: No such process
>bg 5036
[1] 5036   foo 100 &
>/bin/kill 5036
Job 5036 terminated by signal: Terminated
> fg %2
Ожидает завершения задания
>quit
Возврат в командную оболочку Unix
linux>

```

## Решения упражнений

### Решение упражнения 8.1

Процессы А и В выполняются конкурентно по отношению друг к другу, так же как процессы В и С, потому что выполняются, перекрываясь во времени, то есть один процесс запускается до завершения другого. Процессы А и С выполняются неконкурентно, потому что не перекрываются во времени: процесс А завершается раньше, чем запускается С.

### Решение упражнения 8.2

В нашем примере программы в листинге 8.2 родительский и дочерний процессы выполняют непересекающиеся наборы инструкций. Однако в этой программе родительская и дочерняя программы выполняют пересекающиеся наборы инструкций, что вполне возможно, потому что родитель и потомок имеют идентичные сегменты кода. Это упражнение может показаться сложным концептуально, поэтому постарайтесь разобраться в его решении. На рис. 8.19 показан граф процессов.

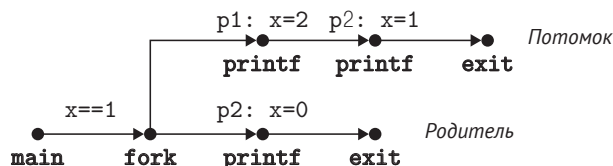


Рис. 8.19. Граф процессов для упражнения 8.2

1. Дело в том, что дочерний процесс выполнит оба оператора printf. После того как fork вернет управление, потомок выполнит printf в строке 6. После этого потомок выйдет из оператора if и выполнит printf в строке 7. Вот вывод дочернего процесса:

```

p1: x=2
p2: x=1

```

2. Родительский процесс выполнит только printf в строке 7:

```

p2: x=0

```

### Решение упражнения 8.3

Мы знаем, что возможны последовательности *acbc*, *abcc* и *bacc*, потому что они соответствуют топологическим сортировкам графа процессов (рис. 8.20). Однако такие последовательности, как *bcac* и *cbca*, не соответствуют топологическим сортировкам и, следовательно, невозможны.

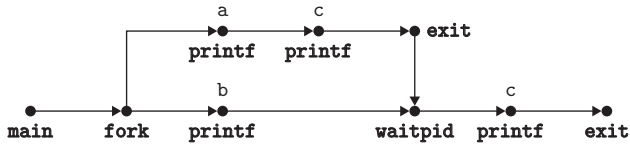


Рис. 8.20. Граф процессов для упражнения 8.3

### Решение упражнения 8.4

1. Мы можем определить количество строк вывода, просто подсчитав количество вершин `printf` в графе процессов (рис. 8.21). В данном случае таких вершин шесть, поэтому программа выведет шесть строк.

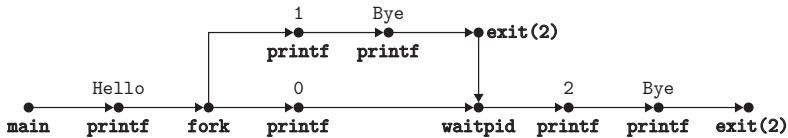


Рис. 8.21. Граф процессов для упражнения 8.4

2. Строки могут выводиться в любом порядке, соответствующем топологической сортировке графа. Например, возможен такой вариант:

Hello, 1, 0, Bye, 2, Bye.

### Решение упражнения 8.5

[code/ecf/snooze.c](#)

```

1 unsigned int snooze(unsigned int secs) {
2     unsigned int rc = sleep(secs);
3
4     printf("Slept for %d of %d secs.\n", secs-rc, secs);
5     return rc;
6 }
  
```

[code/ecf/snooze.c](#)

### Решение упражнения 8.6

[code/ecf/myecho.c](#)

```

1 #include "csapp.h"
2
3 int main(int argc, char *argv[], char *envp[])
4 {
5     int i;
6
7     printf("Command-line arguments:\n");
8     for (i=0; argv[i] != NULL; i++)
9         printf(" argv[%2d]: %s\n", i, argv[i]);
10
11     printf("\n");
12     printf("Environment variables:\n");
  
```

```

13     for (i=0; envp[i] != NULL; i++)
14         printf(" envp[%2d]: %s\n", i, envp[i]);
15
16     exit(0);
17 }

```

*code/ecf/myecho.c*

## Решение упражнения 8.7

Функция `sleep` преждевременно возвращает управление всякий раз, когда приостановленному процессу посылается неигнорируемый сигнал. Поскольку по умолчанию `SIGINT` предполагает завершение процесса (табл. 8.4), необходимо установить обработчик `SIGINT`, чтобы дать возможность вернуться из функции `sleep`. Обработчик просто перехватывает сигнал `SIGINT` и возвращает управление функции `sleep`, которая тут же возвращает управление вызвавшему ее процессу.

*code/ecf/snooze.c*

```

1 #include "csapp.h"
2
3 /* Обработчик SIGINT */
4 void handler(int sig)
5 {
6     return; /* Перехватить сигнал и вернуть управление */
7 }
8
9 unsigned int snooze(unsigned int secs) {
10     unsigned int rc = sleep(secs);
11
12     printf("Slept for %d of %d secs.\n", secs-rc, secs);
13     return rc;
14 }
15
16 int main(int argc, char **argv) {
17
18     if (argc != 2) {
19         fprintf(stderr, "usage: %s <secs>\n", argv[0]);
20         exit(0);
21     }
22
23     if (signal(SIGINT, handler) == SIG_ERR) /* Установить обработчик SIGINT */
24         unix_error("signal error\n"); /* Обработчик */
25     (void)snooze(atoi(argv[1]));
26     exit(0);
27 }

```

*code/ecf/snooze.c*

## Решение упражнения 8.8

Эта программа выведет строку 213 – сокращенное название курса CS:APP в университете Карнеги–Меллона. Сначала родитель выведет «2», затем запустит дочерний процесс, выполняющий бесконечный цикл. Потом родитель отправит сигнал потомку и дожждется его завершения. Потомок перехватит сигнал (прервав бесконечный цикл), уменьшит счетчик (с начальным значением 2), выведет «1» и завершится. Когда родитель утилизирует зомби потомка, он увеличит счетчик (с начальным значением 2), выведет «3» и завершится.

## Виртуальная память

- 9.1. Физическая и виртуальная адресация.
- 9.2. Пространства адресов.
- 9.3. Виртуальная память как средство кеширования.
- 9.4. Виртуальная память как средство управления памятью.
- 9.5. Виртуальная память как средство защиты памяти.
- 9.6. Преобразование адресов.
- 9.7. Практический пример: система памяти Intel Core i7/Linux.
- 9.8. Отображение памяти.
- 9.9. Динамическое распределение памяти.
- 9.10. Сборка мусора.
- 9.11. Часто встречающиеся ошибки.
- 9.12. Итоги.

Библиографические заметки.

Домашние задания.

Решения упражнений.

Процессы в системе совместно используют центральный процессор и оперативную память. Однако совместное использование оперативной памяти различными процессами порождает некоторые специфические проблемы. С увеличением потребления вычислительных ресурсов процессора скорость выполнения процессов постепенно уменьшается. Но если слишком много процессов потребуют слишком много памяти, то некоторые из них просто не смогут выполняться. Если программе не хватит выделенной ей памяти, то это может повлечь за собой катастрофические последствия для программы. Кроме того, информация в памяти может повреждаться. Если один процесс по недосмотру запишет свои данные в память, используемую другим процессом, то выполнение того второго процесса может пойти по непредсказуемому пути, совершенно не соответствующая логике программы.

Для эффективного управления памятью с минимально возможным количеством ошибок современные системы используют абстракцию оперативной памяти, известную как *виртуальная память* (Virtual Memory, VM). Виртуальная память – это элегантная композиция взаимодействующих между собой аппаратных исключений, аппаратного преобразования адресов; основной памяти, файлов на диске и программного обеспечения ядра системы, которые обеспечивают каждый процесс большим, однородным и изолированным адресным пространством. Хорошо продуманный механизм виртуальной памяти обеспечивает три важные особенности: (1) эффективное исполь-

зование основной памяти как кеша для адресного пространства, хранимого на диске, когда в основной памяти находятся только активно используемые области, а остальные области перемещаются между диском и основной памятью по мере необходимости; (2) простота управления памятью, благодаря которой каждый процесс получает однородное адресное пространство; (3) защита адресного пространства каждого процесса от повреждения другими процессами.

Виртуальная память – одна из самых замечательных идей, реализованных в компьютерных системах. Главная причина успеха – в том, что она работает автоматически, без всякого вмешательства со стороны прикладного программиста. Но если виртуальная память работает так хорошо, оставаясь в тени, то зачем тогда программисту разбираться в ее устройстве? Для этого есть несколько причин.

- *Виртуальная память находится в центре событий.* Она пронизывает все уровни компьютерных систем, играя ключевую роль в организации аппаратных исключений, в работе ассемблеров, компоновщиков, загрузчиков, совместном использовании объектов, файлов и процессов. Знание устройства виртуальной памяти поможет вам лучше понимать, как вообще работают такие системы.
- *Виртуальная память обладает большими возможностями.* Она дает приложениям широкие возможности выделять и освобождать фрагменты памяти, отображать их в файлы на диске, совместно использовать несколькими процессами. Например, знаете ли вы, что можно читать или изменять содержимое файла на диске, читая и изменяя значения в памяти? Или что можно загрузить содержимое файла в память, не выполняя никаких явных операций копирования? Понимание устройства виртуальной памяти поможет вам использовать ее мощные возможности в ваших приложениях.
- *Виртуальная память таит в себе опасности.* Приложения взаимодействуют с виртуальной памятью всякий раз, когда ссылаются на переменную, разыменовывают указатель или вызывают инструменты динамического распределения памяти, такие как `malloc`. Неправильное использование виртуальной памяти может завести приложение в тупик. Например, используя ошибочный указатель, программа может потерпеть аварию с диагнозом «ошибка сегментации» или «ошибка защиты памяти», но может тихо выполняться в течение многих часов, прежде чем произойдет авария, или, что еще хуже, завершится как обычно, но с неправильными результатами. Знание основ виртуальной памяти и механизмов ее распределения, например семейства функций `malloc`, может помочь избежать таких ошибок.

В этой главе виртуальная память рассматривается с двух точек зрения. В первой половине главы рассказывается, как работает виртуальная память. Во второй половине мы опишем, как приложения используют виртуальную память и управляют ею. Мы не можем игнорировать тот факт, что виртуальная память – сложный механизм, и наше обсуждение будет отражать эту сложность. Однако в этом есть и свои плюсы: знание деталей механизма виртуальной памяти поможет вам самостоятельно смоделировать его для небольшой системы, а сама идея виртуальной памяти навсегда потеряет для вас свою мистическую ауру.

Во второй части главы мы будем предполагать, что вы хорошо усвоили основные понятия, представленные в первой половине, и покажем вам, как использовать и управлять виртуальной памятью в ваших программах. Вы узнаете, как явно управлять виртуальной памятью, используя механизмы отображения памяти и динамического распределения, такие как `malloc`. Мы также расскажем вам о множестве распространенных ошибок, обусловленных неправильным использованием памяти в программах на языке C, и как их не допустить.



## 9.1. Физическая и виртуальная адресация

Основная память компьютерной системы организована как массив из  $M$  последовательно расположенных однобайтных ячеек. Каждый байт имеет уникальный *физический адрес*. Первый байт имеет адрес 0, следующий за ним – адрес 1, следующий – адрес 2 и т. д. При такой простой организации памяти самый естественный способ доступа к памяти состоит в использовании физических адресов. Мы называем такой подход *физической адресацией*. На рис. 9.1 показан пример физической адресации в контексте инструкции загрузки, которая считает слово, начинающееся с физического адреса 4. Когда процессор (CPU) выполняет инструкцию загрузки, он генерирует эффективный физический адрес и передает его основной памяти по шине памяти. Основная память выбирает четырехбайтное слово, начиная физического адреса 4, и возвращает его процессору, который сохраняет это слово в регистре.

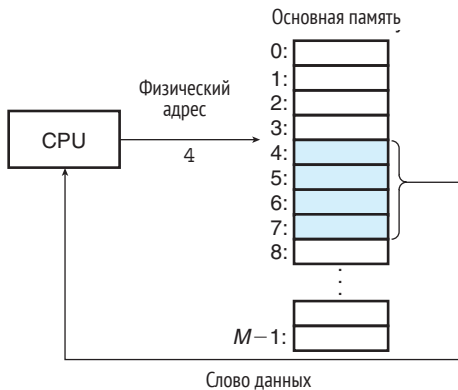


Рис. 9.1. Система с физической адресацией

Самые первые персональные компьютеры использовали физическую адресацию, и многие системы, такие как процессоры цифровых сигналов, встраиваемые микроконтроллеры, и супер-ЭВМ фирмы Cray продолжают использовать этот вид адресации. Однако современные процессоры используют форму адресации, известную как *виртуальная адресация* (рис. 9.2).

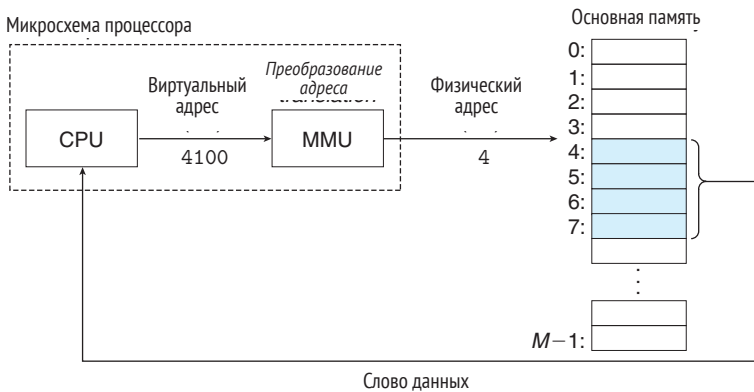


Рис. 9.2. Система, использующая виртуальную адресацию

Используя виртуальную адресацию, процессор осуществляет доступ к основной памяти, генерируя *виртуальный адрес*, который преобразуется в соответствующий физический адрес перед фактическим обращением к памяти. Задача преобразования вир-

туального адреса в физический известна как *трансляция адреса*. Подобно обработке исключений, трансляция адреса требует тесного взаимодействия между аппаратурой центрального процессора и операционной системой. Специализированный аппаратный модуль в микросхеме процессора, который называется *модулем управления памятью* (Memory Management Unit, MMU), в процессе выполнения транслирует виртуальные адреса в физические, используя таблицу преобразования, хранящуюся в основной памяти, которой управляет операционная система.

### Упражнение 9.1 (решение в конце главы)

Заполните следующую таблицу, вписав значения в пустые ячейки, и замените каждый вопросительный знак соответствующим целым числом. Используйте следующие единицы: K =  $2^{10}$  (кило), M =  $2^{20}$  (мега), G =  $2^{30}$  (гига), T =  $2^{40}$  (тера), P =  $2^{50}$  (пета) или E =  $2^{60}$  (экса).

Количество разрядов в виртуальном адресе ( $n$ )	Количество виртуальных адресов ( $N$ )	Наибольший возможный виртуальный адрес
8		
	$2^7 = 64$ K	
		$2^{32} - 1 = ?$ G - 1
	$2^7 = 256$ T	
64		

## 9.2. Пространства адресов

*Пространство адресов* – это упорядоченное множество неотрицательных целочисленных адресов

$$\{0, 1, 2, \dots\}$$

Если целочисленные значения в пространстве адресов линейно упорядочены, то мы говорим, что это – *линейное адресное пространство*. Чтобы упростить наши рассуждения, мы будем полагать, что всегда используются линейные адресные пространства. В системе с виртуальной памятью процессор генерирует виртуальные адреса из адресного пространства с  $N = 2^n$  адресами, которое называют *виртуальным адресным пространством*:

$$\{0, 1, 2, \dots, N - 1\}$$

Размер адресного пространства определяется количеством двоичных разрядов (битов), необходимых для представления наибольшего адреса. Например, виртуальное адресное пространство с  $N = 2^n$  адресами называют  $n$ -разрядным адресным пространством. Современные системы обычно поддерживают 32- или 64-разрядные виртуальные адресные пространства.

В системе также имеется *физическое адресное пространство*, которое соответствует  $M$  байтам физической памяти в системе:

$$\{0, 1, 2, \dots, M - 1\}.$$

$M$  необязательно должно быть степенью двойки, но для простоты рассуждений мы примем, что  $M = 2^m$ .

Важность понятия адресного пространства объясняется тем, что оно проводит четкое различие между объектами данных (байтами) и их атрибутами (адресами). Осознав

это различие, мы сможем сделать обобщение и допустить, что каждый объект данных может иметь несколько независимых адресов в разных адресных пространствах. Такова основная идея виртуальной памяти. Каждому байту основной памяти ставится в соответствие виртуальный адрес из виртуального адресного пространства и физический адрес, выбранный из физического адресного пространства.

### 9.3. Виртуальная память как средство кеширования

Концептуально виртуальная память организована как массив  $N$  последовательно расположенных однобайтных ячеек, хранящихся на диске. Каждый байт имеет уникальный виртуальный адрес, который служит индексом в этом массиве. Содержимое массива на диске кешируется в основной памяти. По аналогии с любыми другими кешами в иерархии памяти, данные на диске (более низкий уровень) разделены на блоки, которые служат единицами обмена между диском и основной памятью (верхний уровень). Системы с виртуальной памятью управляют ими, разбивая виртуальную память на блоки фиксированного размера, называемые *виртуальными страницами*. Каждая виртуальная страница имеет размер  $P = 2^p$  байт. Точно так же физическая память разделена на *физические страницы*, тоже размером  $P$  байт. Физические страницы называются также *страничными блоками* (page frames).

В любой момент времени множество виртуальных страниц разбито на три непересекающихся подмножества:

- *незанятые страницы*, которые еще не распределены (или не были созданы) системой виртуальной памяти. Незанятые страницы не содержат никаких данных и потому не занимают места на диске;
- *кешированные страницы*, которые в текущий момент отведены под кеш в физической памяти;
- *некешированные страницы*, которые на данный момент уже распределены, но не кешированы в физической памяти.

В примере на рис. 9.3 показана виртуальная память из 8 виртуальных страниц. Виртуальные страницы 0 и 3 еще не были распределены, их нет на диске. Виртуальные страницы 1, 4 и 6 кешированы в физической памяти. Страницы 2, 5 и 7 распределены, но в данный момент не кешированы в физической памяти.

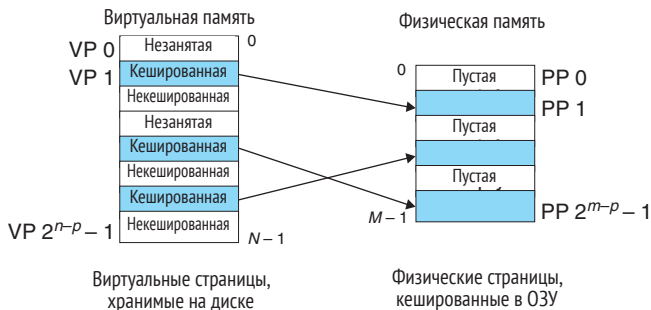


Рис. 9.3. Как система с виртуальной памятью использует основную память в качестве кеша

#### 9.3.1. Организация кеша DRAM

Для простоты ссылок на разные кешы в иерархии памяти будем использовать термин *кеш SRAM* для обозначения кеша первого (L1), второго (L2) и третьего (L3) уровней между процессором и основной памятью и термин *кеш DRAM* для обозначения кеша в

системе виртуальной памяти, предназначенного для хранения виртуальных страниц в физической памяти.

Место, занимаемое кешем DRAM в иерархии памяти, имеет определяющее значение при выборе способа его организации. Еще раз напомним, что DRAM примерно в 10 раз медленнее, чем SRAM, а диск примерно в 100 000 раз медленнее, чем DRAM. Поэтому промахи кеша DRAM обходятся очень дорого (в смысле времени), по сравнению с промахами кеша SRAM, потому что промахи кеша DRAM влекут за собой доступ к диску, тогда как при промахах кеша SRAM часто достаточно обратиться к основной памяти, построенной на базе DRAM. Более того, чтение первого байта из дискового сектора примерно в 100 000 раз медленнее чтения последующих байтов. Проще говоря, огромная стоимость промахов обусловила необходимость организации кеша DRAM.

В силу больших непроизводительных затрат, обусловленных промахами кеша, и большого времени доступа к первому байту существует тенденция делать виртуальные страницы довольно большими, обычно их размеры выбираются в пределах от 4 Кбайт до 2 Мбайт. Из-за дороговизны промахов кеша DRAM делают полностью ассоциативными, т. е. любая виртуальная страница может быть помещена в любую физическую страницу. Стратегия замены при отсутствии нужной страницы тоже имеет большое значение, потому что накладные расходы, связанные с заменой неверно выбранной виртуальной страницы, слишком высоки. По этой причине операционные системы используют намного более сложные алгоритмы управления кешем DRAM, чем в аппаратных кешах SRAM. (Мы не будем обсуждать эти алгоритмы.) Наконец, из-за большого времени доступа к диску кеша DRAM всегда используют алгоритм обратной записи, а не алгоритм сквозной записи.

### 9.3.2. Таблицы страниц

По аналогии с любыми другими кешами, система виртуальной памяти должна иметь возможность определить, кеширована ли виртуальная страница в DRAM. Если да, то система должна определить физическую страницу, где она кеширована. Если нужная страница отсутствует, система должна определить, в каком месте на диске хранится эта виртуальная страница, выбрать в физической памяти страницу, которую можно удалить, и скопировать туда виртуальную страницу с диска.

Эти возможности поддерживаются комбинацией аппаратной трансляции адресов в модуле MMU, программного обеспечения операционной системы и структуры данных, хранящейся в физической памяти, известной как *таблица страниц*, которая отображает виртуальные страницы в физические. Аппаратный модуль трансляции адресов обращается к таблице страниц каждый раз, когда нужно преобразовать виртуальный адрес в физический. Операционная система отвечает за поддержку содержимого таблицы страниц и передачу страниц в обоих направлениях между диском и DRAM.

На рис. 9.4 показана обобщенная организация таблицы страниц. Таблица страниц – это массив *элементов таблицы страниц* (Page Table Entry, PTE). Каждой странице в виртуальном адресном пространстве соответствует свой элемент PTE, смещение которого в таблице страниц фиксировано. Для наших целей будем полагать, что каждый элемент PTE состоит из *разряда достоверности* (valid bit) и *n*-разрядного поля адреса. Разряд достоверности показывает, кеширована ли в настоящее время данная виртуальная страница в DRAM. Если разряд достоверности установлен (т. е. равен 1), то поле адреса указывает на начало соответствующей физической страницы в DRAM, где находится кешированная виртуальная страница. Если разряд достоверности не установлен (т. е. равен 0), то нулевой адрес (Null) свидетельствует о том, что виртуальная страница еще не была закреплена за какой-либо физической страницей. В противном случае адрес указывает на начало виртуальной страницы на диске.

В примере на рис. 9.4 показана таблица страниц для системы с 8 виртуальными страницами (Virtual Page, VP) и 4 физическими страницами (Physical Page, PP). Четыре виртуальные страницы (VP 1, VP 2, VP 4 и VP 7) в настоящий момент кешированы в DRAM. Две страницы (VP 0 и VP 5) еще не были распределены, а остальные (VP 3 и VP 6) выделены и не кешированы. Обратите внимание, что благодаря полной ассоциативности кеша DRAM любая физическая страница может хранить любую виртуальную страницу.

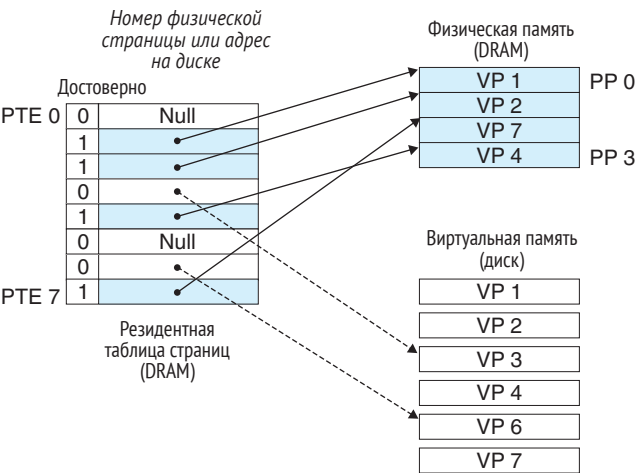


Рис. 9.4. Таблица страниц

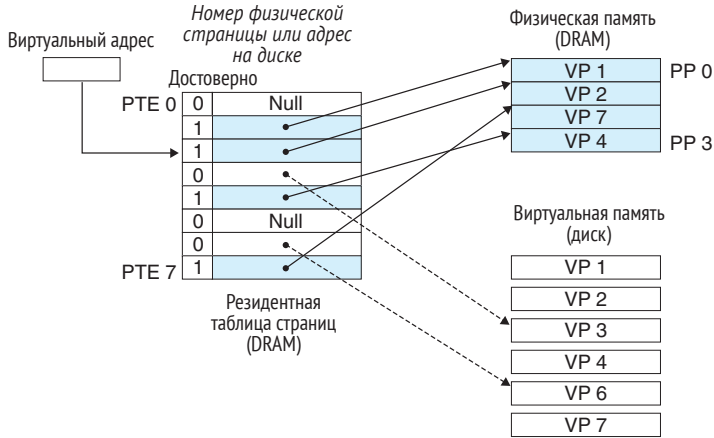
Упражнение 9.2 (решение в конце главы)

Определите, сколько в таблице страниц (PTE) должно быть элементов для следующих комбинаций разрядности виртуальных адресов ( $n$ ) и размеров страниц ( $P$ ):

$n$	$P = 2^p$	Количество элементов в PTE
16	4K	
16	8K	
32	4K	
32	8K	

9.3.3. Попадание в кеш DRAM

Давайте посмотрим, что происходит, когда процессор читает слово из страницы виртуальной памяти VP 2, кешированной в DRAM (рис. 9.5). Используя прием, подробно описанный в разделе 9.6, аппаратный модуль преобразования адресов применяет виртуальный адрес как индекс, чтобы определить местонахождение PTE 2 и прочитать слово из памяти. Разряд достоверности для этой страницы установлен в 1, а значит, она кеширована в памяти, поэтому модуль использует физический адрес в PTE (который указывает на начало кешированной страницы в физической памяти), чтобы сконструировать физический адрес слова.

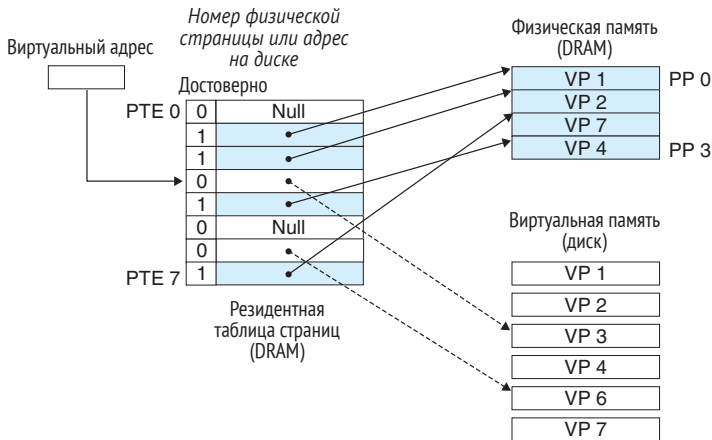


**Рис. 9.5.** Попадание в кеш виртуальной памяти.

При обращении к слову в VP 2 имеет место попадание в кеш

### 9.3.4. Промах кеша DRAM

В терминологии виртуальной памяти промах кеша DRAM называется *сбоем страницы* (page fault). На рис. 9.6 показано состояние таблицы страниц в нашем примере перед сбоем. Процессор ссылается на слово в странице VP 3, которая не кеширована в DRAM. Аппаратный модуль преобразования адресов читает элемент PTE 3 и по состоянию разряда достоверности определяет, что VP 3 не кеширована, после чего генерирует исключение сбоя страницы. Для обработки этого исключения вызывается обработчик, который выбирает, какую страницу удалить, в данном случае это страница VP 4, хранящаяся в PP 3. Если VP 4 изменялась, то ядро копирует ее обратно на диск. Затем ядро корректирует элемент VP 4 таблицы страниц, чтобы отразить факт отсутствия VP 4 в кеше в основной памяти.

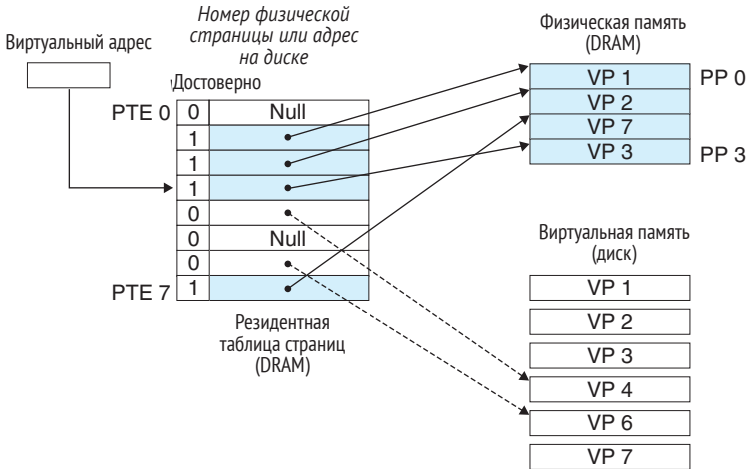


**Рис. 9.6.** Сбой страницы виртуальной памяти (до).

Ссылка на слово в VP 3 вызывает промах кеша и сбой страницы

После этого ядро копирует VP 3 с диска в PP 3 в памяти, обновляет разряд достоверности в PTE 3 и возвращает управление. После возврата из обработчика вновь выпол-

няется инструкция, в которой возник сбой, и та снова обращается по тому же виртуальному адресу. Но на этот раз страница VP 3 уже кеширована в основной памяти и модуль преобразования адресов обнаружит искомую физическую страницу – произойдет попадание в кеш. На рис. 9.7 показано состояние таблицы страниц после обработки сбоя страницы.



**Рис. 9.7.** Сбой страницы виртуальной памяти (после). Обработчик сбоев страниц выбирает VP 4 для удаления и заменяет ее копией VP 3 с диска. Затем обработчик сбоев страниц возвращает управление инструкции, в которой возник сбой, и она читает слово из памяти как обычно, не генерируя исключения

Механизм виртуальной памяти был придуман в начале шестидесятых годов прошлого столетия, задолго до того, как непрерывно увеличивающийся разрыв между оперативной памятью и процессором был ликвидирован с помощью SRAM. По этой причине терминология виртуальной памяти отличается от терминологии кешей SRAM, даже притом что в обоих случаях используются одни и те же идеи. В терминологии виртуальной памяти блоки называются страницами. Действия по пересылке страниц между диском и памятью называются *подкачкой* (swapping) или *подкачкой страниц* (paging). Страницы *загружаются* (swapped in, paged in) с диска в DRAM и *выгружаются* (swapped out, paged out) из DRAM на диск. Стратегия откладывания загрузки страницы до последнего момента, когда происходит промах, называется *замещением страниц по требованию* (demand paging). Возможны и другие подходы, такие как предсказание промахов и предварительная подкачка страницы до фактического обращения к ней. Однако все современные системы используют замещение страниц по требованию.

### Подсчет сбоев страниц

Узнать, как меняется со временем количество сбоев страниц (и получить много другой полезной информации) в Linux можно с помощью функции `getrusage`.

### 9.3.5. Размещение страниц

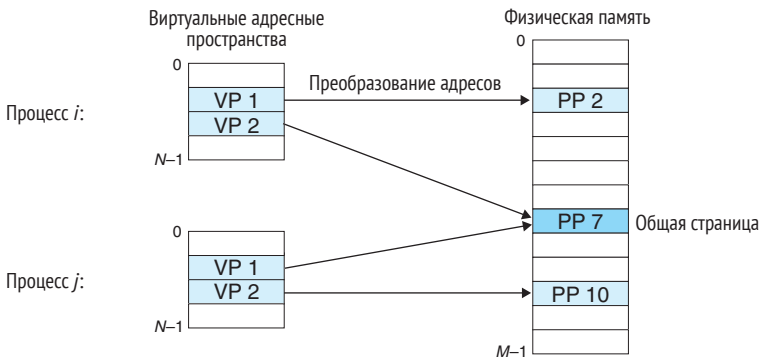
На рис. 9.8 показано, как изменится таблица страниц, если операционная система выделит новую страницу виртуальной памяти, например в результате вызова `malloc`.





виртуального адресного пространства. Интересно, что некоторые ранние системы, такие как, например, DEC PDP-11/70, поддерживали виртуальное адресное пространство, которое было *меньше* объема физической памяти. Однако и в этом случае виртуальная память давала определенную пользу, значительно упрощая управление памятью и обеспечивая естественный способ ее защиты.

До сих пор мы предполагали существование единственной таблицы страниц, которая отображает единственное виртуальное адресное пространство в физическое. На самом деле операционные системы предоставляют каждому процессу отдельную таблицу страниц и, соответственно, отдельное виртуальное адресное пространство. Основная идея такой организации показана на рис. 9.9. В нашем примере таблица страниц для процесса  $i$  отображает VP 1 в PP 2 и VP 2 в PP 7. Аналогично таблица страниц для процесса  $j$  отображает VP 1 в PP 7 и VP 2 в PP 10. Обратите внимание, что несколько виртуальных страниц может отображаться в одну и ту же совместно используемую физическую страницу.



**Рис. 9.9.** Как виртуальная память предоставляет процессам отдельные адресные пространства. Операционная система поддерживает отдельные таблицы страниц для каждого процесса в системе

Сочетание принципа замещения страниц по требованию с использованием отдельных виртуальных адресных пространств оказывает существенное влияние на способ использования памяти и алгоритмы управления ею в системе. В частности, виртуальная память упрощает компоновку и загрузку, совместное использование программного кода и данных, а также выделение памяти приложениям.

- *Упрощение компоновки.* Отдельное адресное пространство позволяет каждому процессу использовать один и тот же формат образа в памяти независимо от того, где в текущий момент находится программный код и данные в физической памяти. Например, как было показано на рис. 8.9, все процессы в системе Linux имеют похожую организацию памяти. В 64-разрядном адресном пространстве раздел кода *всегда* начинается с виртуального адреса 0x400000. Сегмент данных следует за сегментом кода, через промежуток, определяемый правилами выравнивания. Стек занимает старшие адреса в пользовательском адресном пространстве и растет вниз. Такое единообразие значительно упрощает проектирование и реализацию компоновщиков, позволяя им создавать полностью связанные выполняемые файлы, не зависящие от фактического местоположения кода и данных в физической памяти.
- *Упрощение загрузки.* Виртуальная память также упрощает загрузку в память выполняемых и разделяемых объектных файлов. Чтобы загрузить сегменты .text

и `.data` из объектного файла во вновь созданный процесс, загрузчик Linux выделяет виртуальные страницы для сегментов кода и данных, отмечает их как недействительные (т. е. некешированные) и указывает в таблице страниц соответствующие адреса в объектном файле. Интересно отметить, что в действительности загрузчик никогда не копирует данные с диска в память. Данные загружаются автоматически и по требованию системы виртуальной памяти при первом обращении к каждой странице, когда процессор пытается извлечь инструкцию для выполнения или данные, на которые ссылается выполняемая инструкция.

Такой способ отображения набора смежных виртуальных страниц в произвольное место в произвольном файле известен как *отображение памяти*. Linux предоставляет системный вызов `mmap`, с помощью которого прикладные программы могут организовать отображение памяти для своих нужд. Более подробно мы опишем этот механизм в разделе 9.8.

- *Упрощение совместного использования.* Раздельные адресные пространства предоставляют операционной системе согласованный механизм управления совместным использованием памяти пользовательскими процессами и самой операционной системой. В общем случае каждый процесс имеет свои изолированные сегменты кода, данных, динамической памяти и стека, доступные только этому процессу. При этом операционная система создает таблицы страниц, отображающие соответствующие виртуальные страницы на неперекрывающиеся физические страницы.

Однако в некоторых случаях желательно, чтобы процессы могли совместно использовать некоторый программный код и данные. Например, каждому процессу приходится вызывать один и тот же программный код ядра операционной системы, и каждая программа на C вызывает подпрограммы из стандартной библиотеки C, такие как `printf`. Вместо того чтобы включать отдельные копии ядра и стандартной библиотеки C в каждый процесс, операционная система может организовать совместный доступ нескольких процессов к единственной копии этого программного кода, отображая соответствующие виртуальные страницы в различных процессах в одни и те же физические страницы, как показано на рис. 9.9.

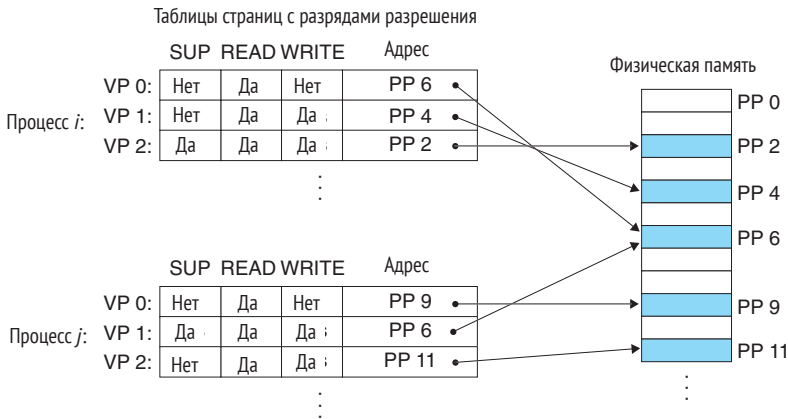
- *Упрощение выделения памяти.* Виртуальная память реализует простой механизм выделения дополнительной памяти пользовательским процессам. Когда программа, выполняющаяся в пользовательском процессе, запрашивает дополнительное пространство в области «динамической памяти» (например, вызовом функции `malloc`), операционная система выделяет соответствующее число, скажем  $k$ , последовательно размещенных страниц виртуальной памяти и отображает их в  $k$  произвольных физических страниц, расположенных в произвольном месте в физической памяти. В силу особенностей функционирования таблиц страниц нет необходимости в том, чтобы операционная система выделяла  $k$  последовательных страниц физической памяти. Страницы могут быть беспорядочно разбросаны по всей физической памяти.

## 9.5. Виртуальная память как средство защиты памяти

Любая современная компьютерная система должна обладать средствами, с помощью которых операционная система могла бы управлять доступом к системе памяти. Пользовательскому процессу нельзя разрешать изменять свой сегмент кода, нельзя разрешать читать или изменять программный код и структуры данных ядра системы, нельзя

разрешать читать и изменять содержимое памяти других процессов, нельзя разрешать изменять любые виртуальные страницы, которые совместно используются другими процессами, пока все вовлеченные процессы явно не дадут на это своего согласия (путем явного вызова системных функций, реализующих взаимодействия между процессами).

Как мы уже видели, наличие отдельных виртуальных адресных пространств упрощает изоляцию памяти различных процессов. Но механизм трансляции адресов можно расширить и обеспечить еще более тонкое управление доступом. Поскольку аппаратный модуль преобразования адресов читает элементы в таблице PTE, то каждый раз, когда процессор генерирует адрес, он напрямую управляет доступом к содержимому виртуальной страницы, добавляя в PTE дополнительные разряды разрешения (permission bits). Основная идея показана на рис. 9.10.



**Рис. 9.10.** Использование виртуальной памяти для защиты памяти на уровне страниц

В этом примере мы добавили в каждый элемент PTE три разряда разрешений. Разряд SUP (supervisor – привилегированный режим) определяет, должен ли процесс выполняться в режиме ядра, чтобы обратиться к этой странице. Процессы, выполняющиеся в привилегированном режиме, могут обратиться к любой странице, но процессы, выполняющиеся в пользовательском режиме, могут обращаться только к страницам, для которых бит SUP имеет значение 0. Биты READ и WRITE управляют доступом к странице для чтения и записи соответственно. Например, если процесс *i* выполняется в пользовательском режиме, он сможет читать страницу VP 0 и читать или изменять страницу VP 1. Но ему запрещено обращаться к странице VP 2.

Если какая-то инструкция попытается нарушить эти ограничения, процессор сгенерирует сбой нарушения общей защиты памяти и передаст управление обработчику исключений в ядре, который пошлет сигнал SIGSEGV процессу-нарушителю. Командные оболочки Linux в ответ на этот сигнал обычно выводят сообщение об ошибке сегментации (segmentation fault).

## 9.6. Преобразование адресов

Этот раздел охватывает основы преобразования адресов. Наша цель – достаточно подробно изучить аппаратные средства поддержки виртуальной памяти, чтобы вы смогли самостоятельно разобрать некоторые конкретные примеры. В то же время следует за-

метить, что мы опускаем отдельные детали, в частности связанные с измерением времени, которые являются немаловажными для разработчиков оборудования, но выходят за рамки наших интересов. Для справки в табл. 9.1 приводятся символы и обозначения, которыми мы будем пользоваться на всем протяжении этого раздела.

**Таблица 9.1.** Символы и обозначения, используемые в преобразовании адресов

Символы и обозначения	Описание
<b>Основные параметры</b>	
$N = 2^n$	Количество адресов в виртуальном адресном пространстве
$M = 2^m$	Количество адресов в физическом адресном пространстве
$P = 2^p$	Размер страницы (в байтах)
<b>Компоненты виртуального адреса</b>	
VPO	Смещение в виртуальной странице (Virtual Page Offset) в байтах
VPN	Номер виртуальной страницы (Virtual Page Number)
TLBI	Индекс в TLB (Translation Lookaside Buffer) – буфере ассоциативной трансляции
TLBT	Тег TLB (TLB Tag)
<b>Компоненты физического адреса</b>	
PPO	Смещение в физической странице (Physical Page Offset) в байтах
PPN	Номер физической страницы (Physical Page Number)
CO	Смещение в блоке кеша (Cache Offset) в байтах
CI	Индекс в кеше (Cache Index)
CT	Тег кеша (Cache Tag)

Формально преобразование адреса – это отображение  $N$ -элементного виртуального адресного пространства (Virtual Address Space, VAS) в  $M$ -элементное физическое адресное пространство (Physical Address Space, PAS):

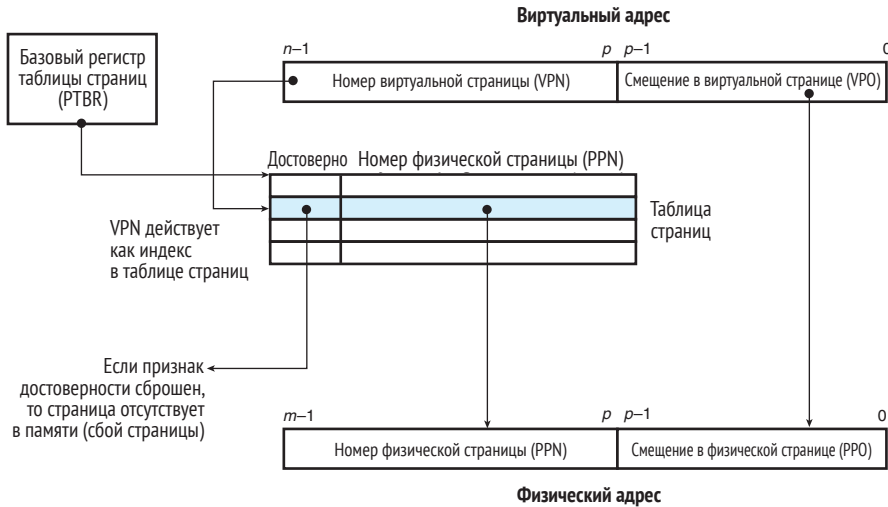
$$\text{MAP: } \text{VAS} \rightarrow \text{PAS} \cup \emptyset,$$

где

$$\text{MAP}(A) = \begin{cases} A', & \text{если данные с вирт. адр. } A \text{ хранятся в физич. памяти с адр. } A' \text{ в PAS;} \\ \emptyset, & \text{если данные с вирт. адр. } A \text{ отсутствуют в физич. памяти.} \end{cases}$$

На рис. 9.11 показано, каким образом модуль управления памятью MMU использует таблицу страниц, чтобы выполнить это отображение. Управляющий регистр процессора – *базовый регистр таблицы страниц* (Page Table Base Register, PTBR), указывает на текущую таблицу страниц.  $n$ -разрядный виртуальный адрес имеет два компонента:  $p$ -разрядное *смещение в виртуальной странице* (Virtual Page Offset, VPO) и  $(n-p)$ -разрядный *номер виртуальной страницы* (Virtual Page Number, VPN). Модуль MMU использует номер VPN, чтобы выбрать соответствующий элемент PTE в таблице страниц. Например, для VPN 0 выбирается элемент PTE 0, для VPN 1 выбирается элемент PTE 1 и т. д. Соответствующий физический адрес – это результат объединения *номера физической страницы* (Physical Page Number, PPN) из элемента в таблице страниц и смещения VPO из виртуального адреса. Обратите внимание, что физические и виртуальные страницы

имеют одинаковый размер  $P$  байт, поэтому *смещение в физической странице* (Physical Page Offset, PPO) совпадает со смещением VPO.



**Рис. 9.11.** Преобразование адресов с помощью таблицы страниц

На рис. 9.12 (а) показана последовательность действий, выполняемых аппаратными модулями процессора, когда искомая страница находится в памяти:

- Шаг 1.** Процессор генерирует виртуальный адрес и передает его в модуль MMU.
- Шаг 2.** Модуль MMU генерирует адрес PTE и запрашивает его из кеша или из основной памяти.
- Шаг 3.** Кеш или основная память возвращает PTE модулю MMU.
- Шаг 4.** Модуль MMU конструирует физический адрес и передает его кешу или основной памяти.
- Шаг 5.** Кеш или основная память возвращает запрошенное слово данных.

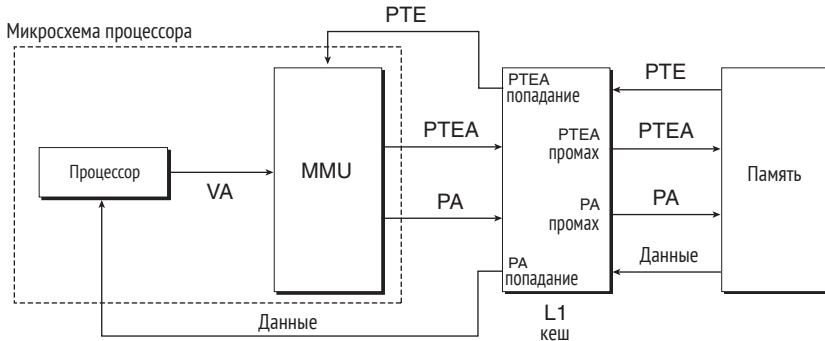
Обработка ситуации, когда страница находится в основной памяти, выполняется исключительно на аппаратном уровне, но обработка противоположной ситуации требует взаимодействия между аппаратным и программным обеспечением ядра операционной системы (рис. 9.12 (б)).

- Шаги 1–3.** Повторяют шаги 1–3 на рис. 9.12 (а).
- Шаг 4.** Разряд достоверности в PTE в этом случае равен нулю, поэтому модуль MMU генерирует исключение, в результате чего управление передается обработчику сбоев страниц в ядре операционной системы.
- Шаг 5.** Обработчик выбирает страницу в физической памяти для удаления, и если эта страница подвергалась изменениям, то выгружает ее на диск.
- Шаг 6.** Обработчик помещает новую страницу на освободившееся место и модифицирует элемент PTE в памяти.
- Шаг 7.** Обработчик ошибки возвращает управление процессу, после чего инструкция, вызвавшая сбой, выполняется повторно. Процессор снова передает виртуальный адрес в модуль MMU. Но теперь искомая виртуальная страница кеширована в физической памяти, поэтому после выполнения шагов, изображенных на рис. 9.12 (а), основная память возвращает запрошенное слово процессору.



зическая адресация. При использовании физической адресации естественным является положение, когда несколько процессов в одно и то же время имеют блоки в кеше и совместно используют блоки одних и тех же виртуальных страниц. Далее, кеш не имеет механизмов защиты, потому что права доступа проверяются на этапе преобразования адресов.

На рис. 9.13 показано, как физически адресуемый кеш можно интегрировать с виртуальной памятью. Основная идея заключается в том, чтобы производить преобразование адреса до поиска в кеше. Обратите внимание, что элементы таблицы страниц могут кешироваться точно так же, как и любые другие данные.



**Рис. 9.13.** Интеграция виртуальной памяти с физически адресуемым кешем.

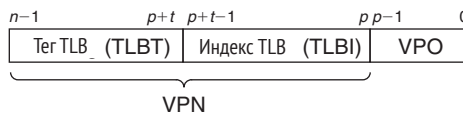
VA: виртуальный адрес. PTEA: адрес элемента в таблице страниц.

PTE: элемент таблицы страниц. PA: физический адрес

## 9.6.2. Ускорение трансляции адресов с помощью TLB

Как мы могли убедиться, каждый раз, когда процессор генерирует виртуальный адрес, модуль MMU должен обратиться к таблице PTE, чтобы преобразовать виртуальный адрес в физический. В худшем случае это потребует дополнительного обращения к памяти, на что придется потратить от нескольких десятков до нескольких сотен тактов. Если окажется, что элемент PTE кеширован в L1, то затраты снижаются до нескольких тактов. Однако многие системы пытаются устранить даже эти затраты, включая в модуль MMU небольшой кеш элементов PTE, так называемый *буфер ассоциативной трансляции* адресов (Translation Lookaside Buffer, TLB).

Буфер TLB – это небольшой, виртуально адресуемый кеш, в котором каждая строка хранит единственный элемент PTE. Буфер TLB обычно имеет высокую степень ассоциативности. Как показано на рис. 9.14, поля индекса и тега, используемые для выбора набора и строки, извлекаются из номера виртуальной страницы в виртуальном адресе. Если буфер TLB может содержать  $T = 2^t$  наборов, то *индекс TLB* (TLBI) состоит из  $t$  младших разрядов VPN, а под *тег TLB* (TLBT) отводятся оставшиеся разряды VPN.



**Рис. 9.14.** Компоненты виртуального адреса, используемые для обращения к TLB

На рис. 9.15 (а) показана последовательность действий при попадании в буфер TLB (обычная ситуация). Важно отметить, что все этапы преобразования адреса выполняются в модуле MMU, благодаря чему обеспечивается необходимое быстроедействие.

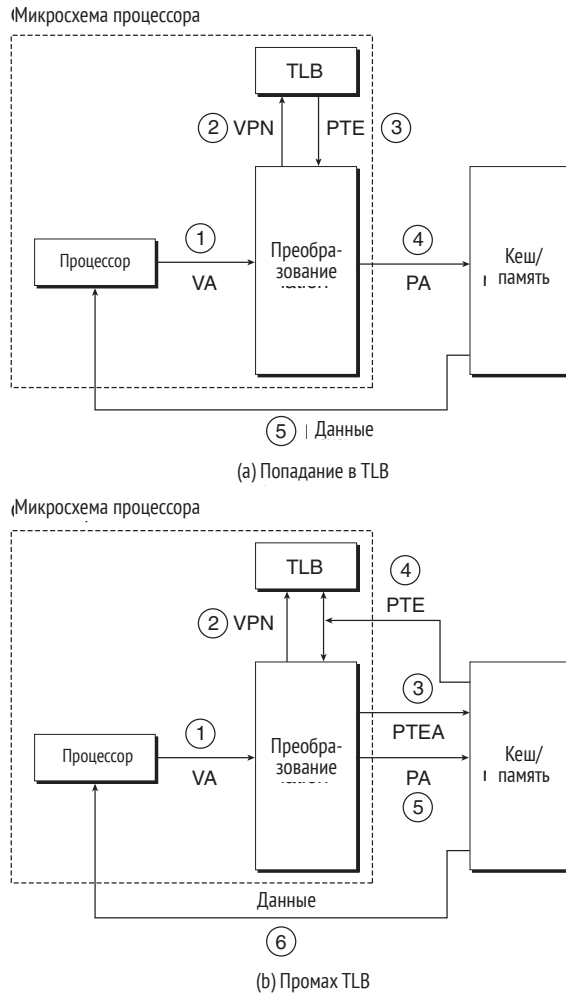
*Шаг 1.* Процессор генерирует виртуальный адрес.

*Шаги 2 и 3.* Модуль MMU выбирает из буфера TLB соответствующий элемент PTE.

*Шаг 4.* Модуль MMU преобразует виртуальный адрес в физический и передает его кешу или основной памяти.

*Шаг 5.* Кеш или память возвращает запрошенное слово данных процессору.

В случае промаха TLB модуль MMU должен получить элемент PTE из кеша L1, как показано на рис. 9.15 (b). Новый элемент PTE сохраняется в TLB, при этом может затираться существующий элемент.



**Рис. 9.15.** Порядок выполнения операций при попадании и промахах TLB

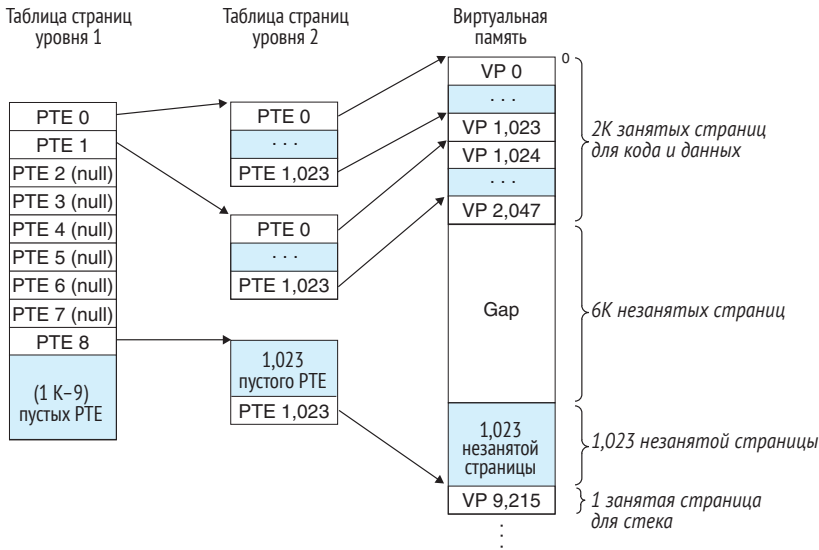
### 9.6.3. Многоуровневые таблицы страниц

До сих пор мы полагали, что для преобразования адресов система использует единственную таблицу страниц. Но если бы у нас было 32-разрядное адресное пространство, страницы размером 4 Кбайта и 4-байтные элементы PTE, то пришлось бы постоянно



держат в памяти таблицу страниц с размером 4 Мбайта, даже если приложение использует лишь небольшую часть виртуального адресного пространства. Задача существенно усложняется для систем с 64-разрядным адресным пространством.

Общепринятый подход уплотнения таблицы страниц предполагает использование их иерархии. Сама по себе эта идея чрезвычайно проста, и мы поясним ее на конкретном примере. Предположим, что 32-разрядное виртуальное адресное пространство разбито на страницы по 4 Кбайта, и каждый элемент таблицы страниц занимает четыре байта. Предположим также, что в некоторый момент времени виртуальное адресное пространство имеет следующую форму: первые 2К страниц памяти отведены для программного кода и данных, следующие 6К страниц свободны, следующие 1023 страницы тоже свободны, а следующая за ними страница отведена для пользовательского стека. На рис. 9.16 показан вариант двухуровневой иерархии для этого виртуального адресного пространства.



**Рис. 9.16.** Иерархия двухуровневых таблиц страниц.

Обратите внимание, что адреса увеличиваются сверху вниз

Каждый элемент первого уровня таблицы PTE представляет фрагмент виртуального адресного пространства размером 4 Мбайт, при этом каждый такой фрагмент состоит из 1024 последовательных страниц. Например, PTE 0 представляет первый фрагмент, PTE 1 – следующий фрагмент и т. д. Если известно, что адресное пространство имеет размер 4 Гбайта, то для охвата всего этого пространства достаточно 1024 элементов PTE.

Если ни одна из страниц во фрагменте  $i$  не занята, то элемент PTE  $i$  уровня 1 будет оставаться пустым. Например, фрагменты 2–7 на рис. 9.16 не заняты. Но если во фрагменте размещена хотя бы одна страница, то элемент PTE  $i$  уровня 1 будет ссылаться на начало таблицы страниц уровня 2. Например, на рис. 9.16 фрагменты 0, 1 и 8 заняты полностью или частично, поэтому соответствующие им элементы PTE уровня 1 ссылаются на таблицы страниц уровня 2.

Каждый элемент PTE в таблице страниц уровня 2 представляет страницы виртуальной памяти размером 4 Кбайта. Обратите внимание, что при использовании 4-байтных элементов PTE каждая таблица страниц уровня 1 и уровня 2 имеет размер 4 Кбайта, который удобен тем, что совпадает с размером страницы.

Эта схема снижает требования к объему памяти. Во-первых, если какой-то элемент PTE в таблице уровня 1 пуст, то соответствующие таблицы страниц уровня 2 просто не могут существовать. Это обстоятельство обещает существенную экономию памяти, потому что большая часть 4-гигабайтного виртуального адресного пространства, выделенного типичной программе, остается невостребованной. Во-вторых, только таблица уровня 1 должна постоянно находиться в основной памяти. Таблицы страниц уровня 2 могут создаваться и использоваться системой виртуальной памяти для перекачки содержимого в обоих направлениях, что уменьшает нагрузку на основную память. В основной памяти должны кешироваться только наиболее часто используемые таблицы страниц уровня 2.

На рис. 9.17 подводится итог описанию механизма преобразования адресов с использованием иерархии  $k$ -уровневых таблиц страниц. Виртуальные адреса разбиты на  $k$  номеров VPN плюс смещение VPO. Каждый номер VPN  $i$ ,  $1 \leq i \leq k$ , является индексом в таблице страниц уровня  $i$ . Каждый элемент PTE в таблице уровня  $j$ ,  $1 \leq j \leq k-1$ , указывает на начало некоторой таблицы страниц уровня  $j+1$ . Каждый элемент PTE в таблице уровня  $k$  содержит либо номер PPN некоторой физической страницы, либо адрес дискового блока. Чтобы построить физический адрес, модуль MMU должен обратиться к  $k$  элементам PTE для получения номера PPN. Так же как в случае одноуровневой иерархии, смещение PPO идентично смещению VPO.

На первый взгляд процедура доступа к  $k$  элементам PTE может показаться дорогостоящей и непрактичной. Однако здесь на помощь приходит буфер TLB, в котором кешируются элементы PTE таблиц различных уровней. На практике преобразование адреса с помощью многоуровневых таблиц по быстродействию ненамного отличается от преобразования с использованием одноуровневых таблиц.

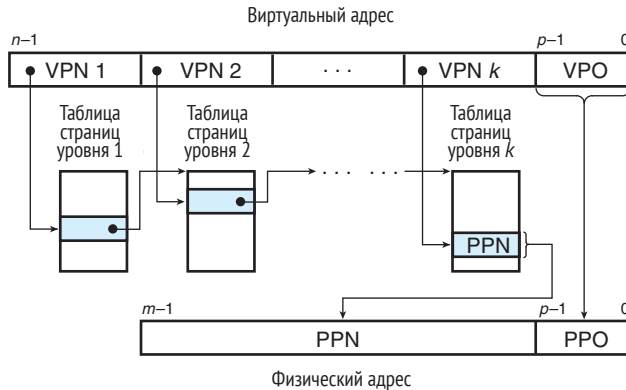


Рис. 9.17. Иерархия  $k$ -уровневых таблиц страниц

#### 9.6.4. Все вместе: сквозное преобразование адресов

В этом разделе мы объединим вместе все, что узнали, и рассмотрим конкретный пример реализации механизма сквозного преобразования адресов в небольшой системе с буфером TLB и кешем данных L1. Для определенности сделаем следующие предположения:

- память адресуется побайтно;
- из памяти выбираются 1-байтные слова (не 4-байтные);
- используются 14-разрядные виртуальные адреса ( $n = 14$ );
- физические адреса имеют 12-разрядов ( $m = 12$ );
- размер страницы составляет 64 байта ( $P = 64$ );
- буфер TLB состоит из 16 четырехстрочных наборов;

- кеш данных L1 является физически адресуемым с прямым отображением и имеет 16 наборов по 4 строки в каждом.

На рис. 9.18 показаны форматы виртуальных и физических адресов. Каждая страница имеет размер  $2^6 = 64$  байта, поэтому младшие шесть разрядов виртуальных и физических адресов используются для задания смещений VPO и PPO соответственно. Старшие восемь разрядов виртуального адреса отводятся под номер VPN. Старшие шесть бит физического адреса отводятся под номер PPN.



**Рис. 9.18.** Механизм адресации небольшой системы памяти. Предполагаются 14-битные виртуальные адреса ( $n = 14$ ), 12-битные физические адреса ( $m = 12$ ) и 64-байтные страницы ( $P = 64$ )

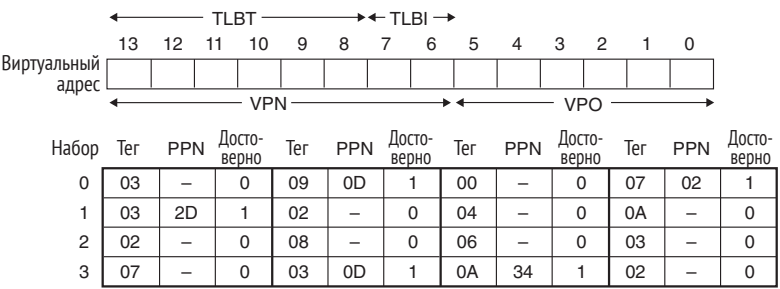
На рис. 9.19 показано состояние в некоторый момент времени разрабатываемой нами небольшой системы памяти, включая буфер TLB (рис. 9.19 (а)), часть таблицы страниц (рис. 9.19 (b)) и кеш L1 (рис. 9.19 (с)). Над изображениями буфера TLB и кеша показано распределение разрядов виртуальных и физических адресов в аппаратных модулях, посредством которых осуществляется доступ к этим устройствам.

**TLB.** Буфер TLB является виртуально адресуемым с использованием разрядов номера VPN. Поскольку TLB имеет четыре набора, два младших разряда в VPN используются как индекс набора (TLBI). Остальные шесть разрядов используются как тег (TLBT), группирующий разные номера VPN, которые могут отображаться в один и тот же набор в буфере TLB.

**Таблица страниц.** Таблица страниц – это одноуровневая конструкция с  $2^8 = 256$  элементами таблицы страниц (PTE). Однако мы рассмотрим только первые шестнадцать элементов. Для удобства мы подписали каждый элемент PTE соответствующим номером VPN, но имейте в виду, что эти номера VPN не являются частью таблицы страниц и не хранятся в памяти. Обратите также внимание, что номер PPN каждого отсутствующего элемента PTE отмечен прочерком, чтобы подчеркнуть тот факт, что содержимое отдельных разрядов не имеет никакого значения.

**Кеш.** Кеш прямого отображения адресуется полями из физического адреса. Поскольку каждый блок состоит из 4 байт, младшие 2 разряда физического адреса используются как смещение блока (CO). Поскольку всего имеется 16 наборов, следующие 4 разряда используются как индекс набора (CI). Остальные 6 разрядов используются как тег кеша (CT).

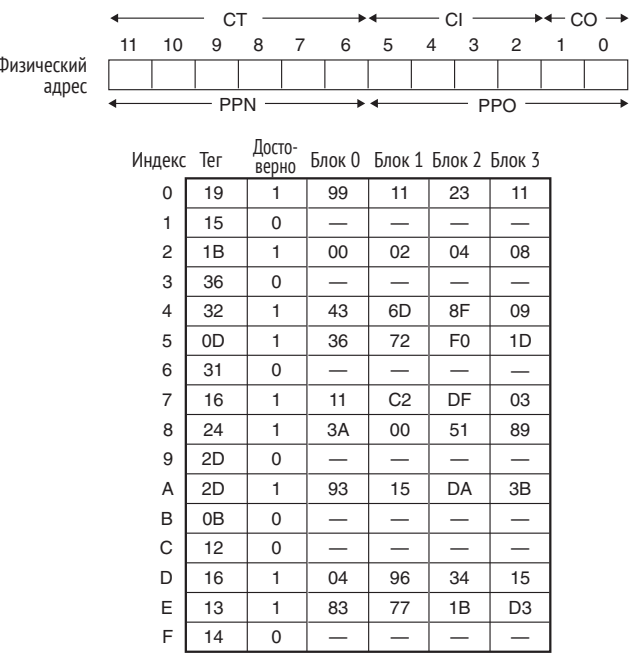
Теперь, оговорив начальные условия, посмотрим, что произойдет, когда процессор выполнит инструкцию загрузки байта, находящегося по адресу `0x03d4`. (Напомним, что наш гипотетический процессор читает однобайтные слова, а не четырехбайтные.) При таком моделировании вручную полезно записать виртуальный адрес побитно, определить поля, которые понадобятся, и вычислить их шестнадцатеричные значения. Аппаратные модули решают аналогичную задачу, выполняя декодирование адреса.



(а) TLB: 4 набора, 16 элементов, 4-строчные ассоциативные наборы

VPN	PPN	Досто- верно	VPN	PPN	Досто- верно
00	28	1	08	13	1
01	—	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	—	0
04	—	0	0C	—	0
05	16	1	0D	2D	1
06	—	0	0E	11	1
07	—	0	0F	0D	1

(б) Таблица страниц: показаны только первые 16 элементов 16 PTE



(с) Кеш: 16 наборов, 4-байтные блоки, прямое отображение

**Рис. 9.19.** TLB, таблица страниц и кеш для небольшой системы памяти.  
Все значения в TLB, таблице страниц и в кеше приводятся  
в шестнадцатеричной форме

	TLBT							TLBI							
	0x03							0x03							
Позиция бита	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
VA=0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0	
	VPN								VPO						
	0x0f								0x14						

Сначала модуль MMU извлекает из виртуального адреса (0x0F) номер виртуальной страницы (VPN) и сверяет его с соответствующим значением в буфере TLB, чтобы проверить, является ли он кешированной копией элемента PTE 0x0F, созданного при некотором предыдущем обращении к памяти. TLB извлекает из номера VPN индекс TLB 0x03 и тег TLB 0x03, обнаруживает подходящее соответствие во второй строке набора 0x03 и возвращает модулю MMU кешированный номер физической страницы (PPN) 0x0D.

В случае промаха TLB модуль MMU должен был бы выбрать элемент PTE из основной памяти. Однако в данном случае нам повезло, и TLB нашел нужный элемент там, где он и должен быть. Теперь модуль MMU имеет все необходимое, чтобы сформировать физический адрес. Объединяя номера PPN 0x0D из PTE со смещением VPO 0x14 из виртуального адреса, он получает физический адрес 0x354.

Далее модуль MMU посылает в кеш физический адрес, из которого извлекается смещение в кеше CO (0x0), индекс набора в кеше CI (0x5) и тег кеша CT (0x0D).

Позиция бита	CT						CI				CO	
	0x0d						0x05				0x0	
	11	10	9	8	7	6	5	4	3	2	1	0
PA=0x354	0	0	1	1	0	1	0	1	0	1	0	0
PPN										PPO		
0x0d										0x14		

Поскольку тег в наборе (0x5) соответствует значению поля CT, кеш определяет попадание, читает байт данных 0x36 со смещением CO и возвращает его модулю MMU, который затем передает его назад процессору.

Возможны также другие варианты в процессе преобразования адресов. Например, если буфер TLB отсутствует, то модуль MMU должен выбрать номер PPN из соответствующего элемента PTE в таблице страниц. Если полученный элемент недействителен, то это означает попытку обращения к отсутствующей странице, и ядро должно прочитать с диска соответствующую страницу, а затем повторно выполнить инструкцию загрузки. Возможен другой случай, когда элемент PTE является достоверным, но необходимый блок памяти отсутствует в кеше.

#### Упражнение 9.4 (решение в конце главы)

Покажите, как в примере системы памяти из раздела 9.6.4 преобразуется виртуальный адрес в физический и осуществляется доступ к кешу. Для заданного виртуального адреса укажите, к какому элементу буфера TLB происходит обращение, физический адрес и возвращаемое значение байта из кеша. Укажите, имеет ли место промах TLB, возникают ли сбои страниц и промахи кеша. Если имеет место промах кеша, поставьте прочерк в графе «Возвращаемый байт из кеша». Если имеет место сбой страницы, поставьте прочерк в графе «PPN» и не заполняйте части 3 и 4 упражнения.

Виртуальный адрес: 0x03d7

1. Формат виртуального адреса.

13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Преобразование адреса.

Параметр	Значение
VPN	_____
Индекс TLB	_____
Тег TLB	_____
Попадание в TLB? (Да/Нет)	_____
Сбой страницы? (Да/Нет)	_____
PPN	_____

3. Формат физического адреса.

11	10	9	8	7	6	5	4	3	2	1	0

4. Ссылка на физическую память.

Параметр	Значение
Смещение байта	_____
Индекс в кеше	_____
Тег кеша	_____
Попадание в кеш? (Да/Нет)	_____
Возвращаемый байт из кеша	_____

## 9.7. Практический пример: система памяти Intel Core i7/Linux

Мы завершаем наше обсуждение механизмов виртуальной памяти практическим примером действующей системы памяти в Linux на архитектуре Intel Core i7. Базовая микроархитектура Haswell поддерживает полноценные 64-разрядные виртуальное и физическое адресные пространства, однако текущие реализации Core i7 (и те, что появятся в обозримом будущем) поддерживают лишь 48-разрядное (256 Тбайт) виртуальное адресное пространство и 52-разрядное (4 Пбайт) физическое адресное пространство, а также режим совместимости с 32-разрядным (4 Гбайт) виртуальным и физическим адресными пространствами.

На рис. 9.20 показано устройство системы памяти Core i7. *Микросхема процессора* (processor package) включает четыре ядра, большой кеш L3, совместно используемый всеми ядрами, и контроллер памяти DDR3. Каждое ядро содержит иерархию буферов преобразования адресов (TLB), иерархию кешей данных и инструкций, а также набор скоростных соединений типа «точка–точка», основанных на технологии QuickPath, для прямой связи с другими ядрами и внешним мостом ввода/вывода. Буферы TLB имеют виртуальную адресацию и 4-строчную ассоциативность. Кеши L1, L2 и L3 имеют физическую адресацию с размером блока 64 байт. L1 и L2 являются ассоциативными с 8 строками, а L3 – ассоциативным с 16 строками. Размер страницы можно настроить во время запуска: 4 Кбайта или 4 Мбайта. В Linux используются страницы размером 4 Кбайта.

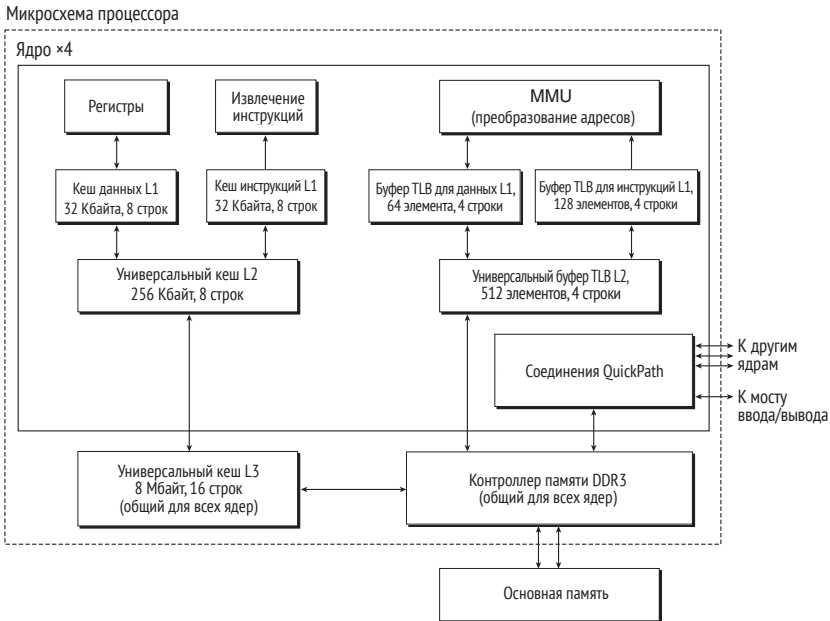


Рис. 9.20. Система памяти в Core i7

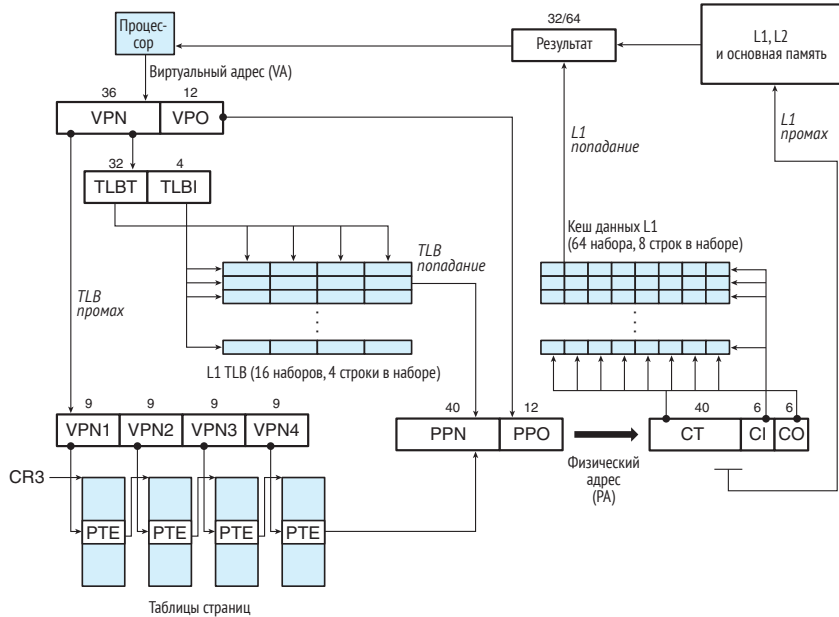
### 9.7.1. Преобразование адресов в Core i7

На рис. 9.21 показана общая схема преобразования адреса в Core i7 с момента, когда процессор генерирует виртуальный адрес, до получения слова данных из памяти. В Core i7 используется четырехуровневая иерархия таблиц страниц. Каждый процесс имеет свою иерархию таблиц страниц. После запуска процесса Linux все его таблицы страниц с информацией о выделенных страницах постоянно хранятся в памяти, хотя архитектура Core i7 позволяет загружать и выгружать эти таблицы. Регистр управления CR3 содержит физический адрес начала таблицы страниц уровня 1 (L1). Значение CR3 является частью контекста каждого процесса и восстанавливается при каждом переключении контекста.

В табл. 9.2 показан формат элементов таблиц страниц уровня 1, 2 и 3. Когда  $P = 1$  (это условие всегда выполняется в Linux), поле адреса содержит 40-разрядный номер физической страницы (PPN), определяющий адрес начала соответствующей таблицы страниц. Обратите внимание, что это условие накладывает требование выравнивания таблиц страниц по границам, кратным 4 Кбайтам.

В табл. 9.3 показан формат элементов в таблице страниц уровня 4. Когда  $P = 1$ , поле адреса содержит 40-разрядный номер PPN, определяющий адрес начала некоторой страницы в физической памяти. И снова это условие накладывает требование выравнивания физических страниц по границам, кратным 4 Кбайтам.

В элементе PTE есть три бита разрешений, управляющих доступом к странице. Бит *R/W* определяет доступность содержимого страницы для чтения/записи или только для чтения. Бит *U/S* определяет возможность доступа к странице в пользовательском режиме и защищает код и данные ядра операционной системы от пользовательских программ. Бит *XD* (eXecute Disable – запрет выполнения), появившийся в 64-разрядных системах, может использоваться для отключения выборки инструкций из отдельных страниц памяти. Это важное новшество, позволяющее ядру операционной системы снизить риск атак переполнения буфера, ограничивая возможность выполнения лишь для сегментов кода, доступных только для чтения.



**Рис. 9.21.** Обобщенная схема преобразования адресов в Core i7. Для простоты кэши инструкций, буферы TLB инструкций и универсальные буферы TLB L2 не показаны

**Таблица 9.2.** Формат элементов таблиц страниц уровней 1, 2 и 3.

Каждый элемент ссылается на дочернюю таблицу страниц размером 4 Кбайта

63	62	52	51	12		11	9	8	7	6	5	4	3	2	1	0
XD	Не исп.	Базовый физический адрес таблицы страниц			Не исп.		G	PS			A	CD	WT	U/S	R/W	P=1
Доступно для операционной системы (таблица расположена на диске)																P=0

Поле	Описание
P	Дочерняя таблица страниц находится (1) или отсутствует (0) в физической памяти
R/W	Разрешение доступа только для чтения или для чтения/записи для всех доступных страниц
U/S	Разрешение доступа в пользовательском или привилегированном режиме (в режиме ядра) для всех доступных страниц
WT	Режим сквозной или отложенной записи в кеш для дочерней таблицы страниц
CD	Кеш заблокирован (1) или задействован (0) для дочерней таблицы страниц
A	Бит-признак обращения (устанавливается модулем MMU при чтении и записи, сбрасывается программно)
PS	Размер страницы 4 Кбайта (0) или 4 Мбайта (1); определяется только для элементов PTE уровня 1
Базовый адрес	40 старших разрядов физического адреса дочерней таблицы страниц
XD	Разрешена или запрещена выборка инструкций со всех страниц, доступных из этого PTE



**Таблица 9.3.** Формат элементов таблицы страниц уровня 4.  
Каждый элемент ссылается на дочернюю страницу размером 4 Кбайта

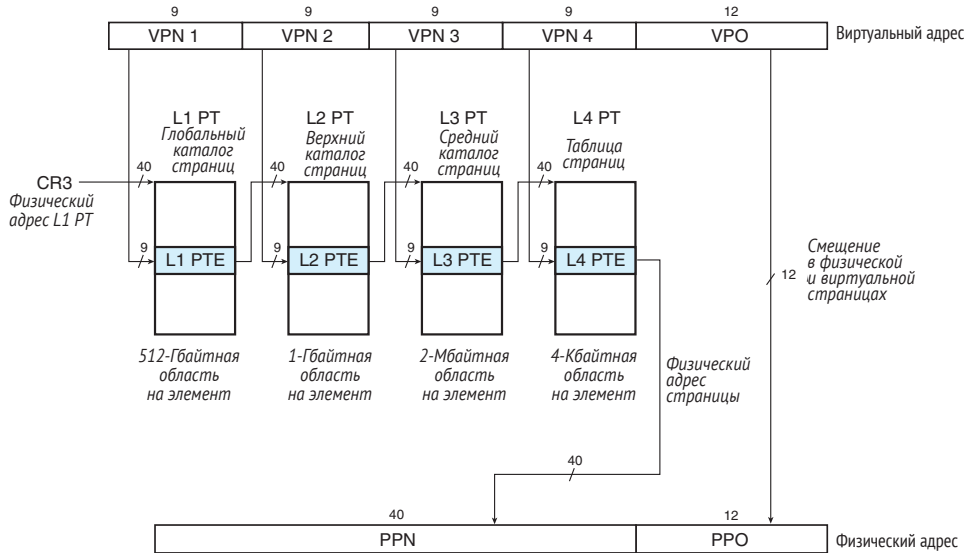
63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	Не исп.	Базовый физический адрес таблицы страниц			Не исп.	G	0	D	A	CD	WT	U/S	R/W	P=1	
Доступно для операционной системы (таблица расположена на диске)														P=0	
Поле		Описание													
P		Дочерняя страница находится (1) или отсутствует (0) в физической памяти													
R/W		Разрешение доступа только для чтения или для чтения/записи к дочерней странице													
U/S		Разрешение доступа в пользовательском или привилегированном режиме (в режиме ядра) к дочерней странице													
WT		Режим сквозной или отложенной записи в кеш для дочерней страницы													
CD		Кеш заблокирован (1) или задействован (0)													
A		Бит-признак обращения (устанавливается модулем MMU при чтении и записи, сбрасывается программно)													
D		Бит-признак изменения (устанавливается модулем MMU при записи, сбрасывается программно)													
G		Глобальная страница (не вытесняется из TLB при переключении задач)													
Базовый адрес		40 старших разрядов физического адреса дочерней страницы													
XD		Разрешена или запрещена выборка инструкций из дочерней страницы													

Когда модуль MMU преобразует каждый виртуальный адрес, он также обновляет два других бита, которые могут использоваться обработчиком сбоев страниц ядра. MMU устанавливает бит *A*, известный как *бит ссылки*, при каждом обращении к странице. Ядро может использовать бит ссылки для реализации своего алгоритма замещения страниц. MMU устанавливает бит *D* или *бит изменения* (dirty bit) после каждой операции записи в страницу. Измененную страницу иногда называют *грязной страницей* (dirty page). Бит изменения сообщает ядру, следует ли записать замещаемую страницу на диск, прежде чем копировать на ее место новую страницу. Ядро может вызвать специальные инструкции для очистки битов ссылки или изменения.

На рис. 9.22 показано, как модуль MMU в процессоре Core i7 использует четыре уровня таблиц страниц для преобразования виртуального адреса в физический. 36-разрядный номер VPN разбит на четыре 9-разрядных фрагмента, каждый из которых используется как смещение в таблице страниц. Регистр CR3 содержит физический адрес таблицы страниц L1. Номер VPN 1 определяет смещение для PTE L1, которое содержит базовый адрес таблицы страниц L2. VPN 2 – смещение для PTE L2 и т. д.

### 9.7.2. Система виртуальной памяти Linux

Виртуальная память ядра содержит код и структуры данных ядра. Некоторые области виртуальной памяти ядра отображаются в физические страницы, общие для всех процессоров. Например, все процессы совместно используют код ядра и глобальные структуры данных. Интересно отметить, что Linux также отображает набор смежных виртуальных страниц (равный по размеру общему объему DRAM в системе) в соответствующий набор смежных физических страниц. Это дает ядру удобную возможность доступа к любому конкретному адресу в физической памяти, например когда ему необходимо получить доступ к таблицам страниц или выполнить операции ввода/вывода с отображением в память на устройствах, которые отображены в определенные области физической памяти.



**Рис. 9.22.** Преобразование адресов с помощью таблиц страниц в Core i7.

PT: таблица страниц; PTE: элемент таблицы страниц; VPN: номер виртуальной страницы; VPO: смещение в виртуальной странице; PPN: номер физической страницы; PPO: смещение в физической странице. Также показаны названия четырех уровней таблиц страниц в Linux

Система виртуальной памяти требует тесного взаимодействия программного и аппаратного обеспечения. В зависимости от версии ядра и оборудования детали взаимодействия могут меняться и полное их описание не входит в нашу задачу. Тем не менее в этом разделе мы рассмотрим систему виртуальной памяти Linux, чтобы вы могли получить представление о том, как реальная операционная система организует виртуальную память и как она обрабатывает сбои страниц.

Linux поддерживает отдельное виртуальное адресное пространство для каждого процесса, как показано на рис. 9.23. Мы уже несколько раз видели эту схему с сегментами кода, данных, кучи, разделяемых библиотек и стека. Теперь, разобравшись с трансляцией адресов, мы можем добавить некоторые детали, касающиеся виртуальной памяти ядра, располагающейся над стеком пространства пользователя.

Другие области виртуальной памяти ядра содержат данные, различающиеся для разных процессов, например таблицы страниц, стек, который ядро использует при выполнении кода в контексте процесса, и различные структуры данных, определяющие текущую организацию виртуального адресного пространства.

### Система виртуальной памяти в Linux

В Linux виртуальная память организована как набор *областей* (также называемых *сегментами*). Область – это непрерывный фрагмент существующей (распределенной) виртуальной памяти, страницы которой взаимосвязаны некоторым образом. Например: сегмент кода, сегмент данных, сегмент динамической памяти (кучи), сегмент разделяемых библиотек и пользовательский стек. Каждая существующая виртуальная страница хранится в некоторой области, и любая виртуальная страница, не являющаяся частью какой-либо области, не существует и не может использоваться процессом. Понятие области играет важную роль, потому что позволяет виртуальному адресному пространству иметь разрывы. Ядро не следит за несуществующими виртуальными страницами, и такие страницы не занимают места в памяти, на диске или в самом ядре.

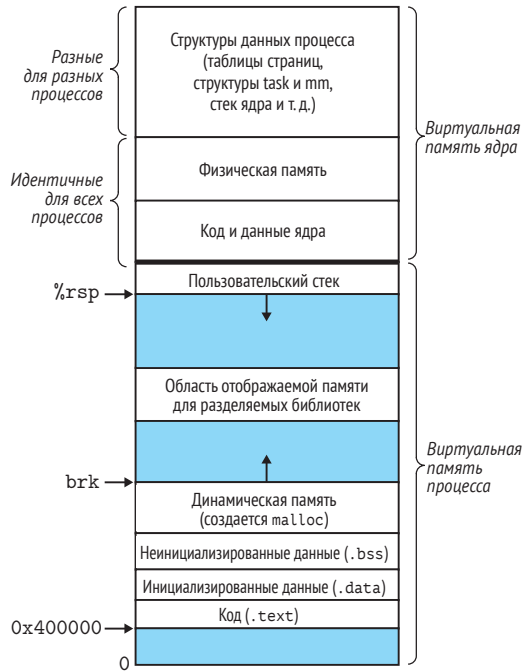


Рис. 9.23. Виртуальная память процесса в Linux

### Оптимизация преобразования адресов

При обсуждении механизма преобразования адресов мы описали последовательный двухэтапный процесс, в ходе которого модуль MMU преобразует виртуальный адрес в физический и затем передает физический адрес в кеш уровня L1. Однако реальные аппаратные реализации используют искусный прием, который позволяет частично перекрывать по времени эти этапы, ускоряя тем самым доступ к кешу L1. Например, виртуальный адрес в Core i7 с 4-Кбайтными страницами имеет 12-битное смещение VPO, и эти биты идентичны 12 битам смещения PPO в соответствующем физическом адресе. 8-строчные ассоциативные кешы L1 имеют по 64 набора и 64-байтные блоки, поэтому каждый физический адрес имеет 6 ( $\log_2 64$ ) разрядов смещения и 6 ( $\log_2 64$ ) разрядов индекса. Эти 12 разрядов точно соответствуют разрядам смещения VPO виртуального адреса, и это не случайное совпадение! Когда процессору требуется преобразовать виртуальный адрес, он посылает номер VPN в модуль MMU и смещение VPO в кеш L1. Пока блок MMU запрашивает из соответствующего буфера TLB элемент таблицы страниц, кеш L1 занят тем, что, используя разряды VPO, ищет соответствующий набор и читает из него восемь тегов и соответствующие слова данных. Когда блок MMU снова получает номер PPN из буфера TLB, кеш уже готов выполнить попытку сравнения номера PPN с одним из этих восьми указанных выше тегов.

На рис. 9.24 показаны структуры данных ядра с информацией об областях виртуальной памяти процесса. Ядро поддерживает отдельную структуру задач (`task_struct` в исходном коде) для каждого процесса в системе. Элементы этой структуры содержат либо саму информацию, либо ссылки на информацию, которая необходима ядру для запуска процесса (например, идентификатор процесса, указатель на пользовательский стек, имя выполняемого объектного файла и счетчик инструкций).

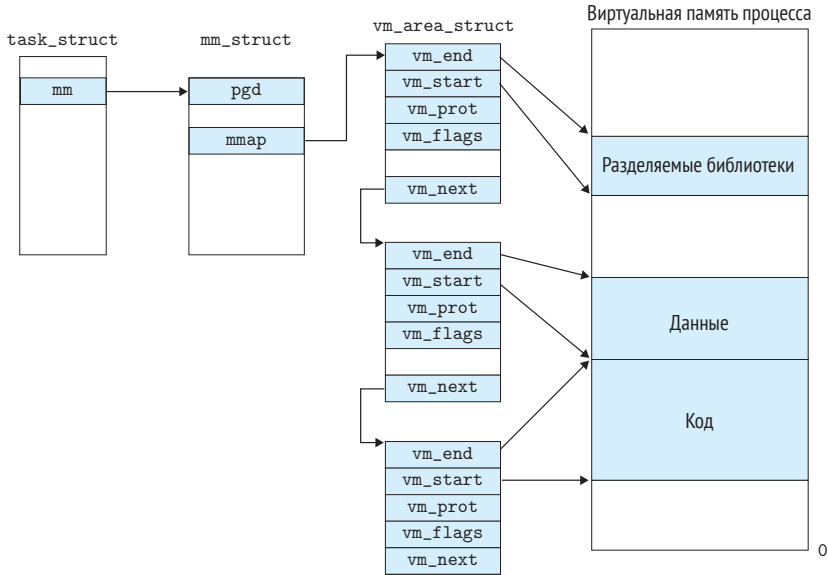


Рис. 9.24. Организация виртуальной памяти в Linux

Один из элементов в структуре `task_struct` ссылается на структуру `mm_struct`, которая характеризует текущее состояние виртуальной памяти. Для нас интерес представляют два поля: `pgd`, с базовым адресом каталога страниц уровня 1, и `mmap`, указывающее на список `vm_area_structs` структур областей, каждая из которых характеризует одну область текущего виртуального адресного пространства. Когда ядро выполняет процесс, содержимое `pgd` хранится в управляющем регистре CR3.

На данном этапе вам достаточно знать, что структура для каждой отдельной области имеет следующие поля:

- `vm_start` – указывает на начало области;
- `vm_end` – указывает на конец области;
- `vm_prot` – определяет разрешения на чтение/запись для всех страниц в данной области;
- `vm_flags` – определяет (кроме всего прочего), используются ли страницы данной области совместно с другими процессами или предназначены только для данного конкретного процесса;
- `vm_next` – указывает на следующую структуру области в списке.

## Обработка исключений при обращении к несуществующим страницам в Linux

Предположим, что при попытке выполнить преобразование некоторого виртуального адреса *A* модуль MMU сгенерировал исключение. Это исключение передает управление в ядро – обработчику сбоев страниц, который выполняет следующие действия:

1. Проверяет допустимость виртуального адреса *A*. То есть находится ли *A* в пределах некоторой области, определяемой какой-либо структурой? Чтобы ответить на этот вопрос, обработчик просматривает список структур областей, сравнивая *A* с `vm_start` и `vm_end` в каждой структуре. Если данная инструкция недопустима, то обработчик генерирует исключение ошибки сегментации, которое завершает процесс. На рис. 9.25 эта ситуация отмечена цифрой «1».

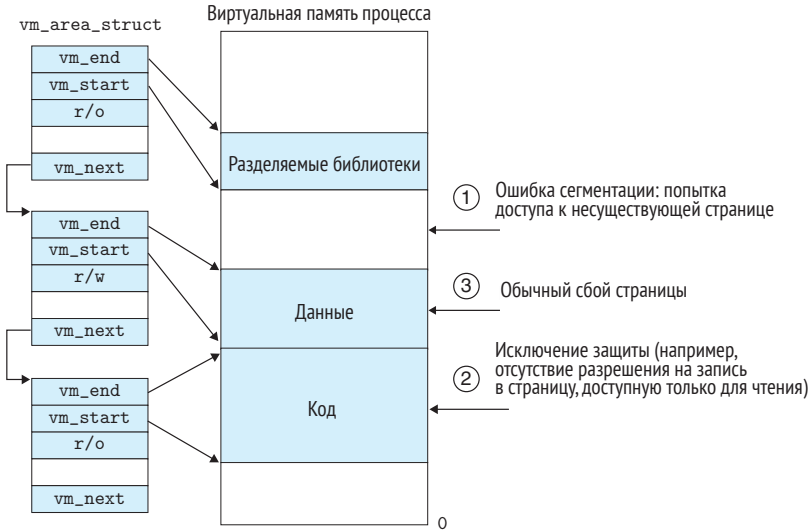


Рис. 9.25. Обработка сбоев страниц в Linux

- Ввиду того, что процесс может создать произвольное число новых областей в виртуальной памяти (используя функцию `mmap`, которая будет описана в следующем разделе), последовательный перебор списка структур областей может оказаться весьма дорогостоящим. По этой причине Linux осуществляет наложение дерева на список, используя с этой целью некоторые поля, которые мы здесь не показали, и выполняет поиск в этом дереве.
- Проверяет допустимость попытки доступа к памяти. То есть имеет ли данный процесс право выполнять операции чтения или записи со страницами в этой области? Например, возник ли сбой страницы из-за попытки записи в страницу, доступную только для чтения, в сегменте кода? Является ли сбой страницы результатом попытки процесса, выполняющегося в пользовательском режиме, обратиться к виртуальной памяти ядра? Если попытка доступа незаконна, то обработчик генерирует исключение защиты, которое завершает процесс. На рис. 9.25 эта ситуация отмечена цифрой «2».
- Достигнув этого этапа, ядро знает, что сбой страницы возник в результате выполнения допустимой операции с допустимым виртуальным адресом. Поэтому обработчик выбирает страницу для удаления, загружает ее, если она подверглась изменению, и загружает новую страницу, одновременно обновляя таблицу страниц. Когда обработчик сбоя страницы возвращает управление, процессор повторно выполняет инструкцию, вызвавшую сбой, которая, как и раньше, отправляет `A` в модуль MMU. Но на этот раз MMU выполняет преобразование адреса `A` без возбуждения ошибки обращения к отсутствующей странице.

## 9.8. Отображение в память

Система Linux инициализирует содержимое области виртуальной памяти, связывая их с объектами на диске. Этот процесс называется *отображением в память* (memory mapping). В области памяти могут отображаться объекты двух типов:

- Обычный файл в файловой системе Linux:** в область может быть отображен непрерывный участок обычного файла на диске, такого как выполняемый объектный

файл. Файл делится на участки размером в одну страницу, и каждый такой участок хранит начальное содержимое виртуальной страницы. Поскольку подкачка страниц выполняется по требованию, ни одна из этих виртуальных страниц фактически не загружается в физическую память, пока процессор сначала не *затронет* эту страницу (т. е. пока не выдаст виртуальный адрес, который попадет в адресное пространство, занимаемое этой страницей). Если область больше участка файла, то неиспользованная ее часть заполняется нулями.

2. *Анонимный файл*: в область может также отображаться анонимный файл, созданный ядром и содержащий только двоичные нули. Когда процессор в первый раз затронет виртуальную страницу в такой области, то ядро найдет в физической памяти подходящую страницу для удаления, выгрузит ее, если она подвергалась изменению, заполнит страницу двоичными нулями и внесет в таблицу страниц коррективы, относящие эту страницу к числу резидентных. Следует отметить, что на самом деле никакого обмена данными между диском и памятью при этом не происходит. По этой причине страницы, отображаемые в анонимные файлы, иногда называются *нулевыми страницами по требованию* (demand-zero pages).

В любом случае, если виртуальная страница инициализирована, она будет выгружаться в специальный *файл подкачки* (swap file), поддерживаемый ядром, и загружаться обратно. Файл подкачки иначе называют *пространством подкачки* (swap space), или *областью подкачки* (swap area). Важно понимать, что в любой момент времени пространство подкачки ограничивает общее количество виртуальных страниц, которые могут распределяться процессами, выполняемыми в текущий момент.

### 9.8.1. И снова о разделяемых объектах

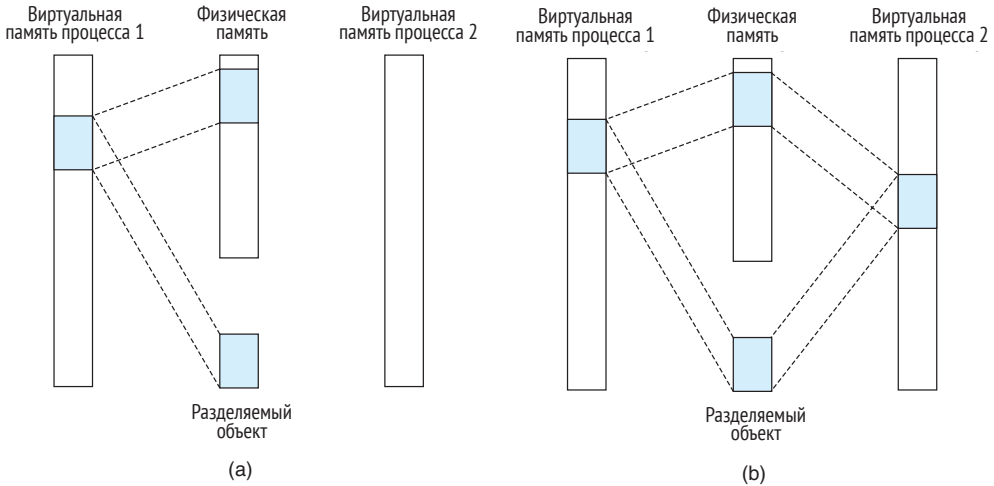
Сама идея отображения в память возникла из четкого понимания, что если систему виртуальной памяти можно интегрировать в стандартную файловую систему, то она сможет обеспечить простой и эффективный способ загрузки программ и данных в память.

Как мы уже видели, абстракция процесса предполагает предоставление каждому процессу своего изолированного виртуального адресного пространства, недоступного другим процессам. В то же время некоторые процессы могут иметь идентичные области с кодом, доступные только для чтения. Например, все процессы, в рамках которых выполняется программа `bash` – командная оболочка системы Linux, – имеют абсолютно идентичные области с кодом. Более того, многие программы используют одни и те же копии библиотечного кода, доступные только для чтения. Например, каждая программа на языке C подключает стандартную библиотеку C с функциями, такими как `printf`. Было бы большим расточительством создавать для каждого процесса дубликаты такого общего программного кода в физической памяти. К счастью, механизм отображения в память позволяет управлять объектами, которые используются сразу несколькими процессами.

Любой объект может быть отображен в область виртуальной памяти как *совместно используемый (разделяемый)* или как *закрытый объект*. Если процесс отображает разделяемый объект в некоторую область своего виртуального адресного пространства, то любое изменение в этой области смогут увидеть все другие процессы, которые тоже отображают этот разделяемый объект в свою виртуальную память. Более того, все изменения отражаются также на исходном объекте на диске.

С другой стороны, изменения в области, в которую отображается закрытый объект, не видны другим процессам, и все изменения в этой области *не* отражаются на объекте на диске. Область виртуальной памяти, куда отображается разделяемый объект, часто называют *совместно используемой (разделяемой) областью*. То же можно сказать и о *закрытой области*.

Предположим, что процесс 1 отобразил разделяемый объект в область своей виртуальной памяти, как показано на рис. 9.26 (а). Предположим также, что процесс 2 отобразил тот же разделяемый объект в свое адресное пространство (не обязательно с тем же виртуальным адресом, что в процессе 1).



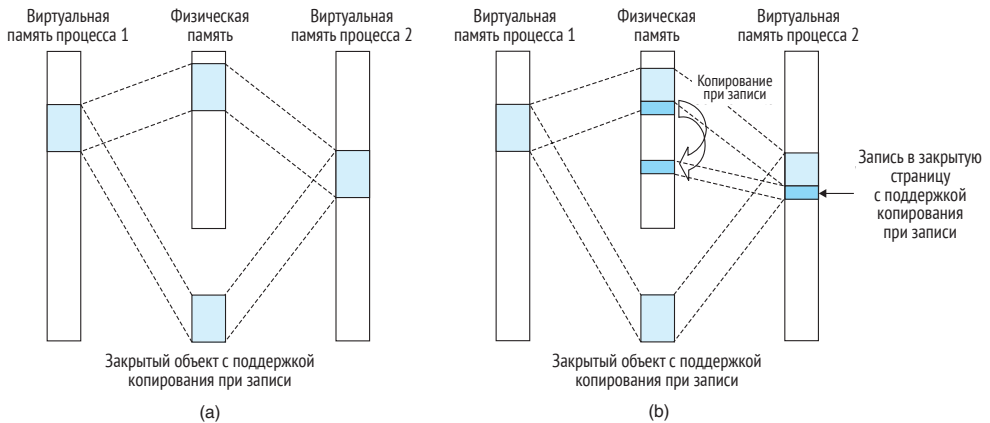
**Рис. 9.26.** Разделяемый объект. (а) После того, как процесс 1 отобразил разделяемый объект. (б) После того, как процесс 2 отобразил тот же разделяемый объект. (Обратите внимание, что физические страницы не обязательно должны быть смежными)

Поскольку каждый объект имеет уникальное имя файла, ядро может быстро определить, что процесс 1 уже отобразил этот объект, и установить указатели в элементах таблиц страниц в процессе 2 на соответствующие физические страницы. В этом случае в физической памяти будет храниться только одна-единственная копия разделяемого объекта, даже если он будет отображен в нескольких различных разделяемых областях. Для удобства на рисунке физические страницы изображены смежными, но вообще это совсем необязательно.

Закрытые объекты отображаются в виртуальную память с использованием тщательно продуманной поддержки *копирования при записи* (copy-on-write). Закрытый объект начинает свой жизненный путь точно так же, как разделяемый, имея одну-единственную копию в физической памяти. Например, на рис. 9.27 (а) показана ситуация, когда два процесса отображали закрытый объект в различные области своей виртуальной памяти, но в то же время совместно используют одну и ту же физическую копию. Для каждого процесса, отображающего закрытый объект, элементы таблицы страниц для соответствующей закрытой области отмечены как доступные только для чтения, а структура с информацией об этой области отмечена как поддерживающая *копирование при записи*. Пока никакой из процессов не пытается изменить свою закрытую область, они продолжают совместно использовать единственную копию объекта в физической памяти. Однако как только любой из них попытается изменить какую-либо страницу в своей закрытой области, операция записи сгенерирует исключение нарушения защиты.

Когда обработчик исключения заметит, что ошибка инициирована системой защиты при попытке записи в страницу закрытой области, он создаст новую копию страницы в физической памяти, скорректирует соответствующий элемент таблицы страниц, чтобы тот указывал на новую копию, и затем установит разрешение на запись в эту страницу, как показано на рис. 9.27 (б). Когда обработчик вернет управление, процес-

сор повторно выполнит инструкцию записи, которая на этот раз преуспевает и изменит данные во вновь созданной странице.



**Рис. 9.27.** Закрытый объект с поддержкой копирования при записи.

(а) После того, как оба процесса отображали закрытый объект с поддержкой копирования при записи. (б) После того, как процесс 2 изменил содержимое страницы в закрытой области

Откладывая копирование страниц закрытых объектов до самого последнего момента, копирование при записи существенно повышает эффективность использования дефицитной физической памяти.

## 9.8.2. И снова о функции `fork`

Теперь, когда вы получили представление о виртуальной памяти и отображении в память, мы можем прояснить, как функция `fork` создает новый процесс, обладающий своим изолированным виртуальным адресным пространством.

Когда функция `fork` вызывается текущим процессом, ядро создает различные структуры данных для *нового процесса* и назначает ему уникальный идентификатор PID. Чтобы выделить виртуальную память для нового процесса, ядро создает точные копии структуры `mm_struct` текущего процесса, структур областей и таблиц страниц, а также отмечает каждую страницу в обоих процессах как доступную только для чтения, а каждую структуру области – как закрытую, с копированием при записи.

Когда функция `fork` возвращает управление новому процессу, тот уже имеет точную копию виртуальной памяти, какой она была на момент вызова функции `fork`. Когда какой-либо из процессов попытается выполнить запись в память, механизм копирования при записи создаст новую страницу, обеспечивая тем самым изолированность адресного пространства для каждого процесса.

## 9.8.3. И снова о функции `execve`

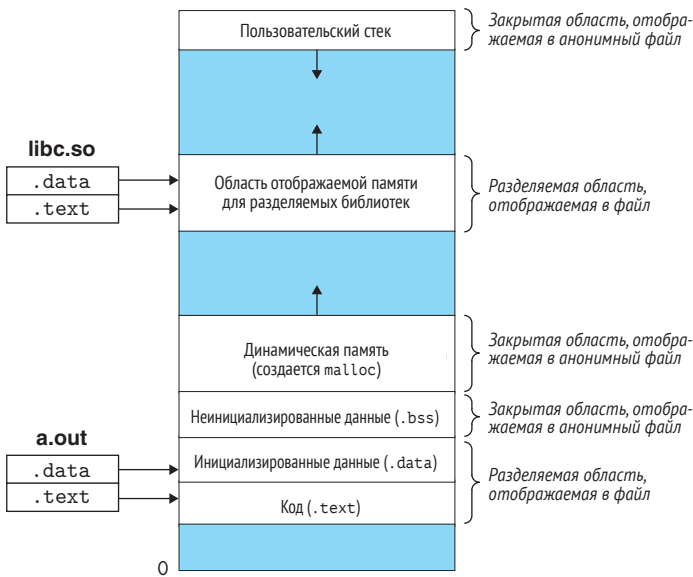
Механизмы виртуальной памяти и отображения в память играют ключевую роль в процессе загрузки программ в память. Теперь, после подробного знакомства с этими понятиями, мы вполне можем разобраться, как функция `execve` на самом деле загружает и выполняет программы. Предположим, что программа, выполняемая в текущем процессе, делает следующий вызов функции `execve`:

```
execve("a.out", NULL, NULL);
```



Как рассказывалось в главе 8, функция `execve` загружает и запускает в текущем процессе программу из выполняемого объектного файла `a.out`, фактически замещая текущую программу программой `a.out`. Загрузка и запуск `a.out` требуют выполнения следующих шагов.

1. *Удалить существующие пользовательские области:* удалить все структуры областей в пользовательской части виртуального адресного пространства текущего процесса.
2. *Отобразить закрытые области:* создать новые структуры областей для кода, данных, неинициализированных данных и стека новой программы. Все эти новые области являются закрытыми, копируемыми при записи. В области кода и данных отображаются разделы кода и данных из файла `a.out`. Область неинициализированных данных заполняется нулями, и в нее отображается анонимный файл, размер которого содержится в `a.out`. Стек и область динамической памяти тоже заполняются нулями и первоначально имеют нулевую длину. На рис. 9.28 показаны отображения закрытых областей.



**Рис. 9.28.** Как загрузчик отображает области пользовательского адресного пространства

3. *Отобразить совместно используемые области:* если программа `a.out` была скомпонована с разделяемыми объектами, такими как стандартная библиотека `libc.so`, то эти объекты динамически связываются с программой и затем отображаются в совместно используемую область виртуального адресного пространства пользователя.
4. *Установить счетчик инструкций (Program Counter, PC):* последнее, что должна сделать `execve`, – установить счетчик инструкций в контексте текущего процесса на точку входа в области кода.

Когда в следующий раз система запланирует этот процесс, его выполнение начнется с этой точки входа. По мере необходимости операционная система Linux будет подкачивать с диска страницы с кодом и данными.

### 9.8.4. Отображение в память на уровне пользователя с помощью функции `mmap`

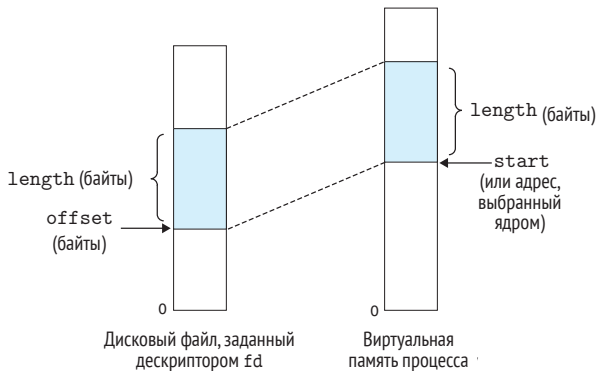
Процессы в операционной системе Linux могут использовать функцию `mmap` для создания новых областей в виртуальной памяти и отображения объектов в эти области.

```
#include <unistd.h>
#include <sys/mman.h>
```

```
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Возвращает указатель на отображенную область в случае успеха, `MAP_FAILED` (`-1`) в случае ошибки

Функция `mmap` запрашивает у ядра создать новую область виртуальной памяти, желательно начинающуюся с адреса `start`, в которую отображается непрерывный участок объекта, заданного дескриптором файла `fd`. Этот непрерывный участок имеет размер `length` байт и смещение `offset` байт от начала файла. Адрес `start` представляет условную точку отсчета, обычно в этом аргументе передается `NULL`. Далее мы будем полагать, что начальный адрес всегда `NULL`. Графическая интерпретация этих аргументов приводится на рис. 9.29.



**Рис. 9.29.** Визуальная интерпретация аргументов функции `mmap`

В аргументе `prot` передается набор битов, описывающих правила доступа к вновь отображенной области виртуальной памяти (т. е. разряды `vm_prot` в соответствующей структуре области).

**PROT\_EXEC.** Страницы в области содержат инструкции, которые могут выполняться процессором.

**PROT\_READ.** Страницы в области доступны для чтения.

**PROT\_WRITE.** Страницы в области доступны для записи.

**PROT\_NONE.** Доступ к страницам в области запрещен.

В аргументе `flags` передается набор битов, описывающих тип отображаемого объекта. Если установлен флаг `MAP_ANON`, то страница хранится в анонимном файле и соответствующие виртуальные страницы должны создаваться по требованию и заполняться нулями. Флаг `MAP_PRIVATE` означает закрытый объект, копируемый при записи, а разряд `MAP_SHARED` – совместно используемый объект. Например, вызов

```
bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE|MAP_ANON, 0, 0);
```

требует от ядра создать новую закрытую область виртуальной памяти размером `size`, доступную только для чтения и заполненную нулями. В случае успеха в `bufp` будет записан адрес созданной области.

Функция `munmap` удаляет указанную область виртуальной памяти:

```
#include <unistd.h>
#include <sys/mman.h>
```

```
int munmap(void *start, size_t length);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Функция `munmap` удаляет область, начинающуюся с виртуального адреса `start` с размером `length` байт. Последующие попытки обратиться к удаленной области завершатся ошибкой сегментации.

#### Упражнение 9.5 (решение в конце главы)

Напишите программу `mparsору.с`, которая использует `mmap` для копирования дискового файла произвольного размера в стандартный вывод. Программа должна принимать имя входного файла в аргументе командной строки.

## 9.9. Динамическое распределение памяти

Для создания и удаления областей виртуальной памяти всегда можно воспользоваться низкоуровневыми функциями `mmap` и `munmap`, однако в большинстве программ на С для получения дополнительной виртуальной памяти используют функции динамического распределения памяти.

Функции динамического распределения памяти обслуживают область виртуальной памяти процесса, называемую *динамической памятью*, или *кучей* (рис. 9.30). В разных системах динамическая память организована по-разному, но мы будем полагать, что это область памяти, заполненная нулями, которая начинается сразу же за областью неинициализированных данных и растет вверх, в направлении старших адресов. Для каждого процесса ядро поддерживает переменную `brk` (произносится как «брейк»), которая указывает на вершину кучи.

Динамическая память поддерживается как совокупность блоков разных размеров. Каждый блок – это непрерывный участок виртуальной памяти, который либо *распределен* (allocated), либо *свободен* (free). Распределенный блок явно резервируется для использования приложением. Свободный блок доступен для распределения. Свободный блок остается таковым до тех пор, пока он явно не будет распределен приложением. Распределенный блок остается таковым, пока он не будет освобожден приложением явно или самим механизмом динамической памяти.

Существует два вида механизмов распределения динамической памяти. Оба требуют, чтобы приложение явно выделяло блоки памяти. Они отличаются тем, как происходит освобождение выделенных блоков.

- *Механизмы явного распределения памяти* (explicit allocator) требуют, чтобы приложение явно освобождало любой распределенный блок. Например, стандартная библиотека языка С предоставляет функцию явного распределения памяти с именем `malloc`. Программы на С выделяют блоки памяти с помощью `malloc` и освобождают с помощью `free`. В программах на С++ используются аналогичные им функции `new` и `delete`.

- *Механизмы неявного распределения памяти* (implicit allocator), напротив, автоматически обнаруживают, когда выделенный блок памяти выходит из употребления и должен быть освобожден. Такие механизмы неявного распределения памяти называются также *сборщиками мусора* (garbage collector), а сам процесс автоматического освобождения неиспользуемых распределенных блоков называется *сборкой мусора* (garbage collection). Например, языки высокого уровня Lisp, ML и Java полагаются на механизм сборки мусора.

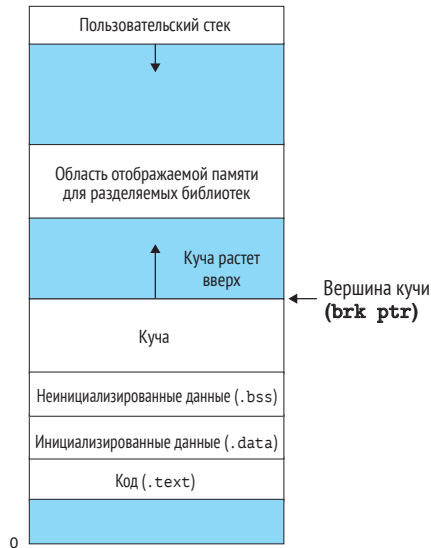


Рис. 9.30. Динамическая память (куча)

Оставшуюся часть этого раздела мы посвятим вопросам проектирования и реализации механизмов явного распределения памяти. Механизмы неявного распределения памяти мы обсудим в разделе 9.10. Более конкретно мы сосредоточимся на изучении механизмов распределения памяти, которые манипулируют динамической памятью. При этом вы должны помнить, что распределение памяти – это лишь базовая идея, которая в различных контекстах интерпретируется по-разному. Например, приложения, обрабатывающие такие структуры, как графы, часто используют стандартный механизм распределения памяти для получения крупных блоков виртуальной памяти, а затем – специальный механизм для управления памятью в пределах выделенного блока, по мере создания и уничтожения тех или иных узлов графа.

### 9.9.1. Функции malloc и free

Стандартная библиотека языка C предоставляет механизм явного распределения памяти в виде семейства функций malloc. С его помощью программы могут выделять блоки динамической памяти, вызывая функцию malloc.

```
#include <stdlib.h>
```

```
void *malloc(size_t size);
```

Возвращает указатель на выделенный блок в случае успеха,  
NULL в случае ошибки

Функция `malloc` возвращает указатель на блок памяти с размером не менее `size` байт, выровненный по границе адресов так, что может ранить любые объекты. На практике выравнивание зависит от режима компиляции кода – 32- (`gcc -m32`) или 64-разрядного (по умолчанию). В 32-разрядном режиме `malloc` возвращает блок, всегда начинающийся с адреса, кратного 8. В 64-разрядном режиме адрес всегда кратен 16.

### Каков размер слова?

При обсуждении машинного кода в главе 3 мы уже говорили о том, что в Intel называют 4-байтные объекты *двойными словами* (*double words*). Однако далее в этом разделе мы будем полагать, что *слова* – это 4-байтные объекты, а *двойные слова* – 8-байтные, что согласуется с общепринятой терминологией.

Если функция `malloc` сталкивается со сложностями (например, программа запросила памяти больше, чем доступно в виртуальной памяти), то она возвращает `NULL` и записывает номер ошибки в `errno`. Функция `malloc` не инициализирует возвращаемую память. Приложения, которым нужна инициализированная динамическая память, могут воспользоваться функцией `calloc` – тонкой оберткой вокруг функции `malloc`, которая инициализирует выделенную память нулями. Если приложению потребуется изменить размер ранее выделенного блока, оно может воспользоваться функцией `realloc`.

Механизмы распределения динамической памяти, такие как функция `malloc`, могут выделить или освободить память явно с помощью функций `mmap` и `munmap`, а также `sbrk`:

```
#include <unistd.h>
```

```
void *sbrk(intptr_t incr);
```

Возвращает старое значение `brk` в случае успеха,  
–1 в случае ошибки

Функция `sbrk` увеличивает или уменьшает кучу, добавляя приращение `incr` к указателю `brk` в ядре. Она возвращает старое значение `brk` в случае успеха и –1 в случае ошибки с записью кода ошибки `ENOMEM` в переменную `errno`. Если в `incr` передать ноль, то `sbrk` вернет текущее значение `brk`. В `incr` можно передать отрицательное значение приращения `incr`, но при этом возвращаемое значение (старое значение `brk`) будет указывать на новую вершину кучи со смещением `abs(incr)` байт.

Освободить динамическую память можно вызовом функции `free`:

```
#include <stdlib.h>
```

```
void free(void *ptr);
```

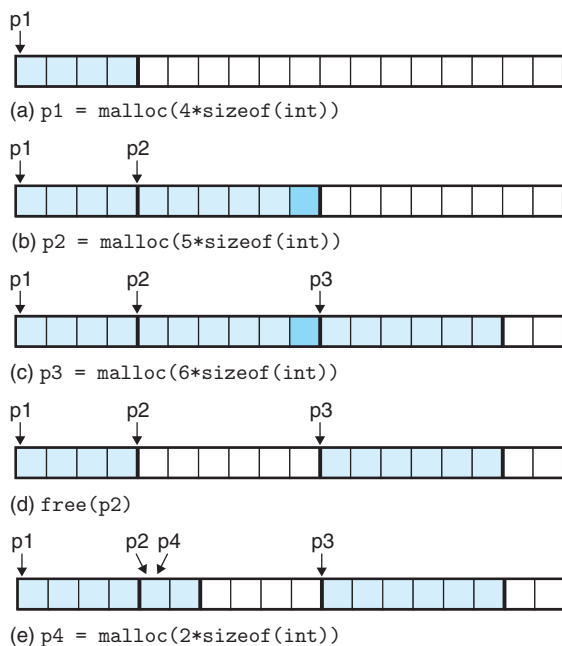
Ничего не возвращает

Аргумент `ptr` должен указывать на начало выделенного блока, который был получен вызовом функции `malloc`, `calloc` или `realloc`, иначе поведение `free` не определено. Хуже того, так как `free` ничего не возвращает, приложение останется в неведении, что что-то пошло не так. Как будет показано в разделе 9.11, это обстоятельство может вызвать определенные затруднения и появление ошибок времени выполнения.

На рис. 9.31 показано, как реализации `malloc` и `free` управляют небольшой кучей, вмещающей 18 слов. Каждая клетка – это 4-байтное слово. Прямоугольники, выделенные жирными границами, соответствуют блокам, распределенным (заштрихованы) и

свободным (не заштрихованы). Первоначально куча состоит из единственного блока длиной в 18 слов, выровненного по границе двойного слова<sup>1</sup>.

- Рисунок 9.31 (а). Программа запрашивает блок размером в 4 слова. Функция `malloc` выделяет блок с 4 словами в начале кучи и возвращает указатель на первое слово выделенного блока.
- Рисунок 9.31 (b). Программа запрашивает блок размером в 5 слов. Функция `malloc` выделяет блок с 6 словами в начале свободного блока. В этом примере `malloc` присоединяет к блоку дополнительное слово, чтобы обеспечить выравнивание свободного блока по границе двойного слова.
- Рисунок 9.31 (c). Программа запрашивает блок размером в 6 слов, и `malloc` выделяет блок с 6 словами в начале свободного блока.
- Рисунок 9.31 (d). Программа освобождает блок из 6 слов, который был выделен на рис. 9.31 (b). Обратите внимание, что когда функция `free` возвращает управление, указатель `p2` все еще указывает на освобожденный блок. Как и раньше, всю ответственность за использование `p2` несет само приложение. Последующее использование `p2` допустимо только после его инициализации новым обращением к функции `malloc`.
- Рисунок 9.31 (e). Программа запрашивает блок размером в 2 слова. В этой ситуации `malloc` выделяет часть блока, который был освобожден на предыдущем шаге, и возвращает указатель на этот новый блок.



**Рис. 9.31.** Выделение и освобождение блоков с помощью `malloc` и `free`. Каждая клетка соответствует слову. Прямоугольники, выделенные жирными границами, соответствуют блокам. Распределенные блоки заштрихованы. Области, добавленные в распределенные блоки для выравнивания, заштрихованы более темным цветом. Свободные блоки не заштрихованы. Адреса в куче увеличиваются слева направо

<sup>1</sup> Далее в этом разделе будем полагать, что все блоки выравниваются по границам 8-байтных двойных слов.

### 9.9.2. Что дает динамическое распределение памяти

Наиболее важной причиной использования в программах динамического распределения памяти является невозможность знать заранее, сколько памяти потребуется, пока программа фактически не запустится. Например, предположим, что нас попросили написать программу на С, которая читает из `stdin` целые числа и записывает их в массив. Сначала пользователь вводит число  $n$  – количество целых чисел, а потом сами числа, по одному в строке. Самое простое решение – объявить статический массив некоторого жестко заданного размера:

```
1 #include "csapp.h"
2 #define MAXN 15213
3
4 int array[MAXN];
5
6 int main()
7 {
8     int i, n;
9
10    scanf("%d", &n);
11    if (n > MAXN)
12        app_error("Input file too big");
13    for (i = 0; i < n; i++)
14        scanf("%d", &array[i]);
15    exit(0);
16 }
```

Выделение массива с жестко заданным размером, как в данном случае, часто оказывается не лучшим решением. Значение `MAXN` выбирается произвольно и не имеет никакого отношения к фактическому объему доступной виртуальной памяти. Кроме того, если пользователю потребуется ввести больше чисел, чем `MAXN`, то единственная возможность, которая позволит ему сделать это, – перекомпиляция программы с новым значением `MAXN`. В таком простом примере, как этот, сделать это совсем не сложно, однако в более крупных приложениях с миллионами строк кода и большим количеством пользователей такое жесткое ограничение размеров массива может превратиться в кошмар для тех, кто будет обслуживать программу.

Более удобное решение – размещать массив динамически во время выполнения программы, после того как значение  $n$  станет известно. При таком подходе максимальный размер массива ограничен только объемом доступной виртуальной памяти.

```
1 #include "csapp.h"
2
3 int main()
4 {
5     int *array, i, n;
6
7     scanf("%d", &n);
8     array = (int *)Malloc(n * sizeof(int));
9     for (i = 0; i < n; i++)
10         scanf("%d", &array[i]);
11     free(array);
12     exit(0);
13 }
```

Динамическое распределение памяти – полезный и важный прием, широко используемый в программировании. Однако, чтобы правильно и эффективно использовать механизмы распределения памяти, программисты должны обладать пониманием осо-

бенностей их работы. В разделе 9.11 мы рассмотрим некоторые из «ужасных» ошибок, которые могут возникать при неправильном использовании механизмов распределения памяти.

### 9.9.3. Цели механизмов распределения памяти и требования к ним

Механизмы явного распределения памяти должны сохранять работоспособность в условиях некоторых довольно строгих ограничений:

*Обработка произвольных последовательностей запросов.*

Приложение может запрашивать выделение и освобождение памяти в произвольном порядке, при условии что каждый запрос на освобождение памяти должен соответствовать блоку, распределенному предыдущим запросом на выделение памяти. То есть механизм распределения памяти не может делать никаких прогнозов относительно упорядоченности запросов на распределение и освобождение. Например, механизм распределения памяти не должен предполагать, что каждый запрос на выделение будет сопровождаться соответствующим запросом на освобождение или что соответствующие запросы на распределение и освобождение будут попарно упорядочены.

*Немедленное реагирование на запросы.*

Механизм распределения памяти должен реагировать немедленно на запросы выделения памяти. То есть он не должен менять порядок запросов или сохранять запросы в буфере с целью повышения эффективности их обслуживания.

*Использование только динамической памяти.*

Для масштабируемости механизма распределения памяти любые нескаллярные структуры данных, используемые этим механизмом, должны размещаться в динамической памяти.

*Выравнивание блоков (требование выравнивания).*

Механизм распределения памяти должен выравнивать блоки так, чтобы в них можно было хранить объекты данных любого типа.

*Неизменность содержимого выделенных блоков.*

Механизмы распределения памяти могут манипулировать только свободными блоками или менять число свободных блоков. В частности, им запрещается модифицировать или перемещать выделенные блоки. Такие методы, как уплотнение выделенных блоков, не разрешаются.

Работая в условиях этих ограничений, авторы механизмов распределения памяти часто пытаются удовлетворить противоречивые требования: обеспечение максимальной производительности и максимально эффективное использование памяти.

*Требование 1: максимальная производительность.*

Пусть дана некоторая последовательность из  $n$  запросов на распределение и освобождение памяти

$$R_0, R_1, \dots, R_k, \dots, R_{n-1}$$

и требуется обеспечить максимальную *производительность* механизма распределения памяти, которая определяется как число запросов, выполненных в единицу времени. Например, если механизм распределения памяти выполняет 500 запросов на выделе-



ние и 500 запросов на освобождение памяти за 1 секунду, то его производительность составляет 1000 операций в секунду. В общем случае достичь максимальной производительности можно за счет уменьшения среднего промежутка времени, необходимого для удовлетворения запроса на распределение или освобождение памяти. Как будет показано далее, не так уж трудно создать механизм распределения памяти с достаточно высокой производительностью в условиях, когда в худшем случае продолжительность обработки запроса линейно зависит от числа свободных блоков, а время обработки запроса на освобождение постоянно.

*Требование 2: максимальная эффективность использования памяти.*

Некоторые начинающие программисты часто неправильно полагают, что виртуальная память – это неограниченный ресурс. На самом деле суммарный объем виртуальной памяти, отведенной всем процессам в системе, ограничен объемом дискового пространства, отведенного для файла или раздела подкачки. Опытные программисты понимают, что виртуальная память – это ограниченный ресурс, который следует использовать эффективно. Это особенно верно для механизмов распределения динамической памяти, работающих с большими блоками памяти.

Имеется несколько способов оценки эффективности использования динамической памяти. В нашем случае наиболее подходящим является *пиковое потребление*. Как и прежде, пусть дана некоторая последовательность запросов из  $n$  операций выделения и освобождения памяти

$$R_0, R_1, \dots, R_k, \dots, R_{n-1}.$$

Если приложение запрашивает блок с размером  $p$  байт, то выделенный в результате блок будет иметь размер *полезной области*  $p$  байт. Предположим, что было обработано  $R_k$  запросов, и обозначим через  $P_k$  *суммарный размер полезных областей* всех выделенных блоков, а через  $H_k$  – текущий (монотонно неубывающий) размер кучи. Тогда пиковое потребление памяти для первых  $k$  запросов, обозначаемое как  $U_k$ , будет определяться следующим выражением:

$$U_k = \frac{\max_{i \leq k} P_i}{H_k}.$$

Теперь цель механизма распределения памяти будет заключаться в достижении максимума пикового потребления памяти  $U_{n-1}$  на всей последовательности. Как мы увидим далее, между максимизацией производительности и пиковым потреблением есть противоречие. В частности, нетрудно написать механизм распределения памяти, обеспечивающий максимальную производительность ценой увеличенного потребления динамической памяти. Одной из интересных проблем в разработке любых механизмов распределения памяти является поиск баланса между этими двумя показателями.

#### Смягчение предположения о монотонности

В нашем определении  $U_k$  можно несколько ослабить требование к монотонности неубывания и позволить динамической памяти расти вверх и вниз, принимая  $H_k$  в качестве предельного максимума по первым  $k + 1$  запросам.

### 9.9.4. Фрагментация

Основная причина неэффективного использования динамической памяти – так называемая *фрагментация*, которая имеет место, когда память, не используемая для

других целей, недоступна для выделения по запросу. Есть две формы фрагментации: *внутренняя фрагментация* и *внешняя фрагментация*.

*Внутренняя фрагментация* возникает при выделении блока, размер которого больше размера полезной области. Это может произойти по нескольким причинам. Например, конкретная реализация механизма распределения памяти может ограничивать минимальный размер выделяемого блока, который может оказаться больше запрошенного размера полезной области. Или, как показано на рис. 9.31 (b), механизм распределения памяти может увеличить размер блока, чтобы удовлетворить ограничениям выравнивания.

Внутреннюю фрагментацию можно выразить количественно. Это не что иное, как сумма разностей между размерами выделенных блоков и размерами их полезных областей. То есть в любой момент времени величина внутренней фрагментации зависит только от последовательности предыдущих запросов и от реализации механизма распределения памяти.

*Внешняя фрагментация* возникает, когда суммарный размер свободной памяти превышает размер запрошенного объема, но нет ни одного свободного блока достаточного размера. Например, если бы запрос на рис. 9.31 (e) потребовал выделить шесть слов, а не два, то этот запрос нельзя было бы удовлетворить без подкачки ядром дополнительной виртуальной памяти, несмотря на то что в динамической памяти имеется шесть свободных слов. Осложнения объясняются тем, что эти шесть слов содержатся в двух свободных блоках.

Выразить количественно внешнюю фрагментацию намного сложнее, чем внутреннюю, потому что она зависит не только от последовательности предыдущих запросов и реализации механизма распределения памяти, но также от последовательности *будущих* запросов. Например, предположим, что после выполнения  $k$  запросов все оставшиеся свободные блоки имеют размер в четыре слова. Страдает ли такая куча от внешней фрагментации? Ответ зависит от последовательности будущих запросов. Если все будущие запросы будут требовать выделения блоков меньше, чем четыре слова, то нет никакой внешней фрагментации. С другой стороны, если один или несколько запросов потребуют выделить блок с размером больше четырех слов, то кучу действительно можно считать страдающей внешней фрагментацией.

Поскольку внешняя фрагментация трудно поддается количественной оценке и ее, как правило, невозможно предсказать, то механизмы распределения памяти обычно используют эвристику, которая помогает держать в резерве небольшое число крупных свободных блоков памяти, вместо большого числа свободных блоков памяти меньших размеров.

### 9.9.5. Вопросы реализации

Простейший воображаемый механизм распределения памяти организует динамическую память как большой массив байтов и указатель  $p$ , который изначально указывает на первый байт массива. Чтобы распределить  $size$  байт, функция `malloc` должна сохранить текущее значение  $p$  в стеке, увеличить  $p$  на величину  $size$  и вернуть вызывающей программе старое значение  $p$ . Функция `free` просто возвращает управление вызывающей программе, не производя при этом никаких действий.

Такой простейший механизм распределения памяти представляет крайнюю точку в пространстве решений. Поскольку функции `malloc` и `free` выполняют очень небольшое число инструкций, производительность такого механизма была бы исключительно высокой. Однако из-за того, что механизм распределения памяти никогда не использует повторно никаких блоков, потребление памяти было бы исключительно высоким. Более практичный механизм распределения памяти, обладающий приемлемым балансом между производительностью и потреблением памяти, должен быть способен ответить на следующие вопросы:

- *организация свободных блоков*: как вести учет свободных блоков?
- *размещение*: как выбрать подходящий свободный блок для размещения в нем распределяемого блока?
- *разбиение*: после размещения вновь выделенного блока в некотором свободном блоке памяти как поступить с оставшейся частью этого свободного блока?
- *объединение*: что делать с только что освобожденным блоком?

Далее эти вопросы рассматриваются более подробно. Поскольку основные методы размещения, разбиения и объединения имеют много общего в различных методах организации свободных блоков, мы представим их в контексте простой организации свободных блоков, которая называется неявным списком свободных блоков.

### 9.9.6. Неявные списки свободных блоков

Любой механизм распределения памяти требует наличия тех или иных структур данных, позволяющих распознавать границы блоков и отличать выделенные блоки от свободных. Многие механизмы распределения памяти встраивают соответствующую информацию непосредственно в блоки. Один простой подход показан на рис. 9.32.



Рис. 9.32. Организация простой кучи блоков

В данном случае блок состоит из *заголовка* размером в одно слово, *полезной области* и, возможно, *дополнения*. Заголовок содержит информацию о размере блока (включая заголовок и дополнение), а также признак выделен/свободен. Если установить выравнивание по границе двойного слова, то размер блока всегда кратен восьми, и три младших бита в значении, определяющем размер блока, всегда будут нулевыми. Поэтому достаточно сохранить только 29 старших разрядов из величины размера блока, а оставшиеся три разряда можно задействовать для хранения другой информации. В данном случае мы используем самый младший из этих трех разрядов для хранения признака выделен/свободен, чтобы показать, выделен данный блок или свободен. Предположим, например, что мы имеем выделенный блок с размером 24 (0x18) байта. Тогда он будет иметь такой заголовок:

0x00000018 | 0x1 = 0x00000019

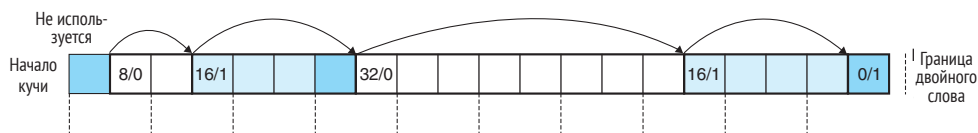
Аналогично заголовок свободного блока с размером 40 (0x28) байт будет иметь такой вид:

0x00000028 | 0x0 = 0x00000028

За заголовком следует полезная область, которая будет использоваться приложением. За полезной областью следует область дополнения, которая в общем случае может иметь любой размер. Имеется несколько причин добавления дополнения в конец выделяемого блока. Например, дополнение может играть особую роль в реализации стра-

тегии механизма распределения памяти для противодействия внешней фрагментации. Также дополнение может понадобиться для выравнивания адресов.

Для блоков, формат которых показан на рис. 9.32, можно организовать кучу в виде некоторой последовательности выделенных и свободных блоков, как показано на рис. 9.33.



**Рис. 9.33.** Организация кучи с использованием неявного списка свободных блоков.

Выделенные блоки заштрихованы. Свободные блоки не заштрихованы.  
Заголовки отмечены числами (размер (в байтах) / бит-признак выделения)

Такую организацию памяти мы будем называть *неявным списком свободных блоков*, потому что свободные блоки неявно связаны между собой полями размера в заголовках. Механизм распределения памяти может просмотреть *все* имеющиеся свободные блоки, последовательно перемещаясь между блоками динамической памяти. Обратите внимание, что для такой организации необходим некоторый специальный блок, отмечающий конец списка. В нашем примере таковым является последний блок, в заголовке которого установлен признак «выделен» и указан размер блока 0. (Как будет показано в разделе 9.9.12, наличие бита-признака выделен/свободен упрощает объединение свободных блоков в связную цепочку.)

Преимущество неявного списка свободных блоков – его простота. Существенным недостатком являются затраты на каждую операцию, такую как размещение выделенного блока, которая требует выполнить поиск в списке свободных блоков, линейно зависящий от *общего* числа выделенных и свободных блоков в куче.

Важно понимать, что системные требования к выравниванию и выбор формата блока для механизма распределения памяти определяют *минимальный размер блока*. Никакой распределенный или свободный блок не может быть меньше этого минимума. Например, при выравнивании по границе двойного слова размер каждого блока должен быть кратен длине двух слов (8 байтам). Соответственно, из формата блока на рис. 9.32 следует, что минимальный размер блока – два слова: одно слово для заголовка и еще одно обеспечивает удовлетворение требования к выравниванию. Даже если бы приложение запросило всего лишь один байт, механизм распределения памяти выделил бы блок размером в два слова.

#### Упражнение 9.6 (решение в конце главы)

Определите размеры блоков и значения в полях заголовка для следующей последовательности запросов к функции `malloc`. Исходите из следующих условий: (1) механизм распределения памяти поддерживает выравнивание по границе двойного слова и использует неявный список свободных блоков с форматом, изображенным на рис. 9.32; (2) размеры блоков округляются до ближайшего большего целого числа, кратного восьми.

Запрос	Размер блока (в десятичном формате)	Размер блока (в шестнадцатеричном формате)
<code>malloc(1)</code>	_____	_____
<code>malloc(5)</code>	_____	_____
<code>malloc(12)</code>	_____	_____
<code>malloc(13)</code>	_____	_____

### 9.9.7. Размещение распределенных блоков

Когда приложение запрашивает блок размером  $k$  байт, механизм распределения памяти просматривает список свободных блоков и отыскивает незанятый блок, размер которого достаточен, чтобы вместить затребованный блок. Алгоритм поиска определяется *стратегией размещения*. Наиболее распространенные стратегии: первый подходящий (first fit), следующий подходящий (next fit) и наиболее подходящий (best fit).

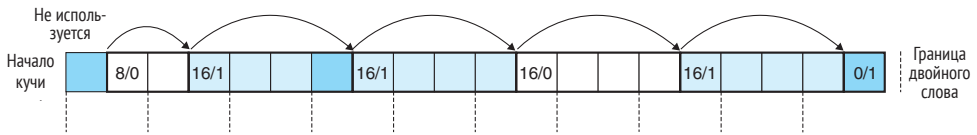
При использовании стратегии поиска *первого подходящего* механизм распределения просматривает список свободных блоков с самого начала и выбирает первый свободный блок с размером, соответствующим запросу. При использовании стратегии поиска *следующего подходящего* механизм распределения действует так же, как при использовании стратегии поиска первого подходящего, но начинает поиск не с начала списка, а с того места, на котором он остановился в прошлый раз. При использовании стратегии поиска *наиболее подходящего* механизм распределения исследует каждый свободный блок и выбирает тот, который соответствует запросу и имеет наименьший размер.

Преимущество стратегии поиска первого подходящего блока в том, что она имеет тенденцию оставлять большие свободные блоки в конце списка. Недостаток: она имеет тенденцию оставлять в начале списка мелкие «осколки» свободных блоков, что увеличивает время поиска больших блоков. Стратегия поиска следующего подходящего блока была впервые предложена Дональдом Кнудом (Donald Knuth) как альтернатива стратегии поиска первого подходящего. Основная идея этой стратегии заключается в том, что если в прошлый раз был обнаружен некоторый пригодный свободный блок, то велика вероятность, что в следующий раз пригодный блок будет обнаружен в оставшейся части списка. Стратегия поиска следующего подходящего блока может дать результат значительно быстрее, чем поиск первого подходящего блока, особенно если начальная часть списка заполнена большим количеством мелких осколков. Однако некоторые исследователи высказывают предположения, что стратегия поиска следующего подходящего блока по эффективности использования памяти несколько хуже стратегии поиска первого подходящего. Также исследователи единодушны в выводах, что стратегия поиска наиболее подходящего блока, как правило, обеспечивает наиболее экономное расходование памяти, чем стратегии поиска первого подходящего и следующего подходящего блока. Однако стратегия поиска наиболее подходящего блока зависит от организации списка свободных блоков, потому что требует просмотра всего списка свободных блоков в динамической памяти. Позже мы рассмотрим некоторые более изощренные способы организации списка свободных блоков, обеспечивающие возможность применения стратегии поиска наиболее подходящего блока без полного перебора всех свободных блоков.

### 9.9.8. Разбиение свободных блоков

Выбрав подходящий свободный блок, механизм распределения памяти должен принять следующее решение: определить, сколько памяти из свободного блока выделить для удовлетворения запроса. Один из возможных вариантов: выделить весь свободный блок. Этот подход прост в реализации и обладает хорошим быстродействием, но имеет большой недостаток – он ведет к появлению внутренней фрагментации. Если такой подход позволяет получить приемлемые результаты, то некоторая дополнительная внутренняя фрагментация может оказаться допустимой.

Однако если выбранный блок памяти далек от оптимального (по размеру намного больше запрашиваемого объема), то механизм распределения памяти обычно разбивает этот свободный блок на две части. Первая часть выделяется для передачи программе, запросившей память, а оставшаяся часть становится новым свободным блоком. На рис. 9.34 показано, как механизм распределения памяти разбивает свободный блок памяти, включающий восемь слов (рис. 9.33), чтобы удовлетворить запрос приложения на выделение трех слов динамической памяти.



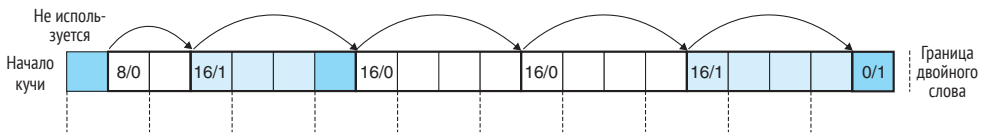
**Рис. 9.34.** Разбиение свободного блока для удовлетворения запроса на выделение из трех слов. Выделенные блоки заштрихованы. Свободные блоки не заштрихованы. Заголовки отмечены числами (размер (в байтах) / бит-признак выделения)

### 9.9.9. Увеличение объема динамической памяти

Что произойдет, если механизм распределения памяти не сможет найти подходящий участок для запрошенного блока? Один возможный вариант – попытаться объединить смежные свободные блоки, чтобы получить большой свободный блок (как рассказывается в следующем разделе). Однако если в результате этого механизму не удастся получить достаточно большой блок, то он обращается к ядру и запрашивает дополнительный объем динамической памяти, вызывая функцию `sbrk` или `mmap`. В любом случае механизм распределения памяти преобразует эту дополнительную память в один большой свободный блок, добавляет его в список свободных блоков, а затем выделяет из него память для удовлетворения запроса.

### 9.9.10. Объединение свободных блоков

Когда механизм распределения памяти освобождает выделенный блок, может оказаться, что смежными с ним могут оказаться другие свободные блоки. Наличие смежных свободных блоков может вызвать эффект, называемый *ложной фрагментацией*, когда имеется много доступной свободной памяти, нарезанной на слишком маленькие свободные блоки. Например, на рис. 9.35 показан результат освобождения блока, который был выделен на рис. 9.34. В результате появились два смежных свободных блока с полезными областями по три слова в каждом. В силу этого обстоятельства последующий запрос на выделение четырех слов окажется невыполнимым, хотя общий размер этих двух свободных блоков достаточен, чтобы удовлетворить запрос.



**Рис. 9.35.** Пример ложной фрагментации. Выделенные блоки заштрихованы. Свободные блоки не заштрихованы. Заголовки отмечены числами (размер (в байтах) / бит-признак выделения)

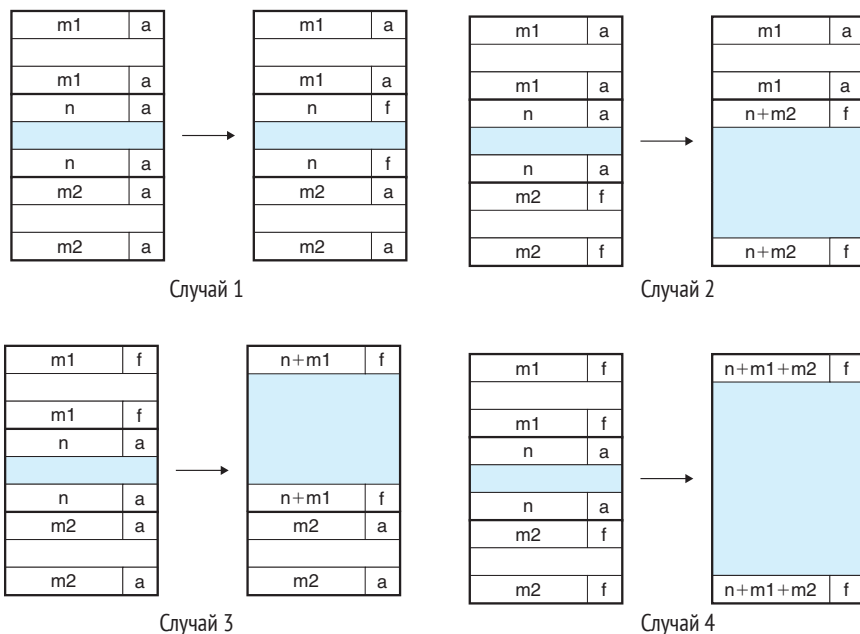
Для предотвращения ложной фрагментации любой механизм распределения памяти должен объединять смежные свободные блоки в процессе так называемого *объединения* (coalescing). Выбор момента, когда выполнять объединение, имеет немаловажное значение. Механизм распределения памяти может осуществлять *объединение немедленно* – при каждом освобождении очередного блока – или откладывать его, например до первой неудачной попытки удовлетворить какой-то запрос, после чего просмотреть всю динамическую память и объединить все смежные свободные блоки.

Немедленное объединение осуществляется достаточно просто и может быть выполнено за постоянное время, но при некоторых особенностях последовательностей запро-





На рис. 9.37 показано, как можно объединить блоки в каждом из этих четырех случаев.



**Рис. 9.37.** Объединение блоков с граничными тегами. *Случай 1:* предыдущий и следующий выделены. *Случай 2:* предыдущий выделен, следующий свободен. *Случай 3:* предыдущий свободен, следующий выделен. *Случай 4:* следующий и предыдущий свободны

В случае 1 оба смежных блока распределены, поэтому объединение невозможно. То есть состояние текущего блока просто изменяется с «выделен» на «свободен». В случае 2 текущий блок объединяется со следующим. Заголовок текущего блока и граничный тег следующего блока корректируются – суммируются размеры текущего и следующего блоков. В случае 3 предыдущий блок объединяется с текущим. Заголовок предыдущего блока и граничный тег текущего блока корректируются – суммируются размеры этих двух блоков. В случае 4 объединяются все три блока и образуется один свободный блок с заголовком в предыдущем блоке и граничным тегом в следующем блоке, отражающими суммарный размер этих трех блоков. В каждом случае объединение выполняется за постоянное время.

Идея граничных тегов проста и элегантна, она обобщает множество видов механизмов распределения памяти и списков свободных блоков. И все же она имеет свой недостаток. Требование наличия в каждом блоке заготовка и граничного тега может привести к существенным непроизводительным затратам памяти, если приложение выделяет большое количество маленьких блоков. Например, если приложение обрабатывает графы, динамически создает и уничтожает узлы графов, многократно вызывая функции `malloc` и `free`, и каждый узел занимает лишь несколько слов, то заголовок и граничный тег будут занимать половину каждого распределенного блока.

К счастью, есть возможность оптимизировать вариант с граничными тегами – использовать их только тогда, когда они действительно необходимы, то есть только в свободных блоках. Как вы помните из описания выше, граничный тег с полем размера в предыдущем блоке необходим, только когда проверяется возможность объединения текущего блока с предыдущим и следующим и только когда предыдущий блок *свободен*.



Если организовать хранение признака «выделен/свободен» для предыдущего блока в одном из неиспользуемых разрядов в заголовке текущего блока, то можно отказаться от граничного тега в выделенных блоках и использовать это пространство для полезной области. Но обратите внимание, что свободные блоки по-прежнему должны иметь граничный тег.

#### Упражнение 9.7 (решение в конце главы)

Определите минимальный размер блока для каждой из следующих комбинаций требований к выравниванию и форматов блоков при следующих условиях: неявный список свободных блоков, полезная область нулевого размера не допускается, а заголовок и граничный тег занимают 4 байта каждый.

Граница выравнивания	Выделенный блок содержит	Свободный блок содержит	Минимальный размер блока (байт)
Одиночное слово	Заголовок и граничный тег	Заголовок и граничный тег	_____
Одиночное слово	Заголовок без граничного тега	Заголовок и граничный тег	_____
Двойное слово	Заголовок и граничный тег	Заголовок и граничный тег	_____
Двойное слово	Заголовок без граничного тега	Заголовок и граничный тег	_____

### 9.9.12. Все вместе: реализация простого механизма распределения памяти

Разработка механизма распределения памяти – достаточно сложная задача. Пространство решений обширно, имеется множество альтернативных форматов блоков, списков свободных блоков, способов размещения, разбиения и объединения блоков. Еще одна сложность связана с необходимостью выйти за пределы безопасной и привычной системы типов и полагаться на небезопасное приведение типов указателей и арифметику с указателями, то есть на приемы, которые обычно применяются для программирования низкоуровневых систем. Несмотря на то что для механизмов распределения памяти не характерны большие объемы программного кода, они требуют определенного искусства и не прощают ошибок. Читатели, знакомые с языками программирования высокого уровня, такими как C++ или Java, часто упираются в концептуальную стену, когда впервые сталкиваются с таким стилем программирования. Чтобы помочь вам преодолеть это препятствие, мы займемся разработкой простого механизма распределения памяти с неявным списком свободных блоков и немедленным объединением с использованием граничных тегов. Максимальный размер блока, поддерживаемый нашим механизмом, будет ограничен  $2^{32} = 4$  Гбайтами. Код будет чисто 64-разрядным, который способен выполняться в 32- (gcc -m32) и 64-разрядных (gcc -m64) процессах.

#### Общая архитектура механизма распределения памяти

Наш механизм распределения памяти будет использовать модель памяти, предоставляемую модулем memlib.c, исходный код которого приводится в листинге 9.1. Назначение модели: дать возможность пользоваться нашим механизмом распределения памяти без обращения к существующему системному пакету malloc.

Листинг 9.1. memlib.c: модель памяти

```

1 /* Закрытые глобальные переменные */
2 static char *mem_heap; /* Указывает на первый байт в куче */
3 static char *mem_brk; /* Указывает на последний байт в куче плюс 1 */
4 static char *mem_max_addr; /* Максимальный допустимый адрес в куче плюс 1 */
5
6 /*
7  * mem_init - Инициализирует модель памяти
8  */
9 void mem_init(void)
10 {
11     mem_heap = (char *)Malloc(MAX_HEAP);
12     mem_brk = (char *)mem_heap;
13     mem_max_addr = (char *)(mem_heap + MAX_HEAP);
14 }
15
16 /*
17  * mem_sbrk - Упрощенная модель функции sbrk. Увеличивает размер кучи
18  * на incr байт и возвращает начальный адрес новой области. В этой
19  * модели куча не может уменьшаться.
20  */
21 void *mem_sbrk(int incr)
22 {
23     char *old_brk = mem_brk;
24
25     if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
26         errno = ENOMEM;
27         fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
28         return (void *)-1;
29     }
30     mem_brk += incr;
31     return (void *)old_brk;
32 }

```

Функция `mem_init` моделирует виртуальную память, доступную для организации динамической памяти, как крупный, выровненный по границе двойного слова массив байтов. Байты, заключенные между `mem_heap` и `mem_brk`, представляют распределенную виртуальную память. Байты, следующие за `mem_brk`, представляют незанятую виртуальную память. Механизм распределения памяти запрашивает дополнительную память для расширения динамической памяти, обращаясь к функции `mem_sbrk`, которая имеет тот же интерфейс, что и системная функция `sbrk`, и ту же семантику, за исключением того, что она не принимает запросы на уменьшение объема динамической памяти.

Реализация самого механизма распределения памяти находится в файле (`mm.c`), который пользователи могут компилировать и включать в свои приложения. Механизм распределения памяти экспортирует прикладным программам три функции:

```

1 extern int mm_init(void);
2 extern void *mm_malloc (size_t size);
3 extern void mm_free (void *ptr);

```

Функция `mm_init` инициализирует механизм распределения памяти, возвращая 0 в случае успеха и -1 в случае ошибки. Функции `mm_malloc` и `mm_free` имеют такие же интерфейсы и семантику, как и их системные аналоги. Механизм распределения памяти использует формат блока, показанный на рис. 9.36. Минимальный размер блока 16 байт.

Список свободных блоков организован как неявный список свободных блоков, изображенный на рис. 9.38.

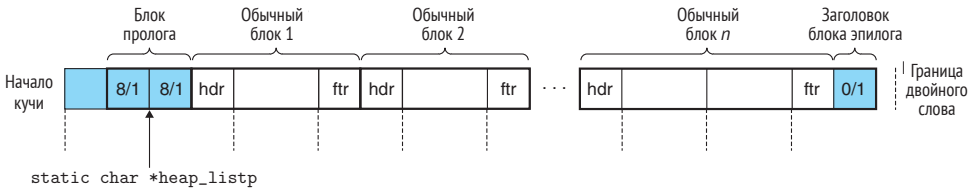


Рис. 9.38. Инвариантная форма неявного списка свободных блоков

Первое слово заполнено незначащей информацией и используется для выравнивания по границе двойного слова. Далее следует специальный выделенный блок *пролога* – 8-байтный выделенный блок, состоящий только из заголовка и граничного тега. Блок пролога формируется в процессе инициализации и никогда не освобождается. За блоком пролога следует некоторое количество обычных блоков, которые формируются при обращении к функциям `malloc` и `free`. В конце динамической памяти располагается специальный блок *эпилога* – выделенный блок нулевого размера, состоящий только из заголовка. Блоки пролога и эпилога служат для устранения граничных эффектов при слиянии блоков. Механизм распределения памяти использует отдельную закрытую (статическую) глобальную переменную (`heap_listp`), которая всегда указывает на блок пролога. (С целью оптимизации в нашей реализации она будет указывать на следующий блок, а не на блок пролога.)

## Основные константы и макроопределения для управления списком свободных блоков

В листинге 9.2 показаны некоторые основные константы, используемые механизмом распределения памяти: в строках 2–4 заданы значения некоторых основных констант, определяющих размеры: размер слова (`WSIZE`), двойного слова (`DSIZE`) и начального свободного блока, который одновременно определяет размер по умолчанию для расширения динамической памяти (`CHUNKSIZE`).

Листинг 9.2. Основные константы и макроопределения для управления списком свободных блоков

*code/vm/malloc/mm.c*

```

1 /* Основные константы и макроопределения */
2 #define WSIZE 4 /* Размер слова и заголовка / граничного тега (байты) */
3 #define DSIZE 8 /* Размер двойного слова (байты) */
4 #define CHUNKSIZE (1<<12) /* Размер увеличения динамической памяти (байты) */
5
6 #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8 /* Упаковывает в слово размер блока и бит-признак "выделен/свободен" */
9 #define PACK(size, alloc) ((size) | (alloc))
10
11 /* Чтение и запись слова по адресу p */
12 #define GET(p) (*(unsigned int *) (p))
13 #define PUT(p, val) (*(unsigned int *) (p) = (val))
14
15 /* Чтение размера и бита-признака "выделен/свободен" по адресу p */
16 #define GET_SIZE(p) (GET(p) & ~0x7)

```

```

17 #define GET_ALLOC(p) (GET(p) & 0x1)
18
19 /* Вычисляют адрес заголовка и граничного тега по указателю bp на блок */
20 #define HDRP(bp) ((char *)(bp) - WSIZE)
21 #define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp))) - DSIZE)
22
23 /* Вычисляют адреса следующего и предыдущего блоков по указателю bp на блок */
24 #define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
25 #define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))

```

*code/vm/malloc/mm.c*

Управление заголовками и граничными тегами в списке свободных блоков сопряжено с некоторыми неудобствами, потому что связано с использованием арифметики указателей. Для преодоления этих неудобств мы сочли полезным реализовать небольшой набор макроопределений, упрощающих операции со списком свободных блоков (строки 9–25). Макроопределение PACK (строка 9) объединяет размер и бит-признак «выделен/свободен» и возвращает значение, которое можно сохранить в заголовке и в граничном теге.

Макроопределение GET (строка 12) читает и возвращает слово, на которое ссылается параметр p. Здесь важную роль играет преобразование типов. Параметр p – это обычно указатель void \*, который нельзя разыменовать непосредственно. Аналогично макроопределение PUT (строка 13) сохраняет значение val в слове, на которое указывает параметр p.

Макроопределения GET\_SIZE и GET\_ALLOC (строки 16–17) возвращают размер и бит-признак «выделен/свободен» блока распределения соответственно из заголовка или из граничного тега по адресу p. Остальные макроопределения оперируют *указателями на блоки* (обозначаются как bp), которые ссылаются на первый байт полезной области. Принимая указатель на блок bp, макроопределения HDRP и FTRP (строки 20–21) возвращают указатели на заголовок и граничный тег блока соответственно. Макроопределения NEXT\_BLKP и PREV\_BLKP (строки 24–25) возвращают указатели на следующий и предыдущий блоки соответственно.

Макроопределения для манипулирования списком свободных блоков можно комбинировать самыми разными способами. Например, вот как можно определить размер следующего блока по указателю bp на текущий блок:

```
size_t size = GET_SIZE(HDRP(NEXT_BLKP(bp)));
```

## Создание начального списка свободных блоков

Прежде чем вызывать функции mm\_malloc и mm\_free, приложение должно инициализировать динамическую память, обратившись к функции mm\_init (листинг 9.3).

**Листинг 9.3.** mm\_init инициализирует динамическую память и создает начальный свободный блок

*code/vm/malloc/mm.c*

```

1 int mm_init(void)
2 {
3     /* Создает пустой блок динамической памяти */
4     if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
5         return -1;
6     PUT(heap_listp, 0); /* Выравнивающее дополнение */
7     PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Заголовок пролога */
8     PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Граничный тег пролога */
9     PUT(heap_listp + (3*WSIZE), PACK(0, 1)); /* Заголовок эпилога */

```

```

10 heap_listp += (2*WSIZE);
11
12 /* Увеличить объем динамической памяти свободным блоком CHUNKSIZE байт */
13 if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14     return -1;
15 return 0;
16 }

```

*code/vm/malloc/mm.c*

Функция `mm_init` получает четыре слова из системы памяти и инициализирует их, образуя пустой список свободных блоков (строки 4–10). После этого она вызывает функцию `extend_heap` (листинг 9.4), которая увеличивает объем динамической памяти на `CHUNKSIZE` байт и формирует начальный свободный блок. После этого механизм распределения динамической памяти инициализирован и готов принимать от приложений запросы на выделение и освобождение памяти.

**Листинг 9.4.** Функция `extend_heap`, увеличивающая объем динамической памяти

```

1 static void *extend_heap(size_t words)
2 {
3     char *bp;
4     size_t size;
5
6     /* Выделить четное число слов, согласно требованиям к выравниванию */
7     size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8     if ((long)(bp = mem_sbrk(size)) == -1)
9         return NULL;
10
11     /* Инициализировать заголовок/тег и заголовок эпилога свободного блока */
12     PUT(HDRP(bp), PACK(size, 0)); /* Заголовок свободного блока */
13     PUT(FTRP(bp), PACK(size, 0)); /* Граничный тег свободного блока */
14     PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* Заголовок нового эпилога */
15
16     /* Объединить, если предыдущий блок свободный */
17     return coalesce(bp);
18 }

```

*code/vm/malloc/mm.c*

Функция `extend_heap` вызывается в двух случаях: (1) при инициализации динамической памяти и (2) когда функция `mm_malloc` не может найти подходящий блок. Для удовлетворения требований к выравниванию функция `extend_heap` округляет запрошенный размер до ближайшего большего целого числа, кратного 2 словам (8 байт), а затем запрашивает у системы памяти дополнительный объем динамической памяти (строки 7–9).

Оставшаяся часть функции `extend_heap` (строки 12–17) несколько сложнее. Динамическая память начинается на границе двойного слова, и каждый вызов функции `extend_heap` возвращает блок с размером, кратным целому числу двойных слов. То есть каждый вызов функции `mem_sbrk` возвращает выровненный по границе двойного слова фрагмент памяти, следующий сразу за заголовком блока эпилога. Этот заголовок становится заголовком нового свободного блока (строка 12), а последнее слово этого фрагмента – новым заголовком блока эпилога (строка 14). Наконец, если предыдущий фрагмент динамической памяти заканчивается свободным блоком, вызывается функция `coalesce`, которая объединяет эти два свободных блока и возвращает указатель на объединенный блок (строка 17).

## Освобождение и объединение блоков

Приложение освобождает выделенный ранее блок, обращаясь к функции `mm_free` (листинг 9.5), которая освобождает указанный блок (`bp`) и объединяет смежные свободные блоки, используя метод граничных тегов, описанный в разделе 9.9.11.

**Листинг 9.5.** `mm_free` освобождает блок и использует метод граничного тега для объединения со смежными свободными блоками

```
code/vm/malloc/mm.c
1 void mm_free(void *bp)
2 {
3     size_t size = GET_SIZE(HDRP(bp));
4
5     PUT(HDRP(bp), PACK(size, 0));
6     PUT(FTRP(bp), PACK(size, 0));
7     coalesce(bp);
8 }
9
10 static void *coalesce(void *bp)
11 {
12     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKPTR(bp)));
13     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKPTR(bp)));
14     size_t size = GET_SIZE(HDRP(bp));
15
16     if (prev_alloc && next_alloc) { /* Случай 1 */
17         return bp;
18     }
19
20     else if (prev_alloc && !next_alloc) { /* Случай 2 */
21         size += GET_SIZE(HDRP(NEXT_BLKPTR(bp)));
22         PUT(HDRP(bp), PACK(size, 0));
23         PUT(FTRP(bp), PACK(size, 0));
24     }
25
26     else if (!prev_alloc && next_alloc) { /* Случай 3 */
27         size += GET_SIZE(HDRP(PREV_BLKPTR(bp)));
28         PUT(FTRP(bp), PACK(size, 0));
29         PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
30         bp = PREV_BLKPTR(bp);
31     }
32
33     else { /* Случай 4 */
34         size += GET_SIZE(HDRP(PREV_BLKPTR(bp))) +
35             GET_SIZE(FTRP(NEXT_BLKPTR(bp)));
36         PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
37         PUT(FTRP(NEXT_BLKPTR(bp)), PACK(size, 0));
38         bp = PREV_BLKPTR(bp);
39     }
40     return bp;
41 }
```

code/vm/malloc/mm.c

Вспомогательная функция `coalesce` реализует четыре случая, изображенных на рис. 9.37. Она имеет одну особенность, на которую следует обратить внимание. Выбранный нами формат списка свободных блоков – с блоками пролога и эпилога, которые всегда отмечаются как выделенные (т. е. занятые), позволяет игнорировать чреватые

неприятностями граничные условия, когда искомый блок `bp` находится в начале или в конце динамической памяти. Без этих специальных блоков код был бы менее прозрачным, более подверженным ошибкам и работал бы медленнее из-за необходимости проверять эти редко встречающиеся граничные условия при каждом освобождении блока памяти.

## Выделение блоков

Приложение запрашивает блок размером `size` байт, вызывая функцию `mm_malloc` (листинг 9.6). После проверки запроса механизм распределения памяти должен откорректировать запрошенный размер блока, чтобы зарезервировать место для заголовка и граничного тега и удовлетворить требование к выравниванию по границе двойного слова. Строки 12–13 задают минимальный размер блока 16 байт: восемь байт, чтобы удовлетворить требование к выравниванию, и еще восемь – для заголовка и граничного тега. Если приложение запросило памяти больше восьми байт запросов (строка 15), то сначала прибавляются дополнительные байты, а затем результат округляется до ближайшего большего целого, кратного восьми.

**Листинг 9.6.** `mm_malloc` выделяет блок и список свободных блоков

```
code/vm/malloc/mm.c
1 void *mm_malloc(size_t size)
2 {
3     size_t asize;      /* Скорректированный размер блока */
4     size_t extendsize; /* Размер для увеличения объема динамической памяти */
5     char *bp;
6
7     /* Игнорировать ошибочные запросы */
8     if (size == 0)
9         return NULL;
10
11     /* Скорректировать размер блока для включения заголовков и выравнивания. */
12     if (size <= DSIZE)
13         asize = 2*DSIZE;
14     else
15         asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
16
17     /* Найти подходящий блок в списке */
18     if ((bp = find_fit(asize)) != NULL) {
19         place(bp, asize);
20         return bp;
21     }
22
23     /* Блок не найден. Получить больше памяти и выделить блок */
24     extendsize = MAX(asize, CHUNKSIZE);
25     if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26         return NULL;
27     place(bp, asize);
28     return bp;
29 }
```

code/vm/malloc/mm.c

После корректировки запрошенного размера функция `mm_malloc` просматривает список свободных блоков, пытаясь найти подходящий свободный блок (строка 18). Если подходящий блок найден, то `mm_malloc` выделит блок и, возможно, отделит избыточную память (строка 19), а затем вернет адрес выделенного блока.

Если `mm_malloc` не найдет подходящий блок, она увеличит объем динамической памяти, добавив новый свободный блок (строки 24–26), выделит запрошенный блок в новом свободном блоке, возможно, отделив избыточную память (строка 27) и вернет указатель на выделенный блок.

#### Упражнение 9.8 (решение в конце главы)

Реализуйте функцию `find_fit` для нашего механизма распределения памяти, описанного в разделе 9.9.12.

```
static void *find_fit (size_t asize)
```

Функция должна выбирать первый блок подходящего размера в неявном списке свободных блоков.

#### Упражнение 9.9 (решение в конце главы)

Реализуйте функцию `place` для механизма распределения памяти.

```
static void place(void *bp, size_t asize)
```

Функция должна размещать выделенный блок в начале свободного блока и разбивать его, только если размер избыточной памяти равен или больше минимального размера блока.

### 9.9.13. Явные списки свободных блоков

Неявный список свободных блоков дает простой способ определения некоторых основных концепций распределения памяти. Однако, поскольку время распределения линейно зависит от общего количества блоков динамической памяти, неявный список свободных блоков не подходит для реализации универсального механизма распределения памяти (хотя он может прекрасно подходить для узкоспециализированных механизмов, если заранее известно, что количество блоков динамической памяти невелико).

Более удачное решение – организовать свободные блоки в виде некоторой явной структуры данных. Поскольку по определению тело свободного блока никак не используется механизмом, то указатели, реализующие структуру данных, можно хранить в телах свободных блоков. Например, динамическую память можно организовать как двусвязный список свободных блоков, содержащий указатели `pred` (предыдущий) и `succ` (следующий) в каждом свободном блоке, как показано на рис. 9.39.

Использование двусвязного списка вместо неявного списка свободных блоков уменьшает время распределения памяти методом выбора первого подходящего блока: с линейно зависящего от общего количества блоков до линейно зависящего от количества свободных блоков. Наряду с этим время освобождения блока может быть либо линейным, либо постоянным, в зависимости от стратегии упорядочивания блоков в списке.

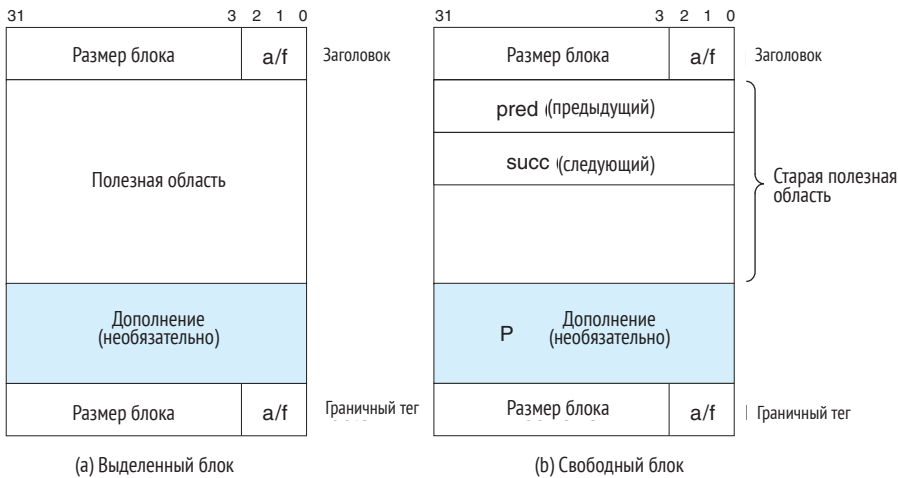
Один из подходов заключается в обслуживании списка в соответствии с дисциплиной «последним пришел – первым ушел» (Last-In First-Out, LIFO), когда последний освобожденный блок вставляется в начало списка. При использовании дисциплины LIFO и стратегии размещения выбором первого подходящего блока механизм распределения памяти просматривает сначала самые последние использованные блоки. В этом случае блоки можно освобождать за постоянное время. Если используются граничные теги, то объединение блоков также можно выполнить за постоянное время.

Другой подход заключается в обслуживании списка *в порядке следования адресов*, когда адрес каждого блока в списке меньше адреса следующего блока. В этом случае



для освобождения блока требуется выполнить поиск соответствующего предыдущего блока. Время такого поиска линейно зависит от числа блоков. Компромиссный вариант: обслуживание списка в порядке следования адресов с использованием стратегии выбора первого подходящего блока. Этот вариант отличается более экономным расходом памяти, чем обслуживание по правилу LIFO с использованием той же стратегии выбора первого подходящего блока, и близок по этому показателю к стратегии выбора наиболее подходящего блока.

В общем случае недостаток явных списков заключается в том, что свободные блоки должны быть достаточно большими, чтобы хранить все необходимые указатели, а также заголовок и, возможно, граничный тег. Это приводит к увеличению минимального размера блока и, потенциально, к более высокой степени внутренней фрагментации.



**Рис. 9.39.** Формат блоков динамической памяти при использовании двусвязного списка свободных блоков

### 9.9.14. Раздельные списки свободных блоков

Как мы видели, для размещения блока механизму распределения памяти, использующему односвязный список свободных блоков, необходимо время, линейно зависящее от числа свободных блоков. Для сокращения времени часто используется прием *разделения памяти*, суть которого состоит в том, чтобы сформировать несколько списков свободных блоков, в каждом из которых блоки имеют примерно одинаковый размер. Общая идея заключается в разбиении множества всех возможных размеров блоков на классы, называемые *классами размеров*. Классы могут определяться по-разному, например выделяются разные классы по степеням двойки:

$$\{1\}, \{2\}, \{3, 4\}, \{5-8\}, \dots, \{1025-2048\}, \{2049-4096\}, \{4097-\infty\}.$$

Или можно каждому небольшому блоку назначить свой класс, равный его размеру, а большие блоки разбивать на классы по степеням двойки:

$$\{1\}, \{2\}, \{3\}, \dots, \{1023\}, \{1024\}, \{1025-2048\}, \{2049-4096\}, \{4097-\infty\}.$$

В таком случае механизм распределения памяти будет управлять массивом списков свободных блоков, по одному списку на класс размера, упорядоченных по возрастанию размера. Когда механизму понадобится выделить блок с размером  $n$ , он просматривает соответствующий список свободных блоков и выделяет блок из этого списка, а если не находит свободный блок, то просматривает следующий список и т. д.

В литературе по динамическому распределению памяти описывается множество вариантов организации раздельных списков свободных блоков памяти, которые отличаются друг от друга организацией классов размеров, алгоритмами слияния блоков, способами обращения к операционной системе для выделения дополнительной динамической памяти, возможностью разбивать блоки и т. д. Чтобы дать вам представление о некоторых возможностях, мы опишем два наиболее часто используемых подхода: *простое разделение памяти* (simple segregated storage) и *разделение с учетом размера* (segregated fits).

### Простое разделение памяти

В методе простого разделения памяти список свободных блоков для каждого размерного класса содержит блоки одинакового размера, равного размеру наибольшего элемента в классе. Например, если некоторый класс размера определен как {17–32}, то список свободных блоков для этого класса содержит только блоки с размером 32.

Чтобы выделить блок некоторого заданного размера, мы просматриваем соответствующий список свободных блоков. Если список не пуст, то мы просто выбираем первый попавшийся блок целиком. Свободные блоки никогда не разбиваются. Если список пуст, то механизм распределения памяти запрашивает у операционной системы порцию дополнительной памяти фиксированного размера (обычно кратного размеру страницы), делит эту порцию на блоки равного размера и связывает их, формируя новый список свободных блоков. Чтобы освободить блок, механизм распределения памяти просто включает блок в начало соответствующего списка.

Эта простая схема обладает множеством преимуществ. Распределение и освобождение блоков выполняются очень быстро и за постоянное время. Кроме того, использование блоков одинаковых размеров и отсутствие необходимости разбивать и объединять блоки означают, что для управления блоками не требуется значительных затрат вычислительных ресурсов. А так как все блоки имеют одинаковый размер, то размер занятого блока можно определить по его адресу. Также благодаря отсутствию необходимости объединять блоки после освобождения бит-признак «выделен/свободен» в заголовке блока становится ненужным. Более того, ненужными становятся заголовки блоков и граничные теги. Операции выделения и освобождения памяти просто вставляют и удаляют блоки из начала списка свободных блоков, соответственно, достаточно, чтобы этот список был односвязным, а не двусвязным. В итоге в каждом блоке необходимо хранить лишь одно поле – указатель `succ` на следующий блок в списке, вследствие чего минимальный размер блока составляет одно слово.

Существенный недостаток этой схемы – подверженность внутренней и внешней фрагментации. Внутренняя фрагментация возможна ввиду того, что свободные блоки никогда не разбиваются. Хуже того, в некоторых особенных ситуациях может возникнуть недопустимая внешняя фрагментация в силу того, что свободные блоки никогда не объединяются (упражнение 9.10).

#### Упражнение 9.10 (решение в конце главы)

Опишите ситуацию с распределением памяти, которая приводит к серьезной внешней фрагментации при использовании механизма распределения, основанного на простом разделении памяти.

### Разделение с учетом размера

При использовании этого подхода механизм распределения памяти точно так же управляет массивом списков свободных блоков. Каждый список принадлежит опреде-

ленному размерному классу и организован как некоторая разновидность явного или неявного списка. Каждый такой список может содержать блоки разных размеров, соответствующих размерному классу списка. Существует много вариантов реализации алгоритма разделения с учетом размера. Далее мы опишем одну из наиболее простых таких реализаций.

Чтобы выделить блок памяти, механизм определяет класс размера и ищет первый блок подходящего размера в списке свободных блоков соответствующего размерного класса. Если блок найден, то он разбивается (при необходимости) и оставшийся неиспользуемый фрагмент вставляется в соответствующий список свободных блоков. Если блок требуемого размера не найден, то просматривается список свободных блоков следующего размерного класса. Эта процедура повторяется до тех пор, пока не будет найден подходящий блок. Если поиск по всем спискам не дал результата, то механизм распределения запрашивает дополнительную динамическую память у операционной системы, после чего выделяет блок в этой новой памяти, а ее остаток помещает в соответствующий размерный класс (обычно класс наибольшего размера). При освобождении блока выполняется объединение с соседними свободными блоками и результат помещается в соответствующий список свободных блоков.

Метод разделения с учетом размера используется во многих механизмах распределения памяти, в том числе и в пакете `malloc` проекта GNU, входящего в состав стандартной библиотеки C. Этот метод отличается высоким быстродействием и эффективностью. Время поиска сокращается за счет того, что поиск производится в узкой области, а не во всей динамической памяти. Эффективность потребления памяти может быть повышена благодаря тому интересному факту, что поиск первого подходящего блока в раздельных списках свободных блоков близок к поиску наиболее подходящего блока в полной динамической памяти.

### Метод близнецов

*Метод близнецов* (buddy system) – это специальный случай разделения с учетом размера, где каждый размерный класс соответствует степени двойки. Основная идея заключается в том, что для заданной динамической памяти размером  $2^m$  слов поддерживаются отдельные списки свободных блоков с размерами, равными  $2^k$ , где  $0 \leq k \leq m$ . Запрашиваемые размеры блоков округляются вверх до ближайшей степени двойки. Первоначально имеется один свободный блок размером  $2^m$  слов.

Чтобы выделить блок с размером  $2^k$ , отыскивается первый доступный блок с размером  $2^j$ , где  $k \leq j \leq m$ . Если  $j = k$ , то задача успешно решена. Иначе блок рекурсивно разбивается пополам, пока не выполнится условие  $j = k$ . После каждого такого разбиения вторая половина (известная как *близнец*), помещается в соответствующий список свободных блоков. При освобождении блока размером  $2^k$  выполняется последовательное объединение всех блоков-близнецов. После встречи с распределенным близнецом объединение прекращается.

Ключевая особенность метода близнецов в том, что для заданного адреса и размера блока легко вычислить адрес его близнеца. Например, блок размером 32 байта с адресом

xxx...x00000

имеет близнеца с адресом

xxx...x10000

Другими словами, адреса блока и его близнеца различаются лишь одним разрядом.

Основное преимущество метода близнецов заключается в быстром поиске и объединении. Главный недостаток – в требовании, чтобы размер блока быть кратен степени двойки, что может вызвать существенную внутреннюю фрагментацию. По этой

причине распределение памяти по методу близнецов подходит не для всех задач. Однако для некоторых специфических приложений, где заранее известно, что размеры блоков являются степенями двойки, использование метода близнецов может оказаться целесообразным.

## 9.10. Сборка мусора

С помощью механизма явного распределения памяти, такого как пакет `malloc` в языке C, приложение выделяет и освобождает блоки динамической памяти, вызывая функции `malloc` и `free`. При этом приложение несет полную ответственность за освобождение любых выделенных им блоков, которые ему больше не нужны.

Неосвобождение выделенной памяти – обычная ошибка в программировании. Например, рассмотрим функцию на языке C, которая выделяет блок памяти всякий раз, когда она вызывается:

```

1 void garbage()
2 {
3     int *p = (int *)Malloc(15213);
4
5     return; /* Массив p после этого навсегда останется в памяти */
6 }
```

Поскольку массив `p` больше не нужен программе, занимаемая им память должна быть освобождена перед возвратом из функции `garbage`. К сожалению, программист забыл освободить эту память, и она останется занятой до конца выполнения программы, недоступная для других нужд.

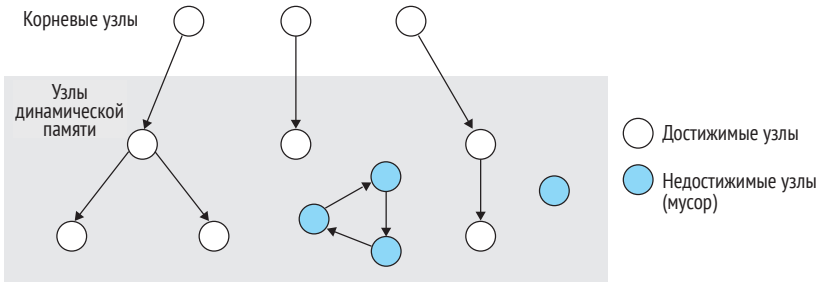
*Сборщик мусора* – это механизм распределения динамической памяти, автоматически освобождающий занятые блоки, которые больше не будут востребованы программой. Такие блоки называют «мусором» (*garbage*), откуда и произошел термин «сборщик мусора». Процесс автоматического освобождения занятой динамической памяти известен как «сборка мусора» (*garbage collection*). В системах, поддерживающих сборку мусора, приложения явно выделяют блоки динамической памяти, но никогда явно их не освобождают. В контексте программ на языке C это означает, что приложение вызывает функцию `malloc`, но никогда не вызывает функцию `free`. Вместо этого сборщик мусора периодически выявляет блоки мусора и освобождает их вызовом функции `free`, возвращая такие блоки в список свободных блоков.

Сборка мусора впервые была реализована в системах Lisp Джоном МакКарти (John McCarthy) из Массачусетского технологического института еще в начале 1960-х. Ныне она является важной частью современных языков программирования, таких как Java, ML, Perl и Mathematica, и тема эта до сих пор остается актуальной и важной областью исследований. В литературе можно найти описания удивительно большого числа подходов к сборке мусора. В своих рассуждениях мы ограничимся знакомством с оригинальным алгоритмом Mark&Sweep, предложенным МакКарти, который интересен прежде всего тем, что может быть реализован на базе существующего пакета `malloc`. Этот алгоритм дает возможность организовать сборку мусора в программах на языках C++ и C.

### 9.10.1. Основы сборки мусора

Сборщик мусора рассматривает память как ориентированный *граф достижимости* в форме, представленной на рис. 9.40. Узлы графа делятся на множество *корневых узлов* и множество *узлов динамической памяти*. Каждый узел динамической памяти соответствует выделенному блоку динамической памяти. Ориентированное ребро  $p \rightarrow q$  означает, что некоторая ячейка в блоке  $p$  ссылается на некоторую ячейку в блоке  $q$ . Корневые узлы соответствуют ячейкам за пределами динамической памяти, ссылающимся на

ячейки в динамической памяти. Такие ячейки могут быть регистрами, переменными в стеке или глобальными переменными в сегменте данных в виртуальной памяти.

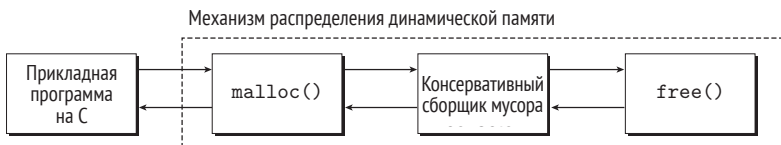


**Рис. 9.40.** Представление памяти с точки зрения сборщика мусора в виде ориентированного графа

Мы говорим, что узел  $p$  *достижим*, если существует ориентированный путь от любого корневого узла к узлу  $p$ . В любой момент времени недостижимые узлы соответствуют мусору, который никогда больше не может быть использован приложением. Роль сборщика мусора заключается в том, чтобы поддерживать некоторое представление графа достижимости и периодически выявлять недостижимые узлы, освобождая их и возвращая в список свободных блоков.

Сборщики мусора для таких языков, как ML и Java, строго контролируют создание и использование указателей в приложениях. Благодаря этому они могут поддерживать точное представление графа достижимости и автоматически утилизировать весь мусор. Однако сборщики мусора для таких языков, как C и C++, в общем случае не могут поддерживать точное представление графа достижимости. Такие сборщики мусора называют *консервативными сборщиками мусора*. Они консервативны в том смысле, что каждый достижимый блок правильно распознается как достижимый, но некоторые недостижимые узлы могут быть неправильно идентифицированы как достижимые.

Сборщики могут предоставлять свои услуги по требованию или выполняться в отдельных потоках, параллельно с приложением, непрерывно модифицируя граф достижимости и освобождая неиспользуемые блоки динамической памяти. Для примера посмотрим, как можно включить консервативный сборщик в существующий пакет malloc, по схеме, изображенной на рис. 9.41.



**Рис. 9.41.** Интеграция консервативного сборщика мусора с пакетом malloc

Приложение вызывает функцию malloc как обычно, когда ему требуется получить дополнительную динамическую память. Если функция malloc не сможет найти свободный блок, соответствующий запрошенному размеру, то она вызывает сборщик мусора в надежде, что тот вернет некоторую память, занимаемую мусором. Сборщик мусора выявляет неиспользуемые занятые блоки и возвращает их в динамическую память, вызывая функцию free. Характерная особенность этого метода заключается в том, что вместо приложения функцию free вызывает сборщик мусора. Когда сборщик возвращает управление, функция malloc снова пытается найти свободный блок соответствующего размера и в случае неудачи обращается к операционной системе за дополнительной

памятью. В конечном счете функция `malloc` возвращает указатель на выделенный блок (если попытка была успешной) или `NULL` (в случае неудачи).

### 9.10.2. Алгоритм сборки мусора Mark&Sweep

Работа алгоритма Mark&Sweep происходит в два этапа: на *этапе маркировки* (mark phase) отмечаются все достижимые и выделенные блоки – потомки корневых узлов, а на *этапе очистки* (sweep phase) освобождаются все неотмеченные выделенные блоки. Для маркировки обычно используется один из резервных младших разрядов в заголовке блока.

В нашем описании алгоритма Mark&Sweep мы будем использовать следующие функции, в которых переменная `ptr` определена как `typedef void *ptr`:

- `ptr isPtr(ptr p)` – возвращает указатель `b` на начало блока, если `p` указывает на некоторое слово в выделенном блоке, иначе возвращает `NULL`;
- `int blockMarked(ptr b)` – возвращает `true`, если блок `b` уже отмечен;
- `int blockAllocated(ptr b)` – возвращает `true`, если блок `b` выделен;
- `void markBlock(ptr b)` – отмечает блок `b`;
- `int length(ptr b)` – возвращает длину блока `b` в словах (исключая заголовок);
- `void unmarkBlock(ptr b)` – снимает отметку с блока `b`;
- `ptr nextBlock(ptr b)` – возвращает наследника блока `b` в динамической памяти.

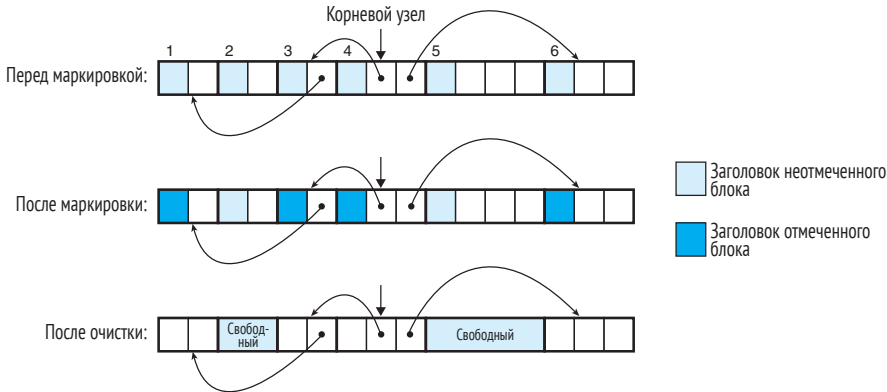
На этапе маркировки функция `mark`, показанная в листинге 9.7 (а), вызывается один раз для каждого корневого узла. Если `p` не указывает на выделенный и неотмеченный блок динамической памяти, то функция `mark` немедленно возвращает управление. Иначе она отмечает блок и рекурсивно вызывает себя для каждого слова в блоке. При каждом вызове `mark` отмечает все неотмеченные и достижимые блоки – потомки некоторого корневого узла. В конце этапа маркировки каждый выделенный блок, который не был отмечен, гарантированно недостижим и, следовательно, является мусором, который можно освободить на этапе очистки.

Этап очистки выполняется единственным вызовом функции `sweep`, показанной в листинге 9.7 (б). Функция `sweep` просматривает каждый блок динамической памяти, освобождая все встречающиеся неотмеченные и выделенные блоки (т. е. мусор).

**Листинг 9.7.** Функции `mark` и `sweep` на псевдокоде

<p>(а) Функция <code>mark</code></p> <pre>void mark(ptr p) {     if ((b = isPtr(p)) == NULL)         return;     if (blockMarked(b))         return;     markBlock(b);     len = length(b);     for (i=0; i &lt; len; i++)         mark(b[i]);     return; }</pre>	<p>(б) Функция <code>sweep</code></p> <pre>void sweep(ptr b, ptr end) {     while (b &lt; end) {         if (blockMarked(b))             unmarkBlock(b);         else if (blockAllocated(b))             free(b);         b = nextBlock(b);     }     return; }</pre>
--	---

На рис. 9.42 в графическом виде показана последовательность действий алгоритма Mark&Sweep, обслуживающего динамическую память небольшого размера. Границы блоков обозначены жирными линиями. Каждая клетка соответствует слову памяти. Каждый блок имеет заголовок с размером в одно слово, который может быть либо отмечен, либо не отмечен.



**Рис. 9.42.** Пример работы алгоритма Mark&Sweep. Обратите внимание, что стрелки в этом примере обозначают ссылки на ячейки памяти, а не на блоки в списке свободных блоков

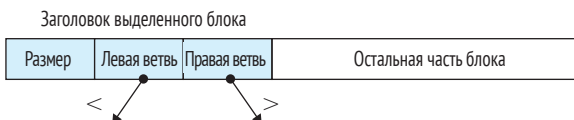
Первоначально динамическая память на рис. 9.42 состоит из шести выделенных блоков, ни один из которых не отмечен. Блок 3 содержит указатель на блок 1. Блок 4 содержит указатели на блоки 3 и 6. Корневой узел указывает на блок 4. На этапе маркировки отмечаются блоки 1, 3, 4 и 6, потому что они достижимы из корневого узла. Блоки 2 и 5 не отмечены, потому что недостижимы. После этапа очистки два недостижимых блока возвращаются в список свободных блоков.

### 9.10.3. Консервативный алгоритм Mark&Sweep для программ на С

Алгоритм Mark&Sweep с успехом можно использовать для сборки мусора в программах на языке С, потому что он не перемещает блоки, оставляя их на месте. С другой стороны, язык С предлагает несколько интересных подходов к реализации функции `isPtr`.

Прежде всего С не связывает ячейки памяти с какой-либо информацией о типе. Поэтому нет никакого очевидного способа, позволяющего функции `isPtr` определять, является ее входной параметр `p` указателем или нет. Во-вторых, даже если бы мы знали, что `p` является указателем, не существует никакого явного способа, позволяющего функции `isPtr` определить, указывает ли `p` на некоторую ячейку в полезной области выделенного блока.

Одно из решений последней проблемы – представить множество выделенных блоков в виде сбалансированного бинарного дерева, чтобы все блоки в левом поддереве были расположены в младших адресах, а все блоки в правом поддереве – в старших адресах. Как показано на рис. 9.43, для этого необходимы два дополнительных поля `left` (левое поддерево) и `right` (правое поддерево) в заголовке каждого выделенного блока. Каждое поле указывает на заголовок некоторого выделенного блока. Функция `isPtr(ptr p)` выполняет поиск в бинарном дереве выделенных блоков. На каждом шаге она, используя поле размера в заголовке блока, определяет, попадает ли `p` в пределы блока.



**Рис. 9.43.** Указатели на левую и правую ветви в сбалансированном дереве выделенных блоков



Подход на основе сбалансированных деревьев корректен в том смысле, что гарантирует маркировку всех узлов, достижимых из корневых узлов. Эта гарантия совершенно необходима, потому что пользователям приложений едва ли понравится, если выделенные ими блоки будут возвращаться в список свободных блоков. Однако этот метод консервативен в том смысле, что может неправильно отметить блоки, которые фактически недостижимы, вследствие чего некоторые неиспользуемые выделенные блоки не удастся освободить. Однако этот аспект не влияет на корректную работу прикладных программ и может лишь вызвать излишнюю внешнюю фрагментацию.

Основная причина предпочтительности консервативной версии сборщика мусора Mark&Sweep в программах на С заключается в том, что язык С не ассоциирует ячейки памяти с информацией о типе данных. Таким образом, скалярные типы, подобные `int` или `float`, могут имитировать указатели. Например, предположим, что некоторый достижимый выделенный блок содержит значение типа `int` в своей полезной области, которое случайно соответствует адресу в полезной области некоторого другого распределенного блока *b*. Не существует никакого способа, позволяющего сборщику мусора определить, что это данные типа `int`, а вовсе не указатель. Поэтому механизм распределения памяти обязательно отмечает блок *b* как достижимый, в то время как на самом деле он таковым не является.

## 9.11. Часто встречающиеся ошибки

Управление виртуальной памятью и ее использование могут оказаться трудной задачей для программистов на С, чреватой многочисленными ошибками. Ошибки, вызванные неправильным использованием памяти, относятся к числу наиболее неприятных, потому что часто проявляются не сразу, а на значительном удалении во времени и в пространстве от их источника. Стоит записать неправильные данные по неправильному адресу, и программа может крутиться часами, прежде чем совершит фатальную ошибку в какой-то другой отдаленной части. Мы завершим наше знакомство с виртуальной памятью обсуждением нескольких типичных ошибок, связанных с неправильным использованием памяти.

### 9.11.1. Разыменование недопустимых указателей

Как рассказывалось в разделе 9.7.2, в виртуальном адресном пространстве процесса имеются большие «дыры», которые не отображаются ни на какие имеющие смысл данные. Если попытаться разыменовывать указатель на одну из таких дыр, операционная система прервет выполнение программы, инициировав исключение нарушения сегментации. Кроме того, некоторые области виртуальной памяти доступны только для чтения. Попытка выполнить запись в одну из них приведет к нарушению защиты.

Известный пример разыменования недопустимого указателя – классическая ошибка при использовании функции `scanf`. Представьте, что мы решили использовать `scanf` для чтения целого числа из `stdin` в некоторую переменную. Корректный способ – передать функции `scanf` строку формата и *адрес* переменной:

```
scanf("%d", &val)
```

Однако программисты, не имеющие достаточного опыта работы с языком С (и опытные тоже!), вместо адреса передают *содержимое* `val`:

```
scanf("%d", val)
```

В этом случае функция `scanf` интерпретирует содержимое `val` как адрес и попытается записать слово по этому адресу. В лучшем случае программа завершится немедленно с исключением. В худшем случае содержимое `val` будет соответствовать некоторой доступной для чтения/записи области виртуальной памяти, и по ошибке будут затерты некоторые данные в памяти, что нередко приводит к непредсказуемым и зачастую пагубным последствиям.



### 9.11.2. Чтение неинициализированной области памяти

Ячейки памяти в сегменте .bss (например, неинициализированные глобальные переменные) всегда инициализируются загрузчиком нулями, но это не относится к динамической памяти. Многие начинающие программисты ошибочно предполагают, что динамическая память инициализируется нулями:

```

1 /* Возвращает y = Ax */
2 int *matvec(int **A, int *x, int n)
3 {
4     int i, j;
5
6     int *y = (int *)Malloc(n * sizeof(int));
7
8     for (i = 0; i < n; i++)
9         for (j = 0; j < n; j++)
10             y[i] += A[i][j] * x[j];
11     return y;
12 }
```

В этом примере программист ошибочно предположил, что вектор *y* инициализирован нулевыми значениями. В правильной реализации следует явно обнулить *y[i]* или использовать функцию *calloc*.

### 9.11.3. Переполнение буфера на стеке

Как рассказывалось в разделе 3.10.3, программа подвержена *ошибке переполнения буфера*, если выполняет запись в целевой буфер на стеке без проверки размера входной строки. Например, следующая ниже функция подвержена ошибке переполнения буфера, потому что функция *gets* копирует в буфер строку произвольной длины. Чтобы не допустить этого, следует использовать функцию *fgets*, которая ограничивает размер входной строки.

```

1 void bufoverflow()
2 {
3     char buf[64];
4
5     gets(buf); /* Здесь может возникнуть ошибка переполнения буфера */
6     return;
7 }
```

### 9.11.4. Предположение о равенстве размеров указателей и объектов, на которые они указывают

Одна из распространенных ошибок – предположение о равенстве размеров указателей и объектов, на которые они указывают:

```

1 /* Создает массив n*m*/
2 int **makeArray1(int n, int m)
3 {
4     int i;
5     int **A = (int **)Malloc(n * sizeof(int));
6
7     for (i = 0; i < n; i++)
8         A[i] = (int *)Malloc(m * sizeof(int));
9     return A;
10 }
```

В данном случае программист намеревался разместить в динамической памяти массив из  $n$  указателей, указывающих на массивы из  $m$  целых чисел. Однако в строке 5 программист написал `sizeof(int)` вместо `sizeof(int *)`, поэтому в действительности программа создает массив целых чисел.

Эта программа будет прекрасно работать на машинах, где целые числа и указатели на целые числа имеют одинаковый размер. Но если запустить эту программу, например, на машине с процессором Core i7, где указатель имеет больший размер, чем значение типа `int`, то цикл в строках 7 и 8 попытается выполнить запись за пределами массива `A`. Поскольку одно из этих слов, по-видимому, будет граничным тегом выделенного блока, то мы не сможем обнаружить ошибку, пока не попытаемся освободить блок в этой программе (гораздо позже), после чего механизм распределения памяти попытается объединить блоки и сгенерирует исключение по непонятной причине. Это пример коварного «действия на расстоянии», так характерного для ошибок, вызванных некорректным использованием памяти.

### 9.11.5. Ошибки занижения или завышения на единицу

Ошибки занижения или завышения на единицу числа подсчитываемых объектов — еще один пример распространенных ошибок переполнения буфера:

```
1 /* Создает массив n*m */
2 int **makeArray2(int n, int m)
3 {
4     int i;
5     int **A = (int **)Malloc(n * sizeof(int *));
6
7     for (i = 0; i <= n; i++)
8         A[i] = (int *)Malloc(m * sizeof(int));
9     return A;
10 }
```

Это еще одна версия программы из предыдущего раздела. Здесь в строке 5 создается массив из  $n$  указателей, но затем в строках 7 и 8 предпринимается попытка инициализировать  $n + 1$  его элементов, в результате чего происходит запись в ячейку памяти, которая следует за массивом `A`.

### 9.11.6. Ссылка на указатель вместо объекта

Если не отнестись с должным вниманием к правилам предшествования и ассоциативности операций в C, то это может привести к некорректному использованию указателя вместо объекта, на который он указывает. Например, рассмотрим следующую функцию, которая удаляет первый элемент из бинарной кучи, содержащей `*size` элементов, и затем переупорядочивает оставшиеся `*size - 1` элементов:

```
1 int *binheapDelete(int **binheap, int *size)
2 {
3     int *packet = binheap[0];
4
5     binheap[0] = binheap[*size - 1];
6     *size--; /* Должно быть (*size)-- */
7     heapify(binheap, *size, 0);
8     return(packet);
9 }
```

Цель строки 6 — уменьшить целое значение, на которое указывает `size`. Однако из-за того, что одноместные операции `--` и `*` имеют одинаковый приоритет и ассоциативность справа налево, оператор в строке 6 фактически уменьшает сам указатель вместо

целочисленного значения, на которое он указывает. Если повезет, то программа прекратит выполняться немедленно, но, скорее всего, нам придется долго искать ошибку, когда эта программа гораздо позднее выдаст неправильный ответ. Отсюда можно сделать вывод, что всегда следует использовать круглые скобки, когда есть сомнение относительно приоритета и ассоциативности. Например, в строке 6 мы могли бы четко сформулировать наше намерение в виде выражения `(*size)--`.

### 9.11.7. Неправильное понимание арифметики указателей

Еще одна распространенная ошибка – иногда начинающие программисты забывают, что арифметические операции с указателями выполняются в единицах размеров объектов, на которые они указывают, а это не обязательно байты. Например, следующая функция должна просмотреть содержимое массива целочисленных значений и вернуть указатель на первое вхождение значения `val`:

```
1 int *search(int *p, int val)
2 {
3     while (*p && *p != val)
4         p += sizeof(int); /* Должно быть p++ */
5     return p;
6 }
```

Однако из-за того, что в каждой итерации цикла (строка 4) указатель увеличивается на четыре (количество байтов в целом числе), функция проверяет только каждое четвертое целое число в массиве.

### 9.11.8. Ссылки на несуществующие переменные

Начинающие программисты, плохо понимающие устройство стека, иногда пытаются ссылаться на локальные переменные, которые уже прекратили свое существование, как показано в следующем примере:

```
1 int *stackref ()
2 {
3     int val;
4
5     return &val;
6 }
```

Эта функция возвращает указатель (пусть это будет `p`) на локальную переменную в стеке и затем выталкивает свой кадр стека. Указатель `p` будет указывать на допустимый адрес в памяти, но допустимой переменной по этому адресу уже не будет. Когда позже программа вызовет другие функции, память стека будет не раз перезаписана кадрами стека этих функций. Если потом программа присвоит некоторое значение по указателю `*p`, то это может изменить кадр стека другой функции и привести к катастрофическим и непредсказуемым последствиям.

### 9.11.9. Ссылка на данные в свободных блоках

Подобная ошибка возникает при попытке сослаться на данные в блоках динамической памяти, которые уже были освобождены. Например, рассмотрим следующий пример программы, которая в строке 6 размещает в динамической памяти целочисленный массив `x`, затем в строке 10 освобождает его, а позже, в строке 14, обращается к нему:

```
1 int *heapref(int n, int m)
2 {
3     int i;
```

```

4     int *x, *y;
5
6     x = (int *)Malloc(n * sizeof(int));
7
8     . // В этот промежуток выполняются другие вызовы malloc и free
9     .
10    free(x);
11
12    y = (int *)Malloc(m * sizeof(int));
13    for (i = 0; i < m; i++)
14        y[i] = x[i]++; /* Ошибка! x[i] находится в свободном блоке */
15
16    return y;
17 }

```

В зависимости от последовательности вызовов функций `malloc` и `free` в промежутке между строками 6 и 10 к моменту ссылки программы на `x[i]` в строке 14 память, которую прежде занимал массив `x`, могла выделяться для других нужд и хранить другие данные. Как и во многих иных случаях неправильного использования памяти, эта ошибка проявится в программе на более поздней стадии, когда обнаружится, что значения в `y` испорчены.

### 9.11.10. Утечки памяти

Утечки памяти можно сравнить с безжалостными молчаливыми убийцами, которые появляются, когда программисты выделяют динамическую память и забывают освобождать ее. Например, следующая функция выделяет блок динамической памяти и затем возвращает управление, не освобождая ее:

```

1 void leak(int n)
2 {
3     int *x = (int *)Malloc(n * sizeof(int));
4
5     return; /* x остается в динамической памяти, продолжая занимать ее */
6 }

```

Если функцию `leak` вызывать часто, то динамическая память постепенно заполнится мусором, и программа исчерпает все виртуальное адресное пространство. Утечки памяти имеют особенно тяжелые последствия для таких программ, например, как демоны и серверы, которые должны работать без перерыва.

## 9.12. Итоги

Виртуальная память – это абстракция основной памяти. Процессоры, поддерживающие виртуальную память, ссылаются на основную память, используя форму косвенной адресации, известную как виртуальная адресация. Процессор генерирует виртуальные адреса, которые преобразуются в физические перед передачей этих адресов устройству основной памяти. Преобразование виртуальных адресов в физические требует тесного взаимодействия аппаратных и программных средств. Специально разработанные аппаратные модули преобразуют виртуальные адреса, используя таблицы страниц, содержание которых поддерживается операционной системой.

Виртуальная память обеспечивает поддержку трех важных системных функций. Во-первых, это автоматическое кеширование в основной памяти только что использованного содержимого виртуального адресного пространства, хранимого на диске. Блок в кеше виртуальной памяти называется страницей. Ссылка на страницу, хранящуюся на диске, вызывает сбой страницы, в ответ на который запускается обработчик оши-

бок в операционной системе. Обработчик копирует страницу с диска в кеш в основной памяти и при необходимости записывает на диск страницу, которую пришлось вытеснить из кеша. Во-вторых, наличие виртуальной памяти упрощает управление памятью, что в свою очередь упрощает компоновку программ, совместное использование данных различными процессами, распределение памяти для процессов, а также загрузку программ. Наконец, виртуальная память упрощает защиту памяти путем включения разряда защиты в каждый элемент таблицы страниц.

Процесс преобразования адреса должен быть интегрирован с работой всех аппаратных кешей в системе. Большинство элементов таблицы страниц хранятся в кеше L1, но высокая стоимость доступа к элементам таблицы страниц в L1 обычно нивелируется кешированием элементов таблицы страниц в буфере ассоциативной трансляции TLB.

Современные системы инициализируют участки виртуальной памяти, связывая их с участками файлов на диске. Этот процесс известен как отображение в память. Отображение в память – эффективный механизм для организации совместного использования данных, создания новых процессов и загрузки программ. Приложения могут самостоятельно создавать и удалять области виртуального адресного пространства, используя функцию `mmap`. Однако большинство программ полагаются на пакеты распределения динамической памяти, такие как `malloc`, управляющие памятью в виртуальном адресном пространстве, которое называют динамической памятью. Механизмы динамического распределения памяти действуют на прикладном уровне и имеют выход на системный уровень. Они управляют памятью независимо от системы типов. Механизмы распределения памяти бывают двух видов. Явные требуют, чтобы приложение явно освобождало свои блоки памяти. Неявные (сборщики мусора) освобождают все неиспользуемые и недостижимые блоки автоматически.

Управление виртуальной памятью и ее использование – трудная задача для программистов на языке C, чреватая всевозможными ошибками. К распространенным ошибкам относятся: неправильное разыменование указателей, чтение неинициализированной памяти, переполнение буфера на стеке, предположение о равенстве размеров указателей и объектов, на которые они указывают, ссылки на указатели вместо ссылок на объекты, непонимание арифметики указателей, ссылки на несуществующие переменные и утечки памяти.

## Библиографические заметки

Первое описание виртуальной памяти было опубликовано Килбурном (Kilburn) и его коллегами [63]. Дополнительные сведения о роли аппаратных средств в организации виртуальной памяти можно найти в [46]. Дополнительную информацию о роли операционной системы можно найти в [102, 106, 113]. Подробное описание организации виртуальной памяти в Linux можно найти в книге Бовета (Bovet) и Цезати (Cesati) [11]. Корпорация Intel предоставляет подробную документацию с описанием преобразования 32- и 64-разрядных адресов на процессорах IA [52].

В 1968 г. Кнут (Knuth) написал классический труд о распределении памяти [64]. С тех пор в этой области появилось огромное количество других работ. Вилсон (Wilson), Джонстон (Johnstone), Нилай (Neely) и Болз (Boles) написали прекрасный обзор, в котором дали оценку производительности механизмов явного распределения памяти [118]. Основу этой публикации составляет описание устройства механизмов распределения памяти и стратегий, реализованных в них. Джонс (Jones) и Линс (Lins) дали всесторонний обзор проблемы сборки мусора в [56]. Керниган (Kernighan) и Ритчи (Ritchie) [61] представили законченную реализацию простого механизма распределения памяти, основанную на использовании явного списка свободных блоков с фиксацией размера блока и указателя на следующий блок. Эта реализация интересна тем, что использует прием объединения для устранения большого количества сложных арифметических



## 2. Преобразование адреса.

Параметр	Значение
VPN	<input type="text"/>
Индекс TLB	<input type="text"/>
Тег TLB	<input type="text"/>
Попадание в TLB? (Да/Нет)	<input type="text"/>
Сбой страницы? (Да/Нет)	<input type="text"/>
PPN	<input type="text"/>

## 3. Формат физического адреса.

11	10	9	8	7	6	5	4	3	2	1	0
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

## 4. Ссылка на физическую память.

Параметр	Значение
Смещение байта	<input type="text"/>
Индекс в кеше	<input type="text"/>
Тег кеша	<input type="text"/>
Попадание в кеш? (Да/Нет)	<input type="text"/>
Возвращаемый байт из кеша	<input type="text"/>

## Упражнение 9.13 ♦

Повторите упражнение 9.11 для виртуального адреса 0x0040.

## 1. Формат виртуального адреса.

13	12	11	10	9	8	7	6	5	4	3	2	1	0
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

## 2. Преобразование адреса.

Параметр	Значение
VPN	<input type="text"/>
Индекс TLB	<input type="text"/>
Тег TLB	<input type="text"/>
Попадание в TLB? (Да/Нет)	<input type="text"/>
Сбой страницы? (Да/Нет)	<input type="text"/>
PPN	<input type="text"/>

## 3. Формат физического адреса.

11	10	9	8	7	6	5	4	3	2	1	0
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

## 4. Ссылка на физическую память.

Параметр	Значение
Смещение байта	<input type="text"/>
Индекс в кеше	<input type="text"/>
Тег кеша	<input type="text"/>
Попадание в кеш? (Да/Нет)	<input type="text"/>
Возвращаемый байт из кеша	<input type="text"/>

### Упражнение 9.14 ♦♦

Дан входной файл `hello.txt`, содержащий строку `Hello, world!\n`, напишите программу на C, которая использует `mmap` для изменения содержимого файла `hello.txt` и записывает в него строку `Jello, world!\n`.

### Упражнение 9.15 ♦

Определите размеры блоков и значения в полях заголовка для следующей последовательности запросов к функции `malloc`. Исходите из следующих условий: (1) механизм распределения памяти поддерживает выравнивание по границе двойного слова и использует неявный список свободных блоков с форматом, изображенным на рис. 9.32; (2) размеры блоков округляются до ближайшего большего целого числа, кратного восьми.

Запрос	Размер блока (в десятичном формате)	Размер блока (в шестнадцатеричном формате)
<code>malloc(3)</code>	_____	_____
<code>malloc(11)</code>	_____	_____
<code>malloc(20)</code>	_____	_____
<code>malloc(21)</code>	_____	_____

### Упражнение 9.16 ♦

Определите минимальный размер блока для каждой из следующих комбинаций требований к выравниванию и форматов блоков при следующих условиях: явный список свободных блоков, 4-байтные указатели `pred` и `succ` в каждом свободном блоке, полезная область нулевого размера не допускается, а заголовок и граничный тег занимают 4 байта каждый.

Граница выравнивания	Выделенный блок содержит	Свободный блок содержит	Минимальный размер блока (байт)
Одиночное слово	Заголовок и граничный тег	Заголовок и граничный тег	_____
Одиночное слово	Заголовок без граничного тега	Заголовок и граничный тег	_____
Двойное слово	Заголовок и граничный тег	Заголовок и граничный тег	_____
Двойное слово	Заголовок без граничного тега	Заголовок и граничный тег	_____

### Упражнение 9.17 ♦♦♦

Разработайте версию механизма распределения памяти, представленного в разделе 9.9.12, которая выполняет поиск следующего подходящего блока вместо первого подходящего.

### Упражнение 9.18 ♦♦♦

Чтобы обеспечить постоянное время объединения свободных блоков, механизму распределения памяти из раздела 9.9.12 необходимы заголовок и граничный тег в каждом блоке. Измените этот механизм так, чтобы в свободных блоках по-прежнему требовались заголовок и граничный тег, но в выделенных блоках достаточно было бы иметь только заголовок.

### Упражнение 9.19 ♦

Ниже приводятся три группы высказываний, касающихся управления памятью и сборки мусора. В каждой группе только одно высказывание является истинным. Укажите, какое высказывание в каждой группе истинно.



1. (a) В методе близнецов до 50 % объема памяти может расходоваться впустую из-за внутренней фрагментации.  
 (b) Алгоритм распределения памяти методом выбора первого подходящего блока обладает меньшим быстродействием, чем алгоритм выбора методом наиболее подходящего (в среднем).  
 (c) Освобождение с использованием граничных тегов обладает достаточным быстродействием, только когда список свободных блоков упорядочен по возрастанию адресов в памяти.  
 (d) Для метода близнецов характерна внутренняя фрагментация и нехарактерна внешняя.
2. (a) Использование алгоритма выбора первого подходящего блока применительно к списку свободных блоков, упорядоченного по убыванию размеров блоков, ухудшает производительность распределения памяти, зато предотвращает внешнюю фрагментацию.  
 (b) Для алгоритма, реализующего метод выбора наиболее подходящего блока, список свободных блоков нужно упорядочить по возрастанию адресов в памяти.  
 (c) При использовании метода выбора наиболее подходящего блока выбирается наибольший свободный блок, в котором может поместиться запрошенный объем.  
 (d) Использование алгоритма выбора первого подходящего блока из списка свободных блоков, упорядоченного по возрастанию размеров блоков, эквивалентно использованию алгоритма выбора наиболее подходящего блока.
3. Сборщики мусора, реализующие алгоритм Mark&Sweep, называются консервативными, если:
  - (a) объединяют освобожденную память, только когда запрос на выделение памяти не получается удовлетворить с имеющимися блоками;
  - (b) рассматривают как указатель все, что выглядит как указатель;
  - (c) выполняют сборку мусора только после исчерпания памяти;
  - (d) не освобождают блоки памяти, образующие циклический список.

### Упражнение 9.20 ♦♦♦♦

Напишите свои версии функций malloc и free и сравните время их выполнения и потребление памяти с версиями malloc и free в стандартной библиотеке языка C.

## Решения упражнений

### Решение упражнения 9.1

Это упражнение дает некоторое представление о размерах различных адресных пространств. Было время, когда 32-разрядное адресное пространство казалось невероятно большим. Но теперь, когда имеются базы данных и научные приложения, требующие еще больших ресурсов, можно ожидать, что эта тенденция сохранится. Можно также ожидать, что наступит такой момент, когда вы будете жаловаться на недостаток 64-разрядного адресного пространства на вашем персональном компьютере!

Количество разрядов в виртуальном адресе ( $n$ )	Количество виртуальных адресов ( $N$ )	Наибольший возможный виртуальный адрес
8	$2^8 = 256$	$2^8 - 1 = 255$
16	$2^{16} = 64 \text{ К}$	$2^{16} - 1 = 64 \text{ К} - 1$
32	$2^{32} = 4 \text{ Г}$	$2^{32} - 1 = 4 \text{ Г} - 1$
48	$2^{48} = 256 \text{ Т}$	$2^{48} - 1 = 256 \text{ Т} - 1$
64	$2^{64} = 16 \text{ Е}$	$2^{64} - 1 = 16 \text{ Е} - 1$

## Решение упражнения 9.2

Поскольку каждая виртуальная страница имеет размер  $P = 2^p$  байт, то всего в системе может быть  $2^n/2^p = 2^{n-p}$  страниц, для каждой из которых должен иметься свой элемент в таблице страниц (PTE).

$n$	$P = 2^p$	Количество элементов в PTE
16	4К	16
16	8К	8
32	4К	1 М
32	8К	512 К

## Решение упражнения 9.3

Вы должны хорошо разбираться в такого рода задачах, чтобы полностью понимать принципы преобразования адресов. Вот как решается первая подзадача: нам даны виртуальные адреса длиной  $n = 32$  разряда и физические адреса длиной  $m = 24$  разряда. Размер страницы  $P = 1$  Кбайт означает, что нам нужно  $\log_2(1 \text{ Кбайт}) = 10$  разрядов и для VPO, и для PPO. (Вспомните, что смещения VPO и PPO идентичны.) Остаточные разряды в адресе – это VPN и PPN соответственно.

$P$	Число			
	разрядов в VPN	разрядов в VPO	разрядов в PPN	разрядов в PPO
1 Кбайт	22	10	14	10
2 Кбайт	21	11	13	11
4 Кбайт	20	12	12	12
8 Кбайт	19	13	11	13

## Решение упражнения 9.4

Построение этих нескольких ручных моделей – хороший способ закрепить понимание принципов преобразования адресов. Полезно было бы выписать все разряды адреса, такие как VPN, TLBI и т. д., и затем заключить их в клеточки. В этом конкретном упражнении нет промахов: буфер TLB хранит копию элемента PTE, а кеш – копии запрошенных слов данных. Некоторые разные комбинации попаданий и промахов можно подсмотреть в упражнениях 9.11–9.13.

1. Формат виртуального адреса.

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	1	1

2. Преобразование адреса.

Параметр	Значение
VPN	0xf
Индекс TLB	0x3
Тег TLB	0x3
Попадание в TLB? (Да/Нет)	Да
Сбой страницы? (Да/Нет)	Нет
PPN	0xd

## 3. Формат физического адреса.

11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	1	1

## 4. Ссылка на физическую память.

Параметр	Значение
Смещение байта	0x3
Индекс в кеше	0x5
Тег кеша	0xd
Попадание в кеш? (Да/Нет)	Да
Возвращаемый байт из кеша	0x1d

## Решение упражнения 9.5

Решение этого упражнения даст вам хорошее представление об идее отображения в память. Попробуйте выполнить его самостоятельно. Мы не рассматривали функции `open`, `fstat` и `write`, поэтому вам придется прочитать соответствующие страницы справочного руководства `man`, чтобы понять, как они работают.

*code/vm/mmapcopy.c*

```

1 #include "csapp.h"
2
3 /*
4  * mmapcopy - использует mmap для копирования содержимого файла fd в stdout
5  */
6 void mmapcopy(int fd, int size)
7 {
8     char *bufp; /* Указатель на область вирт. памяти, куда отображается файл */
9
10    bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
11    Write(1, bufp, size);
12    return;
13 }
14
15 /* Драйвер mmapcopy */
16 int main(int argc, char **argv)
17 {
18     struct stat stat;
19     int fd;
20
21     /* Проверить наличие обязательного аргумента командной строки */
22     if (argc != 2) {
23         printf("usage: %s <filename>\n", argv[0]);
24         exit(0);
25     }
26
27     /* Скопировать входной аргумент в stdout */
28     fd = Open(argv[1], O_RDONLY, 0);
29     fstat(fd, &stat);
30     mmapcopy(fd, stat.st_size);
31     exit(0);
32 }

```

*code/vm/mmapcopy.c*

## Решение упражнения 9.6

Решение этого упражнения требует использования некоторых базовых идей, таких как требования к выравниванию, минимальному размеру блока и формату заголовка. Типичный способ определения размера блока – округление суммы запрашиваемого полезного пространства и заголовка, с последующим выбором размера, равного ближайшему, соответствующему требованиям выравнивания (в нашем случае 8 байт). Например, размер блока для `malloc(1)` вычисляется так: запрошенный размер  $4 + 1 = 5$  округляется до 8. Размер блока для `malloc(13)` вычисляется так: запрошенный размер  $13 + 4 = 17$  округляется до 24.

Запрос	Размер блока (в десятичном формате)	Размер блока (в шестнадцатеричном формате)
<code>malloc(1)</code>	8	0x9
<code>malloc(5)</code>	16	0x11
<code>malloc(12)</code>	16	0x11
<code>malloc(13)</code>	24	0x19

## Решение упражнения 9.7

Значение минимального размера блока может существенно влиять на внутреннюю фрагментацию. Поэтому важно понимать, что минимальный размер блока связан с различиями в реализациях механизмов распределения памяти и в требованиях к выравниванию. Сложность в том, чтобы понять, что в разные моменты времени один и тот же блок может быть выделенным или свободным. Минимальный размер блока – это максимальное значение из минимального размера распределенных блоков и минимального размера свободных блоков. Например, в предыдущем упражнении минимальный размер выделенного блока составляет 5 байт – 4 байта заголовка и 1 байт полезной области, которые после округления превращаются в 8 байт. Минимальный размер свободного блока составляет 8 байт – 4 байта заголовка и 4 байта граничного тега, который кратен 8 и не требует округления. Следовательно, минимальный размер блока для этого механизма выделения памяти составляет 8 байт.

Граница выравнивания	Выделенный блок содержит	Свободный блок содержит	Минимальный размер блока (байт)
Одиночное слово	Заголовок и граничный тег	Заголовок и граничный тег	12
Одиночное слово	Заголовок без граничного тега	Заголовок и граничный тег	8
Двойное слово	Заголовок и граничный тег	Заголовок и граничный тег	16
Двойное слово	Заголовок без граничного тега	Заголовок и граничный тег	8

## Решение упражнения 9.8

Здесь нет ничего сложного. Но решение этого упражнения требует четкого понимания, как работает наш простой механизм распределения памяти с использованием неявного списка свободных блоков и как управлять блоками.

*code/vm/malloc/mm.c*

```
1 static void *find_fit(size_t asize)
2 {
```

```

3      /* Поиск первого подходящего */
4      void *bp;
5
6      for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLK(P(bp))) {
7          if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
8              return bp;
9          }
10     }
11     return NULL; /* Не найдено */
12 }

```

*code/vm/malloc/mm.c*

### Решение упражнения 9.9

Это еще одно упражнение для разминки, помогающее поближе познакомиться с механизмом распределения памяти. Обратите внимание, что минимальный размер блока для рассматриваемого механизма распределения памяти равен 16 байтам. Если остаток после разбиения блока больше или равен минимальному размеру блока, то мы двигаемся дальше и разбиваем блок (строки 6–10). Единственная сложность лишь в том, чтобы понять, что новый выделенный блок следует сначала разместить (строки 6 и 7) и только потом переходить к следующему блоку (строка 8).

*code/vm/malloc/mm.c*

```

1 static void place(void *bp, size_t asize)
2 {
3     size_t csize = GET_SIZE(HDRP(bp));
4
5     if ((csize - asize) >= (2*DSIZE)) {
6         PUT(HDRP(bp), PACK(asize, 1));
7         PUT(FTRP(bp), PACK(asize, 1));
8         bp = NEXT_BLK(P(bp));
9         PUT(HDRP(bp), PACK(csize-asize, 0));
10        PUT(FTRP(bp), PACK(csize-asize, 0));
11    }
12    else {
13        PUT(HDRP(bp), PACK(csize, 1));
14        PUT(FTRP(bp), PACK(csize, 1));
15    }
16 }

```

*code/vm/malloc/mm.c*

### Решение упражнения 9.10

Здесь речь идет о некоторой последовательности операций выделения и освобождения памяти, которая вызывает внешнюю фрагментацию: приложение многократно выделяет и освобождает блоки из первого размерного класса, затем многократно выделяет и освобождает блоки из второго размерного класса, потом многократно выделяет и освобождает блоки из третьего размерного класса и т. д. Для каждого размерного класса механизм распределения памяти отводит большой фрагмент памяти, которая никогда не будет востребована, потому что механизм распределения памяти не выполняет слияний, а приложение никогда повторно не запрашивает блоки из этого размерного класса.



# Часть III

## Взаимодействие программ

До сих пор при изучении компьютерных систем мы полагали, что программы выполняются изолированно, с минимумом информации на входе и на выходе. Однако в реальной практике прикладные программы используют службы операционных систем для взаимодействия с устройствами ввода/вывода, а также с другими программами.

Эта часть книги даст вам понимание основных служб ввода/вывода, предоставляемых операционными системами Unix, а также принципов использования этих служб для создания таких приложений, как веб-клиенты и серверы, взаимодействующие друг с другом через интернет. Вы познакомитесь с приемами разработки конкурентных программ, таких как веб-серверы, способных обслуживать сразу несколько клиентов. Разработка конкурентных программ также позволяет эффективнее использовать современные процессоры с многоядерной архитектурой. Закончив изучать эту часть, вы по-настоящему приблизитесь к цели досконального понимания компьютерных систем и их влияния на вновь создаваемые программы.

# Глава 10

## Системный уровень ввода/вывода

- 10.1. Ввод/вывод в Unix.
- 10.2. Файлы.
- 10.3. Открытие и закрытие файлов.
- 10.4. Чтение и запись файлов.
- 10.5. Надежные чтение и запись с помощью пакета Rio.
- 10.6. Чтение метаданных файла.
- 10.7. Чтение содержимого каталога.
- 10.8. Совместное использование файлов.
- 10.9. Переадресация ввода/вывода.
- 10.10. Стандартный ввод/вывод.
- 10.11. Все вместе: какие функции ввода/вывода использовать?
- 10.12. Итоги.

Библиографические заметки.

Домашние задания.

Решения упражнений.

**В**вод/вывод – это процесс копирования данных между основной памятью и внешними устройствами: дисковыми накопителями, терминалами и сетями. Операция ввода копирует данные из устройства ввода/вывода в основную память, а операция вывода – из памяти в соответствующее устройство.

Среда выполнения любого языка программирования предоставляет высокоуровневые средства ввода/вывода. Например, ANSI C предоставляет *стандартную библиотеку ввода/вывода* с такими функциями, как `printf` и `scanf`, выполняющими буферизованный ввод/вывод. Язык C++ предоставляет похожую функциональность в форме перегруженных операторов `<<` (вывод) и `>>` (ввод). В системах Linux эти высокоуровневые функции ввода/вывода реализуются с использованием *функций ввода/вывода Unix* системного уровня, предоставляемых ядром. В большинстве ситуаций высокоуровневые функции ввода/вывода прекрасно работают, поэтому редко возникает необходимость непосредственно использовать функции ввода/вывода Unix. Тогда возникает вопрос: зачем вообще изучать ввод/вывод Unix?

- Понимание ввода/вывода Unix позволяет понять другие системные концепции. Ввод/вывод неотделим от системы, и по этой причине пользователи часто стал-



квиваются с циклическими зависимостями между вводом/выводом и другими системными концепциями. Например, ввод/вывод играет ключевую роль в создании и выполнении процесса, и наоборот, создание процесса играет ключевую роль в совместном использовании файлов разными процессами. Таким образом, для понимания ввода/вывода по-настоящему необходимо понимание процессов, и наоборот. Мы уже коснулись аспектов ввода/вывода, когда обсуждали иерархию памяти, компоновку и загрузку, процессы и виртуальную память. Теперь, когда эти концепции стали более понятными, можно завершить круг и рассмотреть ввод/вывод подробнее.

- *Иногда нет выбора, кроме как использовать ввод/вывод Unix.* В некоторых ситуациях просто невозможно или неуместно применять высокоуровневые функции ввода/вывода. Например, стандартная библиотека ввода/вывода не дает доступа к метаданным файла, таким как его размер или время создания. Более того, при использовании стандартной библиотеки ввода/вывода могут возникать проблемы, создающие риски при программировании сетевых приложений.

В этой главе представлены общие концепции ввода/вывода систем Unix, стандартного ввода/вывода и рассказывается о том, как максимально надежно использовать их в программах на языке C. Помимо того что глава служит общим введением в предмет, она также закладывает прочный фундамент для последующего изучения сетевого программирования и конкуренции.

## 10.1. Ввод/вывод в Unix

*Файл* в Linux – это последовательность  $m$  байт:

$$B_0, B_1, \dots, B_k, \dots, B_{m-1}.$$

Все устройства ввода/вывода: сети, диски и терминалы – смоделированы в виде файлов. И ввод, и вывод выполняются путем чтения и записи в соответствующие файлы. Такое элегантное отображение устройств в файлы позволяет ядру Linux экспортировать простой низкоуровневый прикладной интерфейс, известный как *ввод/вывод в Unix*, обеспечивающий единообразное и последовательное выполнение операций ввода и вывода.

- *Открытие файлов.* Приложение объявляет о своем намерении получить доступ к устройству ввода/вывода, посылая ядру запрос на *открытие* соответствующего файла. Ядро возвращает небольшое неотрицательное целое число – *дескриптор*, – идентифицирующее файл во всех последующих операциях с ним. Ядро хранит всю информацию об открытом файле, а приложение просто работает с дескриптором.
- Каждый процесс, создаваемый командной оболочкой Linux, начинает существование, имея три открытых файла: *стандартный ввод* (дескриптор 0), *стандартный вывод* (дескриптор 1) и *стандартный вывод ошибок* (дескриптор 2). Заголовочный файл `<unistd.h>` определяет константы `STDIN_FILENO`, `STDOUT_FILENO` и `STDERR_FILENO`, которые можно использовать взамен явных значений дескрипторов.
- *Изменение текущей позиции в файле.* Ядро хранит *позицию в файле*  $k$ , изначально равную 0, для каждого открытого файла. Позиция в файле – это величина смещения в байтах от начала файла. Прикладная программа может явно изменить текущую позицию файла, выполнив операцию *позиционирования* (`seek`).
- *Чтение и запись в файлы.* Операция чтения копирует  $n > 0$  байт из файла в память, начиная с текущей позиции  $k$ , и затем увеличивает  $k$  на  $n$ . Если файл имеет размер



$m$  байт, то операция чтения, такая что  $k \geq m$ , столкнется с ограничением, известным как *конец файла* (end-of-file, EOF), которое определяется приложением. В конце файла нет явного «символа EOF».

Аналогично операция записи копирует  $n > 0$  байт из памяти в файл, начиная с текущей позиции в файле  $k$ , после чего обновляет  $k$ .

- *Заккрытие файлов.* После того как приложение заканчивает работать с файлом, оно посылает ядру запрос на *заккрытие* файла. В ответ ядро освобождает созданные при открытии файла структуры и возвращает дескриптор в пул доступных дескрипторов. Если по какой-то причине выполнение процесса прервется, то ядро закроет все открытые им файлы и освободит ресурсы памяти.

## 10.2. Файлы

Все файлы в Linux имеют *тип*, определяющий их роль в системе:

- *обычный файл* содержит произвольные данные. Прикладные программы часто различают *текстовые файлы*, которые являются обычными файлами, содержащими только символы ASCII или Unicode, и *двоичные файлы*, которые содержат любые другие данные. Ядро не различает текстовые и двоичные файлы.

Текстовый файл в Linux состоит из последовательности *текстовых строк*, в которой каждая строка – это последовательность символов, заканчивающаяся символом *новой строки* ('`\n`'). Символ новой строки аналогичен символу перевода строки в наборе ASCII (Line Feed, LF) и имеет числовое значение `0x0a`;

- *каталог* – это файл, содержащий массив ссылок, каждая из которых отображает *имя файла* в физический файл, который может быть другим каталогом. Каждый каталог содержит как минимум две записи: `.` (точка) – ссылку на сам каталог и `..` (точка-точка) – ссылку на *родительский каталог* в иерархии каталогов (подробнее об этом рассказывается далее в главе). Создать каталог можно с помощью команды `mkdir`, просмотреть его содержимое с помощью `ls` и удалить с помощью `rmdir`;
- *socket* – это файл, который используется для взаимодействий с другим процессом по сети (см. раздел 11.4).

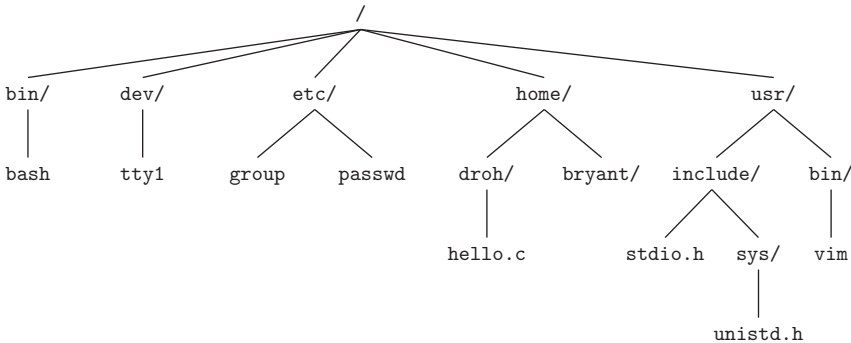
### Индикаторы конца строки (EOL)

Один из неудобных аспектов работы с текстовыми файлами – использование в разных системах разных символов для обозначения конца строки. В Linux и Mac OSX используется символ '`\n`' (`0x0a`) – ASCII-символ перевода строки (LF). Однако в MS Windows и интернет-протоколах, таких как HTTP, используется последовательность '`\r\n`' (`0xd 0xa`) – ASCII-символы возврата каретки (CR) и перевода строки (LF). Если создать файл `foo.txt` в Windows, а затем заглянуть в него в текстовом редакторе в Linux, то можно увидеть раздражающий символ `^M` в конце каждой строки, именно так инструменты Linux отображают символ CR. Удалить эти нежелательные символы CR из `foo.txt` можно простой командой:

```
linux> perl -pi -e "s/\r\n/g" foo.txt
```

В числе других типов файлов можно назвать *именованные каналы*, *символические ссылки*, а также *символьные* и *блочные устройства*, которые мы не будем обсуждать в этой книге.

Ядро Linux организует все файлы в единую *иерархию каталогов*, привязанную к *корневому каталогу* с именем / (косая черта, слеш). Каждый файл в системе является прямым или косвенным потомком корневого каталога. На рис. 10.1 показана часть иерархии каталогов в нашей системе Linux.



**Рис. 10.1.** Часть иерархии каталогов Linux. Косая черта в конце обозначает каталог

Частью контекста каждого процесса является *текущий рабочий каталог*, который идентифицирует текущее местоположение в иерархии каталогов. Изменить текущий рабочий каталог в командной оболочке можно с помощью команды `cd`.

Местоположение в иерархии каталогов определяется *путями*. Путь – это строка, включающая необязательный символ косой черты, за которым следует последовательность имен файлов, разделенных косой чертой. Пути бывают двух видов:

- *абсолютный путь* начинается с косой черты и обозначает путь от корневого каталога. Например, файл `hello.c` на рис. 10.1 имеет абсолютный путь `/home/droh/hello.c`;
- *относительный путь* начинается с имени файла и обозначает путь от текущего рабочего каталога. Например, если принять, что текущим рабочим каталогом в иерархии на рис. 10.1 является `/home/droh`, то файл `hello.c` будет иметь относительный путь `./hello.c`. С другой стороны, если принять, что текущим рабочим каталогом является `/home/bryant`, то файл `hello.c` будет иметь относительный путь `../home/droh/hello.c`.

## 10.3. Открытие и закрытие файлов

Процессы открывают существующие файлы или создают новые вызовом функции `open`:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(char *filename, int flags, mode_t mode);
```

Возвращает новый файловый дескриптор в случае успеха, `-1` в случае ошибки

Функция `open` преобразует имя файла `filename` в файловый дескриптор и возвращает его номер. Возвращаемый дескриптор – всегда наименьший из не открытых процессом до сих пор. Аргумент `flags` определяет, как процесс планирует обращаться к файлу:

- O\_RDONLY – только чтение;
- O\_WRONLY – только запись;
- O\_RDWR – чтение и запись.

Вот как, например, открывается существующий файл для чтения:

```
fd = Open("foo.txt", O_RDONLY, 0);
```

Аргумент flags также может объединять по ИЛИ (OR) дополнительные флаги, когда файл открывается для записи:

- O\_CREAT – если файл не существует, то будет создан новый пустой файл;
- O\_TRUNC – если файл существует, его размер будет усечен до нуля;
- O\_APPEND – перед выполнением каждой операции записи текущая позиция будет переноситься в конец файла.

Например, вот как можно открыть существующий файл для добавления новых данных в конец:

```
fd = Open("foo.txt", O_WRONLY|O_APPEND, 0);
```

Аргумент mode определяет биты разрешения для доступа к новым файлам. Символические имена этих битов перечислены в табл. 10.1.

**Таблица 10.1.** Биты разрешения для доступа к новым файлам. Определяются в sys/stat.h

Бит	Описание
S_IRUSR	Пользователь (владелец) может читать файл
S_IWUSR	Пользователь (владелец) может писать в файл
S_IXUSR	Пользователь (владелец) может выполнять файл
S_IRGRP	Члены группы-владельца могут читать файл
S_IWGRP	Члены группы-владельца могут писать в файл
S_IXGRP	Члены группы-владельца могут выполнять файл
S_IROTH	Все остальные могут читать файл
S_IWOTH	Все остальные могут писать в файл
S_IXOTH	Все остальные могут выполнять файл

Каждый процесс хранит в своем контексте маску umask, которую можно изменить вызовом функции umask. Когда процесс создает новый файл вызовом функции open с некоторым аргументом mode, то биты разрешения доступа к файлу определяются как значение выражения mode & ~umask. Предположим, например, что заданы следующие значения по умолчанию для mode и umask:

```
#define DEF_MODE S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define DEF_UMASK S_IWGRP|S_IWOTH
```

Тогда следующий фрагмент создаст новый файл, доступный владельцу для чтения и записи, а всем другим пользователям – только для чтения:

```
umask(DEF_UMASK);
fd = Open("foo.txt", O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);
```

Закончив работу с файлом, процесс может закрыть его вызовом функции `close`.

```
#include <unistd.h>

int close(int fd);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Попытка закрыть уже закрытый дескриптор вызовет ошибку.

### Упражнение 10.1 (решение в конце главы)

Что выведет следующая программа?

```
1 #include "csapp.h"
2
3 int main()
4 {
5     int fd1, fd2;
6
7     fd1 = Open("foo.txt", O_RDONLY, 0);
8     Close(fd1);
9     fd2 = Open("baz.txt", O_RDONLY, 0);
10    printf("fd2 = %d\n", fd2);
11    exit(0);
12 }
```

## 10.4. Чтение и запись файлов

Прикладные программы вводят и выводят данные вызовом функций `read` и `write` соответственно.

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buf, size_t n);
```

Возвращает число прочитанных байтов в случае успеха,  
0 – по достижении конца файла (EOF) или если прочитано 0 байт, -1 в случае ошибки

```
ssize_t write(int fd, const void *buf, size_t n);
```

Возвращает число записанных байтов в случае успеха,  
-1 в случае ошибки

Функция `read` копирует до `n` байт, начиная с текущей позиции в файле `fd`, в буфер памяти `buf`. Возвращаемое значение -1 сообщает об ошибке, а возвращаемое значение 0 – о достижении конца файла или если прочитано 0 байт. Во всех остальных случаях возвращается число фактически прочитанных байтов.

Функция `write` копирует до `n` байт из буфера памяти `buf` в файл `fd`, начиная с текущей позиции. В листинге 10.1 показана программа, использующая функции `read` и `write` для копирования данных со стандартного ввода в стандартный вывод, по одному байту за раз.

**Листинг 10.1.** Использование функций `read` и `write` для копирования данных со стандартного ввода в стандартный вывод

*code/io/cpstdin.c*

```
1 #include "csapp.h"
2
3 int main(void)
4 {
5     char c;
6
7     while(Read(STDIN_FILENO, &c, 1) != 0)
8         Write(STDOUT_FILENO, &c, 1);
9     exit(0);
10 }
```

*code/io/cpstdin.c*

В некоторых случаях функции `read` и `write` могут читать/записывать меньше байтов, чем определено аргументом `n`. Это не является ошибкой, и подобные ситуации объясняются следующими причинами:

- *достигнут конец файла (EOF).* Предположим, что у нас есть файл, содержащий всего 20 байт, и мы решили читать его содержимое порциями по 50 байт. Тогда вызов `read` вернет число 20, а в следующей попытке чтения сообщит о достижении конца файла, вернув 0;
- *чтение выполняется из терминала.* Если открытый файл является устройством терминала (т. е. клавиатурой и монитором), то каждый вызов `read` будет читать одну строку текста за раз и возвращать ее длину;
- *чтение выполняется из сетевого сокета.* Если открытый файл связан с сокетом (см. раздел 11.4), то из-за ограничений внутренней буферизации и продолжительных задержек в сети `read` и `write` могут прочитать или записать меньше байтов, чем запрошено. Также количество прочитанных/записанных байтов может оказаться меньше запрошенного при работе с каналами Unix – механизмом межпроцессных взаимодействий, которые в этой книге не рассматриваются.

На практике при чтении файлов с диска число прочитанных байтов может оказаться меньше только по достижении конца файла, а при записи в дисковые файлы такое вообще невозможно. Впрочем, когда преследуется цель создания устойчивых (надежных) сетевых приложений, таких как веб-серверы, желательно обрабатывать ситуации, когда читается/записывается меньше байтов, чем запрошено, путем многократного вызова функций `read` и `write`, пока не будут прочитаны/записаны все байты.

#### Чем отличаются `ssize_t` и `size_t`?

Возможно, вы заметили, что функция `read` принимает аргумент типа `size_t` и возвращает значение типа `ssize_t`. Чем отличаются эти два типа? Тип `size_t` определяется как `unsigned long`, а `ssize_t` (*signed size – размер со знаком*) – как `long`. Функция `read` возвращает размер со знаком, потому что в случае ошибки должно возвращаться значение `-1`. Интересно, что для поддержки возврата одного значения `-1` максимальный объем данных, который может прочитать `read`, уменьшается вдвое – с 4 до 2 Гбайт.

## 10.5. Надежные чтение и запись с помощью пакета RIO

В этом разделе мы разработаем пакет ввода/вывода с названием RIO (Robust I/O – надежный ввод/вывод), автоматически обрабатывающий случаи, когда читается/запи-

сывается меньше байтов, чем запрошено. Пакет RIO предлагает удобную, надежную и эффективную поддержку ввода/вывода для таких приложений, как сетевые программы. В состав RIO входят два разных типа функций:

- *функции небуферизованного ввода/вывода.* Эти функции передают данные непосредственно между памятью и файлом без буферизации на уровне приложения. Они особенно полезны для чтения и записи двоичных данных в сеть;
- *функции буферизованного ввода.* Эти функции позволяют эффективно читать текстовые строки и двоичные данные из файлов и кешировать их в буфере на уровне приложения, подобно стандартным функциям ввода/вывода, таким как `printf`. В отличие от процедур буферизованного ввода/вывода, представленных в [110], буферизованные функции ввода в пакете RIO могут использоваться в многопоточном окружении (см. раздел 12.7.1) и попеременно вызываться с одним и тем же дескриптором. Например, из дескриптора можно прочитать некоторые текстовые строки, затем некоторые двоичные данные, а потом снова текстовые строки.

Создание пакета процедур RIO преследует две цели. Во-первых, они будут использоваться при разработке сетевых приложений. Во-вторых, изучая исходный код этих процедур, вы сможете глубже понять общие принципы функционирования ввода/вывода в Unix.

### 10.5.1. Функции RIO небуферизованного ввода/вывода

Приложения могут передавать данные напрямую между памятью и файлом, вызывая функции `rio_readn` и `rio_writen`.

```
#include "csapp.h"
```

```
ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Возвращают число переданных байтов в случае успеха, 0 – по достижении конца файла (только `rio_readn`), -1 в случае ошибки

Функция `rio_readn` читает до `n` байт, начиная с текущей позиции в файле `fd`, в буфер `usrbuf`. Аналогично функция `rio_writen` записывает `n` байт из буфера `usrbuf` в файл `fd`. Функция `rio_readn` может вернуть меньшее количество байтов, чем запрошено, только если достигнут конец файла. Функция `rio_writen` никогда не возвращает счетчик меньше `n`. Вызовы `rio_readn` и `rio_writen` можно произвольно чередовать с одним и тем же дескриптором.

В листинге 10.2 приводится исходный код `rio_readn` и `rio_writen`. Обратите внимание, что обе функции повторно вызывают `read` или `write`, если она была прервана обработчиком сигналов. Для максимальной переносимости мы предусмотрели возможность прерывания системных вызовов и их повторный запуск (при необходимости).

#### Листинг 10.2. Функции `rio_readn` и `rio_writen`

```
1 ssize_t rio_readn(int fd, void *usrbuf, size_t n)
2 {
3     size_t nleft = n;
4     ssize_t nread;
```

*code/src/csapp.c*

```

5 char *bufp = usrbuf;
6
7 while (nleft > 0) {
8     if ((nread = read(fd, bufp, nleft)) < 0) {
9         if (errno == EINTR) /* Если вызов был прерван сигналом, */
10             nread = 0;      /* то перезапустить read() */
11         else
12             return -1;      /* errno установит системный вызов read() */
13     }
14     else if (nread == 0)
15         break; /* EOF */
16     nleft -= nread;
17     bufp += nread;
18 }
19 return (n - nleft); /* Вернуть n >= 0 */
20 }

```

*code/src/csapp.c*

### Происхождение пакета RIO

Вдохновением для создания пакета RIO послужили функции `getline`, `readn` и `written`, описанные У. Ричардом Стивенсом (W. Richard Stevens) в его классической работе по сетевому программированию [110]. Функции `rio_readn` и `rio_writen` идентичны функциям `readn` и `writen` Стивенса. Однако функция `getline` Стивенса имеет определенные ограничения, исправленные в RIO. Во-первых, по причине того, что `getline` – буферизованная функция, а `readn` – нет, их нельзя использовать вместе с одним и тем же дескриптором. Во-вторых, из-за использования буфера `static` функция `getline` Стивенса не является потокобезопасной, что потребовало от Стивенса написать другую потокобезопасную версию `getline_r`. Мы устранили эти два недостатка в функциях `rio_readlineb` и `rio_readnb`, которые являются взаимозаменяемыми и потокобезопасными.

*code/src/csapp.c*

```

1 ssize_t rio_writen(int fd, void *usrbuf, size_t n)
2 {
3     size_t nleft = n;
4     ssize_t nwritten;
5     char *bufp = usrbuf;
6
7     while (nleft > 0) {
8         if ((nwritten = write(fd, bufp, nleft)) <= 0) {
9             if (errno == EINTR) /* Если вызов был прерван сигналом, */
10                 nwritten = 0; /* то перезапустить write() */
11             else
12                 return -1; /* errno установит системный вызов write() */
13         }
14         nleft -= nwritten;
15         bufp += nwritten;
16     }
17     return n;
18 }

```

*code/src/csapp.c*

## 10.5.2. Функции RIO буферизованного ввода

Представьте, что вам понадобилось написать программу, подсчитывающую количество текстовых строк в файле. Как это сделать? Один из способов – использовать

функцию `read` для чтения байтов по одному и сравнивать каждый байт с символом новой строки. Недостаток такого подхода – низкая эффективность, потому что он требует генерировать прерывание (при обращении к системному вызову) в ядре для каждого байта.

Более совершенный подход – вызов функции-обертки `rio_readlineb`, которая копирует текстовую строку из внутреннего буфера чтения и автоматически вызывает `read` для заполнения буфера всякий раз, когда он опустошается. Для файлов, содержащих как текстовые строки, так и двоичные данные (например, HTTP-ответы, которые описываются в разделе 11.5.3), также предусмотрена буферизованная версия `rio_readn` с именем `rio_readnb`, которая копирует прочитанные байты из того же буфера чтения, что и `rio_readlineb`.

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

                                     Ничего не возвращает

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);

                                     Возвращают число прочитанных байтов в случае успеха,
                                     0 по достижении конца файла (EOF), -1 в случае ошибки
```

Функция `rio_readinitb` вызывается один раз с дескриптором файла после его открытия. Она связывает дескриптор `fd` с буфером чтения типа `rio_t`, на который указывает `rp`.

Функция `rio_readlineb` читает следующую текстовую строку из буфера `rp` (включая символ новой строки), копирует ее в буфер `usrbuf` и добавляет в конец строки нулевой символ. Функция `rio_readlineb` читает не более `maxlen-1` байт, оставляя место для завершающего нулевого символа. Текстовые строки, длина которых превышает `maxlen-1` байт, усекаются, и в конец добавляется нулевой символ.

Функция `rio_readnb` читает до `n` байт из буфера `rp` в буфер `usrbuf`. Вызовы `rio_readlineb` и `rio_readnb` можно произвольно чередовать с одним и тем же дескриптором. Однако вызовы буферизованных версий не должны чередоваться с вызовами небуферизованной функции `rio_readn`.

В оставшейся части книги мы часто будем использовать пакет RIO, соответственно, вы увидите множество примеров применения этих функций. В листинге 10.3 показан пример построчного копирования текстового файла из стандартного ввода в стандартный вывод с помощью функций RIO.

**Листинг 10.3.** Копирование текстового файла из стандартного ввода в стандартный вывод

*code/io/cpfile.c*

```
1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     int n;
6     rio_t rio;
7     char buf[MAXLINE];
8
9     Rio_readinitb(&rio, STDIN_FILENO);
10    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
```



```

11         Rio_writen(STDOUT_FILENO, buf, n);
12     }

```

*code/io/cpfile.c*

В листинге 10.4 показан формат буфера чтения и код инициализирующей его функции `rio_readinitb`. Функция `rio_readinitb` создает пустой буфер чтения и связывает дескриптор открытого файла с этим буфером.

**Листинг 10.4.** Буфер чтения типа `rio_t` и функция `rio_readinitb` для его инициализации

*code/src/csapp.c*

```

1 #define RIO_BUFSIZE 8192
2 typedef struct {
3     int rio_fd;          /* Дескриптор файла, ассоциированного с этим буфером */
4     int rio_cnt;         /* Осталось непрочитанных байтов в буфере */
5     char *rio_bufptr;    /* Следующий непрочитанный байт в буфере */
6     char rio_buf[RIO_BUFSIZE]; /* Внутренний буфер */
7 } rio_t;

```

*code/src/csapp.c*

```

1 void rio_readinitb(rio_t *rp, int fd)
2 {
3     rp->rio_fd = fd;
4     rp->rio_cnt = 0;
5     rp->rio_bufptr = rp->rio_buf;
6 }

```

*code/src/csapp.c*

Основой процедурой чтения в пакете RIO является функция `rio_read`, показанная в листинге 10.5. Функция `rio_read` является буферизованной версией системной функции `read`. Когда `rio_read` вызывается для чтения `n` байт, то в буфере чтения имеется `rp->rio_cnt` непрочитанных байтов. Если буфер пуст, то он заполняется вызовом `read`. Если системный вызов `read` вернул меньше байтов, чем было запрошено, то это не является ошибкой – буфер просто заполняется не полностью. Если буфер не пуст, то `rio_read` копирует `n` или `rp->rio_cnt` байт (наименьшее из этих двух чисел) из буфера чтения в пользовательский буфер и возвращает число скопированных байтов.

**Листинг 10.5.** Функция `rio_read`

*code/src/csapp.c*

```

1 static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
2 {
3     int cnt;
4
5     while (rp->rio_cnt <= 0) { /* Заполнить буфер, если он пуст */
6         rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
7             sizeof(rp->rio_buf));
8         if (rp->rio_cnt < 0) {
9             if (errno != EINTR) /* Если вызов не был прерван сигналом */
10                 return -1;
11         }
12     }
13 }

```

```

12     else if (rp->rio_cnt == 0) /* EOF */
13         return 0;
14     else
15         rp->rio_bufptr = rp->rio_buf; /* Переустановить указатель */
16 }
17
18 /* Скопировать min(n, rp->rio_cnt) байт из внутреннего буфера в usrbuf */
19 cnt = n;
20 if (rp->rio_cnt < n)
21     cnt = rp->rio_cnt;
22 memcpy(usrbuf, rp->rio_bufptr, cnt);
23 rp->rio_bufptr += cnt;
24 rp->rio_cnt -= cnt;
25 return cnt;
26 }

```

*code/src/csapp.c*

Для прикладной программы функция `rio_read` имеет ту же семантику, что и функция `read`. В случае ошибки она возвращает `-1` и устанавливает код ошибки в `errno`. По достижении конца файла функция возвращает `0`. Она возвращает меньшее число байтов, если запрошенное количество больше количества байтов в буфере чтения. Сходство семантики этих двух функций упрощает создание разного рода буферизованных функций чтения путем замены `read` на `rio_read`. Например, функция `rio_readnb` в листинге 10.6 имеет ту же структуру, что и `rio_readn`, только в ней вместо `read` вызывается `rio_read`. Аналогично функция `rio_readlineb` в том же листинге 10.6 вызывает `rio_read` максимум `maxlen-1` раз. Каждый вызов возвращает один байт из буфера чтения, который затем сравнивается с символом новой строки.

#### Листинг 10.6. Функции `rio_readlineb` и `rio_readnb`

*code/src/csapp.c*

```

1 ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
2 {
3     int n, rc;
4     char c, *bufp = usrbuf;
5
6     for (n = 1; n < maxlen; n++) {
7         if ((rc = rio_read(rp, &c, 1)) == 1) {
8             *bufp++ = c;
9             if (c == '\n') {
10                 n++;
11                 break;
12             }
13         } else if (rc == 0) {
14             if (n == 1)
15                 return 0; /* Конец файла, больше нет данных для чтения */
16             else
17                 break;    /* Конец файла, какие-то данные были прочитаны */
18         } else
19             return -1;    /* Ошибка */
20     }
21     *bufp = 0;
22     return n-1;
23 }

```

*code/src/csapp.c*

*code/src/csapp.c*

```

1 ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
2 {
3     size_t nleft = n;
4     ssize_t nread;
5     char *bufp = usrbuf;
6
7     while (nleft > 0) {
8         if ((nread = rio_read(rp, bufp, nleft)) < 0)
9             return -1; /* errno устанавливается системным вызовом read() */
10        else if (nread == 0)
11            break; /* EOF */
12        nleft -= nread;
13        bufp += nread;
14    }
15    return (n - nleft); /* Вернуть значение >= 0 */
16 }

```

*code/src/csapp.c*

## 10.6. Чтение метаданных файла

Прикладные программы могут извлекать информацию о файле (иногда ее называют *метаданными файла*) вызовом функций `stat` и `fstat`.

```

#include <unistd.h>
#include <sys/stat.h>

```

```

int stat(const char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);

```

Возвращают 0 в случае успеха, -1 в случае ошибки

Функция `stat` принимает имя файла и заполняет структуру `stat`, которая показана в листинге 10.7. Функция `fstat` действует аналогично, но вместо имени файла принимает дескриптор. При обсуждении веб-серверов в разделе 11.5 мы будем ссылаться на компоненты `st_mode` и `st_size` структуры `stat`. Другие компоненты рассматриваться не будут.

### Листинг 10.7. Структура `stat`

*statbuf.h (подключается в заголовочном файле sys/stat.h)*

```

/* Метаданные, возвращаемые функциями stat и fstat */
struct stat {
    dev_t      st_dev;      /* Устройство */
    ino_t      st_ino;      /* Индексный узел inode */
    mode_t     st_mode;     /* Разрешения и тип файла */
    nlink_t    st_nlink;    /* Количество жестких ссылок на файл */
    uid_t      st_uid;      /* ID пользователя-владельца */
    gid_t      st_gid;      /* ID группы-владельца */
    dev_t      st_rdev;     /* Тип устройства (если это файл устройства) */
    off_t      st_size;     /* Общий размер в байтах */
    unsigned long st_blksize; /* Размер блока для ввода/вывода */
    unsigned long st_blocks; /* Количество выделенных блоков */
    time_t     st_atime;     /* Время последнего доступа */
    time_t     st_mtime;     /* Время последнего изменения содержимого файла */

```

```
time_t      st_ctime;   /* Время последнего изменения состояния файла */
};
```

*statbuf.h (подключается в заголовочном файле sys/stat.h)*

Компонент `st_size` содержит размер файла в байтах. Компонент `st_mode` содержит биты разрешений на доступ к файлу (табл. 10.1) и тип файла (раздел 10.2). В Linux определяется несколько макросов предикатов (в `sys/stat.h`), с помощью которых можно проверить тип файла по значению в поле `st_mode`:

- `S_ISREG(m)` – обычный файл;
- `S_ISDIR(m)` – каталог;
- `S_ISSOCK(m)` – сетевой сокет.

В листинге 10.8 показано, как можно использовать эти макросы и функцию `stat` для получения и интерпретации битов в поле `st_mode`.

#### Листинг 10.8. Получение и интерпретация битов в поле `st_mode`

*code/io/statcheck.c*

```
1 #include "csapp.h"
2
3 int main (int argc, char **argv)
4 {
5     struct stat stat;
6     char *type, *readok;
7
8     Stat(argv[1], &stat);
9     if (S_ISREG(stat.st_mode)) /* Определить тип файла */
10         type = "regular";
11     else if (S_ISDIR(stat.st_mode))
12         type = "directory";
13     else
14         type = "other";
15     if ((stat.st_mode & S_IRUSR)) /* Проверить доступность для чтения */
16         readok = "yes";
17     else
18         readok = "no";
19
20     printf("type: %s, read: %s\n", type, readok);
21     exit(0);
22 }
```

*code/io/statcheck.c*

## 10.7. Чтение содержимого каталога

Приложения могут читать содержимое каталогов с помощью семейства функций `readdir`.

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR *opendir(const char *name);
```

Возвращает указатель на дескриптор в случае успеха,  
NULL в случае ошибки

Функция `opendir` принимает путь к каталогу и возвращает указатель на *поток каталога*. Поток – это абстракция упорядоченного списка элементов, в данном случае – элементов каталога.

```
#include <dirent.h>
```

```
struct dirent *readdir(DIR *dirp);
```

Возвращает указатель на следующий элемент каталога в случае успеха, NULL в случае ошибки или если список элементов исчерпан

Каждый вызов `readdir` возвращает указатель на следующий элемент каталога в потоке `dirp` или NULL, если список элементов прочитан полностью. Каждый элемент каталога – это структура следующего вида:

```
struct dirent {
    ino_t d_ino;        /* Номер индексного узла inode */
    char d_name[256];   /* Имя файла */
};
```

В некоторых версиях Linux эта структура может содержать другие поля, но эти два поля являются стандартными для всех систем. Поле `d_name` – это имя файла, а `d_ino` – индексный узел файла.

В случае ошибки `readdir` возвращает NULL и устанавливает переменную `errno`. К сожалению, единственный способ отличить ошибку от условия конца потока – проверить значение `errno` после вызова `readdir`.

```
#include <dirent.h>
```

```
int closedir(DIR *dirp);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Функция `closedir` закрывает поток и освобождает занятые ресурсы. В листинге 10.9 показано, как с помощью `readdir` прочитать содержимое каталога.

#### Листинг 10.9. Чтение содержимого каталога

*code/io/readdir.c*

```
1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     DIR *stream;
6     struct dirent *dep;
7
8     stream = Opendir(argv[1]);
9
10    errno = 0;
11    while ((dep = readdir(stream)) != NULL) {
12        printf("Found file: %s\n", dep->d_name);
13    }
14    if (errno != 0)
15        unix_error("readdir error");
16
```

```
17     Closedir(streamp);
18     exit(0);
19 }
```

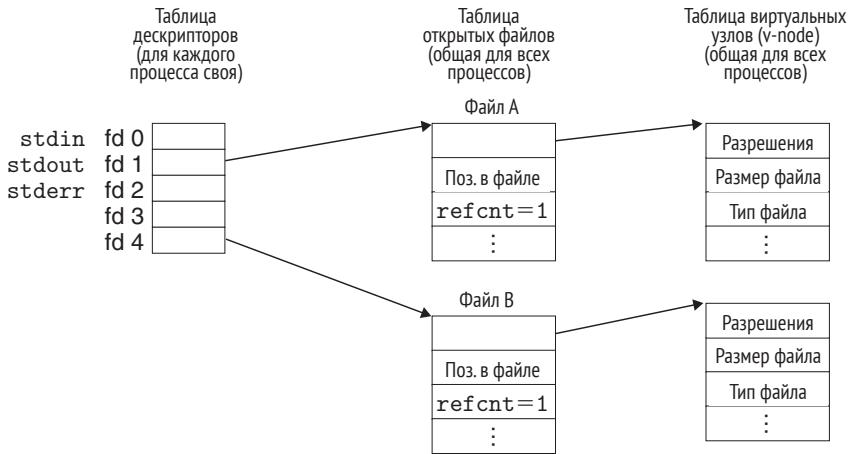
*code/io/readdir.c*

## 10.8. Совместное использование файлов

В Linux файлы можно использовать совместно разными способами. В отсутствие четкого представления о том, как ядро представляет открытые файлы, концепция совместного использования может показаться довольно запутанной. Ядро представляет открытые файлы с помощью трех взаимосвязанных структур данных:

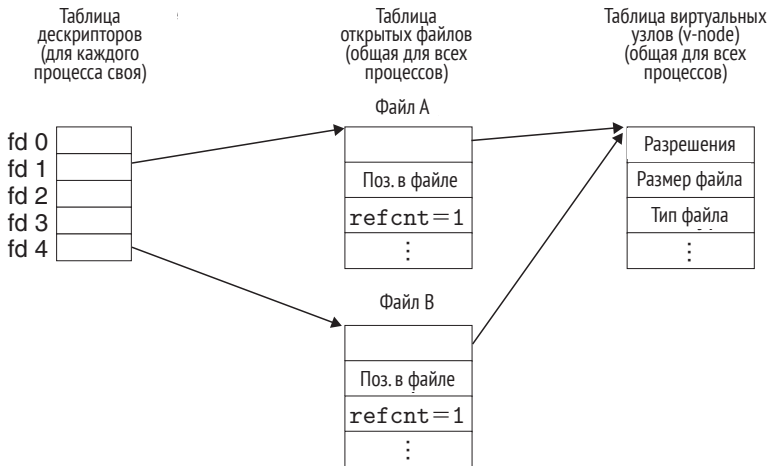
- *таблица дескрипторов*. Каждый процесс имеет свою *таблицу дескрипторов*, элементы в которой индексируются дескрипторами файлов, открытых процессом. Для каждого дескриптора открытого файла имеется свой элемент в *таблице открытых файлов*;
- *таблица открытых файлов*. Набор открытых файлов представлен таблицей открытых файлов, используемой совместно всеми процессами. Каждый элемент в таблице файлов включает (помимо всего прочего) текущую позицию в файле, *счетчик ссылок* – количество дескрипторов, ссылающихся в данный момент на этот файл, и указатель на элемент в *таблице виртуальных узлов (v-node)*. Операция закрытия дескриптора уменьшает счетчик ссылок в соответствующем элементе таблицы файлов. Ядро не удаляет эти элементы, пока счетчик ссылок не достигнет нуля;
- *таблица виртуальных узлов (v-node)*. Подобно таблице файлов, таблица виртуальных узлов совместно используется всеми процессами. В каждом элементе этой таблицы содержится большая часть информации для структуры *stat*, включая компоненты *st\_mode* и *st\_size*.

На рис. 10.2 показан пример структур данных в ядре для случая, когда дескрипторы 1 и 4 ссылаются на два разных файла через элементы таблицы открытых файлов. Это типичная ситуация, когда файлы не используются совместно и каждый дескриптор соответствует своему отдельному файлу.



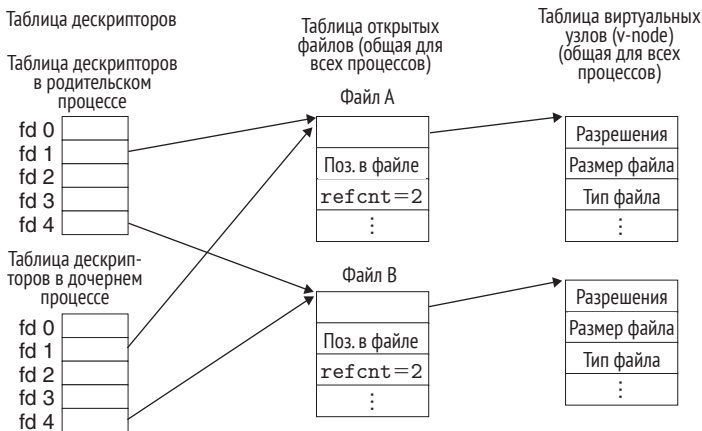
**Рис. 10.2.** Типичное состояние структур данных в ядре, соответствующих открытым файлам. В этом примере два дескриптора ссылаются на разные файлы

Несколько дескрипторов могут также ссылаться на один и тот же файл через различные элементы таблицы открытых файлов, как показано на рис. 10.3. Такое может случиться, например, если дважды вызвать функцию `open` с одним и тем же именем файла. Ключевая идея заключается в том, что каждый дескриптор имеет свою позицию в файле, поэтому операции чтения с разными дескрипторами могут получать данные из разных областей в файле.



**Рис. 10.3.** Совместное использование файлов. В этом примере два дескриптора ссылаются на один и тот же дисковый файл через таблицу файлов

Теперь нужно понять, как родительский и дочерний процессы совместно используют файлы. Предположим, что перед вызовом `fork` родительский процесс открыл файлы, показанные на рис. 10.2. Соответственно, после вызова `fork` сложится ситуация, показанная на рис. 10.4.



**Рис. 10.4.** Как дочерний процесс наследует открытые дескрипторы от родителя. Начальное состояние показано на рис. 10.2

Дочерний процесс получит свою копию таблицы дескрипторов от родительского процесса. В результате оба процесса вместе используют один и тот же набор таблиц открытых файлов и, соответственно, одни и те же текущие позиции в файлах. Важным

следствием такого поведения является необходимость закрывать дескрипторы в обоих процессах, родительском и дочернем, чтобы ядро могло удалить соответствующий элемент из таблицы открытых файлов.

### Упражнение 10.2 (решение в конце главы)

Предположим, что дисковый файл `foobar.txt` содержит 6 символов ASCII: `foobar`. Что выведет следующая программа?

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int fd1, fd2;
6     char c;
7
8     fd1 = Open("foobar.txt", O_RDONLY, 0);
9     fd2 = Open("foobar.txt", O_RDONLY, 0);
10    Read(fd1, &c, 1);
11    Read(fd2, &c, 1);
12    printf("c = %c\n", c);
13    exit(0);
14 }
```

### Упражнение 10.3 (решение в конце главы)

Предположим, что дисковый файл `foobar.txt` содержит 6 символов ASCII: `foobar`. Что выведет следующая программа?

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int fd;
6     char c;
7
8     fd = Open("foobar.txt", O_RDONLY, 0);
9     if (Fork() == 0) {
10         Read(fd, &c, 1);
11         exit(0);
12     }
13     Wait(NULL);
14     Read(fd, &c, 1);
15     printf("c = %c\n", c);
16     exit(0);
17 }
```

### Левая и правая угловые скобки

Во избежание путаницы с другими операторами-скобками, например `]` и `[`, оператор `>` мы будем называть «правой угловой скобкой», а оператор `<` – «левой угловой скобкой».



## 10.9. Переадресация ввода/вывода

Командные оболочки Linux поддерживают операторы переадресации (перенаправления) ввода/вывода, позволяющие пользователям связывать стандартный ввод и стандартный вывод с дисковыми файлами. Например, команда

```
linux> ls > foo.txt
```

заставит оболочку запустить программу `ls` и перенаправить стандартный вывод в дисковый файл `foo.txt`. Как будет показано в разделе 11.5, веб-сервер выполняет аналогичный тип переадресации при выполнении CGI-программ от имени клиента. Так как же работает переадресация ввода/вывода? Один из способов – вызвать функцию `dup2`.

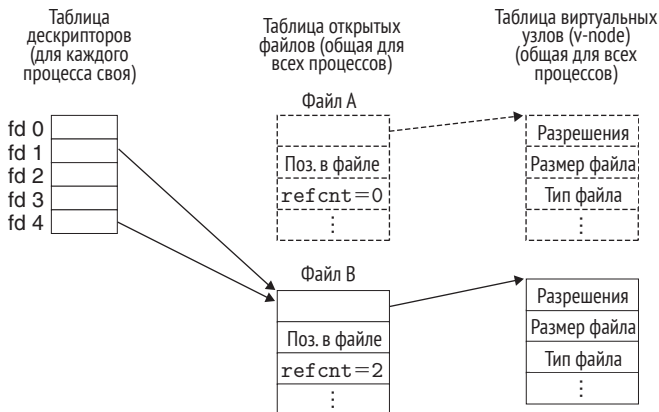
```
#include <unistd.h>
```

```
int dup2(int oldfd, int newfd);
```

Возвращает неотрицательный дескриптор в случае успеха, -1 в случае ошибки

Функция `dup2` копирует элемент `oldfd` в таблице дескрипторов в элемент `newfd`, затирая прежнее содержимое элемента `newfd`. Если дескриптор `newfd` уже открыт, то `dup2` закрывает его до копирования `oldfd`.

Предположим, что перед вызовом `dup2(4, 1)` имела место ситуация, изображенная на рис. 10.2, где дескриптор 1 (стандартный вывод) соответствует файлу *A* (пусть это будет терминал), а дескриптор 4 – файлу *B* (пусть это будет дисковый файл). Количество ссылок на *A* и *B* равно 1. На рис. 10.5 показана ситуация после вызова `dup2(4, 1)`. Теперь оба дескриптора ссылаются на файл *B*, а файл *A* был закрыт, и соответствующие элементы в таблицах открытых файлов и виртуальных узлов были удалены; счетчик ссылок на *B* увеличился на единицу. С этого момента все данные, записанные в стандартный вывод, будут направляться в файл *B*.



**Рис. 10.5.** Структуры данных в ядре после переадресации стандартного вывода вызовом `dup2(4, 1)`. Начальное состояние показано на рис. 10.2

### Упражнение 10.5 (решение в конце главы)

Предположим, что дисковый файл `foobar.txt` содержит 6 символов ASCII: `foobar`. Что выведет следующая программа?

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int fd1, fd2;
6     char c;
7
8     fd1 = Open("foobar.txt", O_RDONLY, 0);
9     fd2 = Open("foobar.txt", O_RDONLY, 0);
10    Read(fd2, &c, 1);
11    Dup2(fd2, fd1);
12    Read(fd1, &c, 1);
13    printf("c = %c\n", c);
14    exit(0);
15 }

```

## 10.10. Стандартный ввод/вывод

Стандарт языка C определяет набор высокоуровневых функций ввода/вывода, который называется *стандартной библиотекой ввода/вывода*. Эта библиотека предоставляет программистам высокоуровневые альтернативы функциям ввода/вывода Unix. Библиотека libc включает функции открытия и закрытия файлов `fopen` и `fclose`, чтения и записи байтов `fread` и `fwrite`, чтения и записи строк `fgets` и `fputs`, а также функции форматированного ввода/вывода `scanf` и `printf`.

Стандартная библиотека ввода/вывода моделирует открытый файл в виде *потока*. С точки зрения программиста поток – это указатель на структуру типа `FILE`. Каждая программа ANSI C первоначально имеет три открытых потока: `stdin`, `stdout` и `stderr`, соответствующих стандартному вводу, стандартному выводу и стандартному выводу ошибок соответственно:

```

#include <stdio.h>
extern FILE *stdin; /* Стандартный ввод (дескриптор 0) */
extern FILE *stdout; /* Стандартный вывод (дескриптор 1) */
extern FILE *stderr; /* Стандартный вывод ошибок (дескриптор 2) */

```

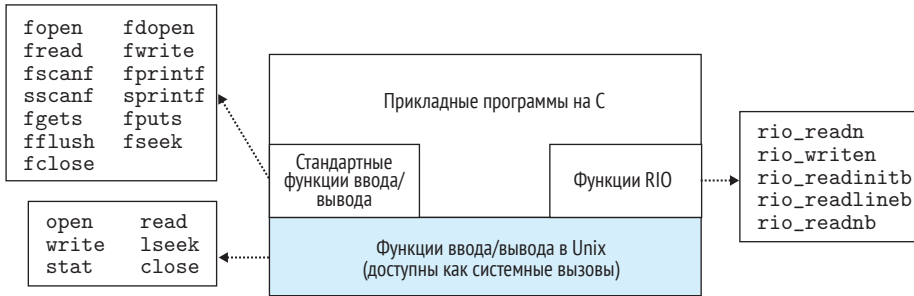
Поток типа `FILE` – это абстракция дескриптора файла и *буферизованного потока*. Буферизованный поток имеет то же назначение, что и буфер чтения в пакете RIO: минимизация количества дорогостоящих обращений к системным вызовам ввода/вывода Linux. Например, предположим, что имеется программа, многократно вызывающая стандартную функцию ввода `getc`, которая возвращает следующий символ из файла. Первый вызов `getc` заполнит буфер потока однократным обращением к функции `read`, после чего будет извлекать очередные байты из буфера, пока в буфере остаются непрочитанные байты.

## 10.11. Все вместе: какие функции ввода/вывода использовать?

На рис. 10.6 показана общая схема различных пакетов ввода/вывода, представленных в этой главе.

Ввод/вывод в Unix реализован в ядре операционной системы. Эти функции доступны приложениям через такие системные вызовы, как `open`, `close`, `lseek`, `read` и `write`. Высокоуровневые функции в пакете RIO и стандартные функции ввода/вывода реализованы «поверх» функций ввода/вывода Unix. Функции в пакете RIO – это надежные обертки

вокруг `read` и `write`, разработанные специально для этой книги. Они автоматически обрабатывают ситуации неполного выполнения запросов и обеспечивают эффективную буферизацию при чтении текстовых строк. Стандартные функции ввода/вывода обеспечивают буферизующие альтернативы функциям ввода/вывода Unix, а также процедуры форматированного ввода/вывода.



**Рис. 10.6.** Взаимосвязи между пакетами ввода/вывода в Unix, в стандартной библиотеке C и RIO

Так какие из этих функций предпочтительнее использовать при разработке программ? Вот несколько основных правил.

**Правило 1:** по возможности используйте стандартные функции ввода/вывода. Стандартные функции ввода/вывода предпочтительнее для ввода/вывода при работе с дисковыми устройствами и терминалами. Большинство программистов на C используют исключительно стандартные функции ввода/вывода, прибегая к низкоуровневым функциям ввода/вывода Unix лишь в крайних случаях (примером может служить функция `stat`, не имеющая аналога в стандартной библиотеке ввода/вывода). Мы советуем поступать так же, если есть такая возможность.

**Правило 2:** не используйте `scanf` или `rio_readlineb` для чтения двоичных файлов. Такие функции, как `scanf` и `rio_readlineb`, разрабатывались специально для чтения текстовых файлов. Многие студенты совершают распространенную ошибку: используют эти функции для чтения двоичных данных, что приводит к странным и непредсказуемым сбоям в их программах. Например, двоичные файлы могут быть замусорены множеством байтов `0xa`, которые не имеют ничего общего с окончаниями текстовых строк.

**Правило 3:** используйте функции RIO для ввода/вывода через сетевые сокеты. К сожалению, стандартная библиотека ввода/вывода страдает некоторыми неприятными проблемами при их использовании для сетевого ввода/вывода. Как будет показано в разделе 11.4, сети в Linux представлены особым типом файлов – сокетами. Операции ввода/вывода с сокетами, как и с любыми другими файлами в Linux, осуществляются посредством файловых дескрипторов, в данном случае дескрипторов сокетов. Прикладные процессы взаимодействуют с процессами на других компьютерах, выполняя операции чтения и записи с дескрипторами сокетов.

Стандартные потоки ввода/вывода являются *полнодуплексными* в том смысле, что программы могут читать и писать в один и тот же поток. Однако потоки имеют некоторые ограничения, плохо согласующиеся с ограничениями сокетов.

**Ограничение 1.** Вызов функций ввода вслед за функциями вывода. Функция ввода не может следовать за функцией вывода без промежуточного вызова `flush`, `fseek`,

`fsetpos` или `rewind`. Функция `fflush` опустошает буфер, связанный с потоком. Три последние функции используют функцию `lseek` для переустановки текущей позиции в файле.

**Ограничение 2.** *Вызов функций вывода вслед за функциями ввода.* Функция вывода не может следовать за функцией ввода без промежуточного вызова `fseek`, `fsetpos` или `rewind`, за исключением случаев, когда функция ввода достигает конца файла.

Эти ограничения вызывают проблемы в сетевых приложениях, потому что использовать функцию `lseek` с сокетом запрещено. Первое ограничение на потоковый ввод/вывод можно «обойти», взяв в привычку выталкивать буфер перед каждой операцией ввода. Однако второе ограничение можно обойти, только открыв два потока с тем же самым открытым дескриптором сокета: один для чтения, а другой для записи:

```
FILE *fpin, *fpout;

fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

Однако этот подход тоже имеет проблему – он требует вызвать `fclose` для обоих потоков, чтобы освободить ресурсы памяти, связанные с каждым потоком:

```
fclose(fpin);
fclose(fpout);
```

Каждая из этих операций пытается закрыть один и тот же базовый дескриптор сокета, поэтому вторая операция `close` терпит неудачу. Для однопоточных программ это не проблема, однако закрытие уже закрытого дескриптора в многопоточной программе – верный путь к аварийному завершению (см. раздел 12.7.4).

Учитывая все вышесказанное, мы рекомендуем не использовать стандартные функции ввода/вывода для операций с сетевыми сокетами – используйте функции RIO. Если необходим форматированный вывод, используйте функцию `sprintf` для форматирования строки в памяти, после чего отправьте ее в сокет вызовом `rio_writen`. Если необходим форматированный ввод, используйте `rio_readlineb` для чтения текстовой строки целиком, после чего применяйте `sscanf` для извлечения различных полей из этой строки.

## 10.12. Итоги

Linux предоставляет небольшое количество системных функций, позволяющих открывать, закрывать, читать и записывать файлы, получать метаданные файлов и выполнять переадресацию ввода/вывода. Операции чтения и записи в Linux могут читать/записывать меньше байтов, чем запрошено, и прикладные программы должны учитывать эту особенность и правильно ее обрабатывать. Вместо непосредственного вызова функций ввода/вывода Unix в прикладных программах следует использовать пакет RIO, автоматически обрабатывающий эти ситуации путем многократного выполнения операций чтения и записи до полного переноса всех запрошенных данных.

Ядро Linux использует три взаимосвязанные структуры данных для представления открытых файлов. Элементы в таблице дескрипторов ссылаются на элементы в таблице открытых файлов, которые, в свою очередь, ссылаются на элементы в таблице виртуальных узлов (*v-node*). Каждый процесс имеет свою таблицу дескрипторов, но все процессы совместно используют одну таблицу открытых файлов и таблицу виртуальных узлов. Понимание общей организации этих структур способствует пониманию принципов совместного использования файлов и переадресации ввода/вывода.

Стандартная библиотека ввода/вывода реализована «поверх» функций ввода/вывода Unix и содержит мощный набор высокоуровневых процедур ввода/вывода. Для большинства прикладных программ стандартная библиотека ввода/вывода является более простой и предпочтительной альтернативой функциям ввода/вывода Unix. Впрочем, из-за определенных взаимно несовместимых ограничений стандартной библиотеки в сетевых приложениях следует использовать функции ввода/вывода Unix.

## Библиографические заметки

Керриск (Kerrisk) дает всестороннее описание ввода/вывода Unix и файловой системы Linux [62]. Стивенс (Stevens) подробно описал систему ввода/вывода Unix [111]. Керниган (Kernighan) и Ритчи (Ritchie) представили четкое и исчерпывающее обсуждение функций из стандартной библиотеки ввода/вывода [61].

## Домашние задания

### Упражнение 10.6. ♦

Что выведет следующая программа?

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int fd1, fd2;
6
7     fd1 = Open("foo.txt", O_RDONLY, 0);
8     fd2 = Open("bar.txt", O_RDONLY, 0);
9     Close(fd2);
10    fd2 = Open("baz.txt", O_RDONLY, 0);
11    printf("fd2 = %d\n", fd2);
12    exit(0);
13 }
```

### Упражнение 10.7 ♦

Измените программу `cpfile` в листинге 10.3 так, чтобы в ней использовались функции R/O для копирования данных из стандартного ввода в стандартный вывод порциями по `MAXBUF` байт.

### Упражнение 10.8 ♦♦

Напишите программу `fstatcheck` – версию программы `statcheck` из листинга 10.8, которая принимает из командной строки номер дескриптора, а не имя файла.

### Упражнение 10.9 ♦♦

Взгляните на следующий вызов программы `fstatcheck` из упражнения 10.8:

```
linux> fstatcheck 3 < foo.txt
```

Можно было бы предположить, что программа `fstatcheck` получит и отобразит метаданные для файла `foo.txt`. Однако в действительности программа завершается с сообщением об ошибке «неверный файловый дескриптор». Учитывая эту особенность, напишите псевдокод, отражающий операции, выполняемые командной оболочкой между вызовами `fork` и `execve`:

```

if (Fork() == 0) { /* потомок */
    /* Какие действия выполняет командная оболочка здесь? */
}
```

```

    Execve("fstatcheck", argv, envp);
}

```

### Упражнение 10.10 ♦♦

Измените программу `cpfile` в листинге 10.3 так, чтобы она принимала необязательный аргумент `infile` командной строки. Если аргумент `infile` задан, то в стандартный вывод должно копироваться содержимое этого файла, иначе копирование должно выполняться, как и прежде, из стандартного ввода в стандартный вывод. «Трюк» заключается в использовании в обоих случаях оригинального цикла копирования (строки 9–11). Вы можете только вставлять дополнительный код, но не изменять существующий.

## Решения упражнений

### Решение упражнения 10.1

Процессы Unix сразу после запуска имеют три открытых дескриптора, привязанных к `stdin` (дескриптор 0), `stdout` (дескриптор 1) и `stderr` (дескриптор 2). Функция `open` всегда возвращает наименьший из неоткрытых дескрипторов, поэтому первый вызов `open` вернет дескриптор 3. Вызов функции `close` освободит дескриптор 3. Последний вызов `open` вновь вернет дескриптор 3, следовательно, программа выведет `fd2 = 3`.

### Решение упражнения 10.2

Дескрипторы `fd1` и `fd2` имеют свои элементы в таблице открытых файлов, поэтому каждому дескриптору соответствует своя позиция в файле `foobar.txt`. Таким образом, операция чтения из `fd2` вернет первый байт из `foobar.txt`, и программа выведет:

```

c = f,
a не
c = o,

```

как можно было бы подумать.

### Решение упражнения 10.3

Как вы наверняка помните, дочерний процесс наследует таблицу дескрипторов от родителя и все процессы совместно используют одну и ту же таблицу открытых файлов. Поэтому дескрипторы `fd` в родительском и дочернем процессах будут ссылаться на один и тот же элемент в таблице открытых файлов. Когда дочерний процесс прочитает первый байт из файла, текущая позиция в файле увеличивается на единицу, соответственно, родительский процесс прочитает второй байт и выведет:

```

c = o.

```

### Решение упражнения 10.4

Для переадресации стандартного вывода (дескриптор 0) в дескриптор 5 необходимо вызвать `dup2(5, 0)`, или, что эквивалентно, `dup2(5, STDIN_FILENO)`.

### Решение упражнения 10.5

На первый взгляд кажется, что программа должна вывести

```

c = f,

```

но, так как произведена переадресация `fd1` в `fd2`, в действительности программа выведет

```

c = o.

```

# Глава 11

## Сетевое программирование

- 11.1. Программная модель клиент–сервер.
- 11.2. Компьютерные сети.
- 11.3. Всемирная сеть интернет.
- 11.4. Интерфейс сокетов.
- 11.5. Веб-серверы.
- 11.6. Все вместе: разработка небольшого веб-сервера TINY.
- 11.7. Итоги.

Библиографические заметки.

Домашние задания.

Решения упражнений.

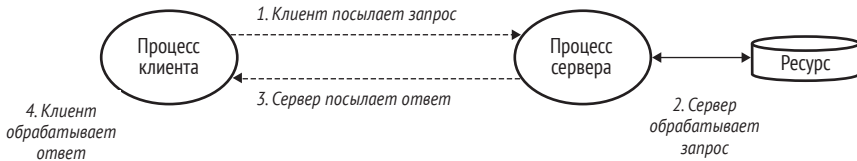
Сетевые приложения окружают нас повсюду. Каждый раз, когда вы просматриваете веб-страницы, посылаете сообщения по электронной почте или играете в онлайн-игру, вы используете сетевое приложение. Интересно отметить, что в основе всех сетевых приложений лежит одна и та же базовая программная модель. Такие приложения имеют общую логическую структуру и используют один и тот же программный интерфейс.

В основе сетевых приложений лежат многие из идей и понятий, которые мы изучали в предыдущих главах. Например, процессы, сигналы, порядок следования байтов, отображение в память и динамическое распределение памяти. Теперь вам придется освоить новые понятия и получить представление о базовой модели программирования клиент–сервер, которая широко используется службами в интернете. После этого мы воспользуемся всеми этими идеями и разработаем небольшой, но вполне пригодный для практического использования веб-сервер, способный обслуживать и статический, и динамический контент с текстом и графикой для реальных веб-браузеров.

### 11.1. Программная модель клиент–сервер

В основу каждого сетевого приложения положена модель *клиент–сервер*. Приложение, построенное на базе этой модели, состоит из процесса *сервера* и процесса *клиента*. Сервер управляет некоторым ресурсом и предоставляет некоторую услугу клиентам, манипулируя этим ресурсом. Например, веб-сервер управляет множеством файлов на дисках и выполняет их по запросам клиентов. Сервер FTP управляет множеством файлов на дисках и предоставляет их содержимое клиентам. Аналогично сервер электронной почты управляет файлом со списком писем и обновляет его по запросам клиентов.

Основной операцией в модели клиент–сервер является *транзакция* (рис. 11.1). Транзакция в модели клиент–сервер предусматривает выполнение четырех шагов.



**Рис. 11.1.** Транзакция в модели клиент–сервер

Когда клиент нуждается в какой-то услуге, он инициирует транзакцию, посылая *запрос*. Например, когда веб-браузеру нужен тот или иной файл, он посылает запрос веб-серверу.

1. Сервер получает запрос, интерпретирует его и выполняет соответствующие манипуляции со своим ресурсом. Например, когда веб-сервер получает запрос от браузера, он читает соответствующий дисковый файл.
2. Сервер *отвечает* клиенту, после чего ждет следующий запрос. Например, веб-сервер посылает файл обратно клиенту.
3. Клиент получает ответ и использует его по своему усмотрению. Например, веб-браузер, получив страницу от сервера, отображает ее на экране.

#### Транзакции в модели клиент–сервер и в базах данных

Транзакции в модели клиент–сервер *отличаются* от транзакций в базах данных и не обладают свойствами, характерными для транзакций баз данных, такими как атомарность. В нашем случае под транзакциями мы будем понимать простую последовательность действий, выполняемых клиентом и сервером.

*Важно понимать, что клиенты и серверы – это процессы, а не машины или хосты*, как их еще часто называют в этом контексте. На одном и том же хосте одновременно может выполняться множество различных клиентов и серверов, а клиент–серверные транзакции могут охватывать несколько разных хостов. Модель клиент–сервер остается неизменной, независимо от особенностей размещения клиентов и серверов на хостах.

## 11.2. Компьютерные сети

Клиенты и серверы часто выполняются на разных хостах и обмениваются данными, используя программные и аппаратные ресурсы *компьютерных сетей*. Сети – это сложные системы, и мы надеемся, что нам удастся помочь вам понять, как они работают. Наша цель – помочь вам сформировать правдоподобную мысленную модель сети с точки зрения программиста.

Для хоста сеть – это еще одно устройство ввода/вывода, которое выступает в роли источника и приемника данных, как показано на рис. 11.2.

Адаптер, подключенный через слот расширения к шине ввода/вывода, обеспечивает физический интерфейс с сетью. Данные, полученные из сети через шины ввода/вывода и памяти, копируются в память, обычно методом прямого доступа к памяти (Direct Memory Access, DMA). Аналогичным образом данные могут копироваться из памяти в сеть.

Физически сеть организована в виде иерархической системы по географическому принципу. Нижний уровень – это локальная сеть (Local Area Network, LAN), охватывающая здание или студенческий лагерь. В настоящее время наиболее популярной из ло-



кальных сетей является сеть *Ethernet*, которая была разработана в середине 1970-х годов компанией Хероx PARC. Сеть Ethernet зарекомендовала себя как исключительно гибкая и живучая локальная сеть, скорость передачи данных которой возросла с 3 Мбит/с до 100 Гбит/с.

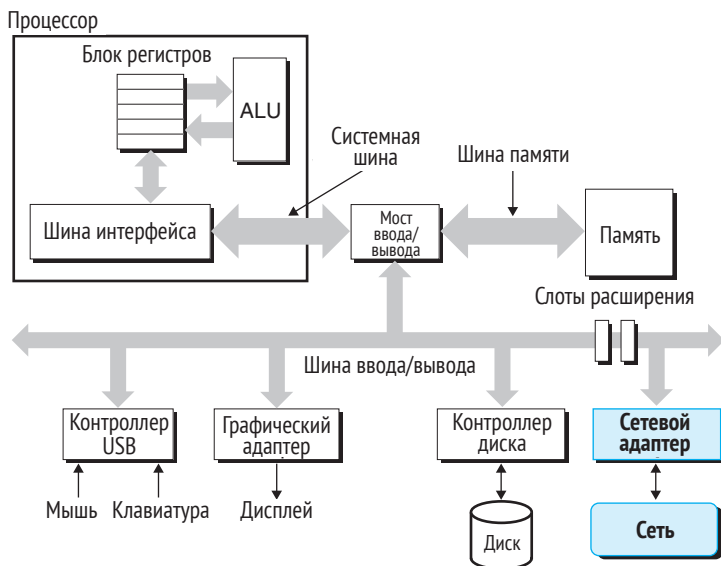


Рис. 11.2. Аппаратная организация сетевого хоста

Сегмент *Ethernet* (Ethernet segment) состоит из проводных соединений (обычно это витые пары) и небольшой коробки, которая называется *концентратором* (hub), как показано на рис. 11.3. Сегменты Ethernet обычно охватывают небольшие области, например комнаты или этаж здания. Все проводные соединения имеют одинаковую пропускную способность, обычно 100 Мбит/с или 1 Гбит/с. Один конец подключен к адаптеру хоста, другой – к *порту* концентратора. Концентратор послушно копирует каждый бит, который он получает, во все остальные порты. То есть каждый хост видит каждый бит, передаваемый по сети.

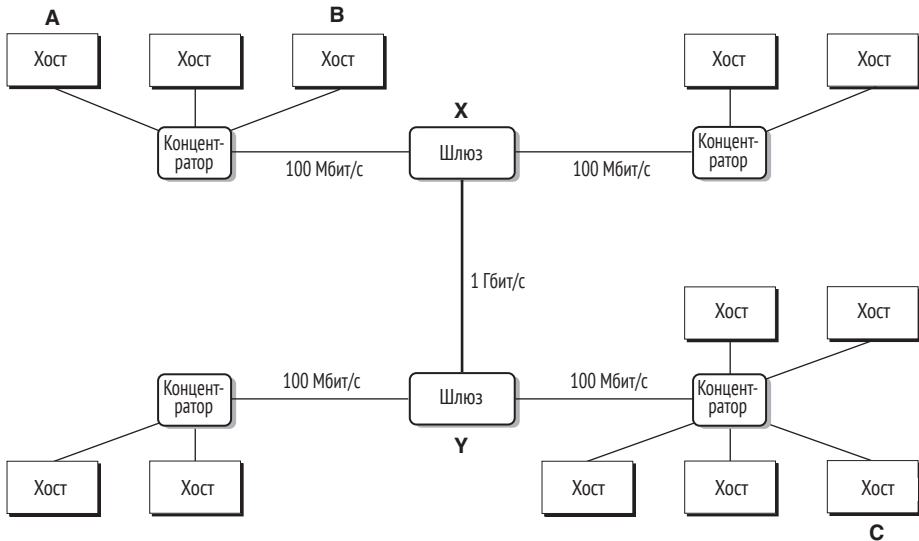


Рис. 11.3. Сегмент Ethernet

Каждый адаптер Ethernet имеет уникальный глобальный 48-разрядный адрес, который хранится в энергонезависимой памяти адаптера. Хост может послать порцию битов, называемую *кадром* (frame), другому хосту в сегменте. Каждый кадр содержит некоторое фиксированное количество битов *заголовка*, в которых определяются отправитель, получатель и длина кадра, а за заголовком следуют фактические данные. Каждый хост видит каждый кадр, но читает его только тот хост, которому этот кадр предназначен.

Многочисленные сегменты сети Ethernet могут объединяться в локальные сети больших размеров, которые еще называют *мостовыми сетями Ethernet* (bridged Ethernet),

как показано на рис. 11.4. В мостовой сети Ethernet одни кабели соединяют шлюз со шлюзом, а другие – шлюзы с концентраторами. Пропускные способности кабелей могут различаться. В рассматриваемом примере кабельное соединение шлюз–шлюз имеет пропускную способность 1 Гбит/с, а четыре соединения концентратор–шлюз – пропускную способность 100 Мбит/с.



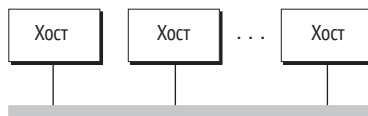
**Рис. 11.4.** Сегменты Ethernet, объединенные в мостовую локальную сеть

Шлюзы с большей эффективностью используют существующие кабельные линии, чем концентраторы. Используя искусный распределенный алгоритм, они автоматически фиксируют, какие хосты и через какие порты доступны, а затем выборочно копируют кадры из одного порта в другой, только если это необходимо. Например, если хост А посылает кадр хосту В, который находится в этом же сегменте, то шлюз X отбрасывает этот кадр, когда тот поступает в его входной порт, тем самым снижая нагрузку на линии передачи данных в других сегментах. В то же время если хост А посылает кадр хосту С, находящемуся в другом сегменте, то шлюз X скопирует этот кадр только в порт, связанный со шлюзом Y, а тот скопирует этот кадр только в порт, подключенный к сегменту с хостом С.

### Интернет или интересеть

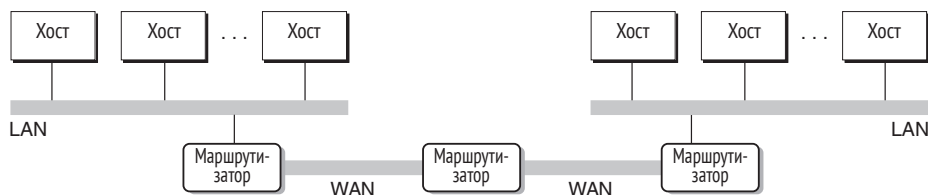
Мы используем термин *интерсеть*, когда подразумеваем обобщенную идею, и термин *интернет*, когда подразумеваем конкретную реализацию – всемирный интернет.

Чтобы упростить наше представление локальной сети, нарисует концентраторы и линии связи, их соединяющие, в виде горизонтальной линии, как показано на рис. 11.5.



**Рис. 11.5.** Концептуальное представление локальной сети

На верхнем уровне иерархии многочисленные несовместимые локальные сети могут соединяться друг с другом посредством специализированных компьютеров, получивших название *маршрутизаторов*, образуя при этом *интерсеть* (сеть с внутренней коммутацией). У каждого маршрутизатора имеется адаптер (порт) для каждой сети, к которой он подключен. Маршрутизаторы могут устанавливать высокоскоростные коммутируемые соединения, которые могут служить примерами глобальных сетей (Wide Area Networks, WAN). Их так называли в силу того факта, что они охватывают намного более обширные территории, чем могут локальные сети. В общем случае маршрутизаторы могут использоваться для построения интерсетей из произвольной совокупности локальных и глобальных сетей. Например, на рис. 11.6 показана интерсеть с парой локальных и парой глобальных сетей, соединенных между собой тремя маршрутизаторами.



**Рис. 11.6.** Небольшая интерсеть. Две локальные и две глобальные сети, связанные тремя маршрутизаторами

Важнейшим свойством интерсетей является их способность объединять совершенно различные локальные и глобальные сети с несовместимыми технологиями. Каждый хост физически связан с остальными хостами, но как сделать так, чтобы некоторый *хост-отправитель* мог посылать биты данных *хосту-получателю* через все эти несовместимые сети?

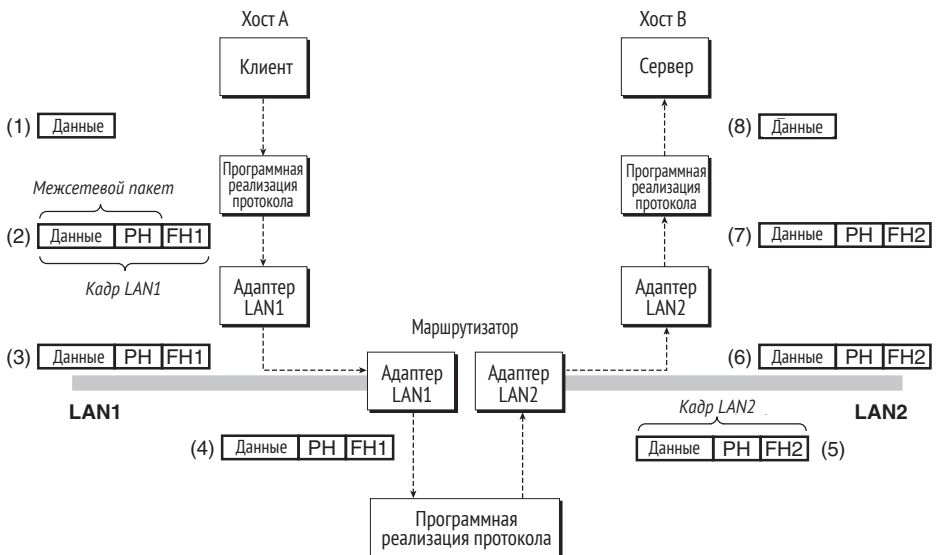
Решением является многоуровневая *программная реализация протоколов*, функционирующая на хосте и маршрутизаторе и сглаживающая различия между разными сетями. Протоколы должны обеспечить две базовые возможности:

- *схема именования.* Технологии передачи данных по различным локальным сетям реализуют различные несовместимые способы адресации хостов. Межсетевой протокол IP сглаживает эти различия, объявляя единообразный формат адресов хостов. Каждому хосту назначается, по меньшей мере, один из таких IP-адресов, которые однозначно его идентифицируют;
- *механизм доставки.* Различные технологии передачи данных по сети предусматривают различные и несовместимые способы кодирования битов в линиях передачи данных и упаковки битов в кадры. Межсетевой протокол IP сглаживает все эти различия, объявляя единый способ упаковки битов данных в отдельные порции, называемые *пакетами*. Пакет состоит из *заголовка*, содержащего данные о размере пакета и адреса хоста-отправителя и хоста-получателя, а также фактические данные, пересылаемые отправителем получателю.

На рис. 11.7 показан пример, как хосты и маршрутизаторы используют межсетевой протокол IP передачи данных через несовместимые локальные сети. Рассматриваемая в этом примере сеть состоит из двух локальных сетей, соединенных маршрутизатором. Клиент, выполняющийся на хосте *A* в локальной сети LAN1, посылает некоторую последовательность байтов данных серверу, выполняющемуся на хосте *B* в локальной сети LAN2. Передача происходит в восемь этапов.

1. Клиент, выполняющийся на хосте *A*, посылает системе запрос на копирование данных из виртуального адресного пространства в буфер ядра.

2. Программная реализация протокола на хосте *A* создает кадр локальной сети LAN1, добавляя к данным межсетевой заголовок и заголовок кадра LAN1. Межсетевой заголовок определяет получателем хост *B*. Заголовок кадра LAN1 определяет получателем маршрутизатор. Затем он передает сформированный кадр адаптеру. Обратите внимание, что кадр LAN1 является межсетевым пакетом, полезное содержимое которого составляют фактические данные пользователя. Такой вид *инкапсуляции* выступает одним из фундаментальных принципов работы сети.
3. Адаптер локальной сети LAN1 копирует кадр в сеть.
4. Когда кадр попадает в маршрутизатор, адаптер маршрутизатора, подключенный к локальной сети LAN1, читает его из линии и передает программной реализации протокола.
5. Маршрутизатор извлекает межсетевой адрес из заголовка меж сетевого пакета и использует его как индекс в таблице маршрутизации, чтобы определить, куда направить этот пакет, в данном случае – в локальную сеть LAN2. Затем маршрутизатор удаляет старый заголовок LAN1, добавляет новый заголовок кадра LAN2 с адресом хоста *B* и передает сформированный кадр адаптеру.
6. Адаптер маршрутизатора, подключенный к локальной сети LAN2, копирует этот кадр в сеть.
7. Когда кадр достигает хоста *B*, его адаптер читает кадр из линии и передает его программной реализации протокола.
8. В завершение реализация протокола хоста *B* удаляет заголовки пакета и кадра, копирует данные в виртуальное адресное пространство сервера, когда сервер выполняет системный вызов, чтобы получить эти данные.



**Рис. 11.7.** Как данные передаются между хостами через интернет. PH: заголовок межсетевого пакета; FH1: заголовок кадра для LAN1; FH2: заголовок кадра для LAN2

Разумеется, здесь мы не заостряем ваше внимание на многих возникающих при этом сложных проблемах. Как быть, когда разные сети работают с разными максимальными размерами кадров? Как маршрутизаторы получают информацию об изменении

топологии сети? И все же наш пример достаточно четко отражает суть передачи данных между сетями, основным принципом которой является инкапсуляция.

### 11.3. Всемирная сеть интернет

Всемирная сеть интернет, функционирующая на базе протокола IP (Internet Protocol – протокол интернета), – наиболее известная и успешная реализация интерсети. В той или иной форме она существует с 1969 года. Несмотря на то что внутренняя архитектура интернета сложна и постоянно меняется, организация приложений типа клиент–сервер остается удивительно устойчивой, начиная с 1980-х годов. На рис. 11.8 показана базовая организация интернет-приложения с архитектурой клиент–сервер.

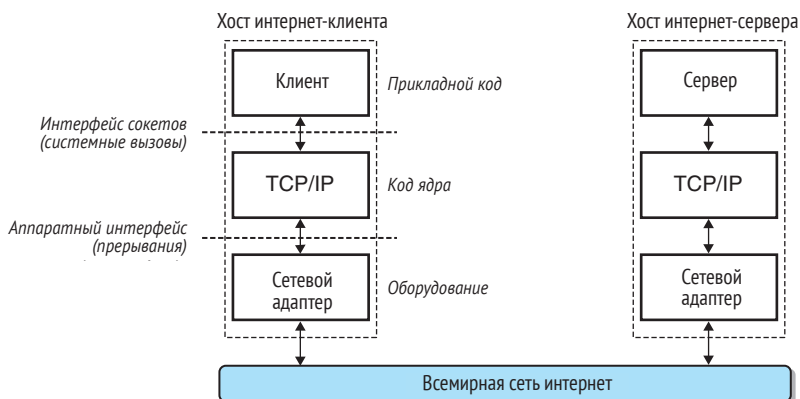


Рис. 11.8. Аппаратная и программная организация интернет-приложения

На каждом хосте, подключенном к интернету, действует программная реализация стека протоколов *TCP/IP* (Transmission Control Protocol/Internet Protocol – протокол управления передачей / протокол интернета), который поддерживается практически всеми современными компьютерными системами. Интернет-клиенты и интернет-серверы обмениваются данными, используя комбинацию функций *интерфейса сокетов* и ввода/вывода системы Unix. (Интерфейс сокетов мы рассмотрим в разделе 11.4.) Функции сокетов обычно доступны как системные вызовы, которые вызывают различные функции в режиме ядра.

TCP/IP – это целое семейство протоколов, каждый из которых выполняет свою задачу. Например, протокол IP определяет базовую схему именования и механизм доставки, обеспечивающий отправку пакетов, известных как *дейтаграммы*, с одного хоста на другой. Механизм IP ненадежен – он не предусматривает возможности восстановления дейтаграмм в случае их утери или дублирования. Протокол UDP (Unreliable Datagram Protocol – протокол ненадежных дейтаграмм) дополняет протокол IP с возможностью передачи пакетов между процессами, а не просто между хостами. Протокол TCP – еще более сложная реализация сетевого протокола, действующая поверх IP и обеспечивающая создание надежных полнодуплексных (двунаправленных) *соединений* между процессами. Для простоты обсуждения будем считать сочетание TCP/IP единым и монолитным протоколом. Мы не будем обсуждать его внутреннее устройство, а рассмотрим лишь некоторые основные свойства, которые протоколы TCP и IP предоставляют прикладным программам. Кроме того, мы не будем рассматривать протокол UDP.

С точки зрения программиста интернет можно рассматривать как совокупность всех хостов в мире, которые обладают следующими свойствами:

- множество хостов отображается в множество 32-разрядных IP-адресов;
- множество IP-адресов отображается в множество идентификаторов, так называемых доменных имен интернета;
- процесс, действующий на одном интернет-хосте, может обмениваться данными с процессом, действующим на другом интернет-хосте через установленное соединение.

В трех следующих разделах мы рассмотрим эти фундаментальные идеи, лежащие в основе интернета, более подробно.

#### IPv4 и IPv6

Оригинальный протокол интернета (IP) с 32-разрядными адресами известен как интернет-протокол версии 4 (Internet Protocol Version 4, IPv4). В 1996 году рабочая группа инженеров интернета (Internet Engineering Task Force, IETF) предложила новую версию IP, названную *протоколом интернета версии 6* (Internet Protocol Version 6, IPv6), использующую 128-разрядные адреса и которая задумывалась как замена IPv4. Однако по состоянию на 2015 год, почти 20 лет спустя, подавляющая часть интернет-трафика по-прежнему передается по сетям IPv4. Например, только 4 % пользователей пользуются услугами Google с применением IPv6 [42].

Из-за низкой скорости внедрения мы не будем подробно обсуждать IPv6 в этой книге и сосредоточимся исключительно на концепциях, лежащих в основе IPv4. Когда мы говорим об интернете, то имеем в виду интернет, основанный на IPv4. Тем не менее методы проектирования клиентов и серверов, о которых мы расскажем далее в этой главе, основаны на современных интерфейсах, не зависящих от какого-либо конкретного протокола.

### 11.3.1. IP-адреса

IP-адрес – это 32-разрядное целое число без знака. Сетевые программы хранят IP-адреса в специальной структуре (листинг 11.1).

#### Листинг 11.1. Структура IP-адреса

```
code/netp/netpfragments.c
/* Структура IP-адреса */
struct in_addr {
    uint32_t s_addr; /* Адрес с сетевым порядком следования байтов (big-endian) */
};
code/netp/netpfragments.c
```

Хранение скалярного адреса в структуре – неудачное наследие более ранних реализаций интерфейса сокетов. Целесообразнее было бы объявить для IP-адресов специальный скалярный тип, но сейчас слишком поздно вносить какие-либо изменения из-за огромной базы действующих приложений.

Поскольку интернет-хосты могут иметь разный порядок следования байтов, протокол TCP/IP объявляет единый порядок – *сетевой порядок следования байтов* (прямой, или big-endian) для любых целочисленных данных, таких как IP-адреса, которые переносятся через сеть в заголовках пакетов. Адреса в структурах IP-адресов всегда хранятся в прямом (сетевом) порядке, даже если аппаратура хоста использует обратный порядок. Система Unix предоставляет следующие функции для изменения порядка следования байтов между хостом и сетью.

```
#include <arpa/inet.h>
```

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
```

Возвращают значение с сетевым порядком следования байтов

```
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Возвращают значение с аппаратным следованием байтов хоста

Функция `htonl` меняет аппаратный порядок следования байтов 32-разрядного целого числа, принятый на хосте, на сетевой. Функция `ntohl` меняет сетевой порядок следования байтов 32-разрядного целого числа на аппаратный, принятый на хосте. Функции `htons` и `ntohs` выполняют соответствующие преобразования для 16-разрядных целых чисел.

Обычно человек работает с IP-адресами, представленными в так называемом *десятично-точечном* формате, в котором каждый байт представлен своим десятичным значением и отделен от других байтов точкой. Например, 128.2.194.242 – это десятично-точечное представление адреса 0x8002c2f2. В операционной системе Linux можно воспользоваться командой `hostname`, чтобы узнать адрес собственного компьютера в десятично-точечном представлении:

```
linux> hostname -i
128.2.210.175
```

Прикладные программы могут преобразовывать IP-адреса в двоично-десятичные строки и обратно, используя функции `inet_pton` и `inet_ntop`:

```
#include <arpa/inet.h>
```

```
int inet_pton(AF_INET, const char *src, void *dst);
```

Возвращает 1 в случае успеха, 0 – если передано недопустимое десятично-точечное представление, -1 в случае ошибки

```
const char *inet_ntop(AF_INET, const void *src, char *dst,
                      socklen_t size);
```

Возвращает указатель на строку с десятично-точечным представлением в случае успеха, NULL в случае ошибки

Буква «*n*» в именах этих функций означает «network» (сетевой), а «*p*» – «presentation» (представление). Они могут манипулировать 32-разрядными адресами IPv4 (AF\_INET), как показано здесь, или 128-разрядными адресами IPv6 (AF\_INET6), которые мы не рассматриваем.

Функция `inet_pton` преобразует строку с десятично-точечным представлением IP-адреса (`src`) в целочисленное представление с сетевым порядком следования байтов (`dst`). Если `src` указывает на строку с недопустимым IP-адресом в десятично-точечном представлении, то `inet_pton` возвращает NULL. В случае любой другой ошибки возвращается -1. Аналогично функция `inet_ntop` преобразует целочисленный IP-адрес с сетевым порядком следования байтов (`src`) в соответствующую строку десятично-точечного представления и копирует в `dst` до `size` байт.

**Упражнение 11.1 (решение в конце главы)**

Заполните следующую таблицу:

Шестнадцатеричный адрес	Десятично-точечный адрес
0x0	_____
0xffffffff	_____
0x7f000001	_____
_____	205.188.160.121
_____	64.12.149.13
_____	205.188.146.23

**Упражнение 11.2 (решение в конце главы)**

Напишите программу `hex2dd.c`, которая преобразует свой шестнадцатеричный аргумент в строку десятично-точечного представления и выводит результат. Например:

```
linux> ./hex2dd 0x8002c2f2
128.2.194.242
```

**Упражнение 11.3 (решение в конце главы)**

Напишите программу `dd2hex.c`, которая преобразует свой строковый аргумент с десятично-точечным представлением в шестнадцатеричное число и выводит результат. Например:

```
linux> ./dd2hex 128.2.194.242
0x8002c2f2
```

## 11.3.2. Доменные имена интернета

При взаимодействии друг с другом клиенты и серверы в интернете используют IP-адреса. Однако люди плохо запоминают длинные числа, поэтому в интернете, кроме IP-адресов, поддерживаются проще запоминающиеся *доменные имена*, а также механизм, отображающий доменные имена в IP-адреса. Доменное имя – это последовательность слов (букв, чисел и дефисов), отделенных друг от друга точками, например: `whaleshark.ics.cs.cmu.edu`.

Множество имен доменов образуют иерархию, и каждое имя в этой иерархии имеет свою позицию. Разберем это на примере. На рис. 11.9 показана часть иерархии доменных имен.

Иерархия имеет форму дерева. Узлы дерева представляют доменные имена, отражающие путь вдоль дерева в направлении его корня. Поддерева называются *поддоменами*. Первый уровень иерархии образует неименованный корневой узел. Следующий уровень образует совокупность имен доменов первого уровня, определенных Международной организацией по распределению номеров и имен (Internet Corporation for Assigned Names and Numbers, ICANN). Вот типичные представители доменов первого уровня: `com`, `edu`, `gov`, `org` и `net`.



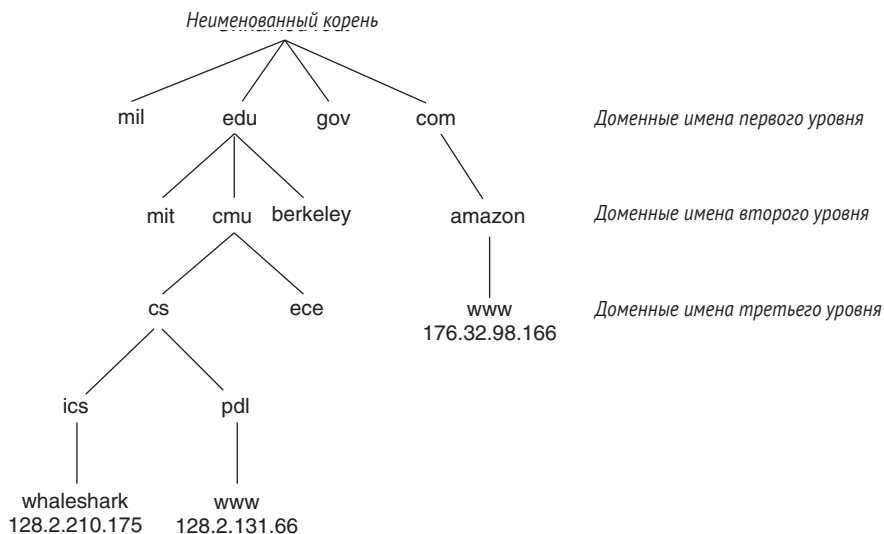


Рис. 11.9. Фрагмент иерархии доменных имен в интернете

Ниже находятся доменные имена *второго уровня*, такие как `cmu.edu`, которые назначаются по принципу очередности различными уполномоченными агентами организации ICANN. Вместе с доменным именем второго уровня организация получает право создавать другие доменные имена в пределах своего поддомена.

Интернет определяет соответствие между множеством доменных имен и множеством IP-адресов. До 1988 года это соответствие устанавливалось вручную в специальном текстовом файле с именем `HOSTS.TXT`. Но затем была реализована распределенная система определения соответствий, известная как *система доменных имен* (Domain Naming System, DNS). Концептуально база данных DNS состоит из многих миллионов записей, отображающих множество доменных имен в множество IP-адресов. В математическом смысле каждую запись можно представить как класс эквивалентности доменных имен и IP-адресов. Мы можем изучить некоторые свойства отображений DNS с помощью программы `NSLOOKUP` в Linux, которая отображает IP-адреса, связанные с доменным именем<sup>1</sup>.

### Сколько хостов в интернете?

С 1987 года дважды в год консорциум Internet Systems Consortium выпускает обзор с перечнем действующих доменов. В отчете дается оценка числа хостов в интернете путем подсчета IP-адресов, назначенных конкретному доменному имени. Так вот, при исследовании этих отчетов была обнаружена любопытная тенденция: начиная с 1987 года, когда в интернете было всего около 20 тыс. хостов, число хостов ежегодно удваивалось примерно вдвое. По состоянию на 2015 год в интернете насчитывалось более 1 млрд хостов!

Каждый хост имеет локальное доменное имя `localhost`, которое всегда отображается в *петлевой адрес* `127.0.0.1`:

```
linux> nslookup localhost
Address: 127.0.0.1
```

<sup>1</sup> В этом примере мы немного изменили формат вывода для большей удобочитаемости.

### Происхождение интернета

Интернет является примером одного из наиболее успешных взаимодействий правительства, университета и промышленности. Многие факторы повлияли на его успех, но, как нам кажется, наибольшую роль сыграли оказание финансовой поддержки правительством Соединенных Штатов на протяжении 30 лет, а также самоотверженность разработчиков, которую Дейв Кларк (Dave Clarke) охарактеризовал как «простые договоренности и рабочий код».

Семена интернета были посеяны в 1957 г., когда в самый разгар холодной войны Советский Союз потряс весь мир, запустив в космос первый искусственный спутник Земли. В ответ правительство Соединенных Штатов создало Управление перспективного планирования научно-исследовательских работ (Advanced Research Projects Agency, ARPA), перед которым была поставлена задача восстановить лидерство США в науке и технологиях. В 1967 году Лоренс Робертс (Lawrence Roberts) из ARPA опубликовал планы проведения работ по созданию новой сети, получившей название ARPANET (Advanced Research Projects Agency network). Первые узлы ARPANET начали работать в 1969-м. В 1971-м в сети ARPANET существовало уже 13 узлов и появилось первое сетевое приложение – электронная почта.

В 1972 г. Роберт Кан (Robert Kahn) сформулировал общие принципы межсетевого обмена, согласно которым обмен данными между сетями производится посредством независимых черных ящиков, получивших название «маршрутизаторы». В 1974-м Кан и Винтон Серф (Vinton Cerf) опубликовали первый вариант протокола TCP/IP, который к 1982-му стал стандартным протоколом межсетевых взаимодействий для сети ARPANET. С 1 января 1983 г. каждый узел в сети ARPANET был переключен на протокол TCP/IP, эта дата стала днем рождения всемирной сети интернет, функционирующей на базе протокола IP.

В 1985 г. Пол Моканетрис (Paul Mockapetris) изобрел систему доменных имен DNS (Domain Name System). На тот момент в интернете было уже около 1000 хостов. В следующем году Национальный научный фонд США (National Science Foundation, NSF) построил магистральную сеть NSFNET, соединившую 13 площадок посредством коммутируемых телефонных линий со скоростью передачи данных 56 Кбит/с. Позже в 1988-м были использованы линии связи T1, обеспечивающие скорость передачи 1,5 Мбит/с, а в 1991-м были использованы линии связи T3, обеспечивающие скорость передачи 45 Мбит/с. В 1988-м в интернете имелось уже более 50 тыс. хостов. В 1989 г. оригинальная сеть ARPANET официально перестала существовать. В 1995 г., когда в интернете имелось порядка 10 млн хостов, фонд NSF отправил в небытие сеть NSFNET, заменив ее современной сетью интернет с архитектурой, основанной на специализированных коммерческих магистральных сетях, связанных общедоступными точками доступа.

Имя localhost позволяет воспользоваться удобным и переносимым способом обращения к клиентам и серверам, которые выполняются на одной машине, что может очень пригодиться при отладке. Определить настоящее доменное имя локального хоста можно с помощью программы HOSTNAME:

```
linux> hostname
whaleshark.ics.cs.cmu.edu
```

В простейшем случае имеет место взаимно однозначное соответствие между доменным именем и IP-адресом:

```
linux> nslookup whaleshark.ics.cs.cmu.edu
Address: 128.2.210.175
```

Но иногда в один IP-адрес может отображаться сразу несколько доменных имен:

```
linux> nslookup cs.mit.edu
Address: 18.62.1.6
```

```
linux> nslookup eeecs.mit.edu
Address: 18.62.1.6
```

В самом общем случае несколько доменных имен могут отображаться в один и тот же набор IP-адресов:

```
linux> nslookup www.twitter.com
Address: 199.16.156.6
Address: 199.16.156.70
Address: 199.16.156.102
Address: 199.16.156.230
```

```
linux> nslookup twitter.com
Address: 199.16.156.102
Address: 199.16.156.230
Address: 199.16.156.6
Address: 199.16.156.70
```

Наконец, некоторые действительные доменные имена не отображаются ни в какие IP-адреса:

```
linux> nslookup edu
*** Can't find edu: No answer
linux> nslookup ics.cs.cmu.edu
*** Can't find ics.cs.cmu.edu: No answer
```

### 11.3.3. Интернет-соединения

Клиенты и серверы в интернете обмениваются между собой данными, посылая и принимая потоки байтов через *соединения*. Соединения называются *двухточечными*, или *двухсторонними* (point-to-point), потому что связывают пару процессов. Соединения называют *полнодуплексными* (full-duplex), потому что данные могут одновременно передаваться в обоих направлениях. И еще соединения называют *надежными* (reliable), потому что если отвлечься от некоторых катастрофических событий, таких как повреждение кабеля пресловутым беспечным экскаваторщиком, то поток байтов, посылаемый процессом-отправителем, в конечном итоге будет получен процессом-получателем в том порядке, в каком он был отправлен.

*Сокет* – это конечная точка соединения. Каждый сокет имеет *адрес сокета*, который состоит из IP-адреса и 16-разрядного целочисленного номера *порта*<sup>2</sup> и обозначается как address:port.

Порт в адресе сокета клиента назначается ядром автоматически, когда клиент посылает запрос на соединение, и называется *эфемерным портом* (ephemeral port). В то же время порт в адресе сокета сервера – это обычно *хорошо известный* номер порта, присвоенный соответствующей службе. Например, веб-серверы обычно используют порт 80, а серверы электронной почты – порт 25. Каждая служба, которой назначен постоянный хорошо известный порт, имеет *хорошо известное имя службы*. Например, веб-служба имеет хорошо известное имя http, а служба электронной почты – имя smtp. Соответствия между хорошо известными именами служб и номерами портов определяются в файле /etc/services.

Соединение однозначно идентифицируется адресами сокетов обеих его конечных точек. Такая пара адресов сокетов называется *парой сокетов* и обозначается кортежем

<sup>2</sup> Это программные порты; они никак не связаны с аппаратными портами в сетевых коммутаторах и маршрутизаторах.

(*cliaddr:cliport, servaddr:servport*), где *cliaddr* – IP-адрес клиента, *cliport* – порт клиента, *servaddr* – IP-адрес сервера и *servport* – порт сервера. Например, на рис. 11.10 показано соединение между веб-клиентом и веб-сервером.

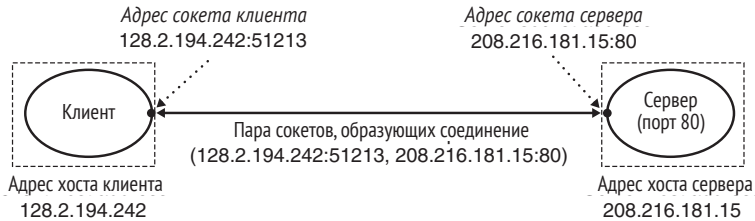


Рис. 11.10. Интернет-соединение

В этом примере сокет веб-клиента имеет адрес 128.2.194.242:51213, где 51213 – номер эфемерного порта, назначенный ядром, а сокет веб-сервера имеет адрес: 208.216.181.15:80, где 80 – хорошо известный номер порта, присвоенный веб-службам. Эти адреса сокетов клиента и сервера однозначно идентифицируют соединение между клиентом и сервером: (128.2.194.242:51213, 1208.216.181.15:80).

## 11.4. Интерфейс сокетов

*Интерфейс сокетов* – это набор функций, использующихся вместе с функциями ввода/вывода Unix для создания сетевых приложений. Он реализован в большинстве современных систем, включая все варианты операционных систем Unix, Windows и Macintosh. На рис. 11.11 показана обобщенная схема использования интерфейса сокетов в контексте обычной транзакции клиент–сервер. Держите этот рисунок перед глазами, пока мы будем обсуждать отдельные функции.

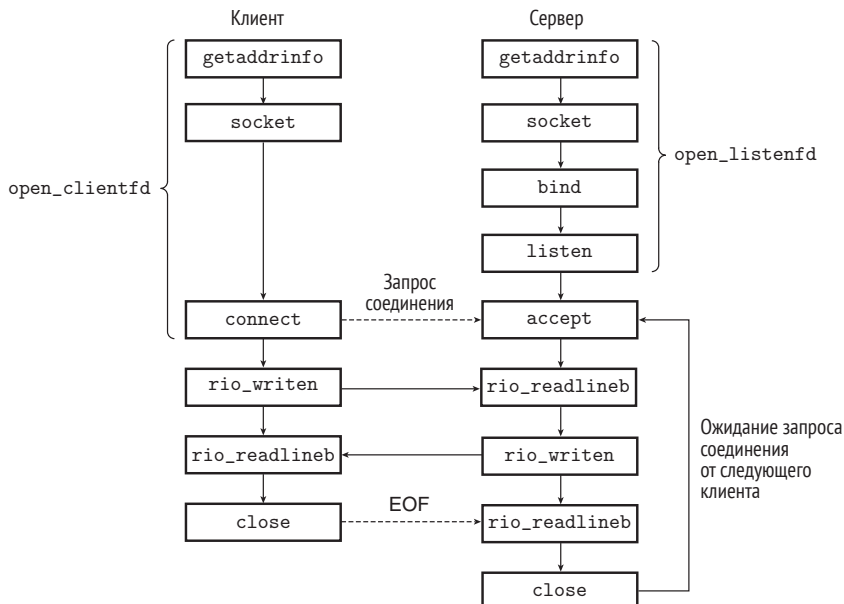


Рис. 11.11. Использование интерфейса сокетов сетевыми приложениями

### 11.4.1. Структуры адресов сокетов

С точки зрения ядра операционной системы Linux, сокет – это конечная точка соединения. С точки зрения программы для Linux сокет – это открытый файл с соответствующим дескриптором.

Адреса сокетов хранятся в виде 16-байтных структур `sockaddr_in`, показанных в листинге 11.2. В интернет-приложениях поле `sin_family` имеет значение `AF_INET`, поле `sin_port` хранит 16-разрядный номер порта, а поле `sin_addr` – 32-разрядный IP-адрес. IP-адрес и номер порта всегда хранятся в сетевом порядке следования байтов (big-endian).

#### Происхождение интерфейса сокетов

Интерфейс сокетов был разработан исследователями Калифорнийского университета в Беркли в начале 1980-х годов. По этой причине его часто называют сокетами Беркли (Berkeley sockets). Исследователи из Беркли разработали интерфейс сокетов, позволяющий работать с любым базовым протоколом. В первой реализации использовался протокол TCP/IP, в который они включили ядро Unix 4.2BSD, распространившееся по многочисленным университетам и лабораториям. Это было важное событие в истории интернета. Практически за одну ночь тысячи пользователей получили доступ к протоколу TCP/IP и его исходному коду. Это вызвало огромный ажиотаж среди пользователей и подтолкнуло дальнейшие исследования в области сетевых технологий.

#### Листинг 11.2. Структура адресов сокетов

```
/* Структура адреса сокета интернета */
struct sockaddr_in {
    uint16_t sin_family;    /* Семейство протоколов (всегда AF_INET) */
    uint16_t sin_port;      /* Номер порта с сетевым порядком байтов */
    struct in_addr sin_addr; /* IP-адрес с сетевым порядком байтов */
    unsigned char sin_zero[8]; /* Дополнение до sizeof(struct sockaddr) */
};

/* Обобщенная структура адреса сокета (для функций connect, bind и accept) */
struct sockaddr {
    uint16_t sa_family; /* Семейство протоколов */
    char sa_data[14];    /* Адрес */
};
```

*code/netp/netpfragments.c*

*code/netp/netpfragments.c*

#### Что означает окончание `_in`?

Окончание `_in` – это сокращение от *internet*, а не от *input*.

Функции `connect`, `bind` и `accept` требуют указателя на структуру с адресом сокета для определенного протокола. Разработчикам интерфейса сокетов пришлось столкнуться с проблемой – как определить эти функции, чтобы иметь возможность использовать любую структуру адреса сокета. В настоящее время принято использовать универсальный тип указателя `void *`. Но в ту пору этот тип отсутствовал в языке C, и было решено определить функции так, чтобы они принимали указатель на универсальную структуру `sockaddr` (листинг 11.2), а затем потребовать от приложений приведения указателя

на конкретную структуру к типу указателя на эту универсальную структуру. Чтобы упростить программный код, мы последуем рекомендациям Стивенса и определим следующий тип:

```
typedef struct sockaddr SA;
```

И будем применять этот тип всякий раз, когда нам понадобится привести указатель на структуру `sockaddr_in` к типу указателя на универсальную структуру `sockaddr`.

### 11.4.2. Функция `socket`

Чтобы получить *дескриптор сокета*, клиенты и серверы используют функцию `socket`.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Возвращает неотрицательный дескриптор в случае успеха,  
-1 в случае ошибки

Чтобы создать сокет – конечную точку соединения, – можно вызвать функцию `socket` с жестко зашитыми аргументами:

```
clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

где `AF_INET` указывает, что мы собираемся использовать 32-разрядные IP-адреса, а `SOCK_STREAM` указывает, что сокет будет конечной точкой соединения. Однако для получения значений аргументов и чтобы сделать программу независимой от протокола, лучше использовать функцию `getaddrinfo` (раздел 11.4.7). Мы покажем, как использовать `getaddrinfo` в паре с `socket`, в разделе 11.4.8.

Дескриптор `clientfd`, возвращаемый функцией `socket`, открыт только частично и не может использоваться в операциях чтения и записи. Порядок завершения процедуры открытия сокета зависит от того, в каком приложении она выполняется – клиентском или серверном. В следующем разделе описывается, как завершить процедуру открытия сокета на стороне клиента.

### 11.4.3. Функция `connect`

Клиент устанавливает соединение с сервером вызовом функции `connect`:

```
#include <sys/socket.h>

int connect(int clientfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Функция `connect` предпринимает попытку установить соединение с сервером, имеющим адрес сокета `addr`, где `addrlen` – значение `sizeof(sockaddr_in)`. Функция `connect` блокирует выполнение процесса до момента, когда соединение будет успешно установлено или возникнет ошибка. Если соединение успешно установлено, то дескриптор `clientfd` будет готов выполнять чтение и запись, а установленное соединение будет описываться парой сокетов:

```
(x:y, addr.sin_addr:addr.sin_port)
```

где  $x$  – IP-адрес клиента, а  $y$  – эфемерный порт, уникально идентифицирующий процесс на хосте клиента. Так же, как в случае с функцией `socket`, для подготовки аргументов лучше использовать функцию `getaddrinfo` (раздел 11.4.8).

### 11.4.4. Функция `bind`

Остальные функции сокетов – `bind`, `listen` и `accept` – используются серверами для установки соединения с клиентами.

```
#include <sys/socket.h>
```

```
int bind(int sockfd, const struct sockaddr *addr,  
        socklen_t addrlen);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Функция `bind` требует от ядра связать адрес сокета сервера в `addr` с дескриптором сокета `sockfd`. Аргумент `addrlen` – это значение `sizeof(sockaddr_in)`. Так же, как в случае с функцией `socket`, для подготовки аргументов лучше использовать функцию `getaddrinfo` (раздел 11.4.8).

### 11.4.5. Функция `listen`

Клиенты действуют как активная сторона и инициируют запросы на соединения. Серверы пассивны и ожидают запросов от клиентов. По умолчанию ядро полагает, что дескриптор, созданный функцией `socket`, соответствует активному сокету, существующему на другом конце соединения. Сервер вызывает функцию `listen`, чтобы сообщить ядру, что дескриптор будет использоваться сервером, а не клиентом.

```
#include <sys/socket.h>
```

```
int listen(int sockfd, int backlog);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Функция `listen` преобразует активный сокет `sockfd` в пассивный (`listen` – слушающий) сокет, который может принимать запросы на соединения от клиентов. Аргумент `backlog` определяет, сколько запросов на соединение ядро должно поставить в очередь, прежде чем начнет отклонять запросы. Точное понимание аргумента `backlog` требует подробного изучения протокола TCP/IP, что выходит за рамки этой книги. В дальнейшем мы будем передавать в этом аргументе большое значение, например 1024.

### 11.4.6. Функция `accept`

Серверы ожидают запросов на соединение от клиентов, вызывая функцию `accept`:

```
#include <sys/socket.h>
```

```
int accept(int listenfd, struct sockaddr *addr, int *addrlen);
```

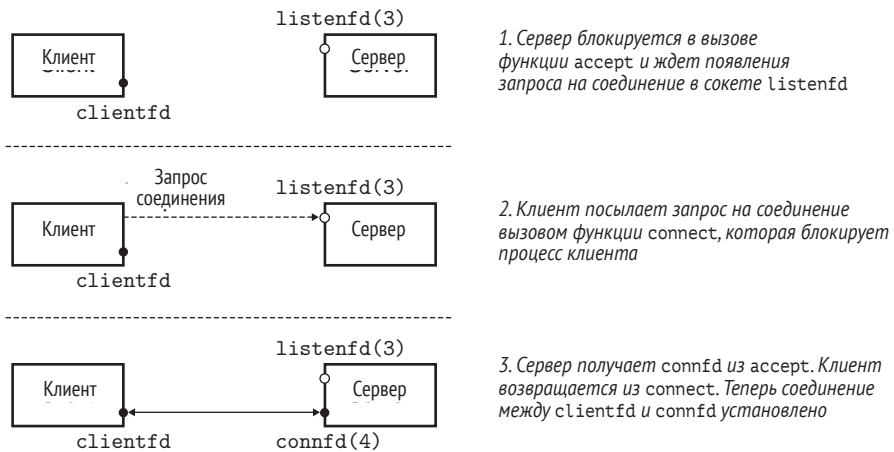
Возвращает неотрицательный дескриптор подключенного сокета в случае успеха, -1 в случае ошибки

Функция `accept` ожидает получения от клиентов запросов на соединение, которые поступают в слушающий сокет `listenfd`, затем записывает адрес сокета клиента в `addr` и

возвращает *подключенный сокет*, который можно использовать для обмена данными с клиентом посредством функций ввода/вывода Unix.

Различия между слушающим и подключенными сокетами приводят в замешательство многих студентов. Слушающий сокет служит конечной точкой для запросов клиента на соединение. Обычно он создается один раз и существует на протяжении всего времени выполнения сервера. Подключенный сокет – это конечная точка соединения, установленного между клиентом и сервером. Он создается всякий раз, когда сервер принимает запрос на соединение, и существует только на протяжении периода обслуживания клиента.

На рис. 11.12 показано, какие роли играют слушающий и подключенный сокеты. На этапе 1 сервер вызывает функцию `ассерт`, которая принимает дескриптор слушающего сокета и ждет получения запроса. Для большей определенности будем считать, что это дескриптор 3. Не забывайте, что дескрипторы 0–2 зарезервированы для стандартных потоков ввода/вывода.



**Рис. 11.12.** Роли слушающего и подключенного сокетов

На этапе 2 клиент вызывает функцию `connect`, которая посылает запрос на соединение сокету `listenfd`. На этапе 3 функция `ассерт` открывает новый подключенный сокет `connfd` (который мы будем обозначать дескриптором 4), устанавливает соединение между `clientfd` и `connfd`, а затем возвращает `connfd` приложению. На стороне клиента функция `connect` возвращает управление, и с этого момента клиент и сервер могут передавать данные, выполняя операции чтения и записи с дескрипторами `clientfd` и `connfd`.

#### Почему возникло деление на слушающий и подключенный сокеты?

Вас, возможно, удивляет, почему интерфейс сокетов различает слушающие и подключенные сокеты. На первый взгляд это выглядит ненужным усложнением. Однако такое деление между этими двумя видами сокетов приносит определенную пользу, потому что позволяет проектировать серверы, выполняющиеся конкурентно и способные обслуживать одновременно множество клиентов. Например, всякий раз, когда слушающий сокет получает запрос на соединение, мы можем запустить новый процесс, обменивающийся данными с клиентом через дескриптор подключенного сокета. Более подробно о конкурентных серверах рассказывается в главе 12.



### 11.4.7. Преобразование имен хостов и служб

Linux предоставляет пару мощных функций – `getaddrinfo` и `getnameinfo` – для преобразования между двоичными структурами адресов сокетов и строковыми представлениями имен хостов, адресов хостов, имен служб и номеров портов. При использовании в сочетании с интерфейсом сокетов они позволяют писать сетевые программы, не зависящие от конкретной версии протокола IP.

#### Функция `getaddrinfo`

Функция `getaddrinfo` преобразует строковое представление имени хоста, адреса хоста, имени службы и номера порта в структуру адреса сокета. Это современная замена устаревшим функциям `gethostbyname` и `getservbyname`. В отличие от устаревших функций, `getaddrinfo` – реентерабельная (раздел 12.7.2) и работает с любыми протоколами.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *host, const char *service,
               const struct addrinfo *hints,
               struct addrinfo **result);
                                   Возвращает 0 в случае успеха, ненулевой
                                   код ошибки в случае ошибки

void freeaddrinfo(struct addrinfo *result);
                                   Ничего не возвращает

const char *gai_strerror(int errcode);
                                   Возвращает текст сообщения об ошибке
```

Для заданных имен хоста и службы (два компонента адреса сокета) `getaddrinfo` возвращает в `result` указатель на список структур `addrinfo`, каждая из которых ссылается на адрес сокета, соответствующего хосту и службе (рис. 11.13).

После вызова `getaddrinfo` клиент выполняет обход этого списка и по очереди пробует каждый адрес сокета, пока вызовы `socket` и `connect` не завершатся успехом и соединение не будет установлено. Точно так же сервер пробует каждый адрес сокета в полученном списке, пока вызовы `socket` и `bind` не завершатся успехом и дескриптор не будет привязан к действительному адресу сокета. Чтобы избежать утечек памяти, по завершении процедуры соединения приложение должно освободить список, вызвав `freeaddrinfo`. Если `getaddrinfo` возвращает ненулевой код ошибки, то приложение может вызвать `gai_strerror`, чтобы преобразовать код в строку с описанием ошибки.

В аргументе `host` функции `getaddrinfo` можно передать строку с доменным именем или числовым адресом (например, IP-адресом в десятично-точечном представлении). В аргументе `service` можно передать имя службы (например, `http`) или десятичный номер порта. Если преобразовывать имя хоста в адрес не требуется, то в аргументе `host` можно передать `NULL`. То же относится к аргументу `service`. Однако хотя бы один из них должен быть указан.

В необязательном аргументе `hints` можно передать структуру `addrinfo` (листинг 11.3), обеспечивающую, чтобы подсказать функции `getaddrinfo`, какие адреса сокетов она может включить в возвращаемый список. В аргументе `hint` требуется определить только поля `ai_family`, `ai_socktype`, `ai_protocol` и `ai_flags`. В остальных полях должны передаваться нули (или `NULL`). На практике обычно вызывается функция `memset` для обнуления всей структуры, а затем устанавливаются выбранные поля:

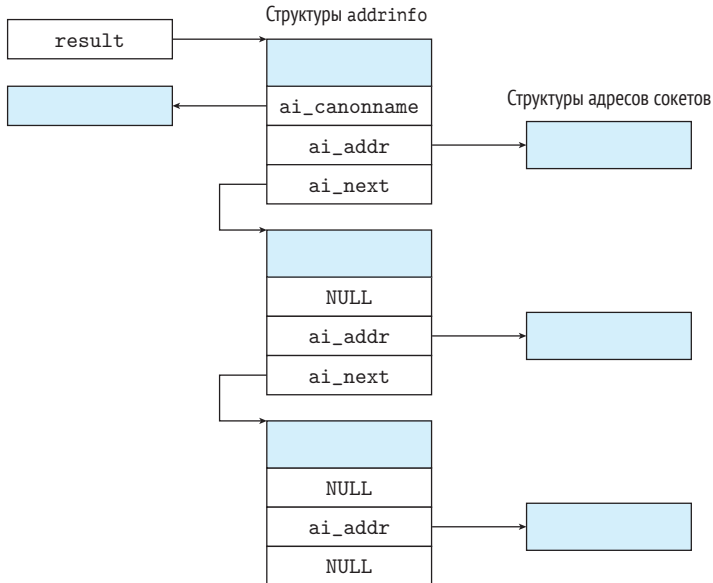


Рис. 11.13. Структура данных, возвращаемая функцией getaddrinfo

Листинг 11.3. Структура addrinfo, используемая функцией getaddrinfo

```

code/netp/netpfragments.c

struct addrinfo {
    int          ai_flags;      /* Аргумент с флагами-подсказками */
    int          ai_family;     /* Первый аргумент для socket */
    int          ai_socktype;    /* Второй аргумент для socket */
    int          ai_protocol;    /* Третий аргумент для socket */
    char         *ai_canonname; /* Каноническое имя хоста */
    size_t       ai_addrlen;    /* Размер структуры ai_addr */
    struct sockaddr *ai_addr;    /* Указатель на структуру адреса сокета */
    struct addrinfo *ai_next;    /* Указатель на следующий элемент списка */
};

code/netp/netpfragments.c

```

- по умолчанию getaddrinfo может возвращать адреса сокетов IPv4 и IPv6. Если передать в поле ai\_family значение AF\_INET, то в возвращаемый список будут включены только адреса IPv4. Если передать значение AF\_INET6, то getaddrinfo вернет только адреса IPv6;
- по умолчанию для каждого уникального адреса, соответствующего аргументу host, функция getaddrinfo может вернуть до трех структур addrinfo, отличающихся полем ai\_socktype: одну для надежных соединений, одну для обмена дейтаграммами (в этой книге не рассматриваются) и одну для низкоуровневых сокетов (в этой книге не рассматриваются). Если в ai\_socktype передать значение SOCK\_STREAM, то для каждого уникального адреса getaddrinfo вернет не более одной структуры addrinfo, определяющей адрес сокета, который можно использовать в качестве конечной точки соединения. Мы будем использовать эту особенность во всех наших примерах программ;

- поле `ai_flags` – это битовая маска, влияющая на поведение по умолчанию. Она создается путем объединения различных значений по ИЛИ. Вот некоторые из флагов, которые мы считаем полезными:
  - ♦ `AI_ADDRCONFIG`. Этот флаг рекомендуется устанавливать при использовании надежных соединений [34]. Он подсказывает функции `getaddrinfo`, что та должна возвращать адреса IPv4 только в том случае, если локальный хост настроен на использование IPv4. Аналогично для IPv6;
  - ♦ `AI_CANONNAME`. По умолчанию поле `ai_canonname` имеет значение `NULL`. Если установить этот флаг, то `getaddrinfo` запишет в поле `ai_canonname` первой структуры `addrinfo` в списке каноническое (официальное) имя хоста (рис. 11.13);
  - ♦ `AI_NUMERICSERV`. По умолчанию в аргументе `service` может передаваться имя службы или номер порта. Этот требует интерпретировать аргумент `service` как номер порта;
  - ♦ `AI_PASSIVE`. По умолчанию `getaddrinfo` возвращает адреса сокетов, которые могут использоваться клиентами (активные сокет) в вызовах `connect`. Этот флаг требует возвращать адреса пассивных сокетов, которые могут использоваться серверами в роли слушающих сокетов. В этом случае в аргументе `host` должно передаваться значение `NULL`, а в возвращаемых структурах адресов сокетов поле адреса будет содержать *подстановочный адрес*, сообщающий ядру, что этот сервер будет принимать запросы к любому из IP-адресов данного хоста. Мы будем использовать эту особенность во всех наших примерах серверов.

Создавая структуры `addrinfo` в возвращаемом списке, `getaddrinfo` заполняет все поля, кроме `ai_flags`. Поле `ai_addr` указывает на структуру адреса сокета, поле `ai_addrlen` определяет размер этой структуры, а поле `ai_next` указывает на следующую структуру `addrinfo` в списке. Другие поля описывают различные атрибуты адреса сокета.

Одна из замечательных особенностей `getaddrinfo` – непрозрачность полей в структуре `addrinfo`, в том смысле, что они могут передаваться непосредственно функциям интерфейса сокетов без дополнительных манипуляций со стороны приложения. Например, `ai_family`, `ai_socktype` и `ai_protocol` можно напрямую передать в вызов `socket`. Точно так же `ai_addr` и `ai_addrlen` можно передать напрямую в вызовы `connect` и `bind`. Эта особенность позволяет писать клиенты и серверы, независимые от конкретной версии протокола IP.

## Функция `getnameinfo`

Функция `getnameinfo` является обратной по отношению к функции `getaddrinfo`. Она преобразует структуру адреса сокета в соответствующие строки с именами хоста и службы. Это современная замена устаревшим функциям `gethostbyaddr` и `getservbyport`, и, в отличие от этих функций, она реентерабельна и не зависит от протокола.

```
#include <sys/socket.h>
#include <netdb.h>
```

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
               char *host, size_t hostlen,
               char *service, size_t servlen, int flags);
```

Возвращает 0 в случае успеха, ненулевой код ошибки в случае ошибки

Аргумент `sa` указывает на структуру адреса сокета с размером `salen`, аргумент `host` указывает на буфер с размером `hostlen`, и аргумент `service` – на буфер с размером `servlen`. Функция `getnameinfo` преобразует структуру адреса сокета `sa` в соответствующие строки с именами хоста и службы и копирует их в буферы `host` и `service`. Если `getnameinfo` возвращает ненулевой код ошибки, то приложение может преобразовать его в строку с описанием, вызвав `gai_strerror`.

Если имя хоста не нужно, то в `host` можно передать `NULL` и в `hostlen` – ноль. То же относится к имени службы, но хотя бы одна пара аргументов должна быть ненулевой.

Аргумент `flags` – это битовая маска, влияющая на поведение по умолчанию. Она создается путем объединения различных значений по ИЛИ. Вот пара флагов, которые могут пригодиться:

- `NI_NUMERICHOST`. По умолчанию `getnameinfo` пытается вернуть доменное имя в `host`. Если передать этот флаг, то `getnameinfo` вернет адрес в числовой форме;
- `NI_NUMERICSERV`. По умолчанию `getnameinfo` выполняет поиск в `/etc/services` и, если возможно, возвращает имя службы вместо номера порта. Если передать этот флаг, то `getnameinfo` просто вернет номер порта.

В листинге 11.4 показана простая программа `HOSTINFO`, которая использует `getaddrinfo` и `getnameinfo` для отображения доменного имени в соответствующие ему IP-адреса. Она похожа на программу `NSLOOKUP` из раздела 11.3.2.

**Листинг 11.4.** `HOSTINFO` отображает доменное имя в соответствующие ему IP-адреса

*code/netp/hostinfo.c*

```

1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     struct addrinfo *p, *listp, hints;
6     char buf[MAXLINE];
7     int rc, flags;
8
9     if (argc != 2) {
10         fprintf(stderr, "usage: %s <domain name>\n", argv[0]);
11         exit(0);
12     }
13
14     /* Получить список структур addrinfo */
15     memset(&hints, 0, sizeof(struct addrinfo));
16     hints.ai_family = AF_INET; /* Только IPv4 */
17     hints.ai_socktype = SOCK_STREAM; /* Только для надежных соединений */
18     if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
19         fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
20         exit(1);
21     }
22
23     /* Выполнить обход списка и вывести IP-адреса */
24     flags = NI_NUMERICHOST; /* Получать адреса вместо доменных имен */
25     for (p = listp; p; p = p->ai_next) {
26         Getnameinfo(p->ai_addr, p->ai_addrlen, buf, MAXLINE, NULL, 0, flags);
27         printf("%s\n", buf);
28     }
29
30     /* Освободить ресурсы */

```

```

31     Freeaddrinfo(listp);
32
33     exit(0);
34 }

```

*code/netp/hostinfo.c*

Сначала она инициализирует структуру `hints`, чтобы `getaddrinfo` возвращала нужные адреса. В данном случае нас интересуют 32-разрядные IP-адреса (строка 16), которые можно использовать в качестве конечных точек надежных соединений (строка 17). Поскольку нам нужны только доменные имена, мы вызываем `getaddrinfo` со значением `NULL` в аргументе `service`.

После вызова `getaddrinfo` программа выполняет обход списка структур `addrinfo` и с помощью `getnameinfo` преобразует каждый адрес сокета в строку с десятично-точечным представлением адреса. Закончив обход, программа освобождает список вызовом `freeaddrinfo` (хотя в такой простой программе этого можно было и не делать).

Запустив `HOSTINFO`, мы увидели, что имя `twitter.com` отображается в четыре IP-адреса. Тот же результат мы получили от команды `NSLOOKUP` в разделе 11.3.2.

```

linux> ./hostinfo twitter.com
199.16.156.102
199.16.156.230
199.16.156.6
199.16.156.70

```

#### Упражнение 11.4 (решение в конце главы)

Функции `getaddrinfo` и `getnameinfo` внутренне используют функции `inet_pton` и `inet_ntop` и, соответственно, обеспечивают более высокий уровень абстракции, не зависящий от конкретного формата адреса. Чтобы ощутить, насколько это удобно, напишите версию `HOSTINFO` (листинг 11.4), используя `inet_ntop` вместо `getnameinfo` для преобразования каждого адреса сокета в строку с адресом в десятично-точечной форме.

### 11.4.8. Вспомогательные функции для интерфейса сокетов

Прямое использование функции `getaddrinfo` и интерфейса сокетов может показаться сложным для тех, кто только познакомился с ними. Поэтому мы посчитали правильным решением написать вспомогательные функции-обертки `open_clientfd` и `open_listenfd`.

#### Функция `open_clientfd`

Функция `open_clientfd` может использоваться на стороне клиента для установки соединения с сервером.

```

#include "csapp.h"

int open_clientfd(char *hostname, char *port);

        Возвращает дескриптор в случае успеха, -1 в случае ошибки

```

Функция `open_clientfd` устанавливает соединение с сервером, действующим на хосте `hostname` и принимающим запросы на соединение на порту `port`. Она возвращает дес-

криптор открытого сокета, готового к вводу/выводу. Реализация функции приводится в листинге 11.5.

**Листинг 11.5.** `open_clientfd`: вспомогательная функция, устанавливающая соединение с сервером; реентерабельная и не зависящая от протокола

*code/src/csapp.c*

```

1 int open_clientfd(char *hostname, char *port) {
2     int clientfd;
3     struct addrinfo hints, *listp, *p;
4
5     /* Получить список потенциальных адресов сервера */
6     memset(&hints, 0, sizeof(struct addrinfo));
7     hints.ai_socktype = SOCK_STREAM; /* Открыть надежное соединение */
8     hints.ai_flags = AI_NUMERICSERV; /* ... номер порта -- число. */
9     hints.ai_flags |= AI_ADDRCONFIG; /* Рекомендуемый флаг для соединений */
10    Getaddrinfo(hostname, port, &hints, &listp);
11
12    /* Найти адрес сокета, с которым удастся установить соединение */
13    for (p = listp; p; p = p->ai_next) {
14        /* Создать дескриптор сокета */
15        if ((clientfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
16            continue; /* Неудача, попробовать следующий адрес */
17
18        /* Установить соединение с сервером */
19        if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
20            break; /* Соединение установлено! */
21        Close(clientfd); /* Ошибка соединения, попробовать следующий адрес */
22    }
23
24    /* Освободить ресурсы */
25    Freeaddrinfo(listp);
26    if (!p) /* Ни с одним адресом не удалось установить соединение */
27        return -1;
28    else /* Вернуть сокет, которому удалось установить соединение */
29        return clientfd;
30 }
```

*code/src/csapp.c*

Наша функция вызывает `getaddrinfo` и получает список структур `addrinfo`, каждая из которых ссылается на структуру адреса сокета, потенциально пригодного для установки соединения с сервером, действующим на хосте `hostname` и прослушивающим порт `port`. Затем выполняется обход списка и производятся попытки установить соединение с каждым из адресов по очереди, пока вызовы `socket` и `connect` не увенчаются успехом. Если соединение установить не удалось, дескриптор сокета закрывается, и предпринимается попытка использовать следующий адрес. Если вызов `connect` завершился успехом, память, занятая списком, освобождается, и вызывающей программе возвращается дескриптор сокета, готовый к выполнению операций ввода/вывода.

Обратите внимание, что в коде нет ничего, что делало бы его зависимым от конкретной версии протокола IP. Аргументы `socket` и `connect` автоматически генерируются вызовом `getaddrinfo`, что делает наш код ясным и переносимым.

## Функция `open_listenfd`

Функция `open_listenfd` может использоваться на стороне сервера и создает дескриптор сокета, готового принимать запросы на соединение.

```
#include "csapp.h"
```

```
int open_listenfd(char *port);
```

Возвращает дескриптор в случае успеха, -1 в случае ошибки

Функция `open_listenfd` возвращает дескриптор сокета, слушающего порт `port` и готового принимать запросы на соединение. Ее реализация приводится в листинге 11.6.

**Листинг 11.6.** `open_listenfd`: вспомогательная функция, которая открывает и возвращает дескриптор слушающего сокета; реентерабельная и не зависящая от протокола

*code/src/csapp.c*

```
1 int open_listenfd(char *port)
2 {
3     struct addrinfo hints, *listp, *p;
4     int listenfd, optval=1;
5
6     /* Получить список потенциальных адресов сервера */
7     memset(&hints, 0, sizeof(struct addrinfo));
8     hints.ai_socktype = SOCK_STREAM;           /* Принимать соединения */
9     hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* ... на любом IP-адресе */
10    hints.ai_flags |= AI_NUMERICSERV;          /* ... номер порта -- число */
11    Getaddrinfo(NULL, port, &hints, &listp);
12
13    /* Найти адрес сокета, который можно связать */
14    for (p = listp; p; p = p->ai_next) {
15        /* Создать дескриптор сокета */
16        if ((listenfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
17            continue; /* Неудача, попробовать следующий адрес */
18
19        /* Предотвратить появление ошибки "Адрес уже используется" */
20        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
21            (const void *)&optval, sizeof(int));
22
23        /* Связать дескриптор с адресом */
24        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
25            break; /* Успех */
26        Close(listenfd); /* Ошибка, попробовать следующий адрес */
27    }
28
29    /* Освободить ресурсы */
30    Freeaddrinfo(listp);
31    if (!p) /* Ни один адрес не подошел */
32        return -1;
33
34    /* Сделать сокет слушающим и готовым принимать запросы на соединение */
35    if (listen(listenfd, LISTENQ) < 0) {
36        Close(listenfd);
37        return -1;
38    }
39    return listenfd;
40 }
```

*code/src/csapp.c*

Она организована так же, как `open_clientfd`. Сначала вызывается `getaddrinfo`, а затем выполняется обход полученного списка, пока вызовы `socket` и `bind` не завершатся успе-

хом. Обратите внимание на вызов функции `setsockopt` (здесь не описывается) в строке 20, с помощью которой производится настройка сокета сервера, чтобы его можно было остановить, перезапустить и начать немедленно принимать запросы на соединение. По умолчанию после перезапуска сервер будет отклонять запросы на соединение в течение примерно 30 секунд, что может затруднить отладку.

Функция `getaddrinfo` вызывается с флагом `AI_PASSIVE` и значением `NULL` в аргументе `host`, поэтому поле адреса во всех возвращаемых структурах адресов сокетов будет содержать подстановочный адрес, сообщающий ядру, что этот сервер будет принимать запросы на любом IP-адресе этого хоста.

В заключение вызывается функция `listen`, которая превращает сокет `listenfd` в слушающий сокет и возвращает результат вызывающей программе. Если `listen` терпит неудачу, то дескриптор закрывается, чтобы избежать утечки памяти.

### 11.4.9. Примеры эхо-клиента и эхо-сервера

Лучший способ изучения интерфейса сокетов – изучение примеров его использования на практике. В листинге 11.7 показана реализация эхо-клиента. После соединения с сервером клиент входит в цикл, в котором многократно читает текстовую строку из стандартного ввода, посылает ее серверу, читает ответ сервера и выводит результат в стандартный вывод. Цикл прерывается, когда функция `fgets` получает признак конца файла EOF из стандартного ввода, когда пользователь нажмет комбинацию **Ctrl+D** на клавиатуре или когда будет достигнут конец файла, переадресованного в стандартный ввод.

После завершения цикла клиент закрывает дескриптор. В результате этого сервер получает признак EOF достижения конца файла, который обнаруживается при получении нулевого значения от функции `rio_readlineb`. Закрыв дескриптор, клиент завершает работу. По завершении процесса клиента ядро автоматически закрывает все открытые им дескрипторы, поэтому вызов `Close` в строке 24 не особенно нужен. Однако в программировании считается хорошим тоном явно закрывать все дескрипторы, которые были открыты.

**Листинг 11.7.** Функция `main` эхо-клиента

*code/netp/echoclient.c*

```

1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     int clientfd;
6     char *host, *port, buf[MAXLINE];
7     rio_t rio;
8
9     if (argc != 3) {
10         fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
11         exit(0);
12     }
13     host = argv[1];
14     port = argv[2];
15
16     clientfd = Open_clientfd(host, port);
17     Rio_readinitb(&rio, clientfd);
18
19     while (Fgets(buf, MAXLINE, stdin) != NULL) {
20         Rio_writen(clientfd, buf, strlen(buf));
21         Rio_readlineb(&rio, buf, MAXLINE);

```



```

22     Fputs(buf, stdout);
23 }
24 Close(clientfd);
25 exit(0);
26 }

```

*code/netp/echoclient.c*

В листинге 11.8 показана функция `main` эхо-сервера. После открытия слушающего дескриптора запускается бесконечный цикл. Каждая итерация этого цикла начинается с того, что сервер переходит в режим ожидания получения запроса на соединение от клиента. При получении такого запроса сервер выводит доменное имя и IP-адрес клиента и вызывает функцию `echo`, которая продолжит обслуживание клиента. После того как `echo` вернет управление, функция `main` закрывает связанный дескриптор, и когда клиент и сервер закроют соответствующие дескрипторы, соединение разрывается.

#### Листинг 11.8. Функция `main` итеративного эхо-сервера

*code/netp/echoserveri.c*

```

1 #include "csapp.h"
2
3 void echo(int connfd);
4
5 int main(int argc, char **argv)
6 {
7     int listenfd, connfd;
8     socklen_t clientlen;
9     struct sockaddr_storage clientaddr; /* Достаточно места для любого адреса */
10    char client_hostname[MAXLINE], client_port[MAXLINE];
11
12    if (argc != 2) {
13        fprintf(stderr, "usage: %s <port>\n", argv[0]);
14        exit(0);
15    }
16
17    listenfd = Open_listenfd(argv[1]);
18    while (1) {
19        clientlen = sizeof(struct sockaddr_storage);
20        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
21        Getnameinfo((SA *)&clientaddr, clientlen, client_hostname, MAXLINE,
22        client_port, MAXLINE, 0);
23        printf("Connected to (%s, %s)\n", client_hostname, client_port);
24        echo(connfd);
25        Close(connfd);
26    }
27    exit(0);
28 }

```

*code/netp/echoserveri.c*

Переменная `clientaddr` в строке 9 – это структура адреса сокета, которая передается в вызов `accept`. Перед возвратом `accept` записывает в `clientaddr` адрес сокета клиента на другом конце соединения. Обратите внимание, что `clientaddr` объявляется с типом `struct sockaddr_storage`, а не `struct sockaddr_in`. По определению, структура `sockaddr_storage` достаточно велика, чтобы вместить адрес сокета любого типа, что обеспечивает независимость реализации от протокола.

Обратите внимание, что простой эхо-сервер может обслуживать клиентов только по одному. Поэтому такие серверы называют *итеративными*. В главе 12 мы покажем, как

строить более сложные *конкурентные серверы*, способные обслуживать одновременно несколько клиентов.

Наконец, в листинге 11.9 показана реализация функции `echo`, которая принимает строки от клиента и посылает их обратно, пока вызов функции `rio_readlineb` в строке 10 не вернет признак конца файла.

**Листинг 11.9.** Функция `echo`, которая принимает строки от клиента и посылает их обратно

```
code/netp/echo.c
1 #include "csapp.h"
2
3 void echo(int connfd)
4 {
5     size_t n;
6     char buf[MAXLINE];
7     rio_t rio;
8
9     Rio_readinitb(&rio, connfd);
10    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
11        printf("server received %d bytes\n", (int)n);
12        Rio_writen(connfd, buf, n);
13    }
14 }
```

code/netp/echo.c

#### Что означает EOF для сетевых соединений?

Идея использования признака EOF вызывает путаницу в умах студентов, особенно в контексте сетевых соединений. Во-первых, вы должны осознавать, что в реальности не существует символа EOF. На самом деле EOF – это условие (условие достижения конца файла), которое обнаруживается ядром. Приложение обнаруживает условие EOF, когда получает ноль от функции `read`. Для дисковых файлов условие EOF возникает, когда позиция в текущем файле выходит за пределы файла. Для сетевых соединений EOF возникает в случаях, когда какой-то процесс закрывает соединение на своем конце. Процесс на другом конце соединения обнаруживает условие EOF, когда делает попытку прочитать последний байт из потока.

## 11.5. Веб-серверы

До сих пор мы обсуждали сетевое программирование в контексте простого эхо-сервера. В данном разделе мы покажем, как, используя базовые идеи сетевого программирования, создать свой небольшой, но в то же время полностью работоспособный веб-сервер.

### 11.5.1. Основные сведения о вебе

Веб-клиенты и веб-серверы взаимодействуют, используя текстовый протокол прикладного уровня, известный как *протокол передачи гипертекста* (Hypertext Transfer Protocol, HTTP). Протокол HTTP достаточно прост. Веб-клиент (*браузер*) открывает соединение с сервером и запрашивает некоторый *контент* (*содержимое*). В ответ сервер посылает затребованный контент, после чего разрывает соединение. Браузер читает контент и отображает его на экране.

Чем отличается веб-сервер от обычной службы поиска файлов, такой как FTP? Основное отличие заключается в том, что веб-контент может быть написан на языке *описания гипертекстовых документов* (Hypertext Markup Language, HTML). Программа (страница) на языке HTML содержит инструкции (теги), которые сообщают браузеру, как отображать различные фрагменты текста и графические объекты. Например, код

```
<b> Make me bold! </b>
```

предписывает вывести текст, заключенный между тегами `<b>` и `</b>`, жирным шрифтом. Однако основное достоинство языка HTML заключается в возможности включать в страницы указатели (гиперссылки) на контент, хранящийся на одном из хостов в интернете. Например, HTML-строка

```
<a href = "http: //www.cmu.edu/inciex.html">Carnegie Mellon</a>
```

представляет собой команду выделения текстового объекта Carnegie Mellon и создания гиперссылки на HTML-файл с именем `index.html`, находящийся на веб-сервере университета Карнеги–Меллона (Carnegie-Melone University, CMU). Если пользователь щелкнет на выделенном текстовом объекте, то браузер затребует соответствующий файл HTML с сервера CMU и отобразит его на экране.

### Происхождение Всемирной паутины

Всемирная паутина (World Wide Web) была изобретена Тимом Бернерсом-Ли (Tim Berners-Lee), специалистом по программному обеспечению в швейцарской физической лаборатории Европейский центр ядерных исследований (European Center for Nuclear Research, CERN). В 1989 г. Бернерс-Ли опубликовал в международных научных изданиях заметку с предложением создания распределенной гипертекстовой системы, которая соединяла бы сеть узлов линиями передачи данных. Назначение предложенной системы заключалось в том, чтобы помочь ученым CERN совместно использовать и управлять информацией. Через два с небольшим года после того, как Бернерс-Ли реализовал первый веб-сервер и веб-браузер, была создана небольшая сеть, охватывающая серверы CERN и несколько других сайтов. В 1993 г. произошло поворотное событие, когда Марк Андрессен (Marc Andreessen; впоследствии основавший компанию Netscape) и его коллеги из Национального центра по применению суперкомпьютеров (National Center for Supercomputing Applications, NCSA) разработали и внедрили графический браузер MOSAIC для всех трех основных платформ: Linux, Windows и Macintosh. С появлением браузера MOSAIC интерес к вебу приобрел характер взрыва, при этом число веб-сайтов ежегодно увеличивалось в 10 и более раз. К 2015 году в мире существовало более 975 млн сайтов (по данным отчета Netcraft Web Survey).

## 11.5.2. Веб-контент

Для веб-клиентов и веб-серверов *контент* (содержимое) представляет собой последовательность байтов со связанным с ними типом *MIME* (Multipurpose Internet Mail Extensions – многоцелевые расширения электронной почты интернета). В табл. 11.1 показаны некоторые общеупотребительные типы MIME.

Веб-серверы формируют контент для клиентов двумя разными способами:

- читают дисковый файл и возвращают его содержимое клиенту. Дисковый файл – это пример *статического контента*, а процесс доставки файла клиенту называется *обслуживанием статического контента*;

Таблица 11.1. Примеры типов MIME

Тип MIME	Описание
text/html	HTML-страница
text/plain	Неформатированный текст
application/postscript	Документ Postscript
image/gif	Двоичные данные, изображение в формате GIF
image/png	Двоичные данные, изображение в формате PNG
image/jpeg	Двоичные данные, изображение в формате JPEG

- запускают выполняемый файл и возвращают его вывод клиенту. Вывод, произведенный выполняемым файлом, называют *динамическим контентом*, а процесс выполнения программы и доставки ее вывода клиенту – *обслуживанием динамического контента*.

Каждый элемент контента, возвращаемый веб-сервером, ассоциируется с некоторым файлом, которым управляет этим контентом. Каждый из этих файлов имеет уникальное имя – унифицированный указатель информационного ресурса (Universal Resource Locator, URL). Например, URL

`http://www.google.com:80/index.html`

идентифицирует HTML-файл с именем `/index.html` на хосте `www.google.com`, который управляется веб-сервером, прослушивающим порт 80. Номер порта можно не указывать, потому что по умолчанию используется хорошо известный порт 80, присвоенный службе HTTP. Указатели URL исполняемых файлов могут включать аргументы после имени файла. Символ `?` отделяет имя файла от аргументов, а между собой аргументы отделяются символом `&`. Например, URL

`http://bluefish.ics.cs.cmu.edu:8000/cgi-bin/adder?15000&213`

идентифицирует выполняемый файл с именем `/cgi-bin/adder`, который вызывается с двумя строковыми аргументами: `15000` и `213`. Клиенты и серверы используют разные части URL во время обработки транзакции. Например, клиент использует префикс

`http://www.google.com:80`

чтобы определить сервер, с которым требуется устанавливать соединение: где находится сервер и какой порт он прослушивает. Сервер использует суффикс

`/index.html`

для поиска файла в своей файловой системе и чтобы определить тип запрошенного контента: статический или динамический.

Выделим несколько моментов, которые необходимо понимать, чтобы правильно интерпретировать суффиксы URL:

- не существует стандартных правил определения типа контента, на который ссылается URL. У каждого сервера свои правила работы с файлами. Обычно разделение контента по типам производится путем определения набора каталогов, таких как `cgi-bin`, в которых должны храниться все выполняемые файлы;
- начальная косая черта `/` в суффиксе означает *не* корневой каталог Linux, а начальный каталог для любого вида запрашиваемого контента. Например, сервер

может быть настроен так, что все файлы со статическим контентом хранятся в каталоге `/usr/httpd/html`, а все файлы с динамическим контентом – в каталоге `/usr/httpd/cgi-bin`;

- минимальный суффикс в URL – это символ `/`, который все серверы интерпретируют как имя файла некоторой стандартной начальной страницы, такой как `/index.html`. Это объясняет, почему можно извлечь начальную страницу простым вводом доменного имени в адресную строку браузера: браузер добавляет отсутствующий символ `/` в конец URL и передает его серверу, который заменяет `/` некоторым стандартным именем файла.

### 11.5.3. Транзакции HTTP

Протокол HTTP основан на текстовых строках, передаваемых через соединения, поэтому для выполнения транзакций с любым веб-сервером можно воспользоваться программой TELNET операционной системы Linux. Программа TELNET практически была вытеснена более защищенной программой SSH, однако она остается весьма полезной при отладке серверов, которые общаются с клиентами, передавая текстовые строки. В листинге 11.10 показан сеанс использования TELNET для запроса домашней страницы веб-сервера AOL.

**Листинг 11.10.** Пример взаимодействия с сервером HTTP, обслуживающим статический контент

1 linux> telnet www.aol.com 80	Клиент: открывает соединение с сервером
2 Trying 205.188.146.23...	Telnet выводит 3 строки в терминал
3 Connected to aol.com.	
4 Escape character is '^'].	
5 GET / HTTP/1.1	Клиент: строка запроса
6 Host: www.aol.com	Клиент: заголовок запроса HTTP/1.1
7	Клиент: пустая строка завершает заголовки
8 HTTP/1.0 200 OK	Сервер: строка ответа
9 MIME-Version: 1.0	Сервер: и пять заголовков ответа
10 Date: Mon, 8 Jan 2010 4:59:42 GMT	
11 Server: Apache-Coyote/1.1	
12 Content-Type: text/html	Сервер: тело ответа – это разметка HTML
13 Content-Length: 42092	Сервер: размер тела ответа -- 42092 байта
14	Сервер: пустая строка завершает заголовки
15 <html>	Сервер: первая строка HTML в теле ответа
16 ...	Сервер: 766 строк HTML в теле ответа (не показаны)
17 </html>	Сервер: последняя строка HTML в теле ответа
18 Connection closed by foreign host.	Сервер: закрывает соединение
19 linux>	Клиент: закрывает соединение и завершается

В строке 1 мы запускаем программу TELNET в командной оболочке Linux и требуем установить соединение с веб-сервером AOL. Программа TELNET выводит в терминал три строки, открывает соединение, а затем ждет, когда мы введем текст (строка 5). Каждый раз, когда мы вводим текстовую строку и нажимаем клавишу **Enter**, программа TELNET читает эту строку, добавляет в конец символы перевода строки (`\r\n`) и отправляет строку серверу. Это соответствует стандарту HTTP, который требует, чтобы каждая строка заканчивалась парой символов возврата каретки и перевода строки. Чтобы инициировать транзакцию, мы вводим HTTP-запрос (строки 5–7). Сервер возвращает HTTP-ответ (строки 8–17) и закрывает соединение (строка 18).

## Запросы HTTP

HTTP-запрос состоит из *строки запроса* (строка 5), за которой может следовать несколько *заголовков запроса* (строка 6), завершающихся пустой текстовой строкой (строка 7). Строка запроса имеет форму:

*метод URI версия*

HTTP имеет несколько разных методов, в том числе GET, POST, OPTIONS, HEAD, PUT, DELETE и TRACE. Мы рассмотрим только наиболее часто используемый метод GET, который применяется в подавляющем большинстве HTTP-запросов. Метод GET требует от сервера сгенерировать и вернуть контент, соответствующий *универсальному идентификатору ресурса* (Uniform Resource Identifier, URI). Идентификатор URI – это суффикс из URL, включающий имя файла и необязательные аргументы<sup>3</sup>.

Поле *версия* в строке запроса сообщает версию HTTP, которой соответствует запрос. Самая последняя версия – HTTP/1.1 [37]. Ей предшествовала более простая версия HTTP/1.0, появившаяся на свет в 1996 г. Версия HTTP/1.1 определяет дополнительные заголовки для поддержки дополнительных возможностей, таких как кэширование и безопасность, а также механизм, позволяющий клиенту и серверу выполнять многочисленные транзакции через одно и то же соединение. На практике две указанные выше версии совместимы, потому что клиенты и серверы, поддерживающие HTTP/1.0, просто игнорируют неизвестные им заголовки HTTP/1.1.

Запрос в строке 5 требует от сервера отыскать и вернуть HTML-файл `/index.html`. Он также сообщает серверу, что остальная часть запроса будет представлена в формате HTTP/1.1.

Заголовки запросов несут дополнительную информацию серверу, такую как название браузера или тип MIME, который тот готов принять. В общем случае каждый заголовок имеет следующий формат:

*Имя-заголовок: данные-заголовок*

Нам пока достаточно знакомства только с одним заголовком – `Host` (строка 6), который обязательно должен присутствовать в запросах HTTP/1.1, но не требуется в запросах HTTP/1.0. Заголовок `Host` используется *прокси-кешами*, служащими посредниками между браузером и сервером, где находится требуемый контент. Между клиентом и сервером может находиться множество прокси-серверов. Данные в заголовке `Host`, который идентифицирует доменное имя оригинального сервера, позволяют прокси-серверу определить кэшированную у себя копию запрошенного контента.

Но вернемся к нашему примеру в листинге 11.10. Пустая строка, завершающая заголовки в строке 7 (получившаяся в результате нажатия клавиши **Enter**), дает серверу команду вернуть запрошенный файл HTML.

## Ответы HTTP

HTTP-ответы подобны HTTP-запросам. HTTP-ответ состоит из строки ответа (строка 8 в листинге 11.10), за которой могут следовать несколько заголовков ответа (строки 9–13), завершающиеся пустой строкой (строка 14), и тело ответа (строки 15–17). Строка ответа имеет вид:

*версия код-состояния сообщение-о-состоянии*

Поле *версия* сообщает версию HTTP, которой соответствует ответ. *Код состояния* – это положительное целое трехзначное число, сообщающее результат обработки запроса.

<sup>3</sup> На самом деле это верно, только когда контент запрашивается браузером. Если контент запрашивает прокси-сервер, то URI должен быть полным URL.

Сообщение о состоянии описывает код состояния. В табл. 11.2 перечислены некоторые наиболее часто встречающиеся коды состояния и соответствующие им сообщения.

**Таблица 11.2.** Некоторые коды состояний HTTP

Код состояния	Сообщение о состоянии	Описание
200	OK («хорошо»)	Запрос обработан без ошибок
301	Moved permanently («перемещено навсегда»)	Контент перемещен на хост, имя которого указано в заголовке «Location» ответа
400	Bad request («неправильный, некорректный запрос»)	Запрос не был понят сервером
403	Forbidden («запрещено (не уполномочен)»)	У сервера недостаточно привилегий для доступа к запрошенному файлу
404	Not found («не найдено»)	Сервер не нашел запрошенный файл
501	Not implemented («не реализовано»)	Сервер не поддерживает метод, указанный в запросе
505	HTTP version not supported («версия HTTP не поддерживается»)	Сервер не поддерживает версию HTTP, указанную в запросе

Заголовки ответа в строках 9–13 (листинг 11.10) несут дополнительную информацию об ответе. Для нас самыми важными заголовками являются Content-Type (тип содержимого; строка 12), который сообщает клиенту тип MIME содержимого в теле ответа, и Content-Length (длина содержимого; строка 13), сообщающий размер тела ответа в байтах.

За пустой строкой, завершающей заголовки (в строке 14), следует тело ответа, которое содержит требуемый контент.

#### 11.5.4. Обслуживание динамического контента

Если задуматься о том, как сервер генерирует динамический контент для клиента, возникают определенные вопросы. Например, как клиент передает серверу аргументы? Как сервер передает эти аргументы дочернему процессу? Как сервер передает другую информацию дочернему процессу, которая может ему понадобиться для создания затребованного контента? Куда дочерний процесс посылает результаты? На все эти вопросы дает ответы фактический стандарт, получивший название *общего шлюзового интерфейса* (Common Gateway Interface, CGI).

##### Как клиент передает аргументы серверу?

При выполнении запроса GET аргументы передаются в URI. Как мы уже знаем, символ ? отделяет имя файла от аргументов, а каждый аргумент отделяется от остальных символом &. Пробелы в аргументах недопустимы и должны быть представлены в виде строк %20. Подобные коды существуют и для других специальных символов.

##### Как сервер передает аргументы дочернему процессу?

Получив такой запрос

```
GET /cgi-bin/adder?15000&213 HTTP/1.1
```

сервер вызывает функцию fork и запускает дочерний процесс, который затем вызывает функцию execve, чтобы запустить программу /cgi-bin/adder. Программы, такие как

adder, часто называют *программами CGI* в силу того, что они подчиняются правилам стандарта CGI. Прежде чем обратиться к функции `execve`, дочерний процесс присваивает переменной `QUERY_STRING` окружения CGI значение `15000&213`, на которое программа `adder` может ссылаться во время выполнения с помощью функции `getenv`.

### Как сервер передает дочернему процессу другую информацию?

Стандарт CGI определяет ряд других переменных окружения, которые программа CGI может устанавливать во время выполнения. Некоторые из этих переменных перечислены в табл. 11.3.

**Таблица 11.3.** Примеры переменных окружения CGI

Переменная окружения	Описание
<code>QUERY_STRING</code>	Аргументы программы
<code>SERVER_PORT</code>	Порт, прослушиваемый родителем
<code>REQUEST_METHOD</code>	GET или POST
<code>REMOTE_HOST</code>	Доменное имя клиента
<code>REMOTE_ADDR</code>	Десятично-точечный IP-адрес клиента
<code>CONTENT_TYPE</code>	Только для POST: тип MIME тела запроса
<code>CONTENT_LENGTH</code>	Только для POST: размер в байтах тела запроса

### Куда дочерний процесс отправляет свой вывод?

Программа CGI выводит сгенерированный контент в стандартный вывод. Прежде чем дочерний процесс загрузит и выполнит программу CGI, он вызывает функцию `dup2`, чтобы переадресовать стандартный вывод в дескриптор, связанный с клиентом. То есть все, что программа CGI выводит в стандартный вывод, передается непосредственно клиенту.

Обратите внимание, что родительский процесс не знает ни типа, ни размера содержимого, которое генерирует дочерний процесс, поэтому вся ответственность за создание заголовков ответа `Content-type` и `Content-length`, а также за вывод пустой строки, завершающей раздел заголовков, возлагается на дочерний процесс.

В листинге 11.11 показана простая программа CGI, складывающая два аргумента и возвращающая клиенту файл HTML с суммой. В листинге 11.12 показана расшифровка транзакции HTTP, соответствующей обслуживанию динамического контента программой `adder`.

**Листинг 11.11.** Программа CGI, складывающая два аргумента

`code/netp/tiny/cgi-bin/adder.c`

```

1 #include "csapp.h"
2
3 int main(void) {
4     char *buf, *p;
5     char arg1[MAXLINE], arg2[MAXLINE], content[MAXLINE];
6     int n1=0, n2=0;
7
8     /* Извлечь аргументы */
9     if ((buf = getenv("QUERY_STRING")) != NULL) {
10         p = strchr(buf, '&');
11         *p = '\0';

```



```

12     strcpy(arg1, buf);
13     strcpy(arg2, p+1);
14     n1 = atoi(arg1);
15     n2 = atoi(arg2);
16 }
17
18 /* Сконструировать тело ответа */
19 sprintf(content, "QUERY_STRING=%s", buf);
20 sprintf(content, "Welcome to add.com: ");
21 sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
22 sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
23         content, n1, n2, n1 + n2);
24 sprintf(content, "%sThanks for visiting!\r\n", content);
25
26 /* Сгенерировать HTTP-ответ */
27 printf("Connection: close\r\n");
28 printf("Content-length: %d\r\n", (int)strlen(content));
29 printf("Content-type: text/html\r\n\r\n");
30 printf("%s", content);
31 fflush(stdout);
32
33 exit(0);
34 }

```

*code/netp/tiny/cgi-bin/adder.c*

#### Листинг 11.12. Расшифровка транзакции HTTP, соответствующей обслуживанию динамического контента

```

1  linux> telnet kittyhawk.cmcl.cs.cmu.edu 8000 Клиент: открывает соединение
2  Trying 128.2.194.242...
3  Connected to kittyhawk.cmcl.cs.cmu.edu.
4  Escape character is '^]'.
5  GET /cgi-bin/adder?15000&213 HTTP/1.0 Клиент: строка запроса
6  Клиент: пустая строка завершает заголовки
7  HTTP/1.0 200 OK Сервер: строка ответа
8  Server: Tiny Web Server Сервер: идентификатор сервера
9  Content-length: 115 Adder: ожидается 115 байт в теле ответа
10 Content-type: text/html Adder: тело ответа в формате HTML
11 Adder: пустая строка завершает заголовки
12 Welcome to add.com: THE Internet addition portal. Adder: первая строка HTML
13 <p>The answer is: 15000 + 213 = 15213 Adder: вторая строка HTML в теле ответа
14 <p>Thanks for visiting! Adder: третья строка HTML в теле ответа
15 Connection closed by foreign host. Сервер: закрывает соединение
16 linux> Клиент: закрывает соединение и завершается

```

#### Упражнение 11.5 (решение в конце главы)

В разделе 10.11 мы предупреждали вас об опасностях использования стандартных функций ввода/вывода языка C в сетевых приложениях. В то же время программа CGI, представленная в листинге 11.11, может без всяких проблем использовать стандартный ввод/вывод. Почему?

**Передача аргументов в запросах HTTP POST**

Чтобы получить тело запроса HTTP POST, дочерний процесс должен также переадресовать стандартный ввод в дескриптор сокета, связанного с клиентом. После этого программа может прочитать аргументы из тела запроса, обратившись к стандартному вводу.

## 11.6. Все вместе: разработка небольшого веб-сервера TINY

В завершение обсуждения сетевого программирования мы разработаем небольшой веб-сервер TINY (крохотный). Сервер TINY – довольно интересная программа. Она сочетает в себе многие из идей, которые мы рассмотрели выше, такие как управление процессами, ввод/вывод Unix, интерфейс сокетов, а также протокол HTTP, и при этом объем программного кода составляет всего 250 строк. И хотя он уступает в функциональности, надежности и безопасности настоящим веб-серверам, он достаточно мощный, чтобы обслуживать статический и динамический контент. Мы настоятельно рекомендуем вам изучить и опробовать его. Это довольно занимательно подключить к своему серверу с помощью браузера и наблюдать, как он отображает на экране сложную веб-страницу с текстом и графикой.

### Функция main сервера TINY

В листинге 11.13 представлена функция main сервера TINY. TINY – это итеративный сервер, прослушивающий порт, номер которого передается в аргументе командной строки. Открыв слушающий сокет вызовом `open_listenfd`, сервер TINY входит в бесконечный цикл, характерный для серверов, принимая запросы на соединение (строка 32), выполняя транзакции (строка 36) и закрывая свой конец соединения (строка 37).

#### Листинг 11.13. Веб-сервер TINY

*code/netp/tiny/tiny.c*

```

1 /*
2  * tiny.c - Простой итеративный веб-сервер HTTP/1.0, принимающий
3  * запросы GET и обслуживающий статический и динамический контент
4  */
5 #include "csapp.h"
6
7 void doit(int fd);
8 void read_requesthdrs(rio_t *rp);
9 int parse_uri(char *uri, char *filename, char *cgiargs);
10 void serve_static(int fd, char *filename, int filesize);
11 void get_filetype(char *filename, char *filetype);
12 void serve_dynamic(int fd, char *filename, char *cgiargs);
13 void clienterror(int fd, char *cause, char *errnum,
14 char *shortmsg, char *longmsg);
15
16 int main(int argc, char **argv)
17 {
18     int listenfd, connfd;
19     char hostname[MAXLINE], port[MAXLINE];
20     socklen_t clientlen;
21     struct sockaddr_storage clientaddr;
22
```

```

23  /* Проверить аргументы командной строки */
24  if (argc != 2) {
25      fprintf(stderr, "usage: %s <port>\n", argv[0]);
26      exit(1);
27  }
28
29  listenfd = Open_listenfd(argv[1]);
30  while (1) {
31      clientlen = sizeof(clientaddr);
32      connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
33      Getnameinfo((SA *) &clientaddr, clientlen, hostname, MAXLINE,
34                  port, MAXLINE, 0);
35      printf("Accepted connection from (%s, %s)\n", hostname, port);
36      doit(connfd);
37      Close(connfd);
38  }
39 }

```

*code/netp/tiny/tiny.c*

## Функция `doit`

Функция `doit`, представленная в листинге 11.14, обрабатывает одну HTTP-транзакцию. Она сначала читает и анализирует строку запроса (строки 11–14). Обратите внимание, что для чтения строки запроса здесь используется функция `rio_readlineb` (листинг 10.6).

**Листинг 11.14.** `doit` – обработчик HTTP-транзакций в TINY

*code/netp/tiny/tiny.c*

```

1 void doit(int fd)
2 {
3     int is_static;
4     struct stat sbuf;
5     char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
6     char filename[MAXLINE], cgiargs[MAXLINE];
7     rio_t rio;
8
9     /* Прочитать строку запроса и заголовки */
10    Rio_readinitb(&rio, fd);
11    Rio_readlineb(&rio, buf, MAXLINE);
12    printf("Request headers:\n");
13    printf("%s", buf);
14    sscanf(buf, "%s %s %s", method, uri, version);
15    if (strcasecmp(method, "GET")) {
16        clienterror(fd, method, "501", "Not implemented",
17                    "Tiny does not implement this method");
18        return;
19    }
20    read_requesthdrs(&rio);
21
22    /* Проанализировать URI из запроса GET */
23    is_static = parse_uri(uri, filename, cgiargs);
24    if (stat(filename, &sbuf) < 0) {
25        clienterror(fd, filename, "404", "Not found",
26                    "Tiny couldn't find this file");
27        return;
28    }

```

```

29
30     if (is_static) { /* Обслужить статический контент */
31         if (!(S_ISREG(sbuf.st_mode)) || !(S_IRUSR & sbuf.st_mode)) {
32             clienterror(fd, filename, "403", "Forbidden",
33                 "Tiny couldn't read the file");
34             return;
35         }
36         serve_static(fd, filename, sbuf.st_size);
37     }
38     else { /* Обслужить динамический контент */
39         if (!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode)) {
40             clienterror(fd, filename, "403", "Forbidden",
41                 "Tiny couldn't run the CGI program");
42             return;
43         }
44         serve_dynamic(fd, filename, cgiargs);
45     }
46 }

```

*code/netp/tiny/tiny.c*

Сервер TINY поддерживает только метод GET. Если клиент пошлет запрос с другим методом (например, POST), то сервер вернет ему сообщение об ошибке и вернется в функцию `main` (строки 15–19), которая затем закроет соединение и перейдет к ожиданию следующего запроса на соединение. В противном случае `doit` читает и (как вы увидите далее) игнорирует любые заголовки в запросе (строка 20).

Далее анализируется URI, из которого извлекается имя файла и, возможно, пустая, строка с аргументами CGI. Попутно устанавливается флаг-признак типа контента – статический или динамический (строка 23). Если файл на диске отсутствует, клиенту возвращается сообщение об ошибке и выполняется возврат в функцию `main`.

Наконец, если клиент запросил статический контент, то проверяется вид файла и наличие разрешений на его чтение (строка 31). Если все в порядке, то содержимое файла отправляется клиенту (строка 36). Аналогично, если клиент запросил динамический контент, проверяется наличие разрешения на выполнение файла (строка 39), и если разрешение имеется, то производится обслуживание динамического контента (строка 44).

## Функция `clienterror`

В сервере TINY отсутствуют многие функции обработки ошибок, которые имеются в настоящих серверах. Тем не менее он выполняет проверку на наличие некоторых очевидных ошибок и уведомляет клиентов о них. Функция `clienterror`, представленная в листинге 11.15, отправляет клиенту HTTP-ответ с соответствующим кодом состояния и сообщением в строке ответа, а также файл HTML в теле ответа, описывающий причину ошибки.

**Листинг 11.15.** `clienterror` посылает сообщение об ошибке клиенту

*code/netp/tiny/tiny.c*

```

1 void clienterror(int fd, char *cause, char *errnum,
2 char *shortmsg, char *longmsg)
3 {
4     char buf[MAXLINE], body[MAXBUF];
5
6     /* Сконструировать тело HTTP-ответа */
7     sprintf(body, "<html><title>Tiny Error</title>");

```

```

8     sprintf(body, "%s<body bgcolor=\"%ffffff\">\r\n", body);
9     sprintf(body, "%s%s: %s\r\n", body, errnum, shortmsg);
10    sprintf(body, "%s<p>%s: %s\r\n", body, longmsg, cause);
11    sprintf(body, "%s<hr><em>The Tiny Web server</em>\r\n", body);
12
13    /* Вывести HTTP-ответ */
14    sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
15    Rio_writen(fd, buf, strlen(buf));
16    sprintf(buf, "Content-type: text/html\r\n");
17    Rio_writen(fd, buf, strlen(buf));
18    sprintf(buf, "Content-length: %d\r\n\r\n", (int)strlen(body));
19    Rio_writen(fd, buf, strlen(buf));
20    Rio_writen(fd, body, strlen(body));
21 }

```

*code/netp/tiny/tiny.c*

Напомним вам, что в HTTP-ответе должен быть указан размер и тип содержимого тела ответа. Поэтому мы решили сконструировать HTML-страницу в виде единой строки, чтобы потом было проще определить ее размер. Обратите также внимание, что для вывода здесь используется надежная функция `rio_writen` (листинг 10.2).

## Функция `read_requesthdrs`

Сервер TINY не использует информацию, содержащуюся в заголовках запросов. Он просто читает ее вызовом функции `read_requesthdrs` (листинг 11.16) и игнорирует. Обратите внимание, что пустая строка, завершающая блок заголовков запроса, состоит из пары символов возврата каретки и перевода строки, наличие которой проверяет строка 6.

**Листинг 11.16.** `read_requesthdrs` читает заголовки запроса и игнорирует их

```

1 void read_requesthdrs(rio_t *rp)
2 {
3     char buf[MAXLINE];
4
5     Rio_readlineb(rp, buf, MAXLINE);
6     while(strcmp(buf, "\r\n")) {
7         Rio_readlineb(rp, buf, MAXLINE);
8         printf("%s", buf);
9     }
10    return;
11 }

```

*code/netp/tiny/tiny.c**code/netp/tiny/tiny.c*

## Функция `parse_uri`

Сервер TINY предполагает, что статический контент хранится в текущем каталоге, а выполняемые файлы, генерирующие динамический контент, – в каталоге `./cgi-bin`. То есть любой URI, содержащий строку `cgi-bin`, считается запросом динамического контента. Если имя файла не указано, используется имя по умолчанию `./home.html`.

Эту стратегию реализует функция `parse_uri` (листинг 11.17). Она выполняет синтаксический анализ URI, выделяя из него имя файла и необязательную строку аргументов CGI. Если запрашивается статический контент (строка 5), то строка аргументов CGI очищается (строка 6), после чего URI преобразуется в строку относительного пути к файлу, например `./index.html` (строки 7–8). Если URI заканчивается символом `/` (стро-

ка 9), то в конец добавляется имя файла по умолчанию (строка 10). С другой стороны, если запрашивается динамический контент (строка 13), то из URI извлекаются все имеющиеся аргументы CGI (строки 14–20), а остальная часть URI преобразуется в относительный путь к файлу (строки 21–22).

**Листинг 11.17.** `parse_uri` анализирует URI HTTP-запроса

*code/netp/tiny/tiny.c*

```

1 int parse_uri(char *uri, char *filename, char *cgiargs)
2 {
3     char *ptr;
4
5     if (!strstr(uri, "cgi-bin")) { /* Статический контент */
6         strcpy(cgiargs, "");
7         strcpy(filename, ".");
8         strcat(filename, uri);
9         if (uri[strlen(uri)-1] == '/')
10             strcat(filename, "home.html");
11         return 1;
12     }
13     else { /* Динамический контент */
14         ptr = index(uri, '?');
15         if (ptr) {
16             strcpy(cgiargs, ptr+1);
17             *ptr = '\0';
18         }
19         else
20             strcpy(cgiargs, "");
21         strcpy(filename, ".");
22         strcat(filename, uri);
23         return 0;
24     }
25 }
```

*code/netp/tiny/tiny.c*

## Функция `serve_static`

Сервер TINY обслуживает пять разных типов статического контента: файлы HTML, обычные текстовые файлы и изображения в форматах GIF, PNG и JPEG.

Функция `serve_static` (листинг 11.18) посылает HTTP-ответ с содержимым локального файла. Сначала функция определяет тип контента по расширению файла (строка 7), а затем посылает строку ответа и заголовки клиенту (строки 8–13). Обратите внимание на то, что раздел заголовков завершает пустая строка.

Затем тело ответа посылается клиенту путем копирования содержимого файла в дескриптор `fd`. Здесь есть некоторые тонкости, поэтому рассмотрим эту операцию подробнее. Строка 18 открывает файл `filename` для чтения и получает дескриптор. В строке 19 вызовом `mmap` файл отображается в виртуальную память. Вспомните, как рассказывалось в разделе 9.8, вызов функции `mmap` отображает первые `filesize` байт из файла `srcfd` в приватную область виртуальной памяти, которая начинается с адреса `srcp`.

**Листинг 11.18.** `serve_static` посылает клиенту статический контент

*code/netp/tiny/tiny.c*

```

1 void serve_static(int fd, char *filename, int filesize)
2 {
3     int srcfd;
```

```

4   char *srcp, filetype[MAXLINE], buf[MAXBUF];
5
6   /* Послать заголовки ответа клиенту */
7   get_filetype(filename, filetype);
8   sprintf(buf, "HTTP/1.0 200 OK\r\n");
9   sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
10  sprintf(buf, "%sConnection: close\r\n", buf);
11  sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
12  sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
13  Rio_writen(fd, buf, strlen(buf));
14  printf("Response headers:\n");
15  printf("%s", buf);
16
17  /* Послать тело ответа клиенту */
18  srcfd = Open(filename, O_RDONLY, 0);
19  srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
20  Close(srcfd);
21  Rio_writen(fd, srcp, filesize);
22  Munmap(srcp, filesize);
23 }
24
25 /*
26  * get_filetype - определяет тип файла по расширению
27  */
28 void get_filetype(char *filename, char *filetype)
29 {
30     if (strstr(filename, ".html"))
31         strcpy(filetype, "text/html");
32     else if (strstr(filename, ".gif"))
33         strcpy(filetype, "image/gif");
34     else if (strstr(filename, ".png"))
35         strcpy(filetype, "image/png");
36     else if (strstr(filename, ".jpg"))
37         strcpy(filetype, "image/jpeg");
38     else
39         strcpy(filetype, "text/plain");
40 }

```

*code/netp/tiny/tiny.c*

После отображения файла в память он становится ненужным, поэтому закрывается (строка 20). Если этого не сделать, возникает риск фатальной утечки памяти. Строка 21 выполняет фактическую отправку файла клиенту. Функция `rio_writen` копирует `filesize` байт, начиная с адреса `srcp` в памяти (куда отображен файл), в дескриптор соединения с клиентом. И наконец, строка 22 освобождает область виртуальной памяти, отведенной для файла. Это необходимо, чтобы избежать возможной фатальной утечки памяти.

## Функция `serve_dynamic`

Сервер TINY поддерживает обслуживание динамического контента любого типа, создавая дочерний процесс, который затем запускает программу CGI. Функция `serve_dynamic` (листинг 11.19) начинается с отправки строки ответа, показывающей клиенту, что все идет нормально, с информационным заголовком `Server`. В обязанности программы CGI входит отправка остальной части ответа. Обратите внимание, что эта операция не так надежна, как хотелось бы, потому что не предусматривает обработку ошибок, которые могут возникнуть в процессе выполнения программ CGI.

**Обработка преждевременного закрытия соединений**

Несмотря на то что базовые функции веб-сервера достаточно просты, мы не хотим вызывать у вас ложное чувство, что написание настоящего сервера – пустяковое дело. Создание надежного веб-сервера, способного безаварийно действовать в течение долгого времени, – трудная задача, требующая глубокого понимания особенностей программирования в Linux. Например, если сервер выполняет последовательность операций записи в соединение, уже закрытое клиентом (например, если пользователь щелкнул на кнопке **Stop** (Остановить)), то первая из таких операций записи выполнится успешно, но вторая и последующие операции повлекут передачу процессу сервера сигнала SIGPIPE, стандартное поведение которого – завершение процесса. Если процесс настроил обработку или игнорирование сигнала SIGPIPE, то вторая и последующие операции записи будут возвращать –1 со значением EPIPE в переменной `errno`. Функции `strerr` и `pergr` возвращают для ошибки EPIPE сообщение «Broken pipe» (разрыв канала) – настолько неинформативное, что ставило в тупик не одно поколение студентов. То есть надежный сервер должен перехватывать сигнал SIGPIPE и проверять, не вернул ли вызов функции `write` ошибку EPIPE.

**Листинг 11.19.** `serve_dynamic` посылает клиенту динамический контент*code/netp/tiny/tiny.c*

```

1 void serve_dynamic(int fd, char *filename, char *cgiargs)
2 {
3     char buf[MAXLINE], *emptylist[] = { NULL };
4
5     /* Отправить первую часть HTTP-ответа */
6     sprintf(buf, "HTTP/1.0 200 OK\r\n");
7     Rio_writen(fd, buf, strlen(buf));
8     sprintf(buf, "Server: Tiny Web Server\r\n");
9     Rio_writen(fd, buf, strlen(buf));
10
11     if (Fork() == 0) { /* Потомок */
12         /* Настоящий сервер в этом месте устанавливает переменные CGI */
13         setenv("QUERY_STRING", cgiargs, 1);
14         /* Переадресовать stdout в дескриптор соединения с клиентом */
15         Dup2(fd, STDOUT_FILENO);
16         Execve(filename, emptylist, environ); /* Запустить CGI-программу */
17     }
18     Wait(NULL); /* Родитель ждет, чтобы утилизировать дочерний процесс */
19 }
```

*code/netp/tiny/tiny.c*

После отправки первой части ответа создается новый дочерний процесс (строка 11). Он инициализирует переменную окружения `QUERY_STRING` аргументами CGI из запроса идентификатора URI (строка 13). Обратите внимание, что в этом месте настоящие серверы устанавливают также другие переменные окружения CGI, но мы для экономии места опускаем эти действия.

Далее дочерний переадресует свой стандартный вывод в дескриптор соединения (строка 15), а затем загружает и запускает программу CGI (строка 16). Поскольку программа CGI выполняется в контексте дочернего процесса, она имеет доступ ко всем открытым файлам и переменным окружения, которые был доступны до вызова функции `execve`. Следовательно, все, что программа CGI записывает в стандартный вывод, будет немедленно отправляться клиентскому процессу без всякого вмешательства родитель-



ского процесса. В то же время родительский процесс блокируется в функции `wait`, дожидаясь завершения процесса-потомка, чтобы утилизировать его (строка 18).

## 11.7. Итоги

В основу каждого сетевого приложения положена модель клиент–сервер. В соответствии с этой моделью приложение состоит из сервера и одного или нескольких клиентов. Сервер управляет ресурсами, предоставляя услуги своим клиентам, для чего необходимо тем или иным способом манипулировать ресурсами. Базовой операцией модели клиент–сервер является *транзакция*, предусматривающая отправку запроса клиентом, за которым следует ответ сервера.

Клиент и серверы обмениваются данными через всемирную сеть, известную как интернет. С точки зрения программиста интернет можно рассматривать как коллекцию хостов, разбросанных по всему миру и обладающих следующими свойствами: (1) каждый хост имеет уникальный 32-разрядный адрес, который называют IP-адресом; (2) множество IP-адресов отображается в множество доменных имен интернета; (3) процессы на разных хостах в интернете могут обмениваться данными через установленные соединения.

Клиенты и серверы устанавливают соединения с помощью интерфейса сокетов. Сокет – это конечная точка соединения, которая в приложении представлена в форме дескриптора файла. Интерфейс сокетов предоставляет функции для открытия и закрытия дескрипторов. Клиенты и серверы обмениваются данными, записывая и читая содержимое этих дескрипторов.

Веб-серверы и их клиенты (такие как браузеры) обмениваются данными посредством протокола HTTP. Браузер запрашивает у сервера статический или динамический контент. Запрос статического контента обслуживается путем извлечения файлов с диска сервера и доставки его клиенту. Запрос динамического контента обслуживается путем выполнения программы в контексте дочернего процесса сервера и возврата его вывода клиенту. Стандарт CGI формулирует множество правил, регламентирующих передачу клиентами аргументов серверу, передачу этих аргументов и другой информации дочернему процессу и передачу вывода дочернего процесса обратно клиенту. Простой, но вместе с тем нормально функционирующий процесс, который обслуживает и статический, и динамический контент, можно реализовать несколькими сотнями строк программного кода на языке C.

## Библиографические заметки

Официальный источник информации об интернете содержится во множестве бесплатно распространяемых нумерованных документов, известных как *запросы на комментарии* (Requests for Comments, RFC). Каталог документов RFC с поддержкой поиска доступен по адресу:

<http://rfc-editor.org/rfc.html>

Документы RFC обычно написаны для разработчиков инфраструктуры интернета и часто изобилуют многочисленными деталями, не представляющими интереса для случайного читателя. В то же время не существует более авторитетного источника информации. Протокол HTTP/1.1 задокументирован в RFC 2616. Обязательный для поддержки список типов MIME доступен по адресу:

<http://www.iana.org/assignments/media-types>

Книга Керриска (Kerrisk) – это библия по всем аспектам программирования в Linux, где подробно рассказывается о современном сетевом программировании [62]. Сущест-

вует несколько хороших публикаций общего назначения по организации компьютерных сетей [65, 84, 114]. Великий технический писатель У. Ричард Стивенс (W. Richard Stevens) написал серию классических книг по таким вопросам, как современное программирование в системе Unix [111], протоколы интернета [109, 120, 107] и программирование сетевых задач в системе Unix [108, 110]. Студенты, серьезно изучающие программирование в Unix-подобных системах, возможно, захотят изучить вопрос в полном объеме. К нашему величайшему сожалению, Стивенс умер 1 сентября 1999 года. Нам очень не хватает его книг.

## Домашние задания

### Упражнение 11.6 ♦♦

1. Измените сервер TINY так, чтобы он возвращал клиенту каждую строку запроса и каждый заголовок.
2. Попробуйте из своего браузера запросить статический контент у сервера TINY. Сохраните вывод сервера TINY в файл.
3. Исследуйте вывод сервера TINY и определите, какую версию протокола HTTP использует ваш браузер.
4. Изучите стандарт HTTP/1.1 по документу RFC 2616 и определите содержимое каждого заголовка HTTP-запроса, поступившего от вашего браузера. Документ RFC 2616 можно получить по адресу: [www.rfc-editor.org/rfc.html](http://www.rfc-editor.org/rfc.html).

### Упражнение 11.7 ♦♦

Расширьте сервер TINY так, чтобы он мог обслуживать видеофайлы MPG. Проверьте правильность вашего решения, воспользовавшись настоящим браузером.

### Упражнение 11.8. ♦♦

Измените сервер TINY так, чтобы он утилизировал дочерние в обработчике сигнала SIGCHLD, а не ждал их завершения.

### Упражнение 11.9 ♦♦

Измените сервер TINY так, чтобы при обслуживании статического контента он копировал затребованный файл в подключенный дескриптор, используя функции `malloc`, `rio_readn` и `rio_writen` вместо `mmap` и `rio_writen`.

### Упражнение 11.10 ♦♦

1. Напишите форму HTML для CGI-функции `adder` из листинга 11.11. Форма должна включать два текстовых поля ввода, в которые пользователи могут ввести два числа, чтобы получить их сумму. Форма должна отправлять содержимое полей, используя метод GET.
2. Проверьте, правильно ли работает ваша программа, воспользовавшись настоящим браузером, чтобы запросить форму у сервера TINY, заполнить ее и отправить обратно серверу TINY, а затем отобразить на экране динамический контент, вычисленный программой `adder`.

### Упражнение 11.11 ♦♦

Расширьте сервер TINY, добавив в него поддержку метода HTTP HEAD. Проверьте, правильно ли работает ваш сервер, воспользовавшись для этой цели протоколом TELNET.

### Упражнение 11.12 ♦♦♦

Расширьте сервер TINY, добавив в него способность обслуживать запрос динамического контента методом HTTP POST. Проверьте работу вашего сервера, воспользовавшись для этой цели веб-браузером.

### Упражнение 11.13 ♦♦♦

Измените сервер TINY так, чтобы он надежно (без аварийного завершения) обрабатывал сигнал SIGPIPE и ошибку EPIPE, которые возникают, когда функция write пытается выполнить запись в преждевременно закрытое соединение.

## Решения упражнений

### Решение упражнения 11.1

Шестнадцатеричный адрес	Десятично-точечный адрес
0x0	0.0.00
0xffffffff	255.255.255.255
0x7f000001	127.0.0.1
0xcdbca079	205.188.160.121
0x400c950d	64.12.149.13
0xcdbc9217	205.188.146.23

### Решение упражнения 11.2

*code/netp/hex2dd.c*

```

1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     struct in_addr inaddr; /* Адрес с сетевым порядком байтов */
6     uint32_t addr;         /* Адрес с аппаратным порядком байтов */
7     char buf[MAXBUF];      /* Буфер для строки с представлением адреса */
8
9     if (argc != 2) {
10         fprintf(stderr, "usage: %s <hex number>\n", argv[0]);
11         exit(0);
12     }
13     sscanf(argv[1], "%x", &addr);
14     inaddr.s_addr = htonl(addr);
15
16     if (!inet_ntop(AF_INET, &inaddr, buf, MAXBUF))
17         unix_error("inet_ntop");
18     printf("%s\n", buf);
19
20     exit(0);
21 }
```

*code/netp/hex2dd.c*

### Решение упражнения 11.3

*code/netp/dd2hex.c*

```

1 #include "csapp.h"
2
3 int main(int argc, char **argv)
```

```

4 {
5     struct in_addr inaddr; /* Адрес с сетевым порядком байтов */
6     int rc;
7
8     if (argc != 2) {
9         fprintf(stderr, "usage: %s <dotted-decimal>\n", argv[0]);
10        exit(0);
11    }
12
13    rc = inet_pton(AF_INET, argv[1], &inaddr);
14    if (rc == 0)
15        app_error("inet_pton error: invalid dotted-decimal address");
16    else if (rc < 0)
17        unix_error("inet_pton error");
18
19    printf("0x%x\n", ntohl(inaddr.s_addr));
20    exit(0);
21 }

```

*code/netp/dd2hex.c*

## Решение упражнения 11.4

В следующем решении обратите внимание, насколько сложнее использовать функцию `inet_ntop`, требующую приведения типов и многоуровневых ссылок на вложенные элементы структуры. Функция `getnameinfo` намного проще, потому что всю эту работу она выполняет автоматически.

*code/netp/hostinfo-ntop.c*

```

1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     struct addrinfo *p, *listp, hints;
6     struct sockaddr_in *sockp;
7     char buf[MAXLINE];
8     int rc;
9
10    if (argc != 2) {
11        fprintf(stderr, "usage: %s <domain name>\n", argv[0]);
12        exit(0);
13    }
14
15    /* Получить список структур addrinfo */
16    memset(&hints, 0, sizeof(struct addrinfo));
17    hints.ai_family = AF_INET; /* Только IPv4 */
18    hints.ai_socktype = SOCK_STREAM; /* Только для постоянных соединений */
19    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
20        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
21        exit(1);
22    }
23
24    /* Обход списка и вывод IP-адресов */
25    for (p = listp; p; p = p->ai_next) {
26        sockp = (struct sockaddr_in *)p->ai_addr;
27        Inet_ntop(AF_INET, &(sockp->sin_addr), buf, MAXLINE);
28        printf("%s\n", buf);
29    }
30 }

```

```
31     /* Освобождение ресурсов */
32     Freeaddrinfo(listp);
33
34     exit(0);
35 }
```

---

*code/netp/hostinfo-ntop.c*

### Решение упражнения 11.5

Причина, почему стандартные функции ввода/вывода можно без опаски использовать в программах CGI, заключается в том, что программы CGI выполняются в дочерних процессах и им не нужно явно закрывать любой из потоков ввода/вывода. Когда дочерний процесс прекращает работу, ядро автоматически закрывает все дескрипторы.

# Глава 12

## Конкурентное программирование

- 12.1. Конкурентное программирование с процессами.
  - 12.2. Конкурентное программирование с мультиплексированием ввода/вывода.
  - 12.3. Конкурентное программирование с потоками выполнения.
  - 12.4. Совместное использование переменных несколькими потоками выполнения.
  - 12.5. Синхронизация потоков выполнения с помощью семафоров.
  - 12.6. Использование потоков выполнения для организации параллельной обработки.
  - 12.7. Другие вопросы конкурентного выполнения.
  - 12.8. Итоги.
- Библиографические заметки.
- Домашние задания.
- Решения упражнений.

Как рассказывалось в главе 8, логические потоки управления называются *конкурентными*, если они перекрываются во времени. Это явление, называемое *конкурентным выполнением*, или *конкуренцией*, проявляется на многих уровнях любой компьютерной системы. В число знакомых примеров входят обработчики аппаратных исключений, процессы и обработчики сигналов Linux.

До сих пор конкуренция рассматривалась нами в основном как механизм, используемый ядром для одновременного выполнения множества прикладных программ. Однако конкуренция не ограничивается ядром. Она может играть важную роль в самих прикладных программах. Например, мы уже видели, как обработчики сигналов Linux позволяют приложениям реагировать на асинхронные события, такие как ввод пользователем **Ctrl+C** или обращение программы к произвольной области виртуальной памяти. Конкуренция на уровне приложения также может принести пользу в других областях:

- *доступ к медленным устройствам ввода/вывода.* Когда приложение ожидает поступления данных с медленного устройства ввода/вывода, например с дискового накопителя, то ядро загружает процессор другими задачами. Аналогично отдельные приложения могут использовать конкуренцию чередованием полезных действий с запросами ввода/вывода;

- *взаимодействие с пользователями.* Пользователям необходима возможность одновременного выполнения задач (многозадачность). Например, может возникнуть потребность изменить размер окна, пока приложение выводит документ на печать. В современных оконных системах для этой цели используется поддержка конкурентного выполнения. Всякий раз, когда пользователь запрашивает некоторую операцию (например, щелкает кнопкой мыши), для ее выполнения создается отдельный поток управления;
- *сокращение задержек путем откладывания выполнения.* Иногда приложения могут использовать конкуренцию для сокращения времени задержки при выполнении определенных операций путем откладывания выполнения других операций. Например, механизм распределения динамической памяти может сократить задержки выполнения отдельных операций *free* путем их откладывания и выполнения в отдельном потоке управления, действующем с более низким приоритетом и получающем циклы процессора, когда тот не занят другой работой;
- *одновременное обслуживание нескольких сетевых клиентов.* Итеративные сетевые серверы, о которых рассказывалось в главе 12, непрактичны, потому что в каждый конкретный момент времени они могут обслуживать только одного клиента. В результате, пока обслуживается один клиент, остальные будут вынуждены ждать своей очереди. Для настоящих серверов, от которых требуется обслуживать сотни или тысячи клиентов в секунду, такое положение вещей недопустимо. Более практичное решение – реализовать сервер с использованием механизмов конкуренции, когда для обслуживания каждого клиента создается отдельный поток управления. Это позволяет серверу одновременно обслуживать множество клиентов и предотвращать узурпацию сервера одним клиентом;
- *организация параллельных вычислений на многоядерных процессорах.* Многие современные системы оснащаются многоядерными процессорами. Приложения в таких системах имеют возможность разбить большую задачу на множество подзадач, выполняемых параллельно, и за счет этого увеличить общую производительность.

Приложения, использующие конкуренцию на уровне прикладных задач, называются *конкурентными*. В современных операционных системах поддерживаются три основных механизма конкуренции:

- *процессы* – при использовании этого механизма все потоки управления организованы как отдельные процессы, которые планируются и поддерживаются ядром. Поскольку процессы имеют раздельные виртуальные адресные пространства, то для их взаимодействий между собой необходимо использовать специальные механизмы *межпроцессных взаимодействий* (Interprocess Communication, IPC);
- *мультиплексирование ввода/вывода* – разновидность механизмов конкурентного программирования, когда прикладные программы сами планируют работу своей управляющей логики в контексте одного процесса. Управляющая логика конструируется как конечный автомат, который основная программа явно переводит из одного состояния в другое, например по мере поступления данных из файловых дескрипторов. Поскольку программа выполняется в рамках единственного процесса, все потоки совместно используют одно и то же адресное пространство;
- *потоки выполнения* – логические потоки управления, выполняющиеся в контексте одного процесса и планируемые ядром. Потоки можно рассматривать как

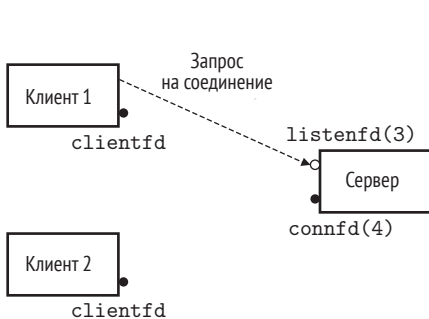
симбиоз двух вышеописанных механизмов: они планируются ядром, подобно процессам, и совместно используют общее виртуальное адресное пространство.

В этой главе мы рассмотрим все эти три механизма поддержки конкуренции. А чтобы дискуссия была максимально конкретной, мы будем работать над одним и тем же приложением – конкурентной версией нашего эхо-сервера, рассматривавшегося в разделе 11.4.9.

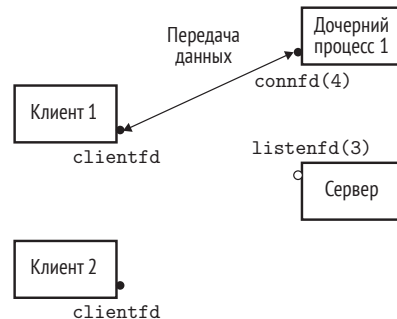
## 12.1. Конкурентное программирование с процессами

Самый простой способ реализации конкурентной программы – создание процессов с использованием знакомых функций: `fork`, `exec` и `waitpid`. Например, в конкурентном веб-сервере можно принимать запросы клиентов в родительском процессе и создавать дочерние процессы для их обработки.

Для большей наглядности предположим, что имеются два клиента и сервер, принимающий запросы через дескриптор слушающего сокета (пусть это будет дескриптор 3). Предположим также, что сервер принимает запрос от клиента 1 и возвращает дескриптор подключенного сокета (скажем, 4), как показано на рис. 12.1. После приема запроса сервер создает дочерний процесс, получающий полную копию таблицы дескрипторов сервера. Дочерний процесс закрывает свою копию слушающего дескриптора 3, а родительский процесс – свою копию подключенного дескриптора 4, потому что он ему больше не нужен. Эта ситуация показана на рис. 12.2, где дочерний процесс «занят» обслуживанием клиента.



**Рис. 12.1.** Шаг 1: сервер принимает запрос от клиента

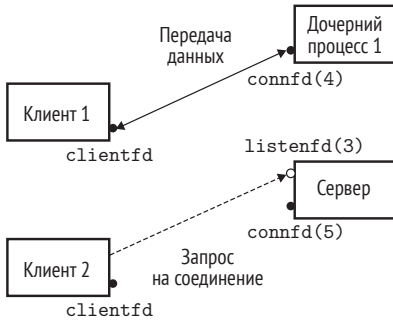


**Рис. 12.2.** Шаг 2: сервер создает дочерний процесс для обслуживания клиента

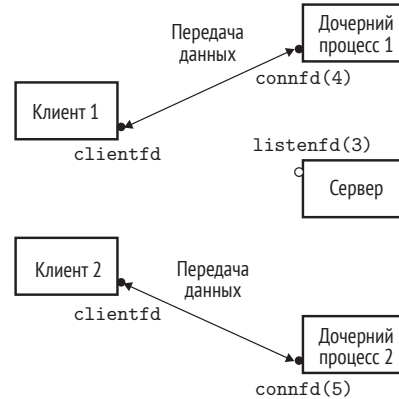
Поскольку дескрипторы подключенного сокета в родительском и дочернем процессах ссылаются на один и тот же элемент в таблице открытых файлов, родительский процесс должен закрыть свою копию дескриптора подключенного сокета. Иначе элемент в таблице открытых файлов, соответствующий дескриптору подключенного сокета 4, никогда не будет удален и образующаяся утечка памяти в конечном итоге приведет к исчерпанию доступной памяти и аварийному завершению сервера.

Теперь предположим, что после того, как родительский процесс создаст потомка для обслуживания клиента 1, он принимает новый запрос от клиента 2 и создает новый дескриптор подключенного сокета (например, 5), как показано на рис. 12.3. После этого родительский процесс создает новый дочерний процесс, который обслуживает своего клиента 2, используя дескриптор 5 подключенного сокета, как показано на рис. 12.4. После этого родительский процесс может перейти к ожиданию следующего запроса, а два созданных дочерних процесса продолжают обслуживание своих клиентов.





**Рис. 12.3.** Шаг 3: сервер принимает запрос от другого клиента



**Рис. 12.4.** Шаг 4: сервер создает еще один дочерний процесс для обслуживания другого клиента

### 12.1.1. Конкурентный сервер, основанный на процессах

В листинге 12.1 представлен код конкурентного эхо-сервера, основанного на процессах. Функция `echo`, используемая сервером в строке 29, была показана в листинге 11.9. Вот несколько интересных моментов, заслуживающих внимания:

- во-первых, серверы обычно работают довольно продолжительное время, поэтому они должны иметь обработчик сигнала `SIGCHLD`, чтобы утилизировать так называемые зомби-процессы (строки 4–9). Поскольку обработчик сигнала `SIGCHLD` блокирует работу процесса на время своего выполнения и сигналы Linux не ставятся в очередь, то обработчик `SIGCHLD` должен быть готов к утилизации нескольких зомби-процессов;
- во-вторых, родительский и дочерний процессы должны закрывать свои копии `connfd` (строки 33 и 30 соответственно). Как уже отмечалось, это особенно важно для родительского процесса, который должен закрыть свою копию дескриптора подключенного сокета, чтобы избежать утечки памяти;
- наконец, из-за особенностей работы механизма подсчета ссылок на элементы таблицы открытых файлов соединение с клиентом не будет разорвано, пока оба процесса, родитель и потомок, не закроют дескриптора подключенного сокета `connfd`.

**Листинг 12.1.** Реализация конкурентного эхо-сервера на основе процессов.  
Родитель создает дочерний процесс для обслуживания каждого нового запроса

*code/conc/echoserverp.c*

```

1 #include "csapp.h"
2 void echo(int connfd);
3
4 void sigchld_handler(int sig)
5 {
6     while (waitpid(-1, 0, WNOHANG) > 0)
7         ;
8     return;
9 }
```

```

10
11 int main(int argc, char **argv)
12 {
13     int listenfd, connfd;
14     socklen_t clientlen;
15     struct sockaddr_storage clientaddr;
16
17     if (argc != 2) {
18         fprintf(stderr, "usage: %s <port>\n", argv[0]);
19         exit(0);
20     }
21
22     Signal(SIGCHLD, sigchld_handler);
23     listenfd = Open_listenfd(argv[1]);
24     while (1) {
25         clientlen = sizeof(struct sockaddr_storage);
26         connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
27         if (Fork() == 0) {
28             Close(listenfd); /* Потомок закрывает слушающий сокет */
29             echo(connfd);    /* Потомок обслуживает клиента */
30             Close(connfd);   /* Потомок закрывает подключенный сокет */
31             exit(0);         /* Потомок завершает работу */
32         }
33         Close(connfd); /* Родитель закрывает подключенный сокет (важно!) */
34     }
35 }

```

*code/conc/echoserverp.c***Unix IPC**

Вы уже видели несколько примеров межпроцессных взаимодействий (IPC) в этой книге. Функция `waitpid` и сигналы, представленные в главе 8, – элементарные механизмы IPC, позволяющие обмениваться короткими сообщениями процессам, выполняющимся на одном хосте. Интерфейс сокетов, описанный в главе 11, – это еще одна важная форма IPC, позволяющая процессам на разных хостах обмениваться произвольными потоками байтов. Однако под термином *Unix IPC* подразумевается великое множество механизмов, позволяющих одним процессам взаимодействовать с другими, выполняющимися на одном и том же хосте. Примерами таких механизмов могут служить каналы, очереди, разделяемая память System V и семафоры System V, однако мы не будем обсуждать их, так как это выходит за рамки нашей книги. Все подробности можно найти в книге Керриска (Kerrisk) [62].

### 12.1.2. Достоинства и недостатки подхода на основе процессов

Процессы реализуют ясную модель совместного использования информации о состоянии между родителем и потомком: таблицы файлов используются совместно, а пространства адресов – нет. Изолированность адресных пространств процессов является как преимуществом, так и недостатком. Никакой процесс не сможет случайно повредить данные, принадлежащие другому процессу, что избавляет от массы досадных ошибок. Это – очевидное преимущество.

С другой стороны, изолированные адресные пространства затрудняют совместное использование процессами информации о состоянии. Для совместного использования ин-

формации они должны прибегать к механизмам межпроцессных взаимодействий. Другой недостаток подхода на основе процессов – меньшая скорость обслуживания клиентов из-за высоких непроизводительных издержек на создание новых процессов и межпроцессные взаимодействия.

### Упражнение 12.1 (решение в конце главы)

После того как родительский процесс закрывает дескриптор подключенного сокета в строке 33, дочерний процесс по-прежнему может взаимодействовать с клиентом, используя свою копию дескриптора. Почему?

### Упражнение 12.2 (решение в конце главы)

Если удалить строку 30 в листинге 12.1, где дочерний процесс закрывает дескриптор подключенного сокета, код по-прежнему оставался бы корректным в том смысле, что это не привело бы к утечке памяти. Почему?

## 12.2. Конкурентное программирование с мультиплексированием ввода/вывода

Предположим, что перед нами поставлена задача написать эхо-сервер, обрабатывающий команды пользователя в интерактивном режиме. В этом случае сервер должен откликаться на два независимых события ввода/вывода: (1) клиент посылает запрос на соединение и (2) пользователь вводит команду с клавиатуры. Какого события следует ожидать первым? Ни один вариант не является идеальным. При ожидании запроса на соединение в ассерт сервер не сможет реагировать на команды. Точно так же, ожидая ввода команды в `read`, сервер не сможет принимать новые запросы на соединение.

Одно из решений этой дилеммы – использовать так называемое *мультиплексирование ввода/вывода*. Основная идея состоит в том, чтобы вызвать функцию `select`, которая вернет управление приложению, только после появления одного или нескольких событий ввода/вывода, перечисленных ниже:

- когда любой дескриптор из множества  $\{0, 4\}$  будет готов к чтению;
- когда любой дескриптор из множества  $\{1, 2, 7\}$  будет готов к записи;
- тайм-аут, если истекли 152,13 секунды в ожидании события ввода/вывода.

Далее мы рассмотрим только первый сценарий: ожидание, когда один или несколько дескрипторов из множества будут готовы к чтению. Более подробное обсуждение этой темы вы найдете в [62, 110].

Функция `select` управляет множествами типа `fd_set`, известными как *наборы дескрипторов*. Логически набор дескрипторов рассматривается как битовая маска размера  $n$ :

$$b_{n-1}, \dots, b_1, b_0$$

Каждый бит  $b_k$  соответствует дескриптору  $k$ . Дескриптор  $k$  является членом набора дескрипторов, если  $b_k = 1$ . Для набора дескрипторов поддерживаются следующие операции: (1) размещение; (2) присваивание значения одной переменной этого типа другой; (3) изменение и проверка битов с помощью макросов `FD_ZERO`, `FD_SET`, `FD_CLR` и `FD_ISSET`.

```
#include <sys/select.h>

int select(int n, fd_set *fdset, NULL, NULL, NULL);

        Возвращает ненулевое количество дескрипторов, готовых
        к чтению, -1 в случае ошибки

FD_ZERO(fd_set *fdset);          /* Сбросить все биты в fdset */
FD_CLR(int fd, fd_set *fdset);   /* Сбросить бит для fd в fdset */
FD_SET(int fd, fd_set *fdset);   /* Установить бит для fd в fdset */
FD_ISSET(int fd, fd_set *fdset); /* Бит для fd в fdset установлен? */

        Макроопределения для управления наборами дескрипторов
```

Для наших целей в этой главе интерес представляют только два аргумента функции `select`: набор дескрипторов `fdset` (*набор чтения*) и количество элементов `n` в наборе чтения. Функция `select` блокирует процесс, пока хотя бы один дескриптор из набора не будет готов к чтению. Дескриптор `k` считается готовым к чтению, если следующая операция чтения из этого дескриптора не заблокирует процесс. Как побочный эффект функция `select` модифицирует набор `fd_set`, на который ссылается аргумент `fdset`, возвращая в нем подмножество из набора чтения, называемое *набором готовых дескрипторов*. Возвращаемое функцией значение указывает количество элементов в наборе готовых дескрипторов. Обратите внимание, что набор чтения необходимо обновлять при каждом вызове `select`.

Лучше всего изучать особенности работы `select` на конкретных примерах. В листинге 12.2 показано, как можно использовать `select` для реализации итеративного эхо-сервера, принимающего команды пользователя через стандартный ввод.

**Листинг 12.2.** Итеративный эхо-сервер, использующий прием мультиплексирования ввода/вывода. Сервер вызывает функцию `select` и ждет запроса на соединение на дескрипторе слушающего сокета и появления команды на дескрипторе стандартного ввода

*code/conc/select.c*

```
1 #include "csapp.h"
2 void echo(int connfd);
3 void command(void);
4
5 int main(int argc, char **argv)
6 {
7     int listenfd, connfd;
8     socklen_t clientlen;
9     struct sockaddr_storage clientaddr;
10    fd_set read_set, ready_set;
11
12    if (argc != 2) {
13        fprintf(stderr, "usage: %s <port>\n", argv[0]);
14        exit(0);
15    }
16    listenfd = Open_listenfd(argv[1]);
17
18    FD_ZERO(&read_set);          /* очистить набор чтения */
19    FD_SET(STDIN_FILENO, &read_set); /* Добавить stdin в набор чтения */
20    FD_SET(listenfd, &read_set);   /* Добавить listenfd в набор чтения */
21
22    while (1) {
```

```

23     ready_set = read_set;
24     Select(listenfd+1, &ready_set, NULL, NULL, NULL);
25     if (FD_ISSET(STDIN_FILENO, &ready_set))
26         command(); /* Обработать команду со стандартного ввода */
27     if (FD_ISSET(listenfd, &ready_set)) {
28         clientlen = sizeof(struct sockaddr_storage);
29         connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
30         echo(connfd); /* Вернуть введенный текст клиенту до EOF */
31         Close(connfd);
32     }
33 }
34 }
35
36 void command(void) {
37     char buf[MAXLINE];
38     if (!Fgets(buf, MAXLINE, stdin))
39         exit(0); /* EOF */
40     printf("%s", buf); /* Обработать введенную команду */
41 }

```

*code/conc/select.c*

Начнем с вызова функции `open_listenfd` (листинг 11.6), чтобы открыть слушающий сокет (строка 16), и затем используем `FD_ZERO` для создания пустого набора чтения (строка 18):

	listenfd			stdin
	3	2	1	0
read_set (0):	0	0	0	0

Далее в строках 19–20 определяется набор чтения, включающий дескрипторы 0 (стандартный ввод) и 3 (слушающий сокет):

	listenfd			stdin
	3	2	1	0
read_set ({0, 3}):	1	0	0	1

В этой точке начинается рабочий цикл сервера. Однако вместо ожидания запроса на соединение вызовом функции `accept` вызывается функция `select`, которая блокирует процесс, пока слушающий дескриптор или дескриптор стандартного ввода не будет готов к чтению (строка 24). Например, ниже приводится значение `ready_set`, которое будет возвращено функцией `select`, если пользователь нажмет клавишу **Enter**, вследствие чего дескриптор стандартного ввода станет готовым к чтению:

	listenfd			stdin
	3	2	1	0
read_set ({0}):	0	0	0	1

После возврата из `select` процесс может определить, какой из дескрипторов готов к считыванию, воспользовавшись макросом `FD_ISSET`. Если готов стандартный ввод (строка 25), то вызывается функция `command`, читающая и обрабатывающая команду. Если готов дескриптор слушающего сокета (строка 27), то вызывается функция `accept` для получения подключенного дескриптора, после чего вызывается функция `echo` (листинг 11.9), возвращающая введенный текст обратно клиенту, пока клиент не закроет соединение со своей стороны.

Несмотря на то что эта программа может служить отличным примером использования функции `select`, она не лишена недостатков. Проблема заключается в том, что после соединения с клиентом она продолжает возвращать ему введенный текст, пока клиент не закроет соединение со своей стороны. То есть если кто-то другой введет команду, сервер не сможет обработать ее, пока не завершит обслуживание клиента. Более удачное решение – производить эхо-вывод построчно, возвращаясь в главный цикл сервера.

### Упражнение 12.3 (решение в конце главы)

В системах Linux ввод комбинации **Ctrl+D** служит признаком конца файла (EOF). Что случится, если ввести **Ctrl+D** в программе в листинге 12.3, пока она заблокирована в вызове `select`?

## 12.2.1. Конкурентный на основе мультиплексирования ввода/вывода, управляемый событиями

Мультиплексирование ввода/вывода можно использовать как основу для создания конкурентных программ, *управляемых событиями*, потоки управления в которых продвигаются вперед от события к событию. Основная идея заключается в моделировании потоков управления в форме *конечных автоматов*. Говоря неформально, конечным автоматом называется набор *состояний*, *событий ввода* и *переходов*, отображающих состояния и коллекцию событий ввода в состояния. Каждый переход отображает пару (состояние ввода, событие ввода) в состояние вывода. *Петлей* в графе состояний называют переход из состояния в то же состояние. Конечные автоматы обычно изображаются в виде ориентированных графов, где узлы обозначают состояния, ребра – переходы, а метки ребер – события ввода. Конечный автомат начинает выполнение в некоем начальном состоянии. Каждое событие ввода запускает переход из текущего состояния в следующее.

Для каждого нового клиента  $k$  параллельный сервер, основанный на мультиплексировании ввода/вывода, создает новый конечный автомат  $s_k$  и ассоциирует его с дескриптором соединения  $d_k$ . Как показано на рис. 12.5, каждый конечный автомат  $s_k$  имеет одно состояние («ожидание готовности дескриптора  $d_k$  к чтению»), одно событие ввода («дескриптор  $d_k$  готов к чтению») и один переход («чтение текстовой строки из дескриптора  $d_k$ »).



**Рис. 12.5.** Конечный автомат, соответствующий потоку управления в конкурентном эхо-сервере, управляемом событиями

С помощью функции `select` сервер мультиплексирует ввод/вывод, чтобы обнаружить появление событий ввода. По мере того как каждый дескриптор сокета, соединенного с клиентом, оказывается готовым к чтению, сервер выполняет переход в соответствующем конечном автомате, читая и возвращая обратно текстовую строку.

В листинге 12.3 показан код конкурентного сервера на основе мультиплексирования ввода/вывода, управляемого событиями. Набор активных клиентов хранится в структуре `pool` (строки 3–11). После инициализации пула `pool` вызовом `init_pool` (строка 27) сервер входит в бесконечный цикл. В каждой итерации сервер вызывает функцию `select` для выявления событий ввода двух типов: запроса на соединение от нового клиента и готовности к чтению одного из дескрипторов соединения, соответствующего тому или иному клиенту. При поступлении запроса на соединение (строка 35) сервер открывает соединение (строка 37) и вызывает функцию `add_client`, чтобы добавить клиента в пул (строка 38). Наконец, когда какой-то дескриптор становится готовым для чтения, сервер вызывает функцию `check_client` и возвращает соответствующему клиенту отправленную им строку (строка 42).

**Листинг 12.3.** Конкурентный на основе мультиплексирования ввода/вывода, управляемый событиями. В каждой итерации сервер возвращает клиенту отправленную им строку, если обнаруживается дескриптор, готовый к чтению

*code/conc/echoservers.c*

```

1 #include "csapp.h"
2
3 typedef struct { /* Пул для хранения дескрипторов установленных соединений */
4     int maxfd; /* Наибольший дескриптор в read_set */
5     fd_set read_set; /* Набор всех активных дескрипторов */
6     fd_set ready_set; /* Подмножество дескрипторов, готовых к чтению */
7     int nready; /* Число дескрипторов, готовых к чтению */
8     int maxi; /* Максимальный индекс в массиве клиентов */
9     int clientfd[FD_SETSIZE]; /* Набор активных дескрипторов соединений */
10    rio_t clientrio[FD_SETSIZE]; /* Набор активных буферов для чтения */
11 } pool;
12
13 int byte_cnt = 0; /* Счетчик байтов, полученных сервером */
14
15 int main(int argc, char **argv)
16 {
17     int listenfd, connfd;
18     socklen_t clientlen;
19     struct sockaddr_storage clientaddr;
20     static pool pool;
21
22     if (argc != 2) {
23         fprintf(stderr, "usage: %s <port>\n", argv[0]);
24         exit(0);
25     }
26     listenfd = Open_listenfd(argv[1]);
27     init_pool(listenfd, &pool);
28
29     while (1) {
30         /* Ждать готовности некоторого дескриптора */
31         pool.ready_set = pool.read_set;
32         pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
33
34         /* Если готов дескриптор слушающего сокета, добавить клиента в пул */
35         if (FD_ISSET(listenfd, &pool.ready_set)) {
36             clientlen = sizeof(struct sockaddr_storage);
37             connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
38             add_client(connfd, &pool);
39         }
40     }

```

```

41      /* Вернуть текст, полученный через дескриптор сокета соединения */
42      check_clients(&pool);
43  }
44 }

```

*code/conc/echoservers.c*

Функция `init_pool` (листинг 12.4) инициализирует пул клиентов. Массив `clientfd` – это набор дескрипторов сокетов открытых соединений с клиентами, в котором значение `-1` означает пустой слот. Первоначально набор дескрипторов сокетов открытых соединений не содержит действительных дескрипторов, а слушающий дескриптор является единственным в наборе чтения, который передается в вызов `select` (строки 10–12).

**Листинг 12.4.** `init_pool` инициализирует пул активных клиентов

```

1 void init_pool(int listenfd, pool *p)
2 {
3     /* Первоначально нет ни одного соединения */
4     int i;
5     p->maxi = -1;
6     for (i=0; i< FD_SETSIZE; i++)
7         p->clientfd[i] = -1;
8
9     /* Первоначально listenfd -- единственный дескриптор в наборе чтения */
10    p->maxfd = listenfd;
11    FD_ZERO(&p->read_set);
12    FD_SET(listenfd, &p->read_set);
13 }

```

*code/conc/echoservers.c*

Функция `add_client` (листинг 12.5) добавляет в пул нового клиента. Отыскав пустой слот в массиве `clientfd`, сервер добавляет в него дескриптор сокета открытого соединения и инициализирует соответствующий буфер чтения `RIO` так, что дескриптор можно передать в вызов `rio_readlineb` (строки 8–9). Затем дескриптор соединения добавляется в набор чтения для функции `select` (строка 12) и обновляются некоторые глобальные свойства пула. В переменной `maxfd` (строки 15–16) сохраняется наибольший дескриптор (для `select`), и в переменной `maxi` (строки 17–18) сохраняется наибольший индекс в массиве `clientfd`, чтобы функции `check_clients` не приходилось просматривать весь массив.

**Листинг 12.5.** `add_client` добавляет нового клиента в пул соединений

```

1 void add_client(int connfd, pool *p)
2 {
3     int i;
4     p->nready--;
5     for (i = 0; i < FD_SETSIZE; i++) /* Найти пустой слот */
6         if (p->clientfd[i] < 0) {
7             /* Добавить дескриптор соединения в пул */
8             p->clientfd[i] = connfd;
9             Rio_readinitb(&p->clientrio[i], connfd);
10
11             /* Добавить дескриптор в набор чтения */
12             FD_SET(connfd, &p->read_set);
13 }

```

*code/conc/echoservers.c*



```

14         /* Обновить maxfd и maxi */
15         if (connfd > p->maxfd)
16             p->maxfd = connfd;
17         if (i > p->maxi)
18             p->maxi = i;
19         break;
20     }
21     if (i == FD_SETSIZE) /* Пустые слоты закончились */
22         app_error("add_client error: Too many clients");
23 }

```

*code/conc/echoservers.c*

Функция `check_clients` (листинг 12.6) читает из дескриптора, готового к чтению, строку, отправленную клиентом. Если чтение строки увенчалось успехом, то она возвращается назад клиенту (строки 15–18). Обратите внимание, что в строке 15 наращивается накопительный счетчик байтов, полученных сервером от всех клиентов. Если обнаруживается признак конца файла (EOF) из-за того, что клиент закрыл свой конец соединения, то сервер закрывает соединение со своей стороны (строка 23) и удаляет дескриптор из пула (строки 24–25).

#### Листинг 12.6. `check_clients` обслуживает подключенных клиентов

*code/conc/echoservers.c*

```

1 void check_clients(pool *p)
2 {
3     int i, connfd, n;
4     char buf[MAXLINE];
5     rio_t rio;
6
7     for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
8         connfd = p->clientfd[i];
9         rio = p->clientrio[i];
10
11         /* Если дескриптор готов для чтения, вернуть клиенту его строку */
12         if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
13             p->nready--;
14             if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
15                 byte_cnt += n;
16                 printf("Server received %d (%d total) bytes on fd %d\n",
17                     n, byte_cnt, connfd);
18                 Rio_writen(connfd, buf, n);
19             }
20
21             /* Обнаружен конец файла, удалить дескриптор из пула */
22             else {
23                 Close(connfd);
24                 FD_CLR(connfd, &p->read_set);
25                 p->clientfd[i] = -1;
26             }
27         }
28     }
29 }

```

*code/conc/echoservers.c*

В контексте модели конечного автомата функция `select` выявляет события ввода, а функция `add_client` создает новый конечный автомат. Функция `check_client` выполняет переходы, отправляя клиентам их строки, а также удаляет данный конечный автомат, когда клиент разрывает соединение.

**Упражнение 12.4 (решение в конце главы)**

В программе сервера, показанной в листинге 12.3, мы аккуратно инициализируем переменную `pool.ready_set` непосредственно перед каждым вызовом `select`. Почему?

### 12.2.2. Достоинства и недостатки мультиплексирования ввода/вывода

Сервер в листинге 12.3 представляет прекрасный пример преимуществ и недостатков событийно-ориентированного программирования, основанного на мультиплексировании ввода/вывода. Одно из преимуществ этого подхода к программированию – возможность для программистов точнее управлять поведением своих программ, чем при использовании моделей процессов. Например, в событийно-ориентированном конкурентном сервере проще обеспечить предпочтительное обслуживание определенных клиентов, чем в конкурентном сервере на основе процессов.

Другое преимущество – событийно-ориентированный сервер с мультиплексированием ввода/вывода, работает в контексте одного процесса, а это значит, что каждый логический поток управления имеет доступ ко всему адресному пространству процесса. Это упрощает совместное использование данных потоками. Дополнительное преимущество выполнения в единственном процессе – конкурентный сервер поддается отладке, как любая последовательная программа, с использованием уже знакомых инструментов, таких как GDB. И наконец, событийно-ориентированные проекты часто значительно более эффективны и производительны, чем проекты, основанные на процессах, потому что не требуют переключения контекста процесса для планирования нового потока.

Значительным недостатком событийно-ориентированных программ является сложность программирования. Например, для реализации конкурентного событийно-ориентированного эхо-сервера потребовалось написать в три раза больше кода, чем для того же сервера на основе процессов, и, к сожалению, эта сложность растет по мере уменьшения степени распараллеливания. Под *распараллеливанием* здесь подразумевается количество команд, выполняемое каждым логическим потоком управления в заданный период времени. Например, в рассматриваемом конкурентном сервере степень распараллеливания определяется количеством команд, необходимых для чтения текстовой строки. Пока один логический поток управления читает текстовую строку, все остальные стоят на месте в ожидании. Для нашего примера в таком положении вещей нет ничего страшного, однако это делает событийно-ориентированный сервер уязвимым для клиентов-злоумышленников, которые могут отправить только часть текстовой строки, после чего останавливают процесс передачи. Адаптация событийно-ориентированного сервера для подобного рода ситуаций – задача нетривиальная, тогда как в сервере на основе процессов она решается автоматически.

#### Веб-серверы, управляемые событиями

Несмотря на недостатки, описанные в разделе 12.2.2, современные высокопроизводительные серверы, такие как Node.js, nginx и Tornado, используют подход мультиплексирования ввода/вывода с управлением событиями, потому что этот прием дает значительный выигрыш в производительности, по сравнению с приемами на основе процессов и потоков.

## 12.3. Конкурентное программирование с потоками выполнения

Итак, мы рассмотрели два подхода к созданию конкурентных приложений: на основе процессов и с мультиплексированием ввода/вывода. Ядро автоматически планирует каждый процесс. Каждый процесс имеет свое изолированное адресное пространство, что затрудняет совместное использование данных. Во втором подходе логические потоки управления планируются с использованием мультиплексирования ввода/вывода. Так как все эти логические потоки действуют в рамках одного процесса, они все вместе используют одно адресное пространство. В этом разделе мы рассмотрим третий подход, основанный на потоках выполнения, объединяющий все преимущества двух первых подходов.

*Поток выполнения* (thread) – это логический поток управления, выполняющийся в контексте процесса. До сих пор программы, описывавшиеся в книге, содержали один поток на процесс. Однако современные системы позволяют писать многопоточные программы, выполняющиеся конкурентно в одном процессе. Каждый поток имеет свой *контекст*, включающий уникальный целочисленный *идентификатор потока* (Thread ID, TID), стек, указатель стека, счетчик команд, регистры общего назначения и флаги состояния. Все выполняющиеся в процессе потоки совместно используют полное виртуальное адресное пространство этого процесса.

Решения на основе потоков выполнения сочетают в себе лучшие качества решений на основе процессов и мультиплексирования ввода/вывода. Потоки выполнения, как и процессы, автоматически планируются ядром и распознаются им по целочисленным идентификаторам. Подобно решениям на основе мультиплексирования ввода/вывода, потоки выполнения действуют в контексте одного процесса и совместно используют все виртуальное адресное пространство процесса, включая код, данные, библиотеки и открытые файлы.

### 12.3.1. Модель выполнения многопоточных программ

Модель выполнения многопоточных программ сходна с моделью выполнения программ, запускающих несколько процессов. Рассмотрим пример на рис. 12.6. Каждый процесс начинает свое существование в виде единственного потока выполнения, который принято называть *основным*, или *главным*, *потомком*. В определенной точке главный поток создает *дочерний поток*, и с этого момента эти два потока выполняются конкурентно. В какой-то момент управление передается этому дочернему потоку в результате переключения контекста, или потому что главный поток обратился к медленному системному вызову, такому как `read` или `sleep`, или потому что он был прерван системным таймером. Одноранговый поток выполняется в течение некоторого времени, после чего управление опять передается главному потоку.

Порядок выполнения потоков имеет некоторые важные отличия от выполнения процессов. Так как контекст потока намного меньше контекста процесса, переключение контекста между потоками происходит быстрее, чем переключение контекста между процессами. Другое важное отличие: отношения «родитель/потомок» между потоками носят чисто номинальный характер. Потоки, выполняющиеся в рамках одного процесса, образуют пул равноправных потоков выполнения. Главный поток отличается от других потоков только тем, что в процессе он всегда запускается первым. Основное следствие такой организации потоков в виде пула равноправных потоков заключается в том, что любой поток может завершить выполнение или дожидаться завершения любого другого потока. Более того, все потоки могут читать и изменять одни и те же совместно используемые данные.

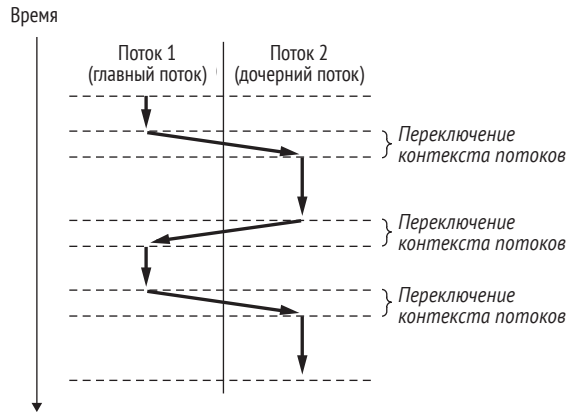


Рис. 12.6. Конкурентное выполнение потоков

### 12.3.2. Потоки Posix

Потоки Posix (Posix threads, Pthreads) – стандартный интерфейс для управления потоками выполнения из программ на С. Он был принят на вооружение в 1995 г. и реализован в большинстве систем Unix. Интерфейс Pthreads определяет порядка 60 функций, позволяющих программам создавать, останавливать и утилизировать потоки выполнения, безопасно использовать данные совместно с другими потоками в том же процессе и уведомлять друг друга об изменениях в состоянии системы.

В листинге 12.7 показана простая программа, использующая потоки Posix. Главный поток создает дочерний поток и ожидает его завершения. Дочерний поток выводит строку «Hello, world!\n» и завершается. Когда главный поток обнаруживает окончание выполнения дочернего потока, он завершает процесс вызовом `exit`. Это первая наша многопоточная программа, поэтому разберем ее подробнее. Код и локальные данные для потока инкапсулируются в *процедуру потока*. Как показывает прототип в строке 2, процедура потока принимает один нетипизированный указатель и возвращает нетипизированный указатель. Если понадобится передать в процедуру потока несколько аргументов, их следует поместить в структуру и передать указатель на эту структуру. Аналогично, если понадобится вернуть из процедуры потока несколько значений, то их следует поместить в структуру и вернуть указатель на нее.

Листинг 12.7. `hello.c`: программа «Hello, world!» на основе интерфейса Pthreads

```

1 #include "csapp.h"
2 void *thread(void *vargp);
3
4 int main()
5 {
6     pthread_t tid;
7     Pthread_create(&tid, NULL, thread, NULL);
8     Pthread_join(tid, NULL);
9     exit(0);
10 }
11
12 void *thread(void *vargp) /* Процедура потока */
13 {

```

*code/conc/hello.c*

```

14     printf("Hello, world!\n");
15     return NULL;
16 }

```

*code/conc/hello.c*

Строка 4 отмечает начало кода главного потока. Главный поток объявляет одну локальную переменную `tid`, которая будет использоваться для хранения идентификатора дочернего потока (строка 6). Главный поток создает новый дочерний поток вызовом функции `pthread_create` (строка 7). При возврате из `pthread_create` главный поток и вновь созданный дочерний поток выполняются конкурентно, а в `tid` сохраняется идентификатор нового потока. Главный поток переходит в ожидание завершения дочернего потока вызовом `pthread_join` в строке 8. Наконец, главный поток вызывает `exit` (строка 9) и завершает выполнение процесса (и всех потоков, которые могли иметься в процессе, в данном случае только главного).

В строках 12–16 определяется процедура потока, которая выполняется дочерними потоками. В этом примере она просто выводит строку, после чего выполнение потока завершается выполнением оператора `return` в строке 15.

### 12.3.3. Создание потоков

Потоки могут создавать другие потоки вызовом функции `pthread_create`:

```

#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                  func *f, void *arg);

```

Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

Функция `pthread_create` создает новый поток и запускает *процедуру потока* в контексте нового потока с аргументом `arg`. Аргумент `attr` можно использовать для изменения атрибутов по умолчанию вновь созданного потока, однако мы не будем обсуждать этот аспект и в наших примерах всегда будем передавать в этом аргументе значение `NULL`.

После возврата из `pthread_create` аргумент `tid` содержит идентификатор вновь созданного потока. Новый поток может узнать свой идентификатор вызовом функции `pthread_self`.

```

#include <pthread.h>

pthread_t pthread_self(void);

```

Возвращает идентификатор вызвавшего потока

### 12.3.4. Завершение потоков

Поток может завершить выполнение одним из следующих способов:

- *неявно*, возвратом из процедуры потока;
- *явно*, вызовом функции `pthread_exit`;

```

#include <pthread.h>

void pthread_exit(void *thread_return);

```

Ничего не возвращает

- некоторый поток может вызвать функцию `exit`, которая завершит процесс и все его потоки;
- другой поток может завершить текущий вызовом функции `pthread_cancel` с идентификатором текущего потока.

```
#include <pthread.h>
```

```
int pthread_cancel(pthread_t tid);
```

Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

### 12.3.5. Утилизация завершившихся потоков

Поток может дожидаться завершения другого потока вызовом функции `pthread_join`.

```
#include <pthread.h>
```

```
int pthread_join(pthread_t tid, void **thread_return);
```

Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

Функция `pthread_join` блокирует выполнение вызывающего потока до остановки потока с идентификатором `tid`, по адресу `void **thread_return` помещает указатель на возвращаемое процедурой потока значение, после чего *утилизирует* (освобождает) ресурсы памяти, удерживаемые прерванным потоком.

Обратите внимание, что, в отличие от функции `wait`, функция `pthread_join` ожидает прекращения выполнения только конкретного потока. С помощью `pthread_join` нельзя дожидаться завершения *произвольного* потока. Это может усложнить код из-за необходимости использовать другие, менее понятные механизмы для обнаружения завершения произвольного потока. Стивенс (Stevens) вполне убедительно заявляет о том, что такое поведение является следствием ошибки в спецификации [110].

### 12.3.6. Обособление потоков

В любой момент времени поток может быть *присоединяемым* (joinable) или *обособленным* (detached). Присоединяемый поток может быть остановлен и утилизирован другими потоками. Его ресурсы памяти (например, стек) не освобождаются, пока он не будет утилизирован другим потоком. Напротив, обособленный поток не может быть остановлен и утилизирован другими потоками. При прекращении выполнения его ресурсы освобождаются системой автоматически.

По умолчанию потоки создаются присоединяемыми. Во избежание утечек памяти каждый присоединяемый поток должен быть утилизирован явно другим потоком или обособлен вызовом функции `pthread_detach`.

```
#include <pthread.h>
```

```
int pthread_detach(pthread_t tid);
```

Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

Функция `pthread_detach` обособляет поток `tid`. Потоки могут обособляться самостоятельно вызовом функции `pthread_detach` с аргументом `pthread_self()`.

Несмотря на то что в некоторых примерах будут использоваться присоединяемые потоки, существуют веские причины использования в реальных программах обособленных потоков. Например, высокопроизводительный веб-сервер может создавать новые потоки при получении запроса на соединение. Поскольку каждое соединение управляется независимо отдельным потоком, то для сервера нет необходимости – и даже нежелательно – явно ждать, когда тот или иной поток завершится. В этом случае каждый поток-обработчик должен самостоятельно обособить себя до того, как приступит к обработке запроса, чтобы его ресурсы памяти могли использоваться повторно после завершения работы.

### 12.3.7. Инициализация потоков

Функция `pthread_once` позволяет инициализировать состояние, связанное с процедурой потока:

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
```

Всегда возвращает 0

Переменная `once_control` – это глобальная или статическая переменная, которая всегда инициализируется значением `pthread_once_init`. При первом вызове `pthread_once` с аргументом `once_control` она вызывает `init_routine` – функцию без входных аргументов, которая ничего не возвращает. Последующие обращения к `pthread_once` с аргументом `pthread_once` ни к чему не приводят. Функция `pthread_once` используется для динамической инициализации глобальных переменных, совместно используемых несколькими потоками. Один из примеров будет показан в разделе 12.5.5.

### 12.3.8. Конкурентный многопоточный сервер

В листинге 12.8 показана реализация конкурентного многопоточного эхо-сервера. Общая структура схожа с версией на основе процессов. Главный поток ожидает запроса на соединение, после чего создает новый поток для обработки этого запроса. Код выглядит простым, однако в нем имеется пара общих и в определенной степени щекотливых моментов, на которые следует обратить особое внимание.

**Листинг 12.8.** Конкурентный многопоточный эхо-сервер

*code/conc/echoservers.c*

```
1 #include "csapp.h"
2
3 void echo(int connfd);
4 void *thread(void *vargp);
5
6 int main(int argc, char **argv)
7 {
8     int listenfd, *conncfdp;
9     socklen_t clientlen;
10    struct sockaddr_storage clientaddr;
11    pthread_t tid;
12
```

```

13     if (argc != 2) {
14         fprintf(stderr, "usage: %s <port>\n", argv[0]);
15         exit(0);
16     }
17     listenfd = Open_listenfd(argv[1]);
18
19     while (1) {
20         clientlen=sizeof(struct sockaddr_storage);
21         connfd = Malloc(sizeof(int));
22         *connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
23         Pthread_create(&tid, NULL, thread, connfd);
24     }
25 }
26
27 /* Процедура потока */
28 void *thread(void *vargp)
29 {
30     int connfd = *((int *)vargp);
31     Pthread_detach(pthread_self());
32     Free(vargp);
33     echo(connfd);
34     Close(connfd);
35     return NULL;
36 }

```

*code/conc/echoservers.c*

Первый момент связан с передачей дескриптора соединения в вызов `pthread_create`. Очевидное решение – передать указатель на дескриптор, как показано ниже:

```

connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
Pthread_create(&tid, NULL, thread, &connfd);

```

Затем дочерний поток мог бы разыменовать указатель и сохранить полученное значение в локальной переменной:

```

void *thread(void *vargp) {
    int connfd = *((int *)vargp);
    .
    .
    .
}

```

Однако это ошибочное решение, потому что возникает состояние гонки между оператором присваивания в дочернем и вызовом `accept` в главном потоке. Если оператор присваивания завершится до следующего вызова `accept`, то локальная переменная `connfd` в дочернем потоке получит корректное значение дескриптора. Но если присваивание завершится *после* `accept`, то локальная переменная `connfd` в дочернем потоке получит номер дескриптора *следующего* соединения. В результате может сложиться ситуация, когда два потока будут обслуживать один и тот же дескриптор. Во избежание потенциальной «гонки на выживание» необходимо сохранить дескриптор соединения, возвращаемый функцией `accept`, в выделенном блоке динамической памяти, как показано в строках 21–22. Мы еще вернемся к вопросу состояния гонки в разделе 12.7.4.

Другой тонкий момент – возможность утечек памяти в процедуре потока. Поскольку потоки не утилизируются явно, каждый из них нужно обособлять, чтобы его ресурсы памяти утилизировались по завершении (строка 31). Более того, необходимо своевременно освобождать блоки памяти, выделенные главным потоком (строка 32).



**Упражнение 12.5 (решение в конце главы)**

В сервере, основанном на процессах (листинг 12.1), дескриптор установленного соединения мы закрывали в обоих процессах, родительском и дочернем. Однако в многопоточном сервере (листинг 12.8) дескриптор соединения закрывается только в дочернем потоке. Почему?

## 12.4. Совместное использование переменных несколькими потоками выполнения

С точки зрения программиста, одним из привлекательных аспектов многопоточности является простота, с какой потоки могут совместно использовать одни и те же переменные. Однако подобное совместное использование может быть коварным. Чтобы написать корректную программу, необходимо четко понимать, что подразумевается под совместным использованием и как это совместное использование работает.

Для понимания особенностей совместного использования переменных в программе рассмотрим несколько основополагающих вопросов: (1) базовая модель памяти потоков, (2) особенности хранения экземпляров переменных в памяти и (3) сколько потоков обращается к каждому из этих экземпляров. Переменная *используется совместно* (является общей), если только потоки обращаются к определенному экземпляру этой переменной.

Для большей конкретики рассмотрим пример программы в листинге 12.9. Она выглядит несколько запутанной, но достаточно наглядно иллюстрирует некоторые тонкости понятия совместного использования переменных. Программа-пример состоит из главного потока, создающего два дочерних потока. Главный поток передает каждому дочернему потоку уникальный идентификатор, используемый для вывода в сообщении вместе со счетчиком вызовов процедуры потока.

**Листинг 12.9.** Пример программы, иллюстрирующей некоторые аспекты совместного использования

*code/conc/sharing.c*

```

1 #include "csapp.h"
2 #define N 2
3 void *thread(void *vargp);
4
5 char **ptr; /* Глобальная переменная */
6
7 int main()
8 {
9     int i;
10    pthread_t tid;
11    char *msgs[N] = {
12        "Hello from foo",
13        "Hello from bar"
14    };
15
16    ptr = msgs;
17    for (i = 0; i < N; i++)
18        Pthread_create(&tid, NULL, thread, (void *)i);
19    Pthread_exit(NULL);
20 }
21
```

```

22 void *thread(void *vargp)
23 {
24     int myid = (int)vargp;
25     static int cnt = 0;
26     printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
27     return NULL;
28 }

```

*code/conc/sharing.c*

### 12.4.1. Модель памяти потоков

Пул потоков выполняется в контексте процесса. Каждый поток имеет свой *контекст потока*, включающий идентификатор потока, указатель стека, счетчик команд, флаги условий и значения регистров общего назначения. И все потоки совместно используют остальную часть контекста процесса. Сюда входят виртуальное адресное пространство процесса, включая код, доступный только для чтения, данные, доступные для чтения и/или записи, динамическую память, любой код и данные разделяемых библиотек. Потоки также используют одно и то же множество открытых файлов.

С одной стороны, потоки не могут читать и изменять значения регистров друг друга. С другой стороны, любой поток может получить доступ к любой ячейке в совместно используемой виртуальной памяти. Если какой-либо поток изменит ячейку памяти, то любой другой поток заметит это изменение, если прочитает эту ячейку. То есть регистры никогда не используются совместно, а виртуальная память – всегда.

Модель памяти для стека потока не так прозрачна. Стеки находятся в области виртуального адресного пространства, отведенной для стеков, и доступ потоков к ней обычно осуществляется независимо. Здесь употреблено слово «обычно», а не «всегда», потому что стек потока не защищен от доступа других потоков. Поэтому если какому-то потоку удастся получить указатель на стек другого потока, он сможет прочитать и записать свои данные в любую часть этого стека. В программе-примере это показано в строке 26, где дочерние потоки косвенно обращаются к содержимому стека главного потока через переменную `ptr`.

### 12.4.2. Особенности хранения переменных в памяти

Переменные в многопоточных программах на C отображаются в виртуальную память в соответствии с их классами памяти.

- *Глобальные переменные.* Глобальной называется любая переменная, объявленная за пределами функции. Во время выполнения область виртуальной памяти, доступная для чтения/записи, хранит ровно один экземпляр каждой глобальной переменной, к которому может обратиться любой поток. Например, глобальная переменная `ptr`, объявленная в строке 5, имеет один экземпляр в виртуальной памяти. При наличии единственного экземпляра переменной он обозначается просто именем переменной, в данном случае – `ptr`.
- *Локальные автоматические переменные.* Это переменные, объявленные внутри функции без атрибута `static`. Во время выполнения каждый поток хранит свои экземпляры любых локальных автоматических переменных в стеке. Это верно, даже если несколько потоков одновременно выполняют одну и ту же процедуру. Например, в этой программе имеется только один экземпляр локальной переменной `tid`; он хранится в стеке главного потока. Обозначим его именем `tid.m`. Другой пример: в программе имеется два экземпляра локальной переменной `myid`: один в стеке дочернего потока 0, а другой – в стеке дочернего потока 1. Обозначим эти экземпляры как `myid.p0` и `myid.p1` соответственно.

- *Локальные статические переменные.* Локальная статическая переменная – это переменная, объявленная внутри функции с атрибутом `static`. Как и в случае с глобальными переменными, в виртуальной памяти хранится только один экземпляр каждой локальной статической переменной, объявленной в программе. Например, несмотря на то что каждый дочерний поток в программе-примере объявляет переменную `cnt` в строке 25, во время выполнения в виртуальной памяти существует только один ее экземпляр. Все дочерние потоки читают и изменяют этот экземпляр.

### 12.4.3. Совместно используемые переменные

Мы говорим, что переменная *v* *используется совместно*, если к одному из ее экземпляров обращается более одного потока. Например, переменная `cnt` в программе-примере является совместно используемой, потому что она имеет только один экземпляр и к этому экземпляру обращаются оба дочерних потока. С другой стороны, `myid` не является совместно используемой, потому что к каждому из двух ее экземпляров обращается только один поток. Важно понимать, что такие локальные автоматические переменные, как `msgs`, также могут использоваться совместно.

#### Упражнение 12.6 (решение в конце главы)

1. Используя анализ из раздела 12.4, заполните следующую таблицу словами «Да» и «Нет» для программы-примера. В первом столбце запись *v.t* обозначает экземпляр переменной *v*, находящийся в локальном стеке потока *t*, где *t* – либо *m* (главный поток), либо *p0* (дочерний поток 0), либо *p1* (дочерний поток 1).

Экземпляр переменной	Доступен		
	главному потоку	дочернему потоку 0	дочернему потоку 1
<code>ptr</code>			
<code>cnt</code>			
<code>i.m</code>			
<code>msgs.m</code>			
<code>myid.p0</code>			
<code>myid.p1</code>			

2. Какая из переменных – `ptr`, `cnt`, `i`, `msgs` и `myid` – является совместно используемой?

## 12.5. Синхронизация потоков выполнения с помощью семафоров

Совместное использование переменных может быть удобным, но при этом не исключается возможность появления *ошибок синхронизации*. Рассмотрим программу `badcnt.c` в листинге 12.10. Она создает два потока, каждый из которых увеличивает совместно используемый счетчик с именем `cnt`.

#### Листинг 12.10. `badcnt.c` : программа с ошибкой синхронизации

[code/conc/badcnt.c](#)

```

1 /* ВНИМАНИЕ: этот код содержит ошибку! */
2 #include "csapp.h"
3
4 void *thread(void *vargp); /* Прототип процедуры потока */

```

```

5
6 /* Совместно используемая глобальная переменная */
7 volatile long cnt = 0; /* Счетчик */
8
9 int main(int argc, char **argv)
10 {
11     long niters;
12     pthread_t tid1, tid2;
13
14     /* Проверить входные аргументы */
15     if (argc != 2) {
16         printf("usage: %s <niters>\n", argv[0]);
17         exit(0);
18     }
19     niters = atoi(argv[1]);
20
21     /* Создать потоки и ждать их завершения */
22     Pthread_create(&tid1, NULL, thread, &niters);
23     Pthread_create(&tid2, NULL, thread, &niters);
24     Pthread_join(tid1, NULL);
25     Pthread_join(tid2, NULL);
26
27     /* Проверить результат */
28     if (cnt != (2 * niters))
29         printf("BOOM! cnt=%ld\n", cnt);
30     else
31         printf("OK cnt=%ld\n", cnt);
32     exit(0);
33 }
34
35 /* Процедура потока */
36 void *thread(void *vargp)
37 {
38     long i, niters = *((long *)vargp);
39
40     for (i = 0; i < niters; i++)
41         cnt++;
42
43     return NULL;
44 }

```

*code/conc/badcnt.c*

Поскольку каждый поток увеличивает счетчик `niters` раз, можно ожидать, что конечное значение будет равно  $2 \times \text{niters}$ . Однако если попробовать запустить `badcnt.c` несколько раз, то мы не только получим неправильный ответ, но и каждый раз ответы будут разными!

```
linux> ./badcnt 1000000
BOOM! cnt=1445085
```

```
linux> ./badcnt 1000000
BOOM! cnt=1915220
```

```
linux> ./badcnt 1000000
BOOM! cnt=1404746
```

В чем же дело? Чтобы разобраться в проблеме, нужно изучить ассемблерный код счетного цикла (строки 40–41), представленный на рис. 12.7.

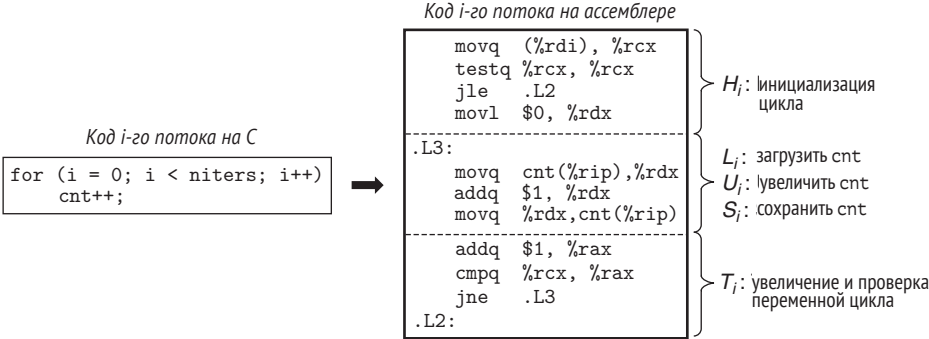


Рис. 12.7. Ассемблерный код цикла из программы badcnt.c (строки 40–41)

Давайте для простоты разобьем код цикла для  $i$ -го потока на пять частей.

- $H_i$ : блок инструкций инициализации цикла.
- $L_i$ : инструкция, загружающая совместно используемую переменную cnt в регистр аккумулятора %rdx<sub>*i*</sub>, где %rdx<sub>*i*</sub> обозначает значение регистра %rdx в  $i$ -м потоке.
- $U_i$ : инструкция, увеличивающая значение %rdx<sub>*i*</sub>.
- $S_i$ : инструкция, сохраняющая увеличенное значение %rdx<sub>*i*</sub> в совместно используемую переменную cnt.
- $T_i$ : блок инструкций, завершающих итерацию цикла.

Обратите внимание, что инструкции в начале и в конце цикла оперируют только локальными переменными в стеке, тогда как  $L_i$ ,  $U_i$  и  $S_i$  оперируют содержимым совместно используемой переменной счетчика.

Когда два дочерних потока в badcnt.c выполняются на единственном процессоре, машинные команды выполняются одна за другой в некотором порядке, то есть инструкции в этих двух потоках чередуются некоторым образом. К сожалению, одни из этих чередований приведут к корректному результату, а другие – нет.

В общем случае невозможно предсказать, выберет ли операционная система корректный порядок выполнения потоков. Например, в табл. 12.1 (а) показан корректный порядок выполнения инструкций. После того как каждый поток обновит совместно используемую переменную cnt, она будет хранить в памяти значение 2, что является ожидаемым результатом.

Таблица 12.1. Порядок выполнения инструкций в первой итерации цикла в программе badcnt.c

(а) Корректный порядок выполнения						(б) Некорректный порядок выполнения					
Шаг	Поток	Инстр.	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt	Шаг	Поток	Инстр.	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
1	1	$H_1$	–	–	0	1	1	$H_1$	–	–	0
2	1	$L_1$	0	–	0	2	1	$L_1$	0	–	0
3	1	$U_1$	1	–	0	3	1	$U_1$	1	–	0
4	1	$S_1$	1	–	1	4	2	$H_2$	–	–	0
5	2	$H_2$	–	–	1	5	2	$L_2$	–	0	0
6	2	$L_2$	–	1	1	6	1	$S_1$	1	–	1

(a) Корректный порядок выполнения

Шаг	Поток	Инстр.	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
7	2	$U_2$	–	2	1
8	2	$S_2$	–	2	2
9	2	$T_2$	–	2	2
10	1	$T_1$	1	–	2

(b) Некорректный порядок выполнения

Шаг	Поток	Инстр.	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
7	1	$T_1$	1	–	1
8	2	$U_2$	–	1	1
9	2	$S_2$	–	1	1
10	2	$T_2$	–	1	1

С другой стороны, в табл. 12.1 (b) показан порядок выполнения инструкций, порождающий некорректный результат. Проблема возникает из-за того, что поток 2 загружает cnt на шаге 5 уже после того, как поток 1 загрузил cnt на шаге 2, но до того, как поток 1 успеет сохранить измененное значение на шаге 6. То есть каждый поток сохраняет в переменную cnt значение 1. Понятия корректного и некорректного порядка выполнения инструкций можно пояснить с помощью *графа выполнения*, который мы представим в следующем разделе.

### Упражнение 12.7 (решение в конце главы)

Заполните таблицу для следующего порядка выполнения инструкций в badcnt.c.

Шаг	Поток	Инстр.	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
1	1	$H_1$	–	–	0
2	1	$L_1$	_____	_____	_____
3	2	$H_2$	_____	_____	_____
4	2	$L_2$	_____	_____	_____
5	2	$U_2$	_____	_____	_____
6	2	$S_2$	_____	_____	_____
7	1	$U_1$	_____	_____	_____
8	1	$S_1$	_____	_____	_____
9	1	$T_1$	_____	_____	_____
10	2	$T_2$	_____	_____	_____

Получится ли в этом случае корректное значение cnt?

## 12.5.1. Граф выполнения

Граф выполнения моделирует выполнение  $n$  конкурентных потоков в виде траектории через  $n$ -мерное декартово пространство. Каждая ось  $k$  соответствует выполнению потока  $k$ . Каждая точка  $(I_1, I_2, \dots, I_n)$  представляет состояние, когда поток  $k$  ( $k = 1, \dots, n$ ) выполнил инструкцию  $I_k$ . Начало графа соответствует *первоначальному состоянию*, в котором ни один из потоков не выполнил ни одной инструкции.

На рис. 12.8 показан двумерный граф выполнения первой итерации цикла в программе badcnt.c. Горизонтальная ось соответствует потоку 1, вертикальная – потоку 2. Точка  $(L_1, S_2)$  соответствует состоянию, в котором поток 1 выполнил  $L_1$  а поток 2 выполнил  $S_2$ .

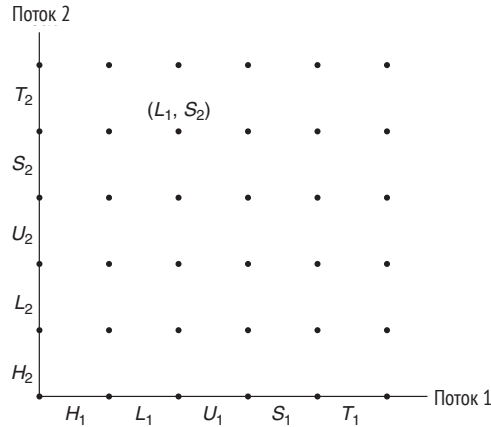


Рис. 12.8. Граф выполнения первой итерации цикла в программе badcnt.c

Граф выполнения представляет выполнение инструкций как переход из одного состояния в другое. Переходы изображаются в виде направленных ребер из одной точки к соседней с ней. Разрешаются переходы вправо (выполнение инструкции в потоке 1) или вверх (выполнение инструкции в потоке 2). Две инструкции не могут выполняться одновременно: диагональные переходы не допускаются. Программы никогда не выполняются в обратном направлении, поэтому переходы вниз или влево также недопустимы.

Порядок выполнения программы моделируется в виде *траектории* в пространстве состояний. На рис. 12.9 показана траектория, соответствующая выполнению инструкций в следующем порядке:

$$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2.$$

Для потока  $i$  инструкции  $(L_i, U_i, S_i)$ , манипулирующие содержимым совместно используемой переменной cnt, образуют *критическую секцию* (относительно совместно используемой переменной cnt), которая не должна пересекаться с критической секцией другого потока. Иными словами, мы должны гарантировать, что каждый поток имеет *исключительный доступ* к общей переменной, пока выполняет инструкции в своей критической секции. В целом такой режим доступа называют *взаимоисключающим*.

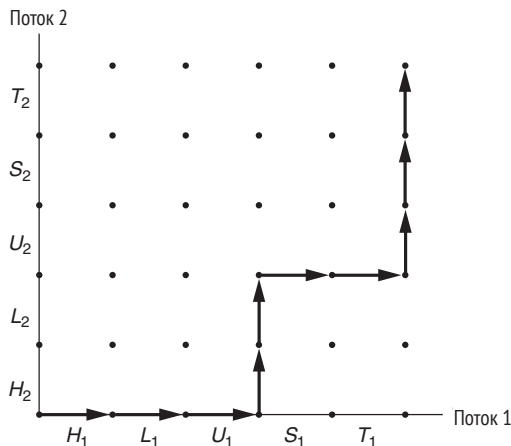
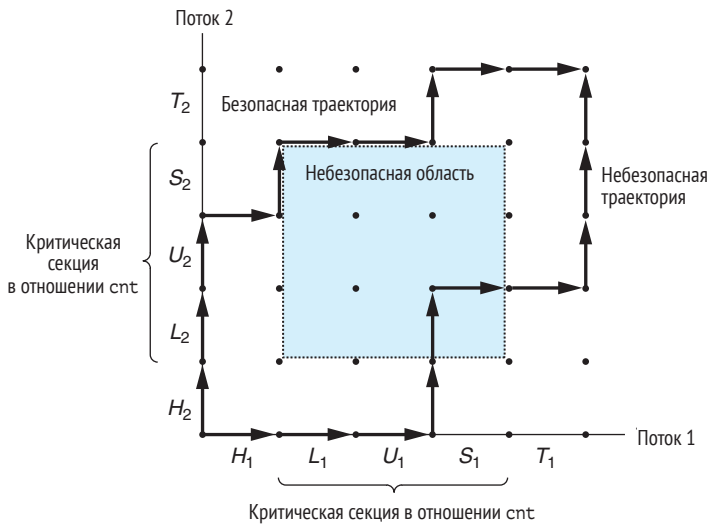


Рис. 12.9. Пример траектории

Пересечение двух критических секций определяет область пространства состояний, называемую *небезопасной областью*. На рис. 12.10 показана небезопасная область для переменной `cnt`. Обратите внимание, что небезопасная область примыкает к областям по своему периметру, но не включает их. Например, состояния  $(H_1, H_2)$  и  $(S_1, U_2)$  примыкают к небезопасной области, но не являются ее частью. Траектория, огибающая небезопасную область, называется *безопасной траекторией*, и наоборот, траектория, пролегающая через любое состояние в небезопасной области, называется *небезопасной траекторией*. На рис. 12.10 показаны примеры безопасной и небезопасной траекторий, проходящих через пространство состояний программы `badcnt.c`. Верхняя траектория огибает небезопасную область по левой и верхней сторонам и, следовательно, является безопасной. Нижняя траектория пересекает небезопасную область и, следовательно, является небезопасной.



**Рис. 12.10.** Безопасная и небезопасная траектории.

Траектории, пересекающие небезопасную область, небезопасны.

Траектории, огибающие небезопасную область, безопасны

Любая безопасная траектория будет корректно изменять совместно используемый счетчик. Для гарантии корректного выполнения рассматриваемой многопоточной программы – да и любой конкурентной программы, в которой совместно используются глобальные структуры данных, – так или иначе необходимо синхронизировать потоки, чтобы они всегда следовали по безопасным траекториям.

#### Упражнение 12.8 (решение в конце главы)

Используя граф выполнения на рис. 12.10, классифицируйте следующие траектории как *безопасные* или *небезопасные*:

1.  $H_1, L_1, U_1, S_1, H_2, L_2, U_2, S_2, T_2, T_1$
2.  $H_2, L_2, H_1, L_1, U_1, S_1, T_1, U_2, S_2, T_2$
3.  $H_1, H_2, L_2, U_2, S_2, L_1, U_1, S_1, T_1, T_2$



### 12.5.2. Семафоры

Эдсгер Дейкстра (Edsger Dijkstra) – первый, кто изучил и сформулировал дисциплину конкурентного программирования, предложил классическое решение проблемы синхронизации различных потоков выполнения, основанное на особом типе переменной, называемой *семафором*. Семафор  $s$  – это глобальная переменная с неотрицательным целочисленным значением, манипулировать которой могут только две специальные операции, называемые  $P$  и  $V$ .

$P(s)$ : если значение  $s$  не равно нулю, тогда  $P$  уменьшает значение  $s$  и немедленно возвращает управление. Если значение  $s$  равно нулю, тогда поток приостанавливается до момента, когда другой поток выполнит операцию  $V$  и  $s$  примет ненулевое значение. После возобновления поток вновь пробует выполнить операцию  $P$ , уменьшает значение  $s$  и возвращает управление вызывающему процессу.

$V(s)$ : операция  $V$  увеличивает значение  $s$  на единицу. При наличии потоков, заблокированных в операции  $P$  и ожидающих, когда  $s$  примет ненулевое значение, операция  $V$  перезапускает только один из этих потоков, который затем завершает свою операцию  $P$  уменьшением значения  $s$ .

#### Происхождение названий $P$ и $V$

Эдсгер Дейкстра – уроженец Нидерландов. Названия  $P$  и  $V$  происходят из голландского языка, от слов *proberen* (проверить) и *verhogen* (увеличить).

Проверка и уменьшение значения в операции  $P$  выполняются атомарно, в том смысле, что как только семафор  $s$  принимает ненулевое значение, уменьшение его значения не может быть прервано ни программными, ни аппаратными прерываниями. Операция увеличения значения в  $V$  тоже выполняется атомарно. Обратите внимание, что определение  $V$  не определяет порядок выбора потока для возобновления из числа ожидающих. Единственное требование: операция  $V$  должна возобновить ровно один ожидающий поток. *То есть когда на семафоре ожидают несколько потоков, невозможно предугадать, какой из них будет возобновлен в результате выполнения  $V$ .*

Определения  $P$  и  $V$  гарантируют, что выполняющаяся программа никогда не окажется в состоянии, когда корректно инициализированный семафор имеет отрицательное значение. Это свойство, называемое *инвариантом семафора*, делает семафоры мощным инструментом управления траекториями конкурентных программ, не позволяющим им пересекать небезопасные области.

Стандарт определяет несколько функций управления семафорами.

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Функция `sem_init` инициализирует семафор `sem` значением `value`. Каждый семафор должен инициализироваться перед использованием. В наших примерах средний аргумент всегда равен 0. Программы выполняют операции  $P$  и  $V$  вызовом функций `sem_wait` и `sem_post` соответственно. Для краткости мы будем использовать функции-обертки  $P$  и  $V$ :

```
#include "csapp.h"
```

```
void P(sem_t *s); /* Функция-обертка для sem_wait */
void V(sem_t *s); /* Функция-обертка для sem_post */
```

Ничего не возвращают

### 12.5.3. Использование семафоров для исключительного доступа к ресурсам

Семафоры реализуют удобный способ организовать исключительный доступ к общим переменным. Основная идея состоит в том, чтобы связать семафор  $s$ , изначально равный 1, с общей переменной (или набором общих переменных), а затем окружить соответствующую критическую секцию в коде операциями  $P(s)$  и  $V(s)$ .

Семафор, используемый таким образом для защиты общих переменных, называется *бинарным семафором*, потому что может принимать только два значения, 0 и 1. Бинарные семафоры, предназначенные для организации исключительного доступа, часто называют *мьютексами*. Выполнение операции  $P$  над мьютексом называется *блокировкой мьютекса*. Аналогично выполнение операции  $V$  называется *разблокировкой мьютекса*. О потоке, который заблокировал, но еще не разблокировал мьютекс, говорят, что он *удерживает мьютекс*. Семафор, который используется как счетчик для набора доступных ресурсов, называется *счетным семафором*.

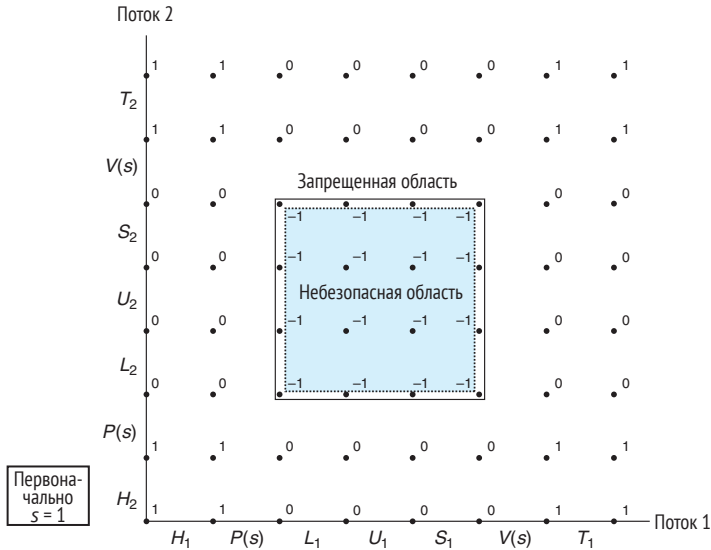
#### Ограничение графов выполнения

Графы выполнения прекрасно справляются с задачей визуализации хода выполнения конкурентных программ в однопроцессорных системах и объяснения необходимости синхронизации. Однако они имеют свои ограничения, в частности в том, что касается конкурентного выполнения в многопроцессорных системах, где множество пар процессор/кеш совместно используют одну и ту же основную память. Поведение многопроцессорной системы нельзя объяснить с помощью графов выполнения. Например, система памяти в многопроцессорной системе может находиться в состоянии, не соответствующем никакой траектории на графе выполнения. Но суть от этого не меняется: доступ к совместно используемым переменным всегда нужно синхронизировать.

Граф выполнения на рис. 12.11 показывает, как мы будем использовать бинарные семафоры для синхронизации потоков в нашей программе-счетчике.

Каждая точка в пространстве состояний подписана значением семафоров в этом состоянии. Ключевая идея состоит в том, что определенные комбинации операций  $P$  и  $V$  создают набор состояний, образующих запрещенную область, где  $s < 0$ . Из-за инвариантности семафора никакая допустимая траектория не может включать состояния из запрещенной области. А поскольку запрещенная область полностью охватывает небезопасную область, то никакая допустимая траектория не сможет коснуться какой-либо части небезопасной области. То есть любая допустимая траектория безопасна, если, следуя ей, программа корректно увеличивает счетчик, независимо от порядка выполнения инструкций.

Проще говоря, запрещенная область, созданная операциями  $P$  и  $V$ , делает невозможным одновременное выполнение инструкций в критической секции несколькими потоками, то есть семафоры обеспечивают исключительный доступ к критической области.



**Рис. 12.11.** Использование семафора для организации исключительного доступа. Недопустимые состояния, где  $s < 0$ , определяют запрещенную область, которая окружает небезопасную область и не позволяет допустимым траекториям касаться небезопасной области

Итак, чтобы правильно синхронизировать потоки в нашей программе-счетчике из листинга 12.10 с использованием семафоров, сначала объявим семафор с именем `mutex`:

```
volatile long cnt = 0; /* Счетчик */
sem_t mutex;          /* Семафор для защиты счетчика */
```

затем инициализируем его в функции `main`:

```
Sem_init(&mutex, 0, 1); /* mutex = 1 */
```

и, наконец, защитим операцию изменения счетчика `cnt` в процедуре потока, окружив ее операциями `P` и `V`:

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

Теперь благодаря семафору наша программа всегда будет давать верный результат:

```
linux> ./goodcnt 1000000
OK cnt=2000000
```

```
linux> ./goodcnt 1000000
OK cnt=2000000
```

## 12.5.4. Использование семафоров для организации совместного доступа к ресурсам

Другое важное применение семафоров – организация совместного доступа к ресурсам. В этом сценарии поток использует операцию с семафором, чтобы уведомить дру-

гой поток о том, что некоторое условие в состоянии программы стало верным. Классический пример: проблемы *производитель–потребитель* и *читателей–писателей*.

### Проблема производитель–потребитель

Суть проблемы *производитель–потребитель* показана на рис. 12.12. Поток-производитель и поток-потребитель совместно используют *буфер ограниченного объема* с емкостью *n* единиц. Поток-производитель создает новые элементы и добавляет их в буфер. Поток-потребитель извлекает элементы из буфера и использует (потребляет) их. Также возможны варианты с разными количествами производителей и потребителей.



**Рис. 12.12.** Проблема производитель–потребитель.

Производитель генерирует элементы и добавляет их в буфер ограниченного объема.

Потребитель извлекает элементы из буфера и «потребляет» их

Поскольку добавление и извлечение элементов связаны с обновлением совместно используемых переменных, необходимо гарантировать исключительный доступ к буферу. Однако одной этой гарантии недостаточно. Необходимо также организовать совместное использование буфера. Если буфер полон (нет места для новых элементов), то производитель должен дождаться, когда потребитель извлечет из буфера хотя бы один элемент. Аналогично, если буфер пуст, то потребитель должен дождаться появления в нем хотя бы одного элемента.

Взаимодействие производителя и потребителя – обычное явление в реальных системах. Например, в мультимедийной системе задачей производителя является кодирование видеокадров, а потребителя – их декодирование и отображение на экране. Цель буфера – обеспечить плавное отображение видеопотока без подтормаживаний, вызываемых различиями в производительности операций кодирования и декодирования отдельных кадров. Для производителя буфер служит пространством, куда можно складывать готовые кадры, а для потребителя – источником закодированных кадров. Другой расхожий пример: создание графических пользовательских интерфейсов. Производитель определяет события мыши и клавиатуры и вставляет их в буфер. Потребитель извлекает эти события из буфера в некой приоритетной последовательности и рисует изображение на экране.

В этом разделе мы разработаем простой пакет под названием SBUF, который можно будет использовать в программах типа производитель–потребитель. В следующем разделе мы используем его для создания конкурентного сервера. Пакет SBUF работает с буферами типа `sbuf_t` (листинг 12.11). Элементы хранятся в целочисленном массиве `buf` с размером `n` в динамической памяти. Переменные `front` и `rear` хранят индексы первого и последнего элементов в массиве. Доступ к буферу синхронизируется тремя семафорами. Семафор `mutex` обеспечивает исключительный доступ к буферу. Семафоры `slots` и `items` служат счетчиками пустых ячеек и доступных для извлечения элементов соответственно.

#### Листинг 12.11. `sbuf_t`: буфер ограниченного объема, используемый пакетом SBUF

*code/conc/sbuf.c*

```

1 typedef struct {
2     int *buf; /* Массив, лежащий в основе буфера */
3     int n; /* Максимальное число ячеек в массиве */
4     int front; /* buf[(front+1)%n] -- первый элемент */
  
```

```

5     int rear;      /* buf[rear%n] - последний элемент */
6     sem_t mutex; /* Защищает доступ к buf */
7     sem_t slots; /* Счетчик пустых ячеек */
8     sem_t items; /* Счетчик готовых к извлечению элементов */
9 } sbuf_t;

```

*code/conc/sbuf.c*

В листинге 12.12 показана полная реализация пакета SBUF. Функция `sbuf_init` выделяет память для буфера, устанавливает значения `front` и `rear`, соответствующие пустому буферу, и присваивает начальные значения трем семафорам. Эта функция вызывается один раз перед обращением к любой из трех других функций. Функция `sbuf_deinit` освобождает память, занятую буфером, когда приложение прекращает его использование. Функция `sbuf_insert` ожидает появления свободной ячейки, блокирует мьютекс, добавляет элемент, разблокирует мьютекс, после чего объявляет о наличии нового элемента. Функция `sbuf_remove` ожидает появления элемента в буфере, блокирует мьютекс, удаляет элемент из начала буфера, разблокирует мьютекс, после чего сигнализирует о появлении пустой ячейки.

**Листинг 12.12.** SBUF: пакет для синхронизации конкурентного доступа к ограниченному буферу

*code/conc/sbuf.c*

```

1 #include "csapp.h"
2 #include "sbuf.h"
3
4 /* Создает пустой буфер FIFO с n ячейками */
5 void sbuf_init(sbuf_t *sp, int n)
6 {
7     sp->buf = Calloc(n, sizeof(int));
8     sp->n = n; /* Буфер вмещает до n элементов */
9     sp->front = sp->rear = 0; /* Буфер пуст, если front == rear */
10    Sem_init(&sp->mutex, 0, 1); /* Бинарный семафор для блокировки */
11    Sem_init(&sp->slots, 0, n); /* Изначально буфер имеет n пустых ячеек */
12    Sem_init(&sp->items, 0, 0); /* Изначально в буфере нет готовых элементов */
13 }
14
15 /* Освобождает буфер sp */
16 void sbuf_deinit(sbuf_t *sp)
17 {
18     Free(sp->buf);
19 }
20
21 /* Добавляет элемент в конец буфера sp */
22 void sbuf_insert(sbuf_t *sp, int item)
23 {
24     P(&sp->slots); /* Ждать появления пустой ячейки */
25     P(&sp->mutex); /* Заблокировать буфер */
26     sp->buf[(++sp->rear)%(sp->n)] = item; /* Добавить элемент */
27     V(&sp->mutex); /* Разблокировать буфер */
28     V(&sp->items); /* Сообщить о новом элементе */
29 }
30
31 /* Извлекает и возвращает первый элемент из буфера sp */
32 int sbuf_remove(sbuf_t *sp)
33 {
34     int item;
35     P(&sp->items); /* Ждать появления элемента */

```

```

36     P(&sp->mutex);                               /* Заблокировать буфер */
37     item = sp->buf[(++sp->front)%(sp->n)];          /* Извлечь элемент */
38     V(&sp->mutex);                                 /* Разблокировать буфер */
39     V(&sp->slots);                                  /* Сообщить о пустой ячейке */
40     return item;
41 }

```

code/conc/sbuf.c

### Упражнение 12.9 (решение в конце главы)

Пусть  $p$  обозначает количество производителей,  $c$  – количество потребителей, а  $n$  – размер буфера в элементах. Для каждого из следующих сценариев укажите, необходим ли мьютекс в `sbuf_insert` и `sbuf_remove`.

1.  $p = 1, c = 1, n > 1$
2.  $p = 1, c = 1, n = 1$
3.  $p > 1, c > 1, n = 1$

## Проблема читателей–писателей

*Проблема читателей–писателей* является обобщением проблемы исключительного доступа. Представьте, что имеется несколько потоков, выполняющихся конкурентно, которые обращаются к общему объекту, такому как структура данных в оперативной памяти или база данных на диске. Одни потоки только читают данные из объекта, другие изменяют его. Потоки, изменяющие объект, называются *писателями*. Потоки, которые только читают, называются *читателями*. Писатели должны иметь исключительный доступ к объекту, но читатели могут читать данные одновременно с неограниченным числом других читателей. В общем случае существует неограниченное количество читателей и писателей, действующих конкурентно.

Подобные сценарии не редкость в реальных системах. Например, в онлайн-системе бронирования авиабилетов неограниченное количество клиентов могут одновременно проверять наличие свободных мест, но только клиент, бронирующий место, должен иметь исключительный доступ к базе данных. Другой пример: веб-прокси с многопоточным кешированием – неограниченное количество потоков может извлекать существующие страницы из общего кеша, но любой поток, записывающий новую страницу в кеш, должен иметь исключительный доступ.

Проблема читателей–писателей имеет несколько вариантов, каждый из которых отличается приоритетами читателей и писателей. В первом варианте предпочтение отдается читателям, чтобы ни одному читателю не приходилось ждать, если только писателю уже не был предоставлен исключительный доступ к объекту. То есть ни один читатель не должен ждать только потому, что своей очереди уже ждет писатель. Во втором варианте предпочтение отдается писателям, чтобы, когда писатель будет готов писать, он мог выполнить запись как можно скорее. В отличие от первого варианта, читатель, запросивший доступ после писателя, должен ждать, даже если писатель тоже ждет.

В листинге 12.13 показано решение первого варианта проблемы читателей–писателей. Так же как решения многих проблем с синхронизацией, это решение только кажется простым. Семафор `w` управляет доступом к критическим секциям, которые обращаются к общему объекту. Семафор `mutex` защищает доступ к общей переменной `readcnt` – счетчику читателей, выполняющих в данный момент критическую секцию. Писатель блокирует мьютекс `w` каждый раз, когда входит в критическую секцию, и разблокирует после выхода из нее. Это гарантирует, что в любой момент времени критическую сек-

цию будет выполнять только один писатель. С другой стороны, только первый читатель, вошедший в критическую секцию, блокирует мьютекс *w*, и только последний читатель, покинувший критическую секцию, разблокирует его. Мьютекс *w* игнорируется читателями, которые входят и выходят, пока присутствуют другие читатели. Это означает, что пока один читатель удерживает мьютекс *w*, неограниченное количество читателей могут беспрепятственно войти в критическую секцию.

**Листинг 12.13.** Решение первого варианта проблемы читателей–писателей.  
Предпочтение отдается читателям

```
/* Глобальные переменные */
int readcnt; /* Первоначально = 0 */
sem_t mutex, w; /* Оба первоначально = 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* Первый читатель */
            P(&w);
        V(&mutex);

        /* Критическая секция */
        /* Выполняется чтение */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Последний читатель */
            V(&w);
        V(&mutex);
    }
}

void writer(void)
{
    while (1) {
        P(&w);

        /* Критическая секция */
        /* Выполняется запись */

        V(&w);
    }
}
```

Правильное решение любой из проблем читателей–писателей может привести к *голоданию*, когда поток блокируется на неопределенный срок и не может продолжать выполняться. Например, в решении, показанном в листинге 12.13, писатель может заблокироваться надолго, пока не иссякнет поток читателей.

**Упражнение 12.10 (решение в конце главы)**

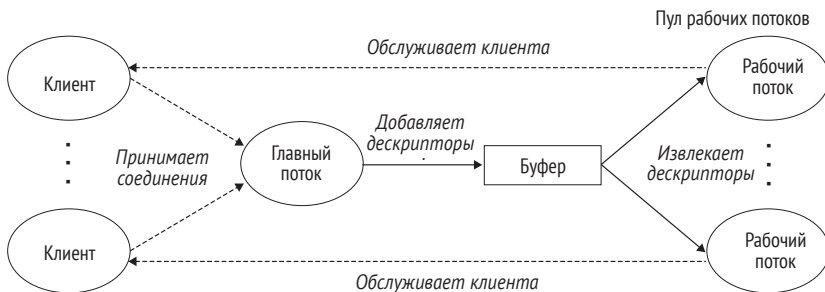
Решение первого варианта проблемы читателей–писателей в листинге 12.13 отдает предпочтение читателям, но это предпочтение не строгое, в том смысле, что поток-писатель,

покидающий критическую секцию, может возобновить ожидающий поток-писатель вместо ожидающего потока-читателя. Опишите сценарий, в котором такое нестрогое предпочтение позволило бы группе потоков-писателей вызвать голодание потоков-читателей.

### 12.5.5. Все вместе: конкурентный сервер на базе предварительно созданных потоков

Выше мы видели, как можно использовать семафоры для доступа к общим переменным и ресурсам. Для закрепления изученного реализуем конкурентный сервер, который будет заранее создавать потоки для обслуживания клиентов и использовать их.

В конкурентном сервере в листинге 12.8 для обслуживания каждого нового клиента создается новый поток. Недостаток такого подхода заключается в значительных затратах, связанных с созданием нового потока для каждого нового клиента. Сервер, основанный на предварительной организации поточной обработки, снижает эти издержки, используя модель производитель–потребитель, показанную на рис. 12.13. Сервер состоит из главного потока и набора рабочих потоков. Главный поток принимает запросы на соединение от клиентов и помещает дескрипторы подключенных сокетов в буфер. Каждый рабочий поток извлекает из буфера очередной дескриптор, обслуживает клиента и затем ждет появления следующего дескриптора.



**Рис.12.13.** Конкурентный сервер на базе предварительно созданных потоков.

Набор существующих потоков извлекает и обрабатывает дескрипторы подключенных сокетов из ограниченного буфера

В листинге 12.14 показано, как можно использовать пакет SBUF для реализации конкурентного эхо-сервера с предварительно созданными потоками. После инициализации буфера `sbuf` (строка 24) главный поток создает множество рабочих потоков (строки 25–26). Затем главный поток входит в бесконечный цикл, в котором принимает запросы на соединение и добавляет дескрипторы подключенных сокетов в `sbuf`. Все рабочие потоки действуют очень просто: ожидают появления в буфере дескриптора подключенного сокета (строка 39), после чего вызывают функцию `echo_cnt` для отправки клиентам отправленных ими строк.

**Листинг 12.14.** Конкурентный сервер на базе предварительно созданных потоков. Сервер основан на модели производитель–потребитель с одним потоком-производителем и несколькими потоками-потребителями

*code/conc/echoserv-pre.c*

```

1 #include "csapp.h"
2 #include "sbuf.h"
3 #define NTHREADS 4
  
```



```

4 #define SBUFSIZE 16
5
6 void echo_cnt(int connfd);
7 void *thread(void *vargp);
8
9 sbuf_t sbuf; /* Общий буфер с дескрипторами подключенных сокетов */
10
11 int main(int argc, char **argv)
12 {
13     int i, listenfd, connfd;
14     socklen_t clientlen;
15     struct sockaddr_storage clientaddr;
16     pthread_t tid;
17
18     if (argc != 2) {
19         fprintf(stderr, "usage: %s <port>\n", argv[0]);
20         exit(0);
21     }
22     listenfd = Open_listenfd(argv[1]);
23
24     sbuf_init(&sbuf, SBUFSIZE);
25     for (i = 0; i < NTHREADS; i++) /* Создать рабочие потоки */
26         Pthread_create(&tid, NULL, thread, NULL);
27
28     while (1) {
29         clientlen = sizeof(struct sockaddr_storage);
30         connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
31         sbuf_insert(&sbuf, connfd); /* Добавить connfd в буфер */
32     }
33 }
34
35 void *thread(void *vargp)
36 {
37     Pthread_detach(pthread_self());
38     while (1) {
39         int connfd = sbuf_remove(&sbuf); /* Извлечь connfd из буфера */
40         echo_cnt(connfd);                /* Обслужить клиента */
41         Close(connfd);
42     }
43 }

```

*code/conc/echoservt-pre.c*

Функция `echo_cnt` (листинг 12.15) является версией функции `echo` из листинга 11.9, которая записывает суммарное число байтов, полученных от всех клиентов, в глобальную переменную `byte_cnt`. Этот код особенно интересен тем, что он демонстрирует общую методику инициализации пакетов, вызываемых из процедур потоков. В данном случае необходимо инициализировать счетчик `byte_cnt` и семафор `mutex`. Один из подходов, которые мы использовали в пакетах `SBUF` и `RIO`, требовал, чтобы главный поток явно вызывал функцию инициализации. Другой подход, представленный здесь, основан на использовании функции `pthread_once` (строка 19) для вызова функции инициализации, как только какой-либо поток в первый раз вызовет функцию `echo_cnt`. Преимущество данного подхода заключается в простоте использования пакета. Недостаток – каждое обращение к `echo_cnt` вызывает функцию `pthread_once`, которая по большей части не делает ничего полезного.

**Листинг 12.15.** echo\_cnt: версия функции echo, подсчитывающей общее количество полученных байтов*code/conc/echo-cnt.c*

```

1 #include "csapp.h"
2
3 static int byte_cnt; /* Счетчик байтов */
4 static sem_t mutex; /* и защищающий его мьютекс */
5
6 static void init_echo_cnt(void)
7 {
8     Sem_init(&mutex, 0, 1);
9     byte_cnt = 0;
10 }
11
12 void echo_cnt(int connfd)
13 {
14     int n;
15     char buf[MAXLINE];
16     rio_t rio;
17     static pthread_once_t once = PTHREAD_ONCE_INIT;
18
19     Pthread_once(&once, init_echo_cnt);
20     Rio_readinitb(&rio, connfd);
21     while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
22         P(&mutex);
23         byte_cnt += n;
24         printf("server received %d (%d total) bytes on fd %d\n",
25             n, byte_cnt, connfd);
26         V(&mutex);
27         Rio_writen(connfd, buf, n);
28     }
29 }

```

*code/conc/echo-cnt.c*

После инициализации пакета функция `echo_cnt` инициализирует пакет RIO буферизованного ввода/вывода (строка 20), после чего возвращает клиенту полученную от него строку. Обратите внимание, что доступ к совместно используемой переменной `byte_cnt` в строках 23–24 защищен операциями *P* и *V*.

### Событийно-ориентированные многопоточные программы

Мультиплексирование ввода/вывода – не единственный способ к созданию событийно-ориентированных программ. Например, возможно, вы заметили, что разработанный нами конкурентный сервер на базе предварительно созданных потоков на самом деле является событийно-ориентированным с простыми конечными автоматами для главного и рабочих потоков. Главный поток имеет два состояния (ожидание запроса на соединение и ожидание появления свободного места в буфере), два события ввода/вывода (поступление запроса на соединение и появление свободного места в буфере) и два перехода (прием запроса на соединение и добавление дескриптора в буфер). Аналогично каждый рабочий поток имеет одно состояние (ожидание появления дескриптора в буфере), одно событие ввода/вывода (появление дескриптора в буфере) и один переход (извлечение дескриптора из буфера).

## 12.6. Использование потоков выполнения для организации параллельной обработки

До сих пор в нашем исследовании конкуренции мы предполагали, что конкурентные потоки выполняются в однопроцессорной системе. Однако большинство современных машин имеют многоядерные процессоры. Конкурентные программы часто работают быстрее на таких машинах, потому что ядро операционной системы планирует конкурентные потоки на нескольких ядрах, а не последовательно на одном ядре. Поддержка такого параллелизма критически важна в таких приложениях, как высоконагруженные веб-серверы, серверы баз данных и большие научные приложения, и становится все более полезной для основных приложений, таких как веб-браузеры, электронные таблицы и процессоры документов.

На рис. 12.14 показана связь между последовательными, конкурентными и параллельными программами. Множество всех программ можно разбить на непересекающиеся множества последовательных и конкурентных программ. Последовательная программа работает как единственный логический поток управления. Конкурентная программа – как несколько конкурентных потоков. А параллельная программа – это конкурентная программа, выполняющаяся на нескольких процессорах. То есть множество параллельных программ являются подмножеством конкурентных программ.



**Рис. 12.14.** Связь между множествами последовательных, конкурентных и параллельных программ

Подробное обсуждение параллельных программ выходит за рамки этой книги, и все же знакомство с некоторыми простыми примерами поможет вам понять некоторые важные аспекты параллельного программирования. Например, давайте посмотрим, как можно организовать параллельное вычисление суммы последовательности целых чисел  $0, \dots, n - 1$ . Конечно, для этой конкретной задачи существует аналитическое решение (формула), но тем не менее этот краткий и простой пример позволит нам сделать несколько интересных замечаний о параллельных программах.

Самый простой подход к распределению работы между разными потоками состоит в том, чтобы разбить последовательность на  $t$  непересекающихся областей, а затем назначить каждому из  $t$  разных потоков работу над своей областью. Для простоты предположим, что  $n$  кратно  $t$ , так что каждая область имеет  $n/t$  элементов. Рассмотрим несколько различных способов организации параллельной обработки отдельных областей несколькими потоками.

Самый простой и понятный вариант – заставить все потоки накапливать сумму в общей глобальной переменной, защищенной мьютексом. В листинге 12.16 показано, как это можно реализовать. В строках 28–33 главный поток создает дочерние потоки, а затем ожидает их завершения. Обратите внимание, что главный поток передает небольшое целое число каждому потоку, который служит уникальным идентификатором потока. Каждый дочерний поток будет использовать свой идентификатор, чтобы определить, с какой частью последовательности он должен работать. Эта идея передачи

небольшого уникального идентификатора дочерним потокам широко используется во многих параллельных приложениях. После завершения дочерних потоков глобальная переменная `gsum` будет содержать окончательную сумму. Затем главный поток использует аналитическое решение для проверки результата (строки 36–37).

**Листинг 12.16.** Функция `main` программы `psum-mutex`. Использует несколько потоков для вычисления суммы элементов последовательности в общей переменной, защищенной мьютексом

```

1 #include "csapp.h"
2 #define MAXTHREADS 32
3
4 void *sum_mutex(void *vargp); /* Процедура потока */
5
6 /* Общие глобальные переменные */
7 long gsum = 0; /* Общая сумма */
8 long nelems_per_thread; /* Количество суммируемых элементов */
9 sem_t mutex; /* Мьютекс для защиты глобальной суммы */
10
11 int main(int argc, char **argv)
12 {
13     long i, nelems, log_nelems, nthreads, myid[MAXTHREADS];
14     pthread_t tid[MAXTHREADS];
15
16     /* Получить аргументы */
17     if (argc != 3) {
18         printf("Usage: %s <nthreads> <log_nelems>\n", argv[0]);
19         exit(0);
20     }
21     nthreads = atoi(argv[1]);
22     log_nelems = atoi(argv[2]);
23     nelems = (1L << log_nelems);
24     nelems_per_thread = nelems / nthreads;
25     Sem_init(&mutex, 0, 1);
26
27     /* Создать потоки и ждать их завершения */
28     for (i = 0; i < nthreads; i++) {
29         myid[i] = i;
30         Pthread_create(&tid[i], NULL, sum_mutex, &myid[i]);
31     }
32     for (i = 0; i < nthreads; i++)
33         Pthread_join(tid[i], NULL);
34
35     /* Проверить окончательный ответ */
36     if (gsum != (nelems * (nelems-1))/2)
37         printf("Error: result=%ld\n", gsum);
38
39     exit(0);
40 }

```

*code/conc/psum-mutex.c*

В листинге 12.17 показана функция, которую выполняет каждый дочерний поток. В строке 4 поток извлекает свой идентификатор из аргумента и на его основе определяет области последовательности, над которой он должен работать (строки 5–6). В строках 9–13 поток перебирает свою часть последовательности, обновляя общую гло-

бальную переменную `gsum` в каждой итерации. Обратите внимание, что мы тщательно защищаем каждое обновление операциями с мьютексом  $P$  и  $V$ .

**Листинг 12.17.** Процедура потока для `psum-mutex`. Каждый дочерний поток прибавляет свои элементы к сумме в глобальной переменной, защищенной мьютексом

*code/conc/psum-mutex.c*

```

1 /* Процедура потока для psum-mutex.c */
2 void *sum_mutex(void *vargp)
3 {
4     long myid = *((long *)vargp); /* Получить ID потока */
5     long start = myid * nelems_per_thread; /* Индекс начального элемента */
6     long end = start + nelems_per_thread; /* Индекс конечного элемента */
7     long i;
8
9     for (i = start; i < end; i++) {
10         P(&mutex);
11         gsum += i;
12         V(&mutex);
13     }
14     return NULL;
15 }
```

*code/conc/psum-mutex.c*

Когда мы попробовали запустить программу `psum-mutex` в системе с четырьмя ядрами, чтобы подсчитать сумму последовательности размером  $n = 2^{31}$  и измерить время его работы (в секундах) в зависимости от количества потоков, то получили неприятный сюрприз:

Версия	Количество потоков				
	1	2	4	8	16
psum-mutex	68	432	719	552	599

Мало того, что программа работает очень медленно в варианте с одним потоком, так она еще почти на порядок замедлилась в варианте с несколькими параллельными потоками. Причина такой низкой производительности заключается в дороговизне операций синхронизации ( $P$  и  $V$ ), по сравнению со стоимостью простого обновления памяти. Это подчеркивает важный урок параллельного программирования: *накладные расходы на синхронизацию обходятся дорого, и их следует по возможности избегать. Если этого нельзя избежать, накладные расходы должны компенсироваться максимально большим объемом полезных вычислений.*

Один из способов избежать операций синхронизации в нашей программе – заставить каждый дочерний поток вычислять сумму своей части последовательности в локальной переменной, которая не используется никакими другими потоками, как показано в листинге 12.18. Главный поток (в этом листинге не показан) определяет глобальный массив с именем `psum`, и каждый дочерний поток  $i$  накапливает свою частичную сумму в `psum[i]`. В этом решении мы предоставляем каждому дочернему потоку свое уникальное место в памяти для обновления, поэтому отпадает необходимость защищать эти места с помощью мьютексов. Единственное место, где необходима синхронизация, – главный поток должен дожидаться завершения всех дочерних потоков, чтобы потом сложить элементы вектора `psum` и получить окончательный результат.

**Листинг 12.18.** Процедура потока для psum-array. Каждый дочерний поток накапливает свою частичную сумму в отдельном элементе массива, который не используется никакими другими потоками

```
code/conc/psum-array.c
1 /* Процедура потока для psum-array.c */
2 void *sum_array(void *vargp)
3 {
4     long myid = *((long *)vargp); /* Получить ID потока */
5     long start = myid * nelems_per_thread; /* Индекс начального элемента */
6     long end = start + nelems_per_thread; /* Индекс конечного элемента */
7     long i;
8
9     for (i = start; i < end; i++) {
10         psum[myid] += i;
11     }
12     return NULL;
13 }
```

code/conc/psum-array.c

Запустив программу psum-array в системе с четырьмя ядрами, мы увидели увеличение производительности на порядки, по сравнению с psum-mutex:

Версия	Количество потоков				
	1	2	4	8	16
psum-mutex	68,00	432,00	719,00	552,00	599,00
psum-array	7,26	3,64	1,91	1,85	1,84

В главе 5 мы узнали, как использовать локальные переменные для устранения ненужных ссылок на память. В листинге 12.19 показано, как применить этот принцип, чтобы каждый дочерний поток накапливал свою частичную сумму в локальной, а не в глобальной переменной. Запустив программу psum-local на нашей четырехъядерной машине, мы получаем еще одно уменьшение времени выполнения почти на два порядка:

Версия	Количество потоков				
	1	2	4	8	16
psum-mutex	68,00	432,00	719,00	552,00	599,00
psum-array	7,26	3,64	1,91	1,85	1,84
psum-local	1,06	0,54	0,28	0,29	0,30

**Листинг 12.19** Процедура потока для psum-local. Каждый дочерний поток накапливает свою частичную сумму в локальной переменной

```
code/conc/psum-local.c
1 /* Процедура потока для psum-local.c */
2 void *sum_local(void *vargp)
3 {
4     long myid = *((long *)vargp); /* Получить ID потока */
5     long start = myid * nelems_per_thread; /* Индекс начального элемента */
6     long end = start + nelems_per_thread; /* Индекс конечного элемента */
```

```

7   long i, sum = 0;
8
9   for (i = start; i < end; i++) {
10      sum += i;
11   }
12   psum[myid] = sum;
13   return NULL;
14 }

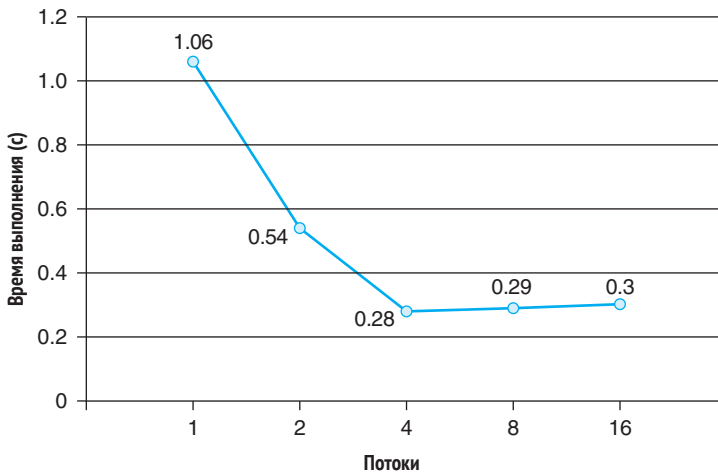
```

*code/conc/psum-local.c*

Из этого примера вытекает важный урок: разработка параллельных программ – непростая задача. Минимальные, на первый взгляд, изменения оказывают поразительное влияние на производительность.

### Оценка производительности параллельных программ

На рис. 12.15 показан график зависимости времени выполнения программы `psum-local` (листинг 12.19) от количества потоков. Во всех случаях программа выполняется в четырехъядерной системе и суммирует последовательность из  $n = 2^{31}$  элементов. Как видите, время выполнения уменьшается с увеличением количества потоков до четырех, после чего выравнивается и даже начинает немного увеличиваться.



**Рис. 12.15.** Производительность программы `psum_local` (листинг 12.19). Суммирование последовательности из  $2^{31}$  целых чисел в системе с четырьмя ядрами

В идеальном случае время выполнения должно уменьшаться линейно с количеством ядер. То есть время выполнения должно уменьшаться вдвое с каждым удвоением количества потоков. И это действительно так до достижения точки ( $t > 4$ ), когда каждое из четырех ядер занято выполнением хотя бы одного потока. Время выполнения немного увеличивается с увеличением количества потоков из-за накладных расходов на переключение контекста между потоками на одном ядре. По этой причине параллельные программы часто пишут так, чтобы каждое ядро выполняло ровно один поток.

Хотя абсолютное время выполнения является конечной мерой производительности любой программы, существуют некоторые полезные относительные показатели, которые помогут оценить, насколько хорошо параллельная программа использует потенциальные возможности параллелизма. Ускорение параллельной программы обычно определяется как

$$S_p = \frac{T_1}{T_p},$$

где  $p$  – количество ядер процессора,  $T_p$  – время выполнения на  $p$  ядрах. Эту формулу иногда называют *формулой строгого масштабирования*. Когда  $T_1$  – это время выполнения последовательной версии программы, то метрику  $S_p$  называют *абсолютным ускорением*. Когда  $T_1$  – это время выполнения параллельной версии программы, выполняющейся на одном ядре, то  $S_p$  называют *относительным ускорением*. Абсолютное ускорение является более точной мерой преимуществ параллелизма, чем относительное. Параллельные программы часто страдают от накладных расходов на синхронизацию, даже если они выполняются на одном процессоре, и эти накладные расходы могут искусственно завышать относительные значения ускорения, поскольку они увеличивают значение в числителе. С другой стороны, абсолютное ускорение измерить труднее, чем относительное, потому что для этого требуются две разные версии программы. Для сложных параллельных программ писать еще и последовательную версию может оказаться нецелесообразным либо из-за высокой сложности кода, либо просто потому, что исходный код недоступен.

Похожая мера, известная как *эффективность*, определяется как

$$E_p = \frac{T_p}{p} = \frac{T_1}{pT_p}$$

и обычно измеряется в процентах в диапазоне (0, 100]. Эффективность – это мера накладных расходов из-за распараллеливания. Программы с высокой эффективностью тратят больше времени на выполнение полезной работы и меньше на синхронизацию и обмен данными.

В табл. 12.2 показаны различные значения показателей ускорения и эффективности для нашей параллельной программы суммирования числовой последовательности. Эффективность выше 90 % – это очень хорошо, но не все так просто. Мы смогли добиться высокой эффективности, потому что нашу задачу легко распараллелить. На практике такое бывает редко. Параллельное программирование активно исследуется на протяжении десятилетий. С появлением многоядерных процессоров, количество ядер в которых удваивается каждые несколько лет, параллельное программирование продолжает оставаться глубокой, сложной и активной областью исследований.

**Таблица 12.2.** Оценки ускорения и эффективности на основе измерений, представленных на рис. 12.15

Потоки ( $t$ )	1	2	4	8	16
Ядра ( $p$ )	1	2	4	4	4
Время выполнения ( $T_p$ )	1,06	0,54	0,28	0,29	0,30
Ускорение ( $S_p$ )	1	1,9	3,8	3,7	3,5
Эффективность ( $E_p$ )	100 %	98 %	95 %	91 %	88 %

Существует еще один взгляд на ускорение, известный как *слабое масштабирование*, когда размер задачи увеличивается вместе с количеством процессоров, так что объем работы, выполняемой каждым процессором, остается постоянным с увеличением количества процессоров. В этой формулировке ускорение и эффективность выражаются через общий объем работы, выполняемой в единицу времени. Например, если мы сможем удвоить количество процессоров и вдвое увеличить объем работы, выполняемой в единицу времени, то получим линейное ускорение и 100%-ную эффективность.



Слабое масштабирование часто является более точной мерой, чем строгое, потому что точнее отражает наше желание использовать более мощные машины для выполнения большего объема работы. Это особенно верно для научных вычислений, где размер задачи легко увеличить, а чем больше размер задачи, тем точнее прогнозы природных явлений. Однако существуют приложения, размер которых не так легко увеличить, и для оценки этих приложений лучше подходит мера строгого масштабирования. Например, объем работы, выполняемой приложениями, которые обрабатывают сигналы в реальном времени, часто определяется свойствами физических датчиков, генерирующих эти сигналы. Для увеличения общего объема работы потребуется использовать большее количество разных физических датчиков, что может оказаться неосуществимыми или нецелесообразным. Параллелизм в таких приложениях обычно используется для выполнения фиксированного объема работы в как можно более короткие сроки.

### Упражнение 12.11 (решение в конце главы)

Вычислите недостающие значения в следующей таблице в предположении строгого масштабирования.

Потоки ( $t$ )	1	2	4
Ядра ( $p$ )	1	2	4
Время выполнения ( $T_p$ )	12	8	6
Ускорение ( $S_p$ )	_____	1,5	_____
Эффективность ( $E_p$ )	100 %	_____	50 %

## 12.7. Другие вопросы конкурентного выполнения

Возможно, вы заметили, что жизнь программиста значительно усложняется, как только появляется требование синхронизировать доступ к совместно используемым данным. До сих пор авторы рассматривали приемы организации исключительного доступа и синхронизации по типу производитель–потребитель, однако это лишь верхушка айсберга. Синхронизация – это исключительно сложная и фундаментальная проблема, поднимающая вопросы, которые просто не могут возникнуть в обычных последовательных программах. В этом разделе мы рассмотрим некоторые положения, которые всегда следует учитывать при разработке параллельных программ. Чтобы придать конкретности нашему обсуждению, мы будем рассуждать в терминах потоков. Но имейте в виду, что все то же самое в равной степени относится к любой организации конкурентных потоков управления, манипулирующих общими ресурсами.

### 12.7.1. Безопасность в многопоточном окружении

В многопоточных программах мы должны уделять особое внимание безопасности функций в многопоточном окружении. Функция считается *потокобезопасной* (thread-safe), если она всегда возвращает корректные результаты при обращении к ней из множества потоков, выполняющихся конкурентно. Если в таком окружении функция может возвращать некорректные результаты, то ее называют *потоконебезопасной* (thread-unsafe).

Различают четыре (пересекающихся) класса потоконебезопасных функций.

Класс 1: *функции, не защищающие совместно используемые переменные*. Эта проблема уже встречалась нам в функции thread (листинг 12.10), которая наращивает незащищенную глобальную переменную счетчика. Функции этого

класса сравнительно легко сделать потокобезопасными: достаточно защитить совместно используемые переменные операциями синхронизации, такими как *P* и *V*. Преимущество такого решения – отсутствие необходимости изменять вызывающий код; недостаток – операции синхронизации замедляют выполнение функции.

Класс 2: *функции, сохраняющие состояние между вызовами*. Генератор псевдослучайных чисел – вот простой пример функции из этого класса. Рассмотрим пакет генератора псевдослучайных чисел в листинге 12.20.

**Листинг 12.20.** Потокобезопасный генератор псевдослучайных чисел (на основе [61])

```
code/conc/rand.c
1 unsigned next_seed = 1;
2
3 /* rand - возвращает псевдослучайное целое число в диапазоне 0..32767 */
4 unsigned rand(void)
5 {
6     next_seed = next_seed*1103515245 + 12543;
7     return (unsigned)(next_seed>>16) % 32768;
8 }
9
10 /* srand - устанавливает начальное значение для генератора rand() */
11 void srand(unsigned new_seed)
12 {
13     next_seed = new_seed;
14 }
```

code/conc/rand.c

Функция `rand` является потокобезопасной, потому что результат текущего вызова зависит от промежуточного результата в предыдущей итерации. При многократном вызове `rand` из одного потока после вызова `srand` можно ожидать получения повторяющихся последовательностей чисел. Однако данное предположение не оправдывается, если `rand` вызывается из нескольких потоков.

Единственный способ сделать такую функцию потокобезопасной – переписать ее так, чтобы она не использовала никаких статических данных, а информация о состоянии передавалась в аргументах. Недостаток подобного решения – программист вынужден изменить не только саму функцию, но и вызывающий ее код. В большой программе, где функция может вызываться из сотни мест, внесение таких изменений может оказаться сложной задачей, подверженной ошибкам.

Класс 3: *функции, возвращающие указатель в статической переменной*. Некоторые функции, такие как `gethostbyname`, сохраняют результат в статической переменной и возвращают указатель на нее. Вызов таких функций из конкурентных потоков может закончиться катастрофой, потому что результаты, вычисленные для одного потока, могут быть затерты результатами, вычисленными для другого потока.

Есть два решения этой проблемы. Первое – переписать функцию так, чтобы вызывающая программа передавала адрес переменной, куда следует сохранить результат. Это устраняет из цепочки вычислений все совместно используемые данные, однако требует от программиста изменить также вызывающий код.

Если потоконебезопасную функцию трудно или невозможно модифицировать (например, она находится в библиотеке), то можно порекомендовать альтернативный прием *блокировки и копирования*. Идея заключается в том, чтобы в каждой точке вызова потоконебезопасной функции выполнить такую последовательность действий: захватить мьютекс, вызвать потоконебезопасную функцию, динамически выделить память для результата, скопировать в нее результат и освободить мьютекс. Другой довольно привлекательный способ: определить потокобезопасную функцию-обертку, выполняющую блокировку и копирование, а затем заменить все вызовы небезопасной функции вызовами функции-обертки. Например, в листинге 12.21 представлена потокобезопасная версия `gethostbyname`, использующая прием блокировки и копирования.

**Листинг 12.21.** Потокобезопасная функция-обертка вокруг функции `ctime` из стандартной библиотеки языка C. В этом примере используется прием блокировки и копирования при использовании функций из третьего класса потоконебезопасных функций

```
code/conc/ctime-ts.c
1 char *ctime_ts(const time_t *timep, char *privatep)
2 {
3     char *sharedp;
4
5     P(&mutex);
6     sharedp = ctime(timep);
7     strcpy(privatep, sharedp); /* Скопировать из общей памяти в приватную */
8     V(&mutex);
9     return privatep;
10 }
```

code/conc/ctime-ts.c

Класс 4: *функции, вызывающие потоконебезопасные функции*. Если функция  $f$  вызывает потоконебезопасную функцию  $g$ , то можно ли считать функцию  $f$  потоконебезопасной? Как сказать... Если  $g$  – функция из класса 2, сохраняющего состояние между вызовами, то  $f$  тоже будет потоконебезопасной, и у вас не будет иного выхода, как переписать  $g$ . Однако если функция  $g$  принадлежит классу 1 или 3, тогда  $f$  может оставаться потокобезопасной, если все вызовы и совместно используемые результаты будут защищены мьютексом. Хороший пример такой ситуации показан в листинге 12.21, демонстрирующем прием блокировки и копирования для написания безопасной функции, вызывающей небезопасную.

## 12.7.2. Реентерабельность

Существует важный класс потокобезопасных функций, называемых *реентерабельными*, характеризующихся тем, что они не используют никаких общих данных. Несмотря на то что термины *потокобезопасная* и *реентерабельная* иногда (неверно) используют как синонимы, между ними существует четкое разделение, которое следует учитывать. На рис. 12.16 показано, как связаны потокобезопасные и потоконебезопасные функции. Множество реентерабельных функций – это подмножество потокобезопасных функций.

Как правило, реентерабельные функции более эффективны, чем нереентерабельные потокобезопасные функции, потому что не требуют операций синхронизации. Более того, единственный способ преобразовать небезопасную функцию класса 2 в безопас-

ную – переписать ее так, чтобы она стала реентерабельной. Например, в листинге 12.22 показана реентерабельная версия функции `rand` из листинга 12.20. Идея состоит в том, чтобы заменить статическую переменную: `next_seed` замещается указателем, передаваемым вызывающей программой.

**Листинг 12.22.** `rand_r`: реентерабельная версия функции `rand` из листинга 12.20

```
1 /* rand_r - возвращает псевдослучайное целое число в диапазоне 0..32767 */
2 int rand_r(unsigned int *nextp)
3 {
4     *nextp = *nextp * 1103515245 + 12345;
5     return (unsigned int)(*nextp / 65536) % 32768;
6 }
```

[code/conc/rand-r.c](#)

[code/conc/rand-r.c](#)

Все функции



**Рис. 12.16.** Связь между множествами реентерабельных, потокобезопасных и потокобезопасных функций

Возможно ли, изучив код некоторой функции, объявить ее явно реентерабельной? К сожалению, это зависит от разных факторов. Если все аргументы функции передаются по значению (т. е. не по ссылкам) и все используемые данные хранятся в локальных автоматических переменных в стеке (т. е. функция не обращается к статическим или глобальным переменным), то такую функцию можно смело назвать реентерабельной – ее реентерабельность можно подтвердить независимо от способа обращения к ней.

Однако если сделать допущения менее строгими и позволить передачу некоторых параметров по ссылкам, то такая функция будет *неявно реентерабельной*, в том смысле, что реентерабельной она будет, только если потоки аккуратно передают указатели на их локальные данные. К примеру, функция `rand_r` в листинге 12.22 является неявно реентерабельной.

Используя термин *реентерабельная* в этой книге, мы всегда подразумеваем как явно, так и неявно реентерабельные функции. Однако важно понимать, что реентерабельность иногда является свойством как вызывающей программы, так и вызываемой.

#### Упражнение 12.12 (решение в конце главы)

Функция `ctime_ts` в листинге 12.21 является потокобезопасной, но нереентерабельной. Объясните почему.

### 12.7.3. Использование библиотечных функций в многопоточных программах

Большинство функций Linux, включая функции из стандартной библиотеки C (`malloc`, `free`, `realloc`, `printf`, `scanf` и др.), являются потокобезопасными. В табл. 12.3 перечислены основные исключения. (Полный список вы найдете в [110]). Функция `strtok` – устаревшая (использовать ее не рекомендуется) и используется для парсинга строк. Функ-

ции `asctime`, `ctime` и `localtime` обычно применяются для преобразования в/из разных форматов времени и дат. Функции `gethostbyname`, `gethostbyaddr` и `inet_ntoa` – устаревшие функции, используемые в сетевом программировании, на смену которым пришли реентерабельные версии `getaddrinfo`, `getnameinfo` и `inet_ntop` (глава 11). За исключением `rand` и `strtok`, все эти потокобезопасные функции относятся к классу 3 и возвращают указатель на статическую переменную. При необходимости использовать одну из этих функций в многопоточной программе рекомендуется применять прием блокировки и копирования. Однако этот прием имеет ряд недостатков. Во-первых, операции синхронизации замедляют выполнение программы. Во-вторых, функции, возвращающие указатели на сложные вложенные структуры, требуют *глубокого копирования* всей иерархии структур, возвращаемой в результате. В-третьих, прием блокировки и копирования неприменим к потокобезопасным функциям из класса 2, таким как `rand`, которые сохраняют свое состояние между вызовами.

**Таблица 12.3.** Некоторые потокобезопасные библиотечные функции

Потокобезопасная функция	Класс потокобезопасных функций	Потокобезопасная версия в Linux
<code>rand</code>	2	<code>rand_r</code>
<code>strtok</code>	2	<code>strtok_r</code>
<code>asctime</code>	3	<code>asctime_r</code>
<code>ctime</code>	3	<code>ctime_r</code>
<code>gethostbyaddr</code>	3	<code>gethostbyaddr_r</code>
<code>gethostbyname</code>	3	<code>gethostbyname_r</code>
<code>inet_ntoa</code>	3	–
<code>localtime</code>	3	<code>localtime_r</code>

По этим причинам в Linux реализованы реентерабельные версии большинства потокобезопасных функций. Имена реентерабельных функций всегда оканчиваются на `_r`. Например, реентерабельная версия `asctime` называется `asctime_r`. Мы советуем всегда использовать реентерабельные версии.

### 12.7.4. Состояние гонки

*Состояние гонки* возникает, когда корректность программы зависит от соблюдения условия, требующего, чтобы один поток достиг точки *x* в своем потоке управления раньше, чем другой поток достигнет точки *y*. Состояние гонки обычно возникает, когда программист предполагает, что потоки выберут какую-то особую траекторию в пространстве состояний, забывая правило, гласящее, что многопоточные программы должны работать корректно при выборе любой вероятной траектории.

Природу состояния гонки проще понять на примере. Рассмотрим простую программу в листинге 12.23.

**Листинг 12.23.** Программа с состоянием гонки

*code/conc/race.c*

```

1 /* ВНИМАНИЕ: этот код содержит ошибку! */
2 #include "csapp.h"
3 #define N 4
4
5 void *thread(void *vargp);
6
```

```

7 int main()
8 {
9     pthread_t tid[N];
10    int i;
11
12    for (i = 0; i < N; i++)
13        Pthread_create(&tid[i], NULL, thread, &i);
14    for (i = 0; i < N; i++)
15        Pthread_join(tid[i], NULL);
16    exit(0);
17 }
18
19 /* Процедура потока */
20 void *thread(void *vargp)
21 {
22     int myid = *((int *)vargp);
23     printf("Hello from thread %d\n", myid);
24     return NULL;
25 }

```

*code/conc/race.c*

Главный поток создает четыре дочерних потока и каждому передает указатель на его уникальный целочисленный идентификатор. Каждый дочерний поток копирует этот идентификатор в локальную переменную (строка 22) и выводит сообщение, содержащее идентификатор. Программа выглядит довольно простой, однако, запустив ее у себя, мы получили некорректный результат:

```

linux> ./race
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3

```

Проблема вызвана состоянием гонки между дочерними и главным потоками. Можете заметить причину? Вот что происходит в действительности: когда главный поток создает дочерний поток в строке 13, он передает указатель на локальную переменную *i* в стеке. На этом этапе имеет место гонка между увеличением переменной *i* в строке 12 и разыменованием аргумента в строке 22. Если дочерний поток успеет выполнить строку 22 до того, как главный поток выполнит строку 12, то переменная *myid* получит корректный идентификатор. Иначе она будет содержать идентификатор какого-то другого потока. Самое досадное в этой ситуации, что получение корректного ответа зависит от особенностей планирования потоков в ядре. В системе, описанной в этой книге, мы не смогли получить корректный ответ, однако в других системах такая программа иногда может работать корректно, и программист окажется в блаженном неведении о наличии в его продукте серьезной ошибки.

Чтобы избавиться от состояния гонки, можно динамически выделить отдельный блок для каждого целочисленного идентификатора и передать указатель на блок в процедуру потока, как показано в листинге 12.24 (строки 12–14). Обратите внимание, что процедура потока должна освобождать блок памяти, чтобы избежать утечек памяти.

#### Листинг 12.24. Корректная версия программы из листинга 12.23

*code/conc/norace.c*

```

1 #include "csapp.h"
2 #define N 4
3

```

```

4 void *thread(void *vargp);
5
6 int main()
7 {
8     pthread_t tid[N];
9     int i, *ptr;
10
11     for (i = 0; i < N; i++) {
12         ptr = Malloc(sizeof(int));
13         *ptr = i;
14         Pthread_create(&tid[i], NULL, thread, ptr);
15     }
16     for (i = 0; i < N; i++)
17         Pthread_join(tid[i], NULL);
18     exit(0);
19 }
20
21 /* Процедура потока */
22 void *thread(void *vargp)
23 {
24     int myid = *((int *)vargp);
25     Free(vargp);
26     printf("Hello from thread %d\n", myid);
27     return NULL;
28 }

```

*code/conc/norace.c*

Запустив эту версию в своей системе, мы получили корректный результат:

```

linux> ./norace
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3

```

### Упражнение 12.13 (решение в конце главы)

В листинге 12.24 может возникнуть соблазн освободить выделенный блок памяти сразу после строки 14 в главном потоке, а не в дочернем. Но это будет неправильно. Почему?

### Упражнение 12.14 (решение в конце главы)

1. В листинге 12.24 мы устранили состояние гонки распределением отдельного блока памяти для каждого целочисленного идентификатора. Предложите другой способ, при котором не потребуется обращаться к функциям `malloc` и `free`.
2. Каковы преимущества и недостатки этого подхода?

## 12.7.5. Взаимоблокировка (тупиковые ситуации)

Семафоры потенциально могут привести к одной из ошибок времени выполнения, называемой *взаимоблокировкой*, когда потоки блокируются, ожидая условия, которое никогда не выполнится. Для понимания природы взаимоблокировок (тупиковых ситуа-

ций) ценнейшим инструментом является граф выполнения. Например, на рис. 12.17 показан граф выполнения пары потоков, использующих два семафора для организации исключительного доступа к данным. Из этого графа можно вывести несколько важных выводов, помогающих понять явление взаимной блокировки:

- программист некорректно упорядочил операции  $P$  и  $V$  так, что это привело к наложению запрещенных областей двух семафоров. Если случится, что какая-то траектория выполнения достигнет *состояния взаимоблокировки  $d$* , то дальнейшее выполнение окажется невозможным, потому что в зоне перекрытия запрещенных областей выполнение будет заблокировано в любом допустимом направлении. Другими словами, программа попадает в тупиковую ситуацию из-за того, что каждый поток ожидает, пока другой выполнит операцию  $V$ , чего никогда не произойдет;
- зона перекрытия запрещенных областей охватывает набор состояний, называемый областью тупиковой ситуации (или взаимоблокировки). Если траектория соприкоснется с состоянием в зоне тупиковой ситуации, то взаимоблокировка будет неизбежна. Траектории могут входить в зоны тупиковых ситуаций, но никогда не смогут выйти из них;
- взаимоблокировка – это особенно сложная проблема, потому что ее не всегда можно спрогнозировать. Некоторые «удачные» траектории выполнения обогнут зону тупиковой ситуации; другим же не повезет. На рис. 12.17 есть примеры обеих ситуаций. Последствия для программиста могут оказаться, мягко говоря, не очень приятными. Программа может без проблем выполниться 1000 раз, а на 1001-й возникнет тупиковая ситуация. Или программа будет прекрасно работать на одной машине, но «зависать» на другой. Хуже всего, что воспроизвести эту ошибку удается далеко не всегда, потому что разные выполнения имеют разные траектории.

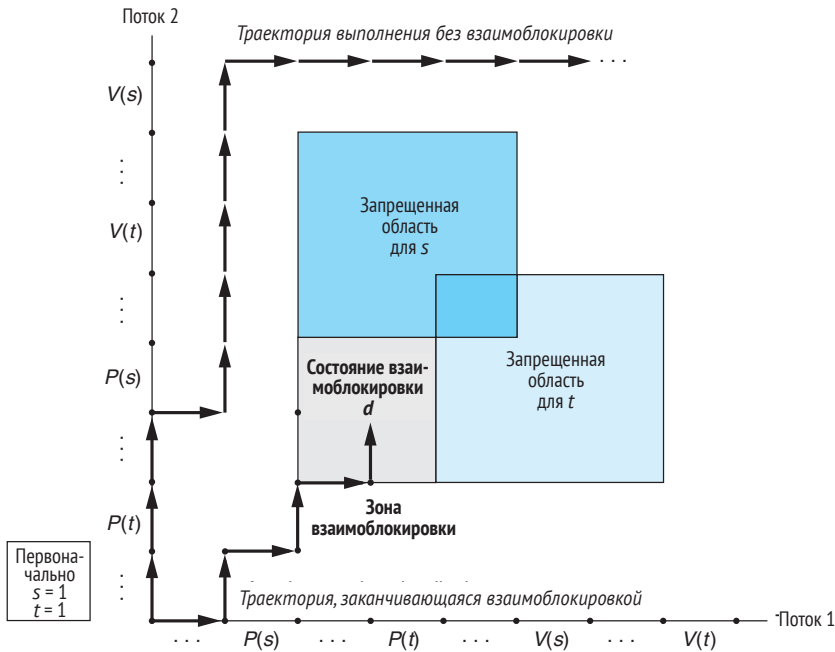


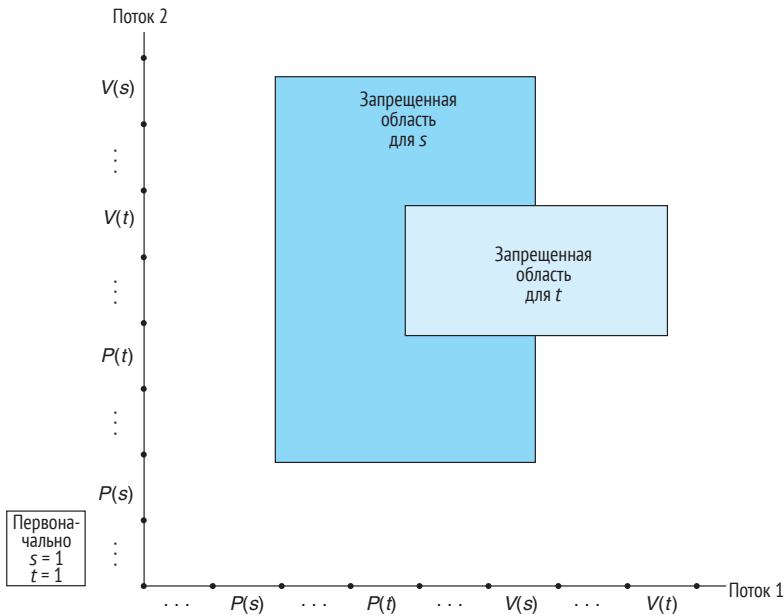
Рис. 12.17. Граф выполнения для программы с проблемой взаимоблокировки



Программы попадают в тупиковые ситуации по многим причинам, и избежать их – задача достаточно сложная. Впрочем, при использовании бинарных семафоров, как в случае на рис. 12.19, можно применить следующее простое и эффективное правило:

*Всегда захватывать и освобождать мьютексы в одном и том же порядке: если все потоки в программе захватывают мьютексы всегда в одном и том же порядке и освобождают их в обратном порядке, то состояние взаимоблокировки в такой программе не возникает.*

Например, состояния взаимоблокировки, показанного на рис. 12.17, можно избежать, если потоки будут захватывать сначала мьютекс  $s$ , а затем  $t$ . На рис. 12.18 показан получившийся в результате граф выполнения.



**Рис. 12.18.** Граф выполнения для программы, не страдающей проблемой взаимоблокировки

### Упражнение 12.15 (решение в конце главы)

Взгляните на следующую программу, в которой используется пара семафоров для организации исключительного доступа к общим данным:

Первоначально:  $s = 1$ ,  $t = 0$ .

Поток 1:	Поток 2:
$P(s)$ ;	$P(s)$ ;
$V(s)$ ;	$V(s)$ ;
$P(t)$ ;	$P(t)$ ;
$V(t)$ ;	$V(t)$ ;

1. Нарисуйте граф выполнения для этой программы.
2. Всегда ли будет возникать тупиковая ситуация?
3. Если да, то какое простое изменение первоначальных значений семафоров устранил риск тупиковой ситуации?

4. Нарисуйте граф выполнения для получившейся программы, не страдающей проблемой взаимоблокировки.

## 12.8. Итоги

Конкурентная программа состоит из набора логических потоков управления, перекрывающихся во времени. В этой главе мы исследовали три разных механизма, используемых при построении конкурентных программ: процессы, мультиплексирование ввода/вывода и потоки. На всем протяжении дискуссии в качестве основного примера использовался конкурентный сетевой сервер.

Процессы автоматически планируются ядром, и так как они имеют изолированные адресные пространства, для применения общих данных им приходится использовать механизмы межпроцессных взаимодействий (IPC). Событийно-ориентированные программы создают свои конкурентные потоки управления, основанные на модели конечных автоматов, а для явного планирования потоков используют мультиплексирование ввода/вывода. Так как такие программы выполняются в рамках одного процесса, совместное использование данных между потоками управления происходит быстро и просто. Подход на основе потоков выполнения являет собой симбиоз (гибрид) двух предыдущих подходов. Подобно процессам, потоки выполнения автоматически планируются ядром. Подобно логическим потокам управления на основе мультиплексирования ввода/вывода, потоки выполнения действуют в контексте одного процесса и могут быстро и просто использовать общие данные.

Независимо от механизма управления, синхронизация конкурентного доступа к общим данным представляет собой сложную проблему. В помощь нам были разработаны операции *P* и *V* с семафорами. Операции с семафорами можно использовать для организации исключительного доступа к общим данным, а также для совместного использования таких ресурсов, как буферы в программах, построенных на основе модели производитель–потребитель. Конкурентный эхо-сервер с предварительно подготовленными потоками выполнения предоставляет убедительный пример этих двух сценариев использования семафоров.

Конкуренция является источником многих других сложностей. Функции, вызываемые потоками, должны обладать свойством потокобезопасности. Мы определили четыре класса потокобезопасных функций и рассмотрели предложения, реализация которых поможет добиться их безопасности в многопоточном окружении. Реентерабельные функции – подмножество потокобезопасных функций, не имеющих доступа к каким бы то ни было общим данным. Реентерабельные функции часто более эффективны, чем нереентерабельные, потому что не требуют синхронизации. В числе других сложностей, возникающих в конкурентных программах, можно назвать состояния гонки и взаимоблокировки. Состояние гонки возникает, когда программист делает некорректные допущения о том, как ядро планирует потоки выполнения. Взаимоблокировки возникают, когда поток ожидает события, которое никогда не произойдет.

## Библиографические заметки

Операции с семафорами предложены Дейкстрой (Dijkstra) [31]. Концепция графов выполнения представлена Коффманом (Coffman) [23] и позже формализована Карсоном (Carson) и Рейнольдсом (Reynolds) [16]. Проблема читателей–писателей описана Куртуа (Courtois) с коллегами в [25]. В книгах об операционных системах подробно описывают классические проблемы синхронизации, такие как проблемы обедающих философов, спящего парикмахера и курильщика [102, 106, 113]. Книга Бутенхофа (Butenhof) [15] включает подробное описание интерфейса переносимой операционной системы

(POSIX). Статья Биррелла (Birrell) [90] – прекрасное введение в многопоточное программирование и его проблемы. В книге Рейндерса (Reinders) [90] описывается библиотека C/C++, упрощающая проектирование и реализацию многопоточных программ. Есть книги, охватывающие основы параллельного программирования на многоядерных системах [47, 71]. Пью (Pugh) описывает слабые стороны методов взаимодействий между потоками выполнения Java с использованием памяти и предлагает модели замещения памяти [88]. Густафсон (Gustafson) предложил модель ускорения слабого масштабирования [43] в качестве альтернативы строгому масштабированию.

## Домашние задания

### Упражнение 12.16 ♦

Напишите версию программы `hello.c`, создающую и утилизирующую  $n$  присоединяемых дочерних потоков, где  $n$  – аргумент командной строки.

### Упражнение 12.17 ♦

1. В листинге 12.25 есть ошибка. Предполагается, что поток приостанавливается на 1 секунду, а затем выводит строку. Однако когда мы запустили эту программу в своей системе, она ничего не вывела. Почему?
2. Эту ошибку можно исправить, заменив функцию `exit` в строке 10 одним из двух разных вызовов функций Pthreads. Что это за функции?

**Листинг 12.25.** Программа с ошибкой для упражнения 12.17

*code/conc/hellobug.c*

```

1 /* ВНИМАНИЕ: этот код содержит ошибку! */
2 #include "csapp.h"
3 void *thread(void *vargp);
4
5 int main()
6 {
7     pthread_t tid;
8
9     Pthread_create(&tid, NULL, thread, NULL);
10    exit(0);
11 }
12
13 /* Процедура потока */
14 void *thread(void *vargp)
15 {
16     Sleep(1);
17     printf("Hello, world!\n");
18     return NULL;
19 }
```

*code/conc/hellobug.c*

### Упражнение 12.18 ♦

Используя граф выполнения на рис. 12.10, классифицируйте следующие траектории как безопасные или небезопасные:

1.  $H_2, L_2, U_2, H_1, L_1, S_2, U_1, S_1, T_1, T_2$
2.  $H_2, H_1, L_1, U_1, S_1, L_2, T_1, U_2, S_2, T_2$
3.  $H_1, L_1, H_2, L_2, U_2, S_2, U_1, S_1, T_1, T_2$

### Упражнение 12.19 ♦♦

Решение первой проблемы читателей–писателей в листинге 12.13 отдает слабое предпочтение потокам–читателям, потому что поток–писатель, покидающий свою критическую секцию, может возобновить ожидающий поток–писатель вместо потока–читателя. Найдите решение, отдающее предпочтение потокам–читателям, чтобы поток–писатель, покидающий свою критическую секцию, всегда возобновлял ожидающий поток–читатель, если он существует.

### Упражнение 12.20 ♦♦♦

Рассмотрим более простой вариант задачи о читателях–писателях, когда имеется не более  $N$  потоков–читателей. Найдите решение, отдающее равное предпочтение потокам–читателям и потокам–писателям, в том смысле, что ожидающие потоки–читатели и потоки–писатели имеют равные шансы на получение доступа к ресурсу. *Подсказка:* эту задачу можно решить, используя один счетный семафор и один мьютекс.

### Упражнение 12.21 ♦♦♦♦

Найдите решение второй проблемы читателей–писателей, когда предпочтение отдается потокам–писателям, а не читателям.

### Упражнение 12.22 ♦♦

Проверьте свое понимание функции `select`, изменив сервер в листинге 12.2 так, чтобы он отображал не более одной текстовой строки в каждой итерации цикла сервера.

### Упражнение 12.23 ♦♦

Событийно-ориентированный конкурентный эхо-сервер в листинге 12.3 имеет уязвимость, позволяющую клиенту–злоумышленнику вызвать отказ в обслуживании других клиентов отправкой неполной текстовой строки. Напишите усовершенствованную версию сервера, который смог бы обрабатывать такие неполные текстовые строки без блокировки.

### Упражнение 12.24 ♦

Функции ввода/вывода в пакете `RIO` (раздел 10.5) безопасны в многопоточном окружении. Являются ли они также реентерабельными?

### Упражнение 12.25 ♦

В конкурентном эхо-сервере с предварительно созданным пулом потоков каждый поток вызывает функцию `echo_cnt` (листинг 12.15). Является ли функция `echo_cnt` потокобезопасной? Является ли она реентерабельной? Почему да или почему нет?

### Упражнение 12.26 ♦♦♦

Используйте прием блокировки и копирования для реализации потокобезопасной нереентерабельной версии `gethostbyname` с именем `gethostbyname_ts`. Правильное решение должно выполнять глубокое копирование структуры `hostent` под защитой мьютекса.

### Упражнение 12.27 ♦♦

В некоторых руководствах по сетевому программированию предлагается следующий подход для реализации чтения и записи с сокетами: перед взаимодействием с клиентом откройте два стандартных потока ввода/вывода на одном дескрипторе сокета соединения: один – для чтения, а другой – для записи:

```
FILE *fpin, *fpout;

fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

Когда сервер закончит взаимодействие с клиентом, закройте оба потока:

```
fclose(fpin);
fclose(fpout);
```

Однако если применить это предложение в конкурентном сервере, основанном на потоках, возникнет состояние гонки. Объясните почему.

### Упражнение 12.28 ♦

Обусловит переменная мест двух операций  $V$ , показанных на рис. 12.18, появление тупиковых ситуаций в программе или нет? Обоснуйте ответ изображением графа выполнения для четырех возможных случаев:

Случай 1		Случай 2		Случай 3		Случай 4	
Поток 1	Поток 2	Поток 1	Поток 2	Поток 1	Поток 2	Поток 1	Поток 2
$P(s)$	$P(s)$	$P(s)$	$P(s)$	$P(s)$	$P(s)$	$P(s)$	$P(s)$
$P(t)$	$P(t)$	$P(t)$	$P(t)$	$P(t)$	$P(t)$	$P(t)$	$P(t)$
$V(s)$	$V(s)$	$V(s)$	$V(t)$	$V(t)$	$V(s)$	$V(t)$	$V(t)$
$V(t)$	$V(t)$	$V(t)$	$V(s)$	$V(s)$	$V(t)$	$V(s)$	$V(s)$

### Упражнение 12.29 ♦

Может ли следующая программа попасть в тупиковую ситуацию? Почему да или почему нет?

Первоначально:  $a = 1$ ,  $b = 1$ ,  $c = 1$ .

Поток 1:	Поток 2:
$P(a)$ ;	$P(c)$ ;
$P(b)$ ;	$P(b)$ ;
$V(b)$ ;	$V(b)$ ;
$P(c)$ ;	$V(c)$ ;
$V(c)$ ;	
$V(a)$ ;	

### Упражнение 12.30 ♦

Следующая программа может попасть в тупиковую ситуацию.

Первоначально:  $a = 1$ ,  $b = 1$ ,  $c = 1$ .

Поток 1:	Поток 2:	Поток 3:
$P(a)$ ;	$P(c)$ ;	$P(c)$ ;
$P(b)$ ;	$P(b)$ ;	$V(c)$ ;
$V(b)$ ;	$V(b)$ ;	$P(b)$ ;
$P(c)$ ;	$V(c)$ ;	$P(a)$ ;
$V(c)$ ;	$P(a)$ ;	$V(a)$ ;
$V(a)$ ;	$V(a)$ ;	$V(b)$ ;

1. Перечислите, какие пары мьютексов каждый поток может удерживать одновременно.
2. Если  $a < b < c$ , то какой поток нарушает правило захвата мьютексов в определенном порядке?
3. Покажите для данных потоков новый порядок блокировки, который гарантирует отсутствие тупиковых ситуаций.

### Упражнение 12.31 ♦♦♦

Реализуйте `tfgets`, версию стандартной функции `fgets`, которая автоматически возвращает `NULL`, если не получит строку из стандартного ввода в течение 5 секунд. Функция должна быть реализована в пакете с названием `tfgets-proc.c` и использовать процессы, сигналы и нелокальные переходы. Она не должна использовать функцию `alarm`. Протестируйте решение, используя программу в листинге 12.26.

**Листинг 12.26.** Тестовая программа для упражнений 12.31–12.33

```
code/conc/tfgets-main.c
1 #include "csapp.h"
2
3 char *tfgets(char *s, int size, FILE *stream);
4
5 int main()
6 {
7     char buf[MAXLINE];
8
9     if (tfgets(buf, MAXLINE, stdin) == NULL)
10         printf("BOOM!\n");
11     else
12         printf("%s", buf);
13
14     exit(0);
15 }
```

code/conc/tfgets-main.c

### Упражнение 12.32 ♦♦♦

Реализуйте версию функции `tfgets` из упражнения 12.31, которая использует функцию `select`. Функция должна быть реализована в пакете с названием `tfgets-select.c`. Протестируйте решение, используя программу в листинге 12.26. Допускается предположить, что стандартный ввод связан с дескриптором 0.

### Упражнение 12.33 ♦♦♦

Реализуйте многопоточную версию функции `tfgets` из упражнения 12.31. Функция должна быть реализована в пакете с названием `tfgets-thread.c`. Протестируйте решение, используя программу в листинге 12.26.

### Упражнение 12.34 ♦♦♦

Напишите параллельную многопоточную версию функции умножения матриц  $N \times M$ . Сравните ее производительность с производительностью последовательной версии.

### Упражнение 12.35 ♦♦♦

Реализуйте конкурентную версию веб-сервера TINY на основе процессов. Ваше решение должно создавать новый дочерний процесс для обработки каждого нового запроса на соединение. Протестируйте решение с помощью настоящего веб-браузера.

### Упражнение 12.36 ♦♦♦

Реализуйте конкурентную версию веб-сервера TINY на основе мультиплексирования ввода/вывода. Протестируйте решение с помощью настоящего веб-браузера.

### Упражнение 12.37 ♦♦♦

Реализуйте конкурентную версию веб-сервера TINY на основе потоков. Ваше решение должно создавать новый дочерний поток для обработки каждого нового запроса на соединение. Протестируйте решение с помощью настоящего веб-браузера.

### Упражнение 12.38 ♦♦♦♦

Реализуйте конкурентную версию веб-сервера TINY с предварительно созданным пулом потоков. Ваше решение должно динамически увеличивать или уменьшать количество потоков в пуле в зависимости от текущей нагрузки. Одна из возможных стратегий – удвоение числа потоков при заполнении буфера дескрипторов и сокращение наполовину при опустошении буфера. Протестируйте решение с помощью настоящего веб-браузера.

### Упражнение 12.39 ♦♦♦♦

Веб-прокси – это программа, играющая роль посредника между веб-сервером и браузером. В этой схеме браузер контактирует не с веб-сервером, а с веб-прокси, который пересылает запрос серверу. Сервер возвращает ответ веб-прокси, а тот пересылает его браузеру. Напишите простой веб-прокси, поддерживающий возможность фильтрации и регистрации запросов.

1. Для начала реализуйте веб-прокси, который принимает запросы, анализирует их, пересылает серверу и возвращает результаты браузеру. Прокси должен регистрировать URL всех запросов в системном журнале на диске, а также блокировать запросы любых URL, содержащихся в файле фильтра на диске.
2. Затем добавьте в прокси возможность обслуживать одновременно множество запросов путем создания отдельного потока для каждого запроса. Ожидая ответа от удаленного сервера, прокси должен продолжать обрабатывать незавершенные запросы от других браузеров.

Проверьте свою реализацию с настоящим веб-сервером.

## Решения упражнений

### Решение упражнения 12.1

Когда родительский процесс создает новый дочерний процесс, тот получает копию дескриптора сокета открытого соединения и счетчик ссылок на соответствующую запись в таблице файлов увеличивается с 1 до 2. Когда родитель закрывает свою копию дескриптора, количество ссылок уменьшается с 2 до 1. Поскольку ядро не закрывает файлы, пока счетчик ссылок в таблице открытых файлов не уменьшится до нуля, дескриптор в дочернем процессе остается открытым.

### Решение упражнения 12.2

Когда по какой-либо причине процесс завершается, ядро закрывает все открытые им дескрипторы. Поэтому копия дескриптора в дочернем процессе будет закрыта автоматически по его завершении.

### Решение упражнения 12.3

Напомним, что дескриптор считается готовым к чтению, если запрос на чтение одного байта из этого дескриптора не будет заблокирован. По достижении конца файла дескриптор тоже считается готовым к чтению, потому что операция чтения в этом слу-

чае вернет нулевой результат, сообщаящий о достижении конца файла. Таким образом, ввод **Ctrl+D** заставляет функцию `select` вернуть дескриптор 0 в множестве дескрипторов, готовых к чтению.

### Решение упражнения 12.4

Переменная `pool.ready_set` повторно инициализируется перед каждым вызовом `select`, потому что она служит и входным, и выходным аргументом. На входе в ней передается множество дескрипторов для наблюдения, а на выходе – множество дескрипторов, готовых к чтению.

### Решение упражнения 12.5

Поскольку потоки выполняются в одном и том же процессе, все они используют одну и ту же таблицу дескрипторов. Независимо от количества потоков, использующих дескриптор соединения, счетчик ссылок на соответствующую запись в таблице файлов равен единице. Поэтому одной операции `close` достаточно для освобождения ресурсов, связанных с дескриптором, когда приложение заканчивает с ним работу.

### Решение упражнения 12.6

Основная идея в том, что переменные на стеке являются локальными для потока, тогда как глобальные и статические переменные являются общими. Такие статические переменные, как `cnt`, довольно коварны, потому что совместное использование их ограничено рамками функций, где они объявлены, в данном случае – рамками процедуры потоков.

1. Вот заполненная таблица:

Экземпляр переменной	Доступен		
	главному потоку	дочернему потоку 0	дочернему потоку 1
<code>ptr</code>	да	да	да
<code>cnt</code>	нет	да	да
<code>i.m</code>	да	нет	нет
<code>msgs.m</code>	да	да	да
<code>myid.p0</code>	нет	да	нет
<code>myid.p1</code>	нет	нет	да

Дополнительные примечания:

- `ptr` – глобальная переменная, изменяется главным потоком и читается дочерними потоками;
- `cnt` – статическая переменная с единственным экземпляром в памяти; используется для чтения и записи обоими дочерними потоками;
- `i.m` – локальная автоматическая переменная, хранящаяся в стеке главного потока. Несмотря на то что ее значение передается дочерним потокам, они никогда не обращаются к ней напрямую, и, следовательно, она не является совместно используемой;
- `msgs.m` – локальная автоматическая переменная, хранящаяся в стеке главного потока. К ней косвенно обращаются оба дочерних потока через `ptr`;
- `myid.p0` и `myid.p1` – экземпляры локальной автоматической переменной, хранящиеся в стеках дочерних потоков 0 и 1 соответственно.

2. Переменные `ptr`, `cnt` и `msgs` используются несколькими потоками, следовательно, они являются совместно используемыми.



### Решение упражнения 12.7

Важно помнить, что нельзя делать никаких предположений о том, в каком порядке ядро будет планировать потоки.

Шаг	Поток	Инстр.	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
1	1	$H_1$	–	–	0
2	1	$L_1$	0	–	0
3	2	$H_2$	–	–	0
4	2	$L_2$	–	0	0
5	2	$U_2$	–	1	0
6	2	$S_2$	–	1	1
7	1	$U_1$	1	–	1
8	1	$S_1$	1	–	1
9	1	$T_1$	1	–	1
10	2	$T_2$	–	1	1

В конце переменная cnt получит некорректное значение 1.

### Решение упражнения 12.8

Это упражнение поможет вам проверить правильность понимания безопасных и небезопасных траекторий в графе выполнения. Траектории, такие как 1 и 3, огибающие критические области, безопасны и приводят к корректным результатам.

1.  $H_1, L_1, U_1, S_1, H_2, L_2, U_2, S_2, T_2, T_1$  *безопасная*
2.  $H_2, L_2, H_1, L_1, U_1, S_1, T_1, U_2, S_2, T_2$  *небезопасная*
3.  $H_1, H_2, L_2, U_2, S_2, L_1, U_1, S_1, T_1, T_2$  *безопасная*

### Решение упражнения 12.9

1.  $p = 1, c = 1, n > 1$ : да, мьютекс необходим, потому что производитель и потребитель могут конкурентно обращаться к буферу.
2.  $p = 1, c = 1, n = 1$ : нет, мьютекс не нужен, потому что непустой буфер эквивалентен полному буферу. Когда буфер содержит элемент, производитель блокируется. Когда буфер пуст, блокируется потребитель. То есть в любой момент времени только один поток имеет доступ к буферу и исключительность доступа гарантируется без использования мьютекса.
3.  $p > 1, c > 1, n = 1$ : нет, мьютекс не нужен, потому что в этом случае применимы те же аргументы, что и в предыдущем.

### Решение упражнения 12.10

Предположим, что конкретная реализация семафора использует стек потоков LIFO, ожидающих его освобождения. Когда поток блокируется на семафоре в операции  $P$ , его идентификатор помещается в стек. Точно так же операция  $V$  извлекает идентификатор верхнего потока из стека и возобновляет этот поток. При такой реализации с использованием стека злонамеренный поток-писатель сможет в своей критической секции просто подождать, пока появится другой поток-писатель, ожидающий освобождения семафора, и затем освободить этот семафор. В этом сценарии ожидающий поток-читатель может ждать вечно, пока два писателя передают управление туда-сюда.

Обратите внимание: использование очереди FIFO вместо стека LIFO может показаться более логичным, и все же применение такого стека не является неправильным и не нарушает семантику операций  $P$  и  $V$ .

### Решение упражнения 12.11

Это упражнение поможет вам проверить свое понимание ускорения и эффективности параллельных вычислений.

Потоки ( $t$ )	1	2	4
Ядра ( $p$ )	1	2	4
Время выполнения ( $T_p$ )	12	8	6
Ускорение ( $S_p$ )	1	1.5	2
Эффективность ( $E_p$ )	100 %	75 %	50 %

### Решение упражнения 12.12

Функция `ctime_ts` не является реентерабельной, потому что каждый вызов использует одну и ту же статическую переменную, возвращаемую функцией `ctime`. Однако она является потокобезопасной, потому что доступ к общей переменной защищен операциями  $P$  и  $V$ , следовательно, каждый вызов имеет исключительный доступ.

### Решение упражнения 12.13

Если освободить блок сразу после обращения к `pthread_create` в строке 14, то возникнет новое состояние гонки, на этот раз между вызовом `free` в главном потоке и оператором присваивания в строке 24 в процедуре потока.

### Решение упражнения 12.14

1. Другое решение: передача целого числа  $i$  напрямую вместо передачи указателя на  $i$ :

```
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, (void *)i);
```

В процедуре потока аргумент приводится обратно к типу `int` и присваивается переменной `myid`:

```
int myid = (int) vargp;
```

2. Преимуществом является уменьшение накладных расходов на вызовы `malloc` и `free`. Недостаток – предположение, что указатели по крайней мере не меньше типа `int`. Это предположение верно для всех современных систем, но в будущем оно может стать ложным.

### Решение упражнения 12.15

1. Граф выполнения для оригинальной программы показан на рис. 12.19.
2. Программа всегда попадает в тупиковые ситуации, потому что любая возможная траектория, в конце концов, окажется в состоянии взаимоблокировки.
3. Для устранения потенциальной взаимоблокировки инициализируйте бинарный семафор `t` значением 1 вместо 0.
4. Граф выполнения для исправленной программы показан на рис. 12.20.

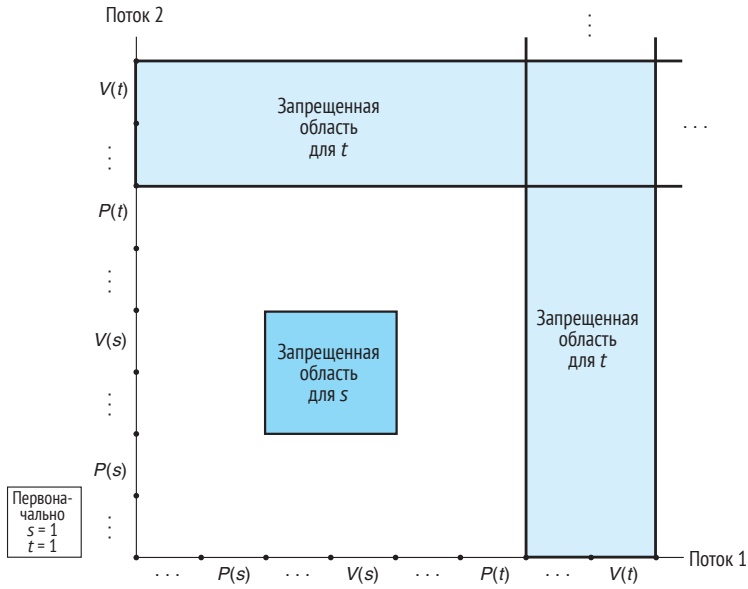


Рис. 12.19. Граф выполнения для программы, страдающей проблемой взаимоблокировки

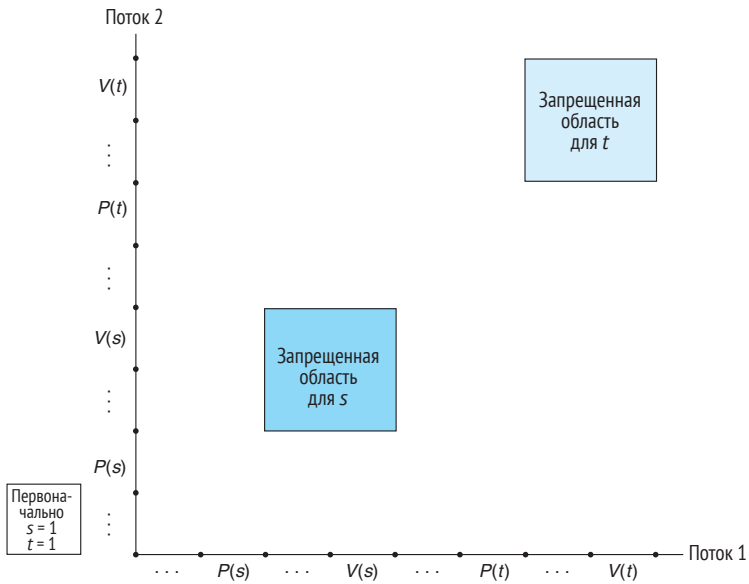


Рис. 12.20. Граф выполнения для исправленной программы, не страдающей проблемой взаимоблокировки

# Приложение А

## Обработка ошибок

Программисты должны *всегда* проверять коды ошибок, возвращаемые функциями системного уровня. Всегда есть вероятность, что что-то пойдет не так, и имеет смысл использовать информацию о состоянии, передаваемую ядром. К сожалению, разработчики часто пренебрегают проверкой ошибок, потому что эти проверки загромаждают код, превращая, например, одну строку кода в многострочный условный оператор. Проверка ошибок также вносит в программу определенного рода путаницу, потому что разные функции по-разному сообщают об ошибках.

Работая над этой книгой, мы постоянно сталкивались с подобными проблемами. С одной стороны, хотелось бы, чтобы примеры были краткими и простыми, а с другой – не хотелось насаждать ложное мнение о том, что проверять наличие ошибок не нужно вообще. Для решения этих проблем на вооружение был принят принцип, основанный на использовании *оберток, реализующих обработку ошибок*, впервые предложенный Ричардом Стивенсом (Richard Stevens) в его работе, посвященной сетевому программированию [110].

Идея заключается в том, что при наличии некоторой системной функции `foo` определяется функция-обертка `Foo` с теми же аргументами, которая вызывает системную функцию и осуществляет проверку ошибок. При выявлении ошибки обертка выводит информативное сообщение и прерывает выполнение процесса. В противном случае возвращает управление вызывающей программе. Обратите внимание, что в отсутствие ошибок поведение обертки ничем не отличается от поведения системной функции. И наоборот, если программа выполняется корректно с обертками, то она будет выполняться корректно, если вместо оберток использовать системные функции.

Обертки упакованы в один файл (`csapp.c`), который компилируется и связывается с каждой программой. Отдельный заголовочный файл (`csapp.h`) содержит прототипы функций-оберток.

В данном приложении представлено руководство по разным способам обработки ошибок в системах Unix, а также примеры различных стилей оформления оберток с обработкой ошибок. Копии файлов `csapp.h` и `csapp.c` вы найдете на веб-сайте CS:APP.

### А.1. Обработка ошибок в системе Unix

Для вызова системных функций, с которыми можно столкнуться в этой книге, используются три разных стиля обработки ошибок: *Unix*, *Posix* и *GAI*.

## Обработка ошибок в стиле Unix

Такие функции, как `fork` и `wait`, разработанные на заре появления систем Unix (как и некоторые старые функции Posix), перегружают возвращаемые значения кодами ошибок и полезными результатами. Например, когда функция `wait` сталкивается с ошибкой (например, отсутствие дочернего процесса), она возвращает `-1` и записывает код ошибки в глобальную переменную `errno`. Если функция `wait` завершается успехом, то возвращается полезный результат – идентификатор утилизированного дочернего процесса. Обработка ошибок в стиле Unix обычно выполняется так:

```
1 if ((pid = wait(NULL)) < 0) {
2     fprintf(stderr, "wait error: %s\n", strerror(errno));
3     exit(0);
4 }
```

Функция `strerror` возвращает текстовую строку с описанием конкретного значения в `errno`.

## Обработка ошибок в стиле Posix

Многие из новейших функций Posix, такие как функции из пакета Pthreads, возвращают только признак успешности (0) или неудачи (ненулевое значение) выполнения. Все полезные результаты возвращаются в аргументах, передаваемых по ссылке. Данный подход называется *обработкой ошибок в стиле Posix*. Например, Posix-функция `pthread_create` указывает в возвращаемом значении признак успеха или неудачи, а идентификатор созданного потока (полезный результат) – в первом аргументе. Обработка ошибок в стиле Posix обычно выполняется так:

```
1 if ((retcode = pthread_create(&tid, NULL, thread, NULL)) != 0) {
2     fprintf(stderr, "pthread_create error: %s\n", strerror(retcode));
3     exit(0);
4 }
```

Функция `strerror` возвращает текстовую строку с описанием конкретного значения в `retcode`.

## Обработка ошибок в стиле GAI

Функции `getaddrinfo` (GAI) и `getnameinfo` возвращают ноль в случае успеха или ненулевое значение в случае ошибки. Обработка ошибок в стиле GAI обычно выполняется так:

```
1 if ((retcode = getaddrinfo(host, service, &hints, &result)) != 0) {
2     fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(retcode));
3     exit(0);
4 }
```

Функция `gai_strerror` возвращает текстовую строку с описанием конкретного значения в `retcode`.

## Функции получения текстовых описаний ошибок

На протяжении всей книги для представления различных стилей обработки ошибок используются следующие функции, возвращающие текстовые описания ошибок:

```
#include "csapp.h"

void unix_error(char *msg);
void posix_error(int code, char *msg);
void gai_error(int code, char *msg);
void app_error(char *msg);
```

Ничего не возвращают

По именам функций `unix_error`, `posix_error` и `gai_error` видно, что они выводят сообщение об ошибке в стиле Unix, Posix и GAI соответственно, после чего завершают процесс. Функция `app_error` включена для удобства обнаружения ошибок на прикладном уровне. Она просто выводит входную строку и завершает процесс. Реализации всех этих функций показаны в листинге A.1.

#### Листинг A.1. Функции вывода сообщений об ошибках

*code/src/csapp.c*

```
1 void unix_error(char *msg) /* Для обработки ошибок в стиле Unix */
2 {
3     fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4     exit(0);
5 }
6
7 void posix_error(int code, char *msg) /* Для обработки ошибок в стиле Posix */
8 {
9     fprintf(stderr, "%s: %s\n", msg, strerror(code));
10    exit(0);
11 }
12
13 void gai_error(int code, char *msg) /* Для обработки ошибок в стиле Getaddrinfo */
14 {
15     fprintf(stderr, "%s: %s\n", msg, gai_strerror(code));
16     exit(0);
17 }
18
19 void app_error(char *msg) /* Для обработки ошибок на прикладном уровне */
20 {
21     fprintf(stderr, "%s\n", msg);
22     exit(0);
23 }
```

*code/src/csapp.c*

## A.2. Функции-обертки обработки ошибок

Ниже приводятся несколько примеров различных функций-оборок обработки ошибок.

- *Обертки обработки ошибок в стиле Unix.* В листинге A.2 представлена функция-обертка обработки ошибок в стиле Unix для функции `wait`. Если `wait` возвращает признак ошибки, то обертка выводит информативное сообщение и завершает процесс. В противном случае вызывающей программе возвращается PID.

**Листинг А.2.** Обертка обработки ошибок в стиле Unix для wait*code/src/csapp.c*

```

1 pid_t Wait(int *status)
2 {
3     pid_t pid;
4
5     if ((pid = wait(status)) < 0)
6         unix_error("Wait error");
7     return pid;
8 }

```

*code/src/csapp.c*

В листинге А.3 представлена функция-обертка обработки ошибок в стиле Unix для функции kill. Обратите внимание, что в отличие от wait эта функция возвращает void в случае успеха.

**Листинг А.3.** Обертка обработки ошибок в стиле Unix для kill*code/src/csapp.c*

```

1 void Kill(pid_t pid, int signum)
2 {
3     int rc;
4
5     if ((rc = kill(pid, signum)) < 0)
6         unix_error("Kill error");
7 }

```

*code/src/csapp.c*

- *Обертки обработки ошибок в стиле Posix.* В листинге А.4 представлена функция-обертка обработки ошибок в стиле Posix для функции pthread\_detach. Подобно большинству функций Posix, она не перегружает возвращаемое значение кодами ошибок, поэтому при успешном выполнении функция-обертка возвращает void.

**Листинг А.4.** Обертка обработки ошибок в стиле Posix для pthread\_detach*code/src/csapp.c*

```

1 void Pthread_detach(pthread_t tid) {
2     int rc;
3
4     if ((rc = pthread_detach(tid)) != 0)
5         posix_error(rc, "Pthread_detach error");
6 }

```

*code/src/csapp.c*

- *Обертки обработки ошибок в стиле GAI.* В листинге А.5 представлена функция-обертка обработки ошибок в стиле GAI для функции getaddrinfo.

**Листинг А.5.** Обертка обработки ошибок в стиле GAI для `getaddrinfo`

```
1 void Getaddrinfo(const char *node, const char *service,  
2 const struct addrinfo *hints, struct addrinfo **res)  
3 {  
4     int rc;  
5  
6     if ((rc = getaddrinfo(node, service, hints, res)) != 0)  
7         gai_error(rc, "Getaddrinfo error");  
8 }
```

*code/src/csapp.c*

*code/src/csapp.c*



# Библиография

- [1] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD64 Processors*, 2005. Publication Number 25112.
- [2] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*, 2013. Publication Number 24592.
- [3] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions*, 2013. Publication Number 24594.
- [4] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual, Volume 4: 128-Bit and 256-Bit Media Instructions*, 2013. Publication Number 26568.
- [5] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Fourth Edition*. Prentice Hall, 2005<sup>1</sup>.
- [6] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. RFC 1945, 1996.
- [7] A. Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, 1989.
- [8] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *SIGOPS Operating Systems Review* 41 (2): 88–93, 2007.
- [9] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–366. ACM, June 2011.
- [10] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Design Automation Conference*, pages 746–749. ACM, 2007.
- [11] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, Inc., 2005.
- [12] A. Demke Brown and T. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44. Usenix, October 2000.
- [13] R. E. Bryant. Term-level verification of a pipelined CISC microprocessor. Technical Report CMU-CS-05-195, Carnegie Mellon University, School of Computer Science, 2005.
- [14] R. E. Bryant and D. R. O'Hallaron. Introducing computer systems from a programmer's perspective. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, pages 90–94. ACM, February 2001.
- [15] D. Butenhof. *Programming with Posix Threads*. Addison-Wesley, 1997.
- [16] S. Carson and P. Reynolds. The geometry of semaphore programs. *ACM Transactions on Programming Languages and Systems* 9 (1): 25–53, 1987.
- [17] J. B. Carter, W. C. Hsieh, L. B. Stoller, M. R. Swanson, L. Zhang, E. L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA)*, pages 70–79. ACM, January 1999.

---

<sup>1</sup> Джеймс Гослинг, Билл Джой, *Язык программирования Java SE 8*, Вильямс, 2015, ISBN: 978-5-8459-1875-8, 978-0-13-390069-9. – Прим. перев.

- [18] K. Chang, D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. Improving DRAM performance by parallelizing refreshes with accesses. In *Proceedings of the 20th International Symposium on High-Performance Computer Architecture (HPCA)*. ACM, February 2014.
- [19] S. Chellappa, F. Franchetti, and M. Püschel. How to write fast numerical code: A small introduction. In *Generative and Transformational Techniques in Software Engineering II, volume 5235 of Lecture Notes in Computer Science*, pages 196–259. Springer-Verlag, 2008.
- [20] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys* 26 (2): 145–185, June 1994.
- [21] S. Chen, P. Gibbons, and T. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 235–246. ACM, May 2001.
- [22] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proceedings of the 1999 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, May 1999.
- [23] E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys* 3 (2): 67–78, June 1971.
- [24] D. Cohen. On holy wars and a plea for peace. *IEEE Computer* 14 (10): 48–54, October 1981.
- [25] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Communications of the ACM* 14 (10): 667–668, 1971.
- [26] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, volume 2, pages 119–129, March 2000.
- [27] J. H. Crawford. The i486 CPU: Executing instructions in one clock cycle. *IEEE Micro* 10 (1): 27–36, February 1990.
- [28] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, pages 222–233, ACM, 1999.
- [29] B. Davis, B. Jacob, and T. Mudge. The new DRAM interfaces: SDRAM, RDRAM, and variants. In *Proceedings of the 3rd International Symposium on High Performance Computing (ISHPC)*, volume 1940 of *Lecture Notes in Computer Science*, pages 26–31. Springer-Verlag, October 2000.
- [30] E. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, University of Aarhus, Denmark, 2002.
- [31] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.
- [32] C. Ding and K. Kennedy. Improving cache performance of dynamic applications through data and computation reorganizations at run time. In *Proceedings of the 1999 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 229–241. ACM, May 1999.
- [33] M. Dowson. The Ariane 5 software failure. *SIGSOFT Software Engineering Notes* 22 (2): 84, 1997.
- [34] U. Drepper. User-level IPv6 programming introduction. Документ доступен по адресу <http://www.akkadia.org/drepper/userapi-ipv6.html>, 2008.

- [35] M. W. Eichen and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November, 1988. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 326–343. IEEE, 1989.
- [36] ELF-64 *Object File Format, Version 1.5 Draft 2*, 1998. Документ доступен по адресу <http://www.uclibc.org/docs/elf-64-gen.pdf>.
- [37] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, 1999.
- [38] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 285–297. IEEE, August 1999.
- [39] M. Frigo and V. Strumpfen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the 18th Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–280. ACM, 2006.
- [40] G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103. ACM, October 1998.
- [41] G. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM* 43 (11): 37–45, November 2000.
- [42] Google. IPv6 Adoption. Документ доступен по адресу <http://www.google.com/intl/en/ipv6/statistics.html>.
- [43] J. Gustafson. Reevaluating Amdahl’s law. *Communications of the ACM* 31 (5): 532–533, August 1988.
- [44] L. Gwennap. New algorithm improves branch prediction. *Microprocessor Report* 9 (4), March 1995.
- [45] S. P. Harbison and G. L. Steele, Jr. *C, A Reference Manual, Fifth Edition*. Prentice Hall, 2002<sup>2</sup>.
- [46] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufmann, 2011<sup>3</sup>.
- [47] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- [48] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM* 17 (10): 549–557, October 1974.
- [49] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Документ доступен по адресу <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>.
- [50] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture*. Документ доступен по адресу <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>.
- [51] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 2: Instruction Set Reference*. Документ доступен по адресу <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>.
- [52] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3a: System Programming Guide, Part 1*. Документ доступен по адресу <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>.

<sup>2</sup> Харбисон Сэмюел П., Стил Гай Л., *Язык программирования C*, Бином-Пресс, 2009, ISBN: 978-5-9518-0334-4. – Прим. перев.

<sup>3</sup> Паттерсон Хеннесси, *Архитектура компьютера и проектирование компьютерных систем*. Питер, 2012, ISBN: 978-5-459-00291-1. – Прим. перев.

- [53] Intel Corporation. *Intel Solid-State Drive 730 Series: Product Specification*. Документ доступен по адресу <http://www.intel.com/content/www/us/en/solid-state-drives/ssd-730-series-spec.html>.
- [54] Intel Corporation. *Tool Interface Standards Portable Formats Specification, Version 1.1*, 1993. Order number 241597.
- [55] F. Jones, B. Prince, R. Norwood, J. Hartigan, W. Vogley, C. Hart, and D. Bondurant. Memory – a new era of fast dynamic RAMs (for video applications). *IEEE Spectrum*, pages 43–45, October 1992.
- [56] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.
- [57] M. Kaashoek, D. Engler, G. Ganger, H. Briceo, R. Hunt, D. Maziers, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 52–65. ACM, October 1997.
- [58] R. Katz and G. Borriello. *Contemporary Logic Design, Second Edition*. Prentice Hall, 2005.
- [59] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, 1999<sup>4</sup>.
- [60] B. Kernighan and D. Ritchie. *The C Programming Language, First Edition*. Prentice Hall, 1978<sup>5</sup>.
- [61] B. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.
- [62] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010<sup>6</sup>.
- [63] T. Kilburn, B. Edwards, M. Lanigan, and F. Sumner. One-level storage system. *IRE Transactions on Electronic Computers* EC-11: 223–235, April 1962.
- [64] D. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition*. Addison-Wesley, 1977<sup>7</sup>.
- [65] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach, Sixth Edition*. Addison-Wesley, 2012.
- [66] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74. ACM, April 1991.
- [67] D. Lea. A memory allocator. Документ доступен по адресу <http://gee.cs.oswego.edu/dl/html/malloc.html>, 1996.
- [68] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica* 6 (1–6), June 1991.
- [69] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 1999.
- [70] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors*. Документ доступен по адресу [https://software.intel.com/sites/products/collateral/hpc/vtune/performance\\_analysis\\_guide.pdf](https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf).

<sup>4</sup> Брайан Керниган, Роб Пайк, *Практика программирования*, Вильямс, 2017, ISBN: 978-5-8459-2005-8. – Прим. перев.

<sup>5</sup> Брайан Керниган, Деннис Ритчи, *Язык программирования C*, Вильямс, 2017, ISBN: 978-5-8459-1874-1, 0-13-110362-8, 978-5-8459-1975-5. – Прим. перев.

<sup>6</sup> Керрис Майкл, *Linux API. Исчерпывающее руководство*, Питер, 2018, ISBN: 978-5-496-02689-5. – Прим. перев.

<sup>7</sup> Кнут Дональд Эрвин, *Искусство программирования. Том 1: Основные алгоритмы*, Вильямс, 2019, ISBN: 978-5-907144-23-1. – Прим. перев.

- [71] C. Lin and L. Snyder. *Principles of Parallel Programming*. AddisonWesley, 2008.
- [72] Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In *Proceedings of the 2000 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 157–168. ACM, June 2000.
- [73] J. L. Lions. Ariane 5 Flight 501 failure. Technical Report, European Space Agency, July 1996.
- [74] S. Macguire. *Writing Solid Code*. Microsoft Press, 1993.
- [75] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhal, and W. W. Hwu. Compiler code transformations for superscalar-based high-performance systems. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 808–817. ACM, 1992.
- [76] E. Marshall. Fatal error: How Patriot overlooked a Scud. *Science*, page 1347, March 13, 1992.
- [77] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V application binary interface AMD64 architecture processor supplement. Technical Report, x86-64.org, 2013. Документ доступен по адресу [http://www.x86-64.org/documentation\\_folder/abi-0.99.pdf](http://www.x86-64.org/documentation_folder/abi-0.99.pdf).
- [78] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, pages 184–201, March 1986.
- [79] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 62–73. ACM, October 1992.
- [80] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [81] S. Nath and P. Gibbons. Online maintenance of very large random samples on flash storage. In *Proceedings of VLDB*, pages 970–983. VLDB Endowment, August 2008.
- [82] M. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, 2001.
- [83] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 109–116. ACM, June 1988.
- [84] L. Peterson and B. Davie. *Computer Networks: A Systems Approach, Fifth Edition*. Morgan Kaufmann, 2011.
- [85] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy* 2 (4): 20–27, 2004.
- [86] S. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann, 1990.
- [87] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM* 35 (8): 102–114, August 1992.
- [88] W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM Conference on Java Grande*, pages 89–98. ACM, June 1999.
- [89] J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits: A Design Perspective, Second Edition*. Prentice Hall, 2003<sup>8</sup>.
- [90] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.
- [91] D. Ritchie. The evolution of the Unix timesharing system. *AT&T Bell Laboratories Technical Journal* 63 (6 Part 2): 1577–1593, October 1984.

<sup>8</sup> Рабаи Жан М., Чандракасан Ананта, Николич Боровож, *Цифровые интегральные схемы. Методология проектирования*, Вильямс, 2016, ISBN: 978-5-8459-1116-2, 0-13-090996-3. – Прим. перев.



- [92] D. Ritchie. The development of the C language. In *Proceedings of the 2nd ACM SIGPLAN Conference on History of Programming Languages*, pages 201–208. ACM, April 1993.
- [93] D. Ritchie and K. Thompson. The Unix timesharing system. *Communications of the ACM* 17 (7): 365–367, July 1974.
- [94] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers* 39 (4): 447–459, April 1990.
- [95] J. Schindler and G. Ganger. Automated disk drive characterization. Technical Report CMUCS-99-176, School of Computer Science, Carnegie Mellon University, 1999.
- [96] F. B. Schneider and K. P. Birman. The monoculture risk put into context. *IEEE Security and Privacy* 7 (1): 14–17, January 2009.
- [97] R. C. Seacord. *Secure Coding in C and C++*, Second Edition. Addison-Wesley, 2013<sup>9</sup>.
- [98] R. Sedgewick and K. Wayne. *Algorithms, Fourth Edition*. Addison-Wesley, 2011<sup>10</sup>.
- [99] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307. ACM, 2004.
- [100] J. P. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw Hill, 2005.
- [101] B. Shriver and B. Smith. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. IEEE Computer Society, 1998.
- [102] A. Silberschatz, P. Galvin, and G. Gagne. *Operating Systems Concepts, Ninth Edition*. Wiley, 2014.
- [103] R. Skeel. Roundoff error and the Patriot missile. *SIAM News* 25 (4): 11, July 1992.
- [104] A. Smith. Cache memories. *ACM Computing Surveys* 14 (3), September 1982.
- [105] E. H. Spafford. The Internet worm program: An analysis. Technical Report CSD-TR-823, Department of Computer Science, Purdue University, 1988.
- [106] W. Stallings. *Operating Systems: Internals and Design Principles, Eighth Edition*. Prentice Hall, 2014<sup>11</sup>.
- [107] W. R. Stevens. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP and the Unix Domain Protocols*. Addison-Wesley, 1996<sup>12</sup>.
- [108] W. R. Stevens. *Unix Network Programming: Interprocess Communications*, Second Edition, volume 2. Prentice Hall, 1998<sup>13</sup>.
- [109] W. R. Stevens and K. R. Fall. *TCP/IP Illustrated, Volume 1: The Protocols, Second Edition*. Addison-Wesley, 2011.
- [110] W. R. Stevens, B. Fenner, and A. M. Rudoff. *Unix Network Programming: The Sockets Networking API, Third Edition*, volume 1. Prentice Hall, 2003.

<sup>9</sup> Роберт Сикорд, *Безопасное программирование на C и C++*, Вильямс, 2016, ISBN: 978-5-8459-1908-3. – Прим. перев.

<sup>10</sup> Роберт Седжвик, Кевин Уэйн. *Алгоритмы на Java, 4-е издание*, Вильямс, 2012, ISBN: 978-5-8459-1781-2. – Прим. перев.

<sup>11</sup> Столлингс Вильям, *Операционные системы. Внутренняя структура и принципы проектирования*, Вильямс, 2020, ISBN: 978-5-907203-08-2. – Прим. перев.

<sup>12</sup> У. Ричард Стивенс, *Протоколы TCP/IP. Практическое руководство*, Невский Диалект, 2004, ISBN: 5-7940-0093-7. – Прим. перев.

<sup>13</sup> У. Р. Стивенс, Б. Феннер, Э. М. Рудофф, *UNIX: разработка сетевых приложений*, Питер, 2007, ISBN: 5-94723-991-4. – Прим. перев.

- [111] W. R. Stevens and S. A. Rago. *Advanced Programming in the Unix Environment*, Third Edition. Addison-Wesley, 2013<sup>14</sup>.
- [112] T. Stricker and T. Gross. Global address space, non-uniform bandwidth: A memory system performance characterization of parallel systems. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, February 1997.
- [113] A. S. Tanenbaum and H. Bos. *Modern Operating Systems*, Fourth Edition. Prentice Hall, 2015<sup>15</sup>.
- [114] A. S. Tanenbaum and D. Wetherall. *Computer Networks*, Fifth Edition. Prentice Hall, 2010<sup>16</sup>.
- [115] K. P. Wadleigh and I. L. Crawford. *Software Optimization for High-Performance Computing: Creating Faster Applications*. Prentice Hall, 2000.
- [116] J. F. Wakerly. *Digital Design Principles and Practices, Fourth Edition*. Prentice Hall, 2005.
- [117] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, EC-14 (2), April 1965.
- [118] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management, volume 986 of Lecture Notes in Computer Science*, pages 1–116. Springer-Verlag, 1995.
- [119] M. Wolf and M. Lam. Adata locality algorithm. In *Proceedings of the 1991 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 30–44, June 1991.
- [120] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.
- [121] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote, and P. Khosla. Survivable information storage systems. *IEEE Computer* 33: 61–68, August 2000.
- [122] T.-Y. Yeh and Y. N. Patt. Alternative implementation of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 451–461. ACM, 1998.

---

<sup>14</sup> Стивенс Уильям Ричард, Раго Стивен А., *UNIX. Профессиональное программирование*, Питер, 2018, ISBN: 978-5-4461-0649-3, 978-0321637734. – Прим. перев.

<sup>15</sup> Бос Херберт, Таненбаум Эндрю, *Современные операционные системы*, Питер, 2018, ISBN: 978-5-4461-1155-8. – Прим. перев.

<sup>16</sup> Таненбаум Эндрю, Уэзеролл Дэвид, *Компьютерные сети*, Питер, 2019, ISBN: 978-5-4461-1248-7. – Прим. перев.

# Предметный указатель

## Символы

8086 микропроцессор 187  
8087 арифметический сопроцессор 135, 187  
80286 микропроцессор 188  
-O1 флаг оптимизации 190  
-Og флаг оптимизации 190  
-pg параметр компилятора 539  
& [C] оператор взятия адреса  
    локальные переменные 258  
    указатели 79, 283  
& [C] операция взятия адреса  
    указатели 206  
\* [C] оператор разыменования  
    указатели 283  
\* [C] операция разыменования указателя 206  
/etc/services файл 866  
/proc файловая система 694  
/sys файловая система 694  
! [HCL] операция NOT 369  
#include директива препроцессора 190  
<inttypes.h> целочисленные типы  
    фиксированного размера 216  
.align директива 362  
.bss секция 639  
.data секция 639  
.debug секция 639  
.interp секция 662  
.line секция 640  
.pos директива 362  
.rel.data секция 639  
.rel.text секция 639  
.rodata секция 638  
.so разделяемые библиотеки 661  
.strtab секция 640  
.symtab секция 639  
.text секция 638  
%ax [x86-64] младшие 16 разрядов  
    регистра %rax 200  
%r8 [Y86-64] программный регистр 352  
%rip счетчик инструкций 191  
%rsp [Y86-64] регистр указателя стека 198

## A

accept [Linux] функция 870  
addq [Y86-64] сложение 354, 394  
ADD [класс инструкций] сложение 210  
Advanced Micro Devices (AMD) 186  
    совместимость с Intel 189  
AFS (Andrew File System) 582  
alarm [Linux] функция 717  
alloca функция выделения памяти  
    на стеке 290  
AMD (Advanced Micro Devices)  
    совместимость с Intel 189

andq [Y86-64] поразрядное И 354  
AND [класс инструкций] И 210  
ANSI (American National Standards Institute Американский национальный институт стандартов) 67  
ANSI C 67  
AOK [Y86-64] код состояния, нормальное выполнение 360  
API (Application Program Interface прикладной программный интерфейс) 60  
ARM A7 микропроцессор 350  
ARM (Acorn RISC Machine) 75  
    микропроцессорная архитектура 358  
AR архиватор Linux 673  
AR архиватор в Linux 650  
ASCII cnfylfhn^juhfybxybz 81  
ASCII стандарт  
    коды символов 80  
asm директива 197  
ATT формат представления ассемблерного кода  
    и формат Intel 196  
AVX (Advanced Vector eXensions усовершенствованные векторные расширения) 518  
AVX (Advanced Vector eXensions усовершенствованные векторные расширения) инструкции 298

## B

Barracuda 7400 572  
Bell Labs 67  
bind [Linux] функция 870, 872  
binutils пакет 673  
break multstore команда в GDB 285  
break инструкция  
    в операторе switch 245

## C

calloc [C] функция 788  
calloc функция [C Stdlib] распределение памяти  
    уязвимость 126  
calloc функция [C Stdlib] распределение памяти  
    объявление 159  
callq [x86-64] вызов процедуры  
    инструкция 253, 254  
call [x86-64] вызов процедур 355  
call [x86-64] вызов процедуры  
    инструкция 252  
cd команда 833  
CF [x86-64] флаг переноса 218



char [C] тип данных 72, 91  
 CISC (Complex Instruction Set Computer  
   полный набор команд) 359  
 closedir [Linux] функция 844  
 close [Linux] функция 835  
 cltd [x86-64] расширение знакового разряда  
   в %eax до %rax 204  
 cmovae [x86-64] переместить если выше  
   или равно 232  
 cmova [x86-64] переместить если выше 232  
 cmovbe [x86-64] переместить если ниже или  
   равно 232  
 cmovb [x86-64] переместить, если ниже 232  
 cmovbe [x86-64] переместить, если равно 232  
 cmovbe [Y86-64] переместить, если равно 355  
 cmovge [x86-64] переместить, если больше  
   или равно 232  
 cmovg [x86-64] переместить, если больше 232  
 cmovle [x86-64] переместить, если меньше  
   или равно 232  
 cmovl [x86-64] переместить, если меньше 232  
 cmovne [x86-64] переместить, если  
   не равно 232  
 cmovns [x86-64] переместить, если  
   неотрицательное 232  
 cmovs [x86-64] переместить, если  
   отрицательное 232  
 cmpb [x86-64] сравнение байтов 219  
 cmpl [x86-64] сравнение двойных слов 219  
 cmpq [x86-64] сравнение четверных слов 219  
 cmpw [x86-64] сравнение слов 219  
 CMP [класс инструкций] сравнение 219  
 cmttest сценарий 450  
 connect [Linux] функция 869, 872  
 continue команда в GDB 285  
 Control Data Corporation 6600 процессор 502  
 copy\_elements функция 126  
 copy\_from\_kernel функция 114  
 Core 2 микропроцессор 188  
 Core 2, микропроцессоры 561  
 Core i7, Haswell микропроцессор 188  
 Core i7, Sandy Bridge микропроцессор 188  
 Core i7, микропроцессоры  
   ландшафт горы памяти 609  
   шина QuickPath 561  
 CPE (Cycles Per Element) циклов  
   на элемент) 484  
 CR3 регистр управления 774  
 Cray 1 суперкомпьютер 350  
 ctest сценарий 450  
 C язык  
   битовые операции 85  
   числа со знаком и без знака 104  
 C, язык  
   для решения практических задач 37  
   история развития 67  
   происхождение 37  
   стандартная библиотека 40  
   тесно связан с операционной системой  
     Unix 37  
 C язык программирования  
   статические библиотеки 648

C++ язык программирования 641  
 программные исключения 681  
 разрешение имен 644

## D

DDD отладчик с графическим  
   интерфейсом 286  
 DEC [класс инструкций] декремент 210  
 delete [C++] функция 786  
 delete команда в GDB 285  
 Digital Equipment Corporation 86  
 DIMM (Dual Inline Memory Module  
   модули памяти с двухрядным  
   расположением контактов) 557  
 disas multstore команда в GDB 285  
 disas команда в GDB 285  
 divq [x86-64] деление без знака 215  
 dlclose [Unix] закрытие разделяемых  
   библиотек 664  
 DLL (Dynamic Link Libraries динамически  
   связываемые библиотеки) 661  
 dlopen [Unix] открытие разделяемых  
   библиотек 663  
 DMA (Direct Memory Access прямой доступ  
   к памяти) 45  
 double [C] тип данных 73  
 double [C] числа с плавающей точкой  
   двойной точности 150  
 double объявление чисел с плавающей  
   точкой двойной точности 198  
 do [C] вариант цикла while 235  
 DRAM  
   быстрая память со страничным  
     режимом 558  
   массивы 555  
   модули памяти 557  
   память с расширенными возможностями  
     вывода 559  
   расширенная память 558  
   синхронная память 559  
   синхронная память с удвоенной  
     скоростью обработки данных 559  
 DRAM (Dynamic Random Access Memory  
   динамическая память  
   с произвольным доступом) 43  
 dup2 [Linux] функция 848

## E

ECHILD код ошибки 703  
 echo функция 287, 292  
 EINTR код ошибки 703  
 errno переменная 703  
 etest сценарий 450  
 Ethernet 856  
 execve [Linux] функция 707, 783  
 exit [Linux] функция 697

## F

fclose [C] функция 849  
 fgetc [C] функция 849

fingerd демон 289  
 FINGER команда 289  
 finish команда в GDB 285  
 float [C] числа с плавающей точкой  
   одинарной точности 150  
 float объявление чисел с плавающей точкой  
   одинарной точности 198  
 fopen [C] функция 849  
 for [C] оператор цикла 242  
 fork [Linux] функция 697, 783  
 fprintf [C Stdlib] функция  
   форматированного вывода 79  
 fputs [C] функция 849  
 fread [C] функция 849  
 freeaddrinfo [Linux] функция 872  
 FreeBSD-SA-02:38.signed-error 115  
 free [C] функция 788  
 fstat [Linux] функция 842  
 fwrite [C] функция 849

## G

gai\_strerror [Linux] функция 872  
 GCC (GNU Compiler Collection)  
   параметры 67  
 GCC компилятор  
   работа с 190  
 GDB GNU отладчик 193  
 GDB отладчик 285  
 getaddrinfo [Linux] функция 872  
 getenv [Linux] функция 708, 887  
 getnameinfo [Linux] функция 872, 874  
 getpgpr [Linux] функция 715  
 getpid [Linux] функция 697  
 getppid [Linux] функция 697  
 getrusage [Linux] функция 758  
 gets функция 287  
 GIPS (Giga-Instructions Per Second млрд  
   инструкций в секунду) 403  
 GNU, проект 40  
 goto [C] оператор передачи управления 225  
 GPROF профилировщик Unix 539

## H

halt [Y86-64] инструкция остановки  
   выполнения 355  
   исключение 360  
 Haswell 230  
 Haswell микропроцессоры 488, 501  
 HCL (Hardware Control Language язык  
   описания аппаратных средств) 368  
   выражения выбора 373  
   целочисленные выражения 371  
 help команда в GDB 286  
 hotnl [Linux] функция 862  
 htest сценарий 450  
 hton [Linux] функция 862  
 HyperTransport шина 561

## I

i386 микропроцессор 188

i486 микропроцессор 188  
 IA32 (Intel Architecture 32 bit)  
   машинный язык 187  
 IA32 (Intel Architecture 32-bit)  
   микропроцессоры 77  
 iaddq [Y86-64] инструкция сложения с  
   непосредственным значением 365  
 IBM  
   обработка инструкций не по порядку 502  
   процессоры Freescale 349  
 icode (код инструкции) 379  
 if [C] условный оператор 227  
 ifun (функция инструкции) 379  
 imem\_error сигнал 401  
 imulq [x86-64] умножение со знаком 215  
 IMUL [класс инструкций] умножение 210  
 incq инструкция 212  
 INC [класс инструкций] инкремент 210  
 inet\_ntop [Linux] функция 862  
 inet\_pton [Linux] функция 862  
 INFINITY константа 150  
 info frame команда в GDB 286  
 info registers команда в GDB 286  
 int [C] целочисленный тип данных 72  
 Intel Core i7  
   практический пример системы памяти 773  
 Intel Corporation 186  
 INT\_MAX константа, максимальное целое  
   со знаком 98  
 INT\_MIN константа, минимальное целое со  
   знаком 98  
 iPhone 5S 350  
 IP-адреса 861  
 ISO C11 стандарт 67  
 ISO C90 стандарт 67  
 ISO C99 стандарт 67  
   типы фиксированного размера 72  
   целочисленные типы данных 95  
 ISO (International Standards Organization  
   Международная организация по  
   стандартизации) 67

## J

jae [x86-64] перейти, если выше или равно 223  
 Java  
   байт-код 313  
 Java Native Interface (JNI) 665  
 Java язык  
   числовые диапазоны 98  
 Java язык программирования 641  
 ja [x86-64] перейти, если выше 223  
 jbe [x86-64] перейти, если ниже или равно 223  
 jb [x86-64] перейти, если ниже 223  
 je [x86-64] перейти, если не равно/не ноль 223  
 je [x86-64] перейти, если равно/ноль 223  
 je [Y86-64] перейти, если равно 354  
 je [Y86-64] переход, если равно 386  
 jge [x86-64] перейти, если больше  
   или равно 223  
 jg [x86-64] перейти, если больше 223  
 jle [x86-64] перейти, если меньше  
   или равно 223

jl [x86-64] перейти, если меньше 223  
 jns [x86-64] перейти, если  
     неотрицательное 223  
 js [x86-64] перейти, если отрицательное 223  
 jtest сценарий 450

## K

kill [Linux] функция 717  
 kill команда в GDB 285

## L

L1 кеш 586  
 L2 кеш 586  
 L3 кеш 586  
 LDD инструмент 673  
 LD-LINUX.SO динамический компоновщик 671  
 LD\_PRELOAD переменная окружения 671  
 LD статический компоновщик Unix 637  
 libc\_start\_main [C] функция 708  
 limits.h файл с определениями числовых  
     границ 98  
 limits.h файл с определениями числовых  
     пределов 106  
 Linux  
     история проекта 55  
 Linux операционная система 77  
 Lisp язык 113  
 listen [Linux] функция 870  
 long double [C] тип данных с плавающей  
     точкой расширенной точности 162  
 long double объявление чисел с плавающей  
     точкой расширенной точности 198  
 longjmp [Linux] функция 736  
 ls команда 832

## M

malloc [C] функция 788  
 malloc [C Stdlib] выделение памяти в куче 67  
 man ascii команда 80  
 Mark&Sweep алгоритм сборки мусора 813  
 maxlen параметр 114  
 memchr функция 115  
 memset функция, объявление 159  
 Microsoft Windows операционная система 77  
 MIME (Multipurpose Internet Mail Exten-  
     sions многоцелевые расширения  
     электронной почты интернета) 882  
 mkdir команда 832  
 mmap [Linux] функция 785  
 MMX мультимедийные инструкции 188, 298  
 MOSAIC веб-браузер 882  
 movabsq [x86-64] перемещение  
     абсолютного четверного слова 202  
 MOVZ [класс инструкций] перемещение  
     с расширением нулями 203  
 mrmovq [Y86-64] память-регистр  
     перемещение 382  
 Multics 51  
 munmap [Linux] функция 786

## N

NaN (не число)  
     представление 141  
 NEG [класс инструкций] отрицание 210  
 new [C++] функция 786  
 nexti команда в GDB 285  
 NFS (Network File System) 582  
 NM инструмент 673  
 нор инструкция, нет операции 420  
 NOT [класс инструкций] дополнение 210  
 ntohl [Linux] функция 862  
 ntohs [Linux] функция 862

## O

O\_APPEND константа 834  
 OBJDUMP дизассемблер 285  
 OBJDUMP инструмент GNU для просмотра  
     выполняемых файлов 654  
 OBJDUMP программа для чтения  
     машинного кода 193  
 O\_CREAT константа 834  
 OF [x86-64] флаг переполнения 218  
 OF флаг переполнения 353  
 once\_control переменная 918  
 open\_clientfd функция 876  
 opendir [Linux] функция 844  
 open [Linux] функция 833  
 open\_listenfd функция 877  
 optest сценарий 450  
 O\_RDONLY константа 834  
 O\_RDWR константа 834  
 OR [класс инструкций] ИЛИ 210  
 O\_TRUNC константа 834  
 O\_WRONLY константа 834

## P

P6 микроархитектура 188  
 pause [Linux] функция 706  
 PCI Express (PCIe) 569  
 Pentium 4E микропроцессор 188  
 Pentium 4 микропроцессор 188  
 Pentium III микропроцессор 188  
 Pentium II микропроцессор 188  
 Pentium/MMX микропроцессор 188  
 Pentium Pro микропроцессор 188  
 PentiumPro микропроцессор 502  
 Pentium микропроцессор 188  
 PF [x86-64] флаг четности 197  
 PID (Process Identifier идентификатор  
     процесса) 697  
 PIPE процессор 416  
     реализация этапов 434  
 PMAP 739  
 popq [x86-64] инструкция выталкивания  
     со стека 208  
 popq [Y86-64] инструкция 355  
     поведение 366  
 Posix 51  
 printf [C] функция 849  
 print команда в GDB 285  
 PS 739

pthread\_cancel [Linux] функция 917  
 pthread\_create [Linux] функция 916  
 pthread\_detach [Linux] функция 917  
 pthread\_exit [Linux] функция 916  
 pthread\_join [Linux] функция 917  
 pthread\_once [Linux] функция 918  
 pthread\_self [Linux] функция 916  
 pushq [x86-64] инструкция вталкивания  
   в стек 208  
 pushq [Y86-64] инструкция 355  
   этапы обработки 366, 384  
 pushq инструкция сохранения четверного  
   слова 193

## Q

qsort функция 541  
 quit команда в GDB 285

## R

readdir [Linux] функция 843  
 READELF инструмент GNU для просмотра  
   объектных файлов 642  
 read [Linux] функция 835  
 realloc [C] функция 788  
 rep [x86-64] инструкция повторения, как  
   по-оп 224  
 ret [x86-64] инструкция возврата из  
   процедуры 224  
 ret инструкция  
   этапы обработки 387  
 rio\_readinitb функция 839  
 rio\_readlineb функция 839  
 rio\_readnb функция 839  
 rio\_readn функция 837  
 rio\_writen функция 837  
 RISC (Reduced Instruction Set Computers  
   сокращенный набор команд) 359  
 rmdir команда 832  
 rmmovq [Y86-64] регистр-память  
   перемещение 382  
 run команда в GDB 285

## S

salb [x86-64] сдвиг влево 213  
 SAR [класс инструкций] арифметический  
   сдвиг вправо 213  
 SAR [класс инструкций] сдвиг вправо  
   арифметический 210  
 SATA интерфейсы 570  
 sbrk [C] функция 788  
 scanf [C] функция 849  
 SCSI интерфейсы 570  
 select [Linux] функция 906  
 sem\_init [Linux] функция 928  
 sem\_post [Linux] функция 928  
 sem\_wait [Linux] функция 928  
 setae [x86-64] установить, если выше или  
   равно 220  
 seta [x86-64] установить, если выше 220  
 setbe [x86-64] установить, если ниже или

  равно 220  
 setb [x86-64] установить, если ниже 220  
 setenv [Linux] функция 709  
 sete [x86-64] установить, если равно 220  
 setge [x86-64] установить, если больше или  
   равно (со знаком) 220  
 setg [x86-64] установить, если больше  
   (со знаком) 220  
 setjmp [Linux] функция 736  
 setle [x86-64] установить, если меньше  
   или равно (со знаком) 220  
 setl [x86-64] установить, если меньше  
   (со знаком) 220  
 setne [x86-64] установить, если не равно 220  
 setns [x86-64] установить, если  
   неотрицательное 220  
 setpgid [Linux] функция 715  
 sets [x86-64] установить, если  
   отрицательное 220  
 SF [x86-64] флаг знака 218  
 SF флаг знака 353  
 SHL [класс инструкций] сдвиг влево 213  
 SHR [класс инструкций] логический сдвиг  
   вправо 213  
 SHR [класс инструкций] сдвиг вправо  
   логический 210  
 SIGABRT сигнал 713  
 sigaction [Linux] функция 729  
 sigaddset [Linux] функция 720  
 SIGALRM сигнал 713  
 sig\_atomic\_t [C] тип 724  
 SIG\_BLOCK константа 720  
 SIGBUS сигнал 713  
 SIGCHLD сигнал 713  
 SIGCONT сигнал 697, 702, 713  
 sigdelset [Linux] функция 720  
 SIG\_DFL константа 718  
 sigemptyset [Linux] функция 720  
 sigfillset [Linux] функция 720  
 SIGFPE сигнал 713  
 SIGHUP сигнал 713  
 SIG\_IGN константа 718  
 SIGILL сигнал 713  
 SIGINT сигнал 713  
 SIGIO сигнал 713  
 sigismember [Linux] функция 720  
 SIGKILL сигнал 713  
 siglongjmp [Linux] функция 737  
 signal [Linux] функция 718  
 signed [C] тип данных 73  
 SIGPIPE сигнал 713  
 sigprocmask [Linux] функция 720  
 SIGPROF сигнал 713  
 SIGPWR сигнал 713  
 SIGQUIT сигнал 713  
 SIGSEGV сигнал 713  
 sigsetjmp [Linux] функция 737  
 SIG\_SETMASK константа 720  
 SIGSTKFLT сигнал 713  
 SIGSTOP сигнал 697, 713  
 sigsuspend [Linux] функция 734  
 SIGTERM сигнал 713

SIGTRAP сигнал 713  
 SIGTSTP сигнал 697, 713  
 SIGTTIN сигнал 697, 713  
 SIGTTOU сигнал 697, 713  
 SIG\_UNBLOCK константа 720  
 SIGURG сигнал 713  
 SIGUSR1 сигнал 713  
 SIGUSR2 сигнал 713  
 SIGVTALRM сигнал 713  
 SIGWINCH сигнал 713  
 SIGXCPU сигнал 713  
 SIGXFSZ сигнал 713  
 SIMD (Single-Instruction, Multiple-Data)  
   одиночный поток команд,  
   множественный поток данных) 60  
 sio\_error [Linux] функция 723  
 sio\_ltoa [Linux] функция 723  
 sio\_putl [Linux] функция 723  
 sio\_puts [Linux] функция 723  
 sio\_strlen [Linux] функция 723  
 S\_IRGRP константа 834  
 S\_IROTH константа 834  
 S\_IRUSR константа 834  
 S\_IWGRP константа 834  
 S\_IWOTH константа 834  
 S\_IWUSR константа 834  
 S\_IXGRP константа 834  
 S\_IXOTH константа 834  
 S\_IXUSR константа 834  
 sizeof [C] вычисляет размер объекта 77  
 size\_t [Unix] тип данных без знака для  
   обозначения размеров 76  
 size\_t [Unix] тип данных без знака для  
   обозначения размеров 128  
 SIZE инструмент 673  
 sleep [Linux] функция 706  
 socket [Linux] функция 869, 872  
 Solaris Sun Microsystems операционная  
   система 77  
 sprintf функция 288  
 sqrtss [x86-64] корень квадратный двойной  
   точности инструкция 306  
 sqrtss [x86-64] корень квадратный  
   одинарной точности инструкция 306  
 SRAM (Static Random Access Memory  
   статическая память с произвольным  
   доступом) 47  
 SSE (streaming SIMD extensions)  
   инструкции 188  
 SSE (Streaming SIMD Extensions потоковые  
   расширения SIMD) 298  
 Standard Unix Specification 51  
 static [C] атрибут функций и переменных 640  
 stat [Linux] функция 842  
 STDERR\_FILENO константа 831  
 STDIN\_FILENO константа 831  
 stdio.h [Unix] заголовочный файл  
   стандартной библиотеки ввода/  
   вывода 112  
 STDOUT\_FILENO константа 831  
 stepi команда в GDB 285  
 STRACE 739

strcat функция 288  
 strcpy функция 288  
 STRINGS инструмент 673  
 STRIP инструмент 673  
 strlen [C Stdlib] функция вычисления длины  
   строки 491  
 struct [C] тип данных 273  
 subq [Y86-64] вычитание 381  
 SUB [класс инструкций] вычитание 210

## T

TEST [класс инструкций] проверка 219  
 TLB (Translation Look-aside Buffer буфер  
   быстрого преобразования адреса) 455  
 TOP 739  
 typedef [C] определение типа 76

## U

UINT\_MAX константа, максимальное целое  
   без знака 98  
 Unicode (Юникод) набор символов 81  
 Unix 51  
 Unix IPC 905  
 Unix операционная система 67  
 unsetenv [Linux] функция 709  
 unsigned [C] тип данных 73, 91  
 UTF-8, кодировка символов 81

## V

vaddss [x86-64] сложение двойной точности  
   инструкция 306  
 vaddss [x86-64] сложение одинарной  
   точности инструкция 306  
 VALGRIND программа 545  
 vandps [x86-64] поразрядное И инструкция 308  
 vdivsd [x86-64] деление двойной точности  
   инструкция 306  
 vdivss [x86-64] деление одинарной точности  
   инструкция 306  
 vmaxsd [x86-64] максимальное двойной  
   точности инструкция 306  
 vmaxss [x86-64] максимальное одинарной  
   точности инструкция 306  
 vminsd [x86-64] минимальное двойной  
   точности инструкция 306  
 vminss [x86-64] минимальное одинарной  
   точности инструкция 306  
 vmovsd [x86-64] перемещение значения  
   двойной точности 300  
 vmovss [x86-64] перемещение значения  
   одинарной точности 300  
 vmulsd [x86-64] умножение двойной  
   точности инструкция 306  
 vmulss [x86-64] умножение одинарной  
   точности инструкция 306  
 void\* [C] нетипизированный указатель 79  
 volatile [C] спецификатор 724  
 vsubsd [x86-64] вычитание двойной  
   точности инструкция 306  
 vsubss [x86-64] вычитание одинарной  
   точности инструкция 306



VTUNE программа 545  
 vuscomisd [x86-64] сравнение значений с  
   двойной точностью инструкция 309  
 vuscomiss [x86-64] сравнение значений с  
   одинарной точностью инструкция 309  
 vxorps [x86-64] поразрядное  
   ИСКЛЮЧАЮЩЕЕ-ИЛИ инструкция 308

## W

wait [Linux] функция 703  
 waitpid [Linux] функция 701  
 WCONTINUED константа 702  
 while [C] оператор 237  
 WIFCONTINUED(status) макрос 702  
 WIFEXITED(status) макрос 702  
 WIFSIGNALED(status) макрос 702  
 WIFSTOPPED(status) макрос 702  
 WNOHANG константа 702  
 write [Linux] функция 835  
 WSTOPSIG(status) макрос 702  
 WTERMSIG(status) макрос 702  
 WUNTRACED константа 702

## X

x86-64 микропроцессоры 77  
   машинный язык 186  
 x87 процессоры 187  
 XDR библиотека, уязвимость 126  
 XMM регистры 300  
 xorq [Y86-64] исключающее или 354  
 XOR [класс инструкций] ИСКЛЮЧАЮЩЕЕ-  
   ИЛИ 210

## Y

Y86-64 архитектура набора команд 351, 352  
   дополнительные сведения 366  
   набор инструкций 353  
   обработка исключений 360  
 YAS ассемблер Y86-64 363  
 YIS имитатор набора команд Y86-64 364  
 YMM регистры 300

## Z

ZF [x86-64] флаг признака нуля 218  
 ZF флаг нуля 353

## A

абелева группа 117  
 абсолютное ускорение 943  
 абстрактная модель работы процессора  
   Core i7 504  
 абстракции  
   важность в компьютерных системах 60  
 аварийное завершение 687  
 адаптер главной шины 570  
 адаптеры 43  
 аддитивная инверсия 84

адреса и адресация  
   Y86-64 354  
   виртуальная память 68  
   возврат из вызова процедуры 252  
   модель плоской адресации 188  
   операнды 200  
   порядок следования байтов 74  
   преобразование адресов в Core i7 774  
 адресное пространство 693  
 активные страницы 759  
 альтернативные представления целых со  
   знаком 99  
 Амдала закон 56, 538, 544  
 Американский национальный институт  
   стандартов (American National  
   Standards Institute, ANSI) 67  
 анализ производительности 452  
 аппаратная организация системы 42  
 аппаратная реализация Y86-64 387  
 аппаратные исключения 683  
 аппаратные модули 389  
 аппаратные прерывания 685  
 арифметика 65, 209

  загрузка эффективного адреса 210, 211  
 задержка и время выпуска 503  
 насыщающая 158  
 обсуждение 213  
 операция сдвига 130  
 произвольной точности 113  
 специальная 215  
 с плавающей точкой 298, 305  
 унарная и бинарная 212  
 целочисленная 113  
 арифметика с плавающей точкой IEEE 135  
 арифметико-логическое устройство (ALU)  
   на этапе выполнения 379  
 архивы 649  
 архитектура набора команд (Instruction Set  
   Architecture, ISA) 191  
 архитектура набора команд (ISA, Instruction  
   Set Architecture) 349  
 архитектурный набор команд 44, 60  
 ассемблер 39, 184, 190  
 ассемблерный код 184  
   форматирование 195  
 ассоциативность  
   сложения чисел с плавающей точкой 149  
 ациклические цепи 369

## Б

базовые принципы программирования 538  
 базовый регистр таблицы исключений 684  
 базовый регистр таблицы страниц 763  
 базовый указатель 296  
 байты 38, 68  
   кодирование в Y86-64 356  
   копирование 158  
 беззнаковое представление  
   сложение 113  
 безопасная оптимизация 481  
 безопасная траектория 927

безопасные функции 721  
 бесконечность  
   представление 141  
 библиотеки  
   разделяемые 661, 662  
 биграмм подсчет 540  
 бинарные (двухместные) операции 212  
 бинарные семафоры 929  
 бит достоверности, в кешах 586  
 бит знака 95  
 бит изменения 776  
 битовые векторы 82  
 бит режима 693  
 бит ссылки 776  
 биты  
   обзор 64  
 блок-жертва 583  
 блоки  
   кеша 583  
 блокирование сигналов 714  
 блокировка мьютекса 929  
 блокировка сигналов 720  
 блоки управления инструкциями 389  
 блок пролога 802  
 блок регистров 355  
 блок списания (retirement unit) 501  
 блок управления инструкциями (Instruction Control Unit, ICU) 499  
 блок эпилога 802  
 булева алгебра 82  
 булева алгебра и функции  
   свойства 84  
 булевы выражения в HCL 369  
 булевы кольца 84  
 Буль, Джордж 82  
 буфер ассоциативной трансляции адресов  
   (Translation Lookaside Buffer, TLB) 766  
 буфер быстрого преобразования адреса 455  
 буферизованный ввод 839  
 буферы  
   операций сохранения 533

## B

ввод/вывод 830  
   правила выбора функций для  
   использования 850  
 ввод/вывод с отображением в память 570  
 веб-контент 882  
 веб-серверы 881  
 векторные регистры 191  
 ветвление, условное 225, 528  
   switch 245  
 взаимоблокировка 950  
 взаимодействие с пользователями 902  
 видеопамять (VRAM) 559  
 виртуальная адресация 752  
 виртуальная машина 60  
 виртуальная память 51, 68, 660, 750  
   абстракция 49  
   интегрирование кешей и виртуальной  
   памяти 765  
   как средство кеширования 754

как средство управления памятью 760  
 механизмы неявного распределения  
   памяти 787  
 организация в Linux 777  
 сегменты 777  
 упрощение компоновки 760  
   часто встречающиеся ошибки 815  
 виртуальное адресное пространство 52, 68, 753  
 виртуальные адреса  
   Y86-64 353  
   в программировании на машинном  
   уровне 191  
 виртуальные страницы 294, 754  
 вирусы 290  
 внешние исключения  
   в конвейерной обработке 432  
 внешняя фрагментация памяти 793  
 внутренняя фрагментация памяти 793  
 возврат из процедуры, инструкция 355  
 вращающиеся диски 563  
 временная локальность 577, 585  
 время выполнения  
   компоновка 634  
 время доступа к диску 566  
 время загрузки 634  
 время компиляции 634  
 время ожидания  
   конвейерная обработка 402  
 время передачи, диски 566  
 время позиционирования для дисков 566  
 Всемирная паутина (World Wide Web) 882  
 встраиваемые процессоры  
   Y86-64 358  
 встраивание ассемблерного кода 197  
 встраивание функций 483  
 выборки этап  
   обработка инструкций 379  
 выборки, этап  
   обработка инструкций 381  
   последовательная обработка 388  
   процессор PIPE 434  
   трассировка 396  
 вызовы  
   и производительность 494  
 вызываемые процедуры 261  
 вынос кода 490  
 выполнение  
   трассировка 386  
 выполнение не по порядку 456, 498  
 выполнения, этап  
   обработка инструкций 379  
   последовательная обработка 389  
   процессор PIPE 439  
   трассировка 398  
 выполняемые объектные программы 39  
 выполняемые объектные файлы 39, 657  
   создание 636  
 выполняемые файлы 40  
 выполняемый код 190  
 выполняется всегда, стратегия  
   прогнозирования 417  
 высокоуровневое проектирование 537

выталкивания со стека операция 208  
 вытеснение блоков 583  
 вытеснение процессов 691  
 вытеснение регистров 525  
 вычитание  
   с плавающей точкой 306

## Г

гигагерцы (ГГц) 484  
 гиперпоточность 58, 188  
 главный поток выполнения 914  
 глобальная таблица смещений (Global Offset Table, GOT) 666  
 глобальные переменные 639, 921  
 гора памяти 608  
 Гордон Мур (Gordon Moore) 189  
 Горнер Уильям (William G. Horner) 509  
 границы  
   задержки 498, 504  
 граф выполнения 925  
 графические адаптеры 570  
 граф процессов 699  
 графы  
   поточков данных 505  
 группы процессов 715  
 грязные страницы 776

## Д

данные  
   продвижение 423  
 двойные слова 197  
 двоичная точка 137  
 двоичное представление 64  
   преобразование  
     в шестнадцатеричное представление 69  
 двоичные представления  
   дробные числа 136  
 двоичные файлы 39, 832  
 дейтаграммы 860  
 декодирование инструкций 499  
 декодирования, этап  
   обработка инструкций 379  
   последовательная обработка 389  
   процессор PIPE 436  
   трассировка 397  
 деление  
   инструкции 215  
 деление с плавающей точкой 306  
 дескриптор сокета 869  
 дескрипторы 831  
 Джон Хеннеси (John Hennessy) 359  
 диаграммы  
   аппаратные 389  
   конвейерной обработки 402  
 диапазоны  
   асимметрия 97, 106  
   байтов 68  
   типов данных 72  
 дизассемблеры 76, 97, 193  
 динамическая компиляция 294  
 динамическая память 659

вопросы реализации 793  
 неявные списки свободных блоков 794  
 объединение свободных блоков 794, 797  
 объединение с использованием  
   граничных тегов 798  
 организация свободных блоков 794  
 разбиение свободных блоков 794  
 размещение выделенных блоков 794  
 размещение свободных блоков 796  
 реализация механизма распределения  
   памяти 800  
   стратегии поиска свободных блоков 796  
 явные списки свободных блоков 807  
 динамическая память (куча) 52  
 динамическая память с произвольным  
   доступом (Dynamic Random Access  
   Memory, DRAM) 43  
 динамические компоновщики 661  
 динамический контент 663, 883  
 динамическое распределение памяти 786  
 динамическое связывание 661  
   с разделяемыми библиотеками 661  
 директивы ассемблера 362  
 директивы, ассемблера 196  
 диски 562  
   геометрия 562  
   логические блоки 567  
   подключение 568  
 дисковый привод 563  
 добавление конвейерных регистров 411  
 доменные имена 863  
 Дональд Кнут (Donald Knuth) 796  
 дополнение нулями 106  
 дополнительный код 65, 94  
 дорожки дисков 563  
 доставка сигнала 714, 715  
 доступ  
   для чтения 294  
   к дискам 570  
   к основной памяти 560  
   к регистрам IA32 198  
   к флагам 219  
 доступ к информации в x86-64  
   регистры 198  
 доступ к медленным устройствам ввода/  
   вывода 901  
 дочерний поток выполнения 914  
 драйвер компилятора 39  
 драйверы компиляторов 636  
 древовидные структуры 277  
 Дэвид Паттерсон (David Patterson) 359

## Е

емкость  
   кешей 585  
   функциональных блоков 503  
 емкость диска 564  
 емкость отформатированного диска 568

## З

завершение потока выполнения 916



зависимости по данным в конвейерной обработке 409  
 зависимости по управлению в конвейерной обработке 409  
 зависимость между записью и чтением 533  
 зависимость по данным 418  
 зависимость по управлению 418  
 загрузка выполняемых объектных файлов 659  
 загрузки 637, 660  
 задания 716  
 задержка  
 арифметических операций 503  
 инструкций 403  
 задержка из-за вращения диска 566  
 закон Амдала 56  
 закон Мура 189  
 закрытое адресное пространство 693  
 замещение страниц по требованию (demand paging) 758  
 записи перемещения 637, 653  
 запись без размещения 600  
 запись с размещением 600  
 запрос RAS (Row Access Strobe строб адреса ряда) 556  
 запрос CAS (Column Access Strobe строб адреса колонки) 556  
 запуск на переднем плане 712  
 защита памяти 294  
 защитное значение 292  
 защищенный-do, стратегия трансляции 242  
 знак числа, в представлении с плавающей точкой 139  
 зомби, процесс 701

## И

И (AND) операция  
 булева 82  
 идентификатор потока выполнения (Thread ID, TID) 914  
 идентификаторы, регистров 355  
 иерархия каталогов 833  
 иерархия памяти 47  
 изменения PC, этап  
 трасировка 401  
 ИЛИ (||) логическая операция 87  
 имена  
 глобальные 640  
 локальные 640  
 именованные каналы 832  
 имитатор набора команд Y86-64 364  
 имитаторы Y86-64 458  
 имя службы 866  
 имя файла 832  
 инвариант семафора 928  
 индексные регистры 201  
 инкремента инструкции 212  
 инструкции  
 арифметико-логические операции 44  
 в конвейерной обработке 453  
 декодирование 499  
 загрузка 44

классы 202  
 локальность выборки 579  
 переход 44  
 сохранение 44  
 требующие для выполнения нескольких циклов 454  
 инструкция сложения с непосредственным значением 365  
 инструменты управления процессами 739  
 интернет-черви 290  
 интерпретация комбинаций битов 64  
 интернет 858  
 интерфейс системных вызовов 686  
 интерфейс сокетов 860, 867  
 интерфейс шины 561  
 информация – это биты + контекст 38  
 ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) операция  
 булева 82  
 исключение 682  
 исключение по адресу, код состояния 401  
 исключения  
 обработка 683  
 исключительные инструкции 432  
 исключительные ситуации  
 Y86-64 353  
 исключительный доступ 926  
 исполнительный блок (Execution Unit, EU) 499  
 история архитектур RISC и CISC 358  
 история развития индустрии  
 проектирования процессоров 456  
 история развития языка C 67  
 итеративные серверы 880

## К

кадр стека 251  
 каналы передачи отдельных байтов 390  
 каналы передачи отдельных битов 390  
 каналы передачи слов 390  
 канареечное значение 292  
 квант времени 692  
 Керниган, Брайан 67  
 кеш DRAM 754  
 кеш SRAM 754  
 кеш данных 500  
 кеши и кеш-память 586  
 ассоциативность, влияние на  
 производительность 603  
 ассоциативные 595  
 биты смещения слова в блоке 587  
 биты тега 586  
 виды промахов 584  
 время обработки попадания 602  
 локальность 577  
 назначение 553  
 оптимизация операций записи 600  
 организация 586  
 параметры 587  
 попадания 583  
 практические задачи 604  
 пример 591  
 проблемы с операциями записи 600

- промахи 583
  - размер блока, влияние
    - на производительность 603
  - размер, влияние на производительность 603
  - с прямым отображением 588
  - универсальные 601
  - частота попаданий 602
  - частота промахов 602
  - штраф за промах 602
  - кеш инструкций 499
  - кеширование 582
  - кешированные страницы 754
  - кеш-память
    - виды 46
  - клиент, процесс 854
  - клиент-сервер, программная модель 854
  - код
    - профилирование 538
  - код завершения 697
  - код инструкции (icode) 379
  - кодирование инструкций перехода 223
  - коды
    - инструкций Y86-64 355
  - командная оболочка 44
  - комбинации особых случаев 446
  - компилятор 39, 184
  - компиляторы
    - возможности и ограничения оптимизации 481
    - назначение 191
    - принцип действия 190
  - компоновщик 39, 184
  - компоновщики и компоновка 634
    - инструменты управления объектными файлами 673
    - итоги 673
    - объектные файлы 638
    - разрешение ссылок 643
    - таблицы заголовков 658
  - компоновщик и компоновка 190
  - компьютерные сети 855
  - конвейерная обработка 230, 402, 503
    - глубокая 408
    - инструкций 526
    - ограничения 406
    - работа 404
    - регистры 403
  - конвейерная реализация Y86-64
    - сигналы 415
  - конвейерное выполнение
    - операций сохранения 532
  - конвейерные реализации Y86-64 409
  - конвейерные регистры 403, 411
  - конвейер с тремя этапами 404
  - конечные автоматы 909
  - конкурентное выполнение 692
  - конкурентное выполнение
    - родительского и дочернего процессов 698
  - конкурентное программирование 901
  - конкурентные потоки управления 692
  - конкурентные приложения 902
  - конкурентные серверы 881
  - конкурентный эхо-сервер на основе процессов 904
  - конкуренция 692
    - и параллелизм 57
    - на уровне потоков 58
    - передача управления по исключениям 681
  - константы
    - в Y86-64 356
    - диапазонов 98
    - с плавающей точкой 307
    - умножение на 128
  - контакты DRAM 556
  - контекст потока выполнения 914
  - контекст процесса 694
  - контроллеры 43
    - дисков 567
    - памяти 556
  - конфликтные промахи 584
  - концентраторы 856
  - копирование при записи 782
  - корневой каталог 833
  - косвенный переход 223
  - критическая секция 926
  - критические пути, анализ 480
- ## Л
- ленивое связывание 667
  - Леонардо Пизано (Фибоначчи) 64
  - линейное адресное пространство 753
  - логика предсказания переходов 230
  - логические операции
    - сдвиг 131
  - логический поток 691
  - логический поток управления 691
  - логический синтез 352
  - логическое проектирование 368
  - ложная фрагментация 797
  - локальность 47, 554, 577, 759
    - в программах 616
    - итоги 579
    - обращений к данным 577
    - формы 577
  - локальные автоматические переменные 921
  - локальные переменные 258
    - в регистрах 261
  - локальные статические переменные 922
- ## М
- максимальное из двух чисел с плавающей точкой 306
  - мантисса в представлении с плавающей точкой 139
  - маршрутизаторы 858
  - маска сигналов 715
  - маскирование, операция 86
  - массивы 264
    - арифметика указателей 266
    - базовые принципы 264
    - вложенные 267
    - и указатели 79
    - объявление 265

переменного размера 271  
 представление в машинном коде 191  
 фиксированного размера 268  
 шаг обхода 578  
 масштабный коэффициент в ссылках  
   на память 201  
 материнская плата 43  
 машинный код 184, 190  
 медленные системные вызовы 729  
 Международная организация по  
   стандартизации (International  
   Standards Organization, ISO) 67  
 межпроцессные взаимодействия  
   (Interprocess Communication, IPC) 902  
 метаданные файлов 842  
 метастабильные состояния 555  
 метод Горнера 509  
 механизмы управления конвейером 444  
 механизмы явного распределения памяти 786  
   цели и требования 791  
 микроархитектура 498  
 микрооперации 499  
 микропроцессоры  
   Core i7 Haswell 502  
 микропроцессоры Intel  
   8086 187  
   80286 188  
   Core i7, Haswell 188  
   Core i7, Nehalem 188  
   i386 188  
   Pentium III 188  
   Pentium/MMX 188  
   эволюция 187  
 минимальное из двух чисел с плавающей  
   точкой 306  
 многозадачность 692, 694  
 многозонная запись, технология 564  
 многопоточность 50  
 многопроцессорные системы 49  
 многочленов вычисление 509  
 многоядерные процессоры 49, 188  
 модель памяти потоков 921  
 модули памяти 557  
 модуль управления памятью (Memory  
   Management Unit, MMU) 753  
 монокультура безопасности 290  
 мост ввода/вывода 560  
 мультимедийные инструкции 298  
 мультиплексирование ввода/вывода 902, 906  
 мультиплексоры 370  
 мьютексы 929

## Н

набор, выбор в кеше с прямым  
   отображением 589  
 набор готовых дескрипторов 907  
 набор ожидания 701  
 наборы дескрипторов 906  
 наборы кеша 586  
 наиболее оптимальный, стратегия поиска 796  
 накопленная сумма 484

наложение указателей 482, 484  
 наносекунды (нс) 484  
 НЕ (NOT) операция  
   булева 82  
 небезопасная траектория 927  
 небуферизованный ввод/вывод 837  
 не выполняется никогда, стратегия  
   прогнозирования 417  
 недетерминированное поведение 704  
 незанятые страницы 754  
 неизбежные промахи 584  
 некешированные страницы 754  
 нелокальные переходы 735  
 ненормализованные значения, в  
   представлении с плавающей  
   точкой 140  
 неопределенное целое значение 151  
 непосредственное значение в регистр,  
   инструкция записи 354  
 непосредственное смещение 201  
 непосредственные операнды 200  
 неравномерное разбиение 406  
 неясная ведущая 1, представление 140  
 неясные списки свободных блоков 795  
 Нильса Хенрика Абель 117  
 номер виртуальной страницы 763  
 номер исключения 683  
 номер физической страницы 763  
 нормализованные значения,  
   в представлении с плавающей  
   точкой 140

## О

обертки обработки ошибок в стиле GAI 966  
 обертки обработки ошибок в стиле Posix 966  
 обертки обработки ошибок в стиле Unix 965  
 обертки с поддержкой обработки ошибок 696  
 облуживание динамического контента 883  
 обмен данными в сетях 53  
 обновление, DRAM 555  
 обновления PC, этап  
   обработка инструкций 379  
   последовательная обработка 389  
 обороты в минуту 562  
 обработка ошибок в стиле GAI 964  
 обработка ошибок в стиле Posix 964  
 обработка ошибок в стиле Unix 964  
 обработка сигналов 718, 721  
 обработчики исключений 683  
 обработчики прерываний 685  
 обработчики сигналов 714, 718  
 обратная запись 600  
 обратная совместимость 67  
 обратное проектирование  
   машинный код 185  
 обратной записи, этап  
   обработка инструкций 379  
   последовательная обработка 389  
 обратный код (дополнение до единиц) 99  
 обращения к памяти, этап  
   обработка инструкций 379  
   последовательная обработка 389

- процессор PIPE 440
  - трассировка 400
  - обслуживание статического контента 882
  - общее нарушение защиты 688
  - общий шлюзовый интерфейс (Common Gateway Interface, CGI) 886
  - объединение (union) 76
  - объединения 276
    - доступ к битовым комбинациям 278
  - объектные файлы 193
    - выполняемые 638
    - перемещаемые 637, 638
    - разделяемые 638
  - объектный код 190, 193
  - объявление
    - объединений 276
  - объявление указателей 72
  - объявления
    - public и private 641
  - одиночный поток команд, множественный поток данных (Single-Instruction, Multiple-Data SIMD) 60
  - одновременное обслуживание нескольких сетевых клиентов 902
  - однопроцессорные системы 49
  - ожидающие сигналы 714
  - округление
    - в представлении с плавающей точкой 146
  - округление вверх режим 147
  - округление вниз режим 147
  - округление в сторону нуля режим 147
  - округление до ближайшего целого режим 146
  - округление до четного режим 146
  - округление при делении 131
  - окружение вызова 736
  - операнды регистры 200
  - операции перемещения и преобразования данных с плавающей точкой 300
  - операции сдвига 210, 212
  - операции с плавающей точкой
    - заключительные замечания 312
  - операционная система 48
  - операционные системы
    - Unix 67
  - оптимизация
    - компилятором 190
    - уровни 481
  - оптимизация производительности программ 41
  - основная память 43
  - особые значения, в представлении с плавающей точкой 141
  - останов 421
  - открытие файлов 831
  - отладка 284
  - относительная адресация
    - в Y86-64 356
  - относительное ускорение 943
  - отображение в память 660, 785
  - отображение памяти 761, 780
  - отрицание
    - целых в дополнительном коде 123
    - отрицание целых в дополнительном коде 123
    - отрицательное переполнение 118
  - оценка производительности параллельных программ 942
  - ошибка аппаратного контроля 689
  - ошибка деления 688
  - ошибка обращения к отсутствующей странице 455
  - ошибки времени компоновки 41
  - ошибки при работе с виртуальной памятью
    - неправильное понимание арифметики указателей 818
    - ошибки завышения и занижения на единицу 817
  - переполнение буфера на стеке 816
  - предположение о равенстве указателей и объектов 816
  - разыменование недопустимого указателя 815
  - ссылка на данные в свободных блоках 818
  - ссылка на указатель вместо объекта 817
  - ссылки на несуществующие переменные 818
  - утечки памяти 819
  - чтение неинициализированной памяти 816
  - ошибки синхронизации 922
- П**
- пакеты 858
  - память 553
    - Y86-64 353
    - в программировании на машинном уровне 191
  - иерархия 47
  - иерерхия 581
  - основная 556
  - производительность 530
  - риски по данным 424
  - с произвольным доступом (ОЗУ) 554
  - устройство 378
  - память, обращения
    - производительность 495
  - память с произвольным доступом (ОЗУ) 376
  - параллелизм 515
    - SIMD 518
    - и конкуренция 57
    - на уровне инструкций 59, 480, 498
  - параллельное выполнение 692
  - параллельные вычисления на многоядерных процессорах 902
  - параллельные потоки 692
  - пара сокетов 866
  - первичные входы логических вентилях 369
  - первый подходящий, стратегия поиска 796
  - переадресация ввода/вывода 848
  - перевод строки, символ 39
  - передача DMA 570
  - передача данных, в процедуры 256
  - передача управления 680
    - процедуры 252
  - передача управления по исключению (Exceptional Control Flow, ECF) 681

- важность 681
- исключения 682
- передняя шина (Front Side Bus, FSB) 561
- переименование регистров 502
- переключение контекста 49
- переменные
  - в многопоточных программах 920
- перемещаемый программный код 665
- перемещение 638, 652
  - абсолютных ссылок 656
  - относительных ссылок 655
  - ссылок 654
- перемещение байта в двойное слово с расширением знакового разряда 204
- перемещение байта в двойное слово с расширением нулями 204
- перемещение байта в слово с расширением знакового разряда 204
- перемещение байта в слово с расширением нулями 204
- перемещение байта в четверное слово с расширением знакового разряда 204
- перемещение байта в четверное слово с расширением нулями 204
- перемещение байта, инструкция 202
- перемещение данных 202
- перемещение двойного слова в четверное слово с расширением знакового разряда 204
- перемещение двойного слова, инструкция 202
- перемещение с дополнением нулями 203
- перемещение слова в двойное слово с расширением знакового разряда 204
- перемещение слова в двойное слово с расширением нулями 204
- перемещение слова в четверное слово с расширением знакового разряда 204
- перемещение слова в четверное слово с расширением нулями 204
- перемещение слова, инструкция 202
- перемещение четверного слова, инструкция 202
- перенастройка синхронизации цепи 411
- переносимая обработка сигналов 728
- переносимость и типы данных 73
- переполнение 65
  - арифметическое 159
  - определение 120
  - при умножении 129
  - чисел с плавающей точкой 153
- переполнение буфера 286
  - ограничение областей выполняемого кода 294
- переупорядочение операций 520
- перехват сигналов 714, 718
- переход назад выбирается всегда, вперед никогда, стратегия прогнозирования 417
- переходом в середину, стратегия трансляции 237
- петлевой адрес 864
- пикосекунды (пс) 484
- планирование, процессов 694
- планировщик 694
- пластины, диски 562
- плоская адресация 188
- плотность записи на диски 564
- плотность размещения дорожек на дисках 564
- побочные эффекты 484
- поверхности дисков 562
- поверхностная плотность записи на дисках 564
- повторяющиеся имена 644
- подгонка методом наименьших квадратов 485
- поддомены 863
- подкачка (swapping) 758
- подмена библиотечных функций (library interpositioning) 668
  - во время выполнения 671
  - во время компиляции 669
  - во время компоновки 670
- подсистема ввода-вывода Unix 53
- подставной код (эксплоит exploit code) 289
- подсчет интервалов, схема 540
- позиционирование операция 565
- позиционно-независимые ссылки на данные 666, 667
- позиционно-независимые ссылки на функции 667
- позиционно-независимый код (Position-Independent Code, PIC) 661
- показатель степени в представлении с плавающей точкой 139
- полностью конвейерные функциональные блоки 503
- полностью связанные выполняемые файлы 658
- положительное переполнение 118
- получение сигналов 717
- пользовательский режим 686, 694
- попадание в кеш DRAM 756
- поразрядные операции с плавающей точкой 308
- порты ввода/вывода 570
- порядок байтов
  - в дисассемблированном коде 225
- последним пришел, первым ушел, принцип 208
- последовательная реализация Y86-64 378
  - анализ 401
- последовательное выполнение 218
- постоянная память (ROM) 559
- построение высокопроизводительных веб-серверов 663
- посылка сигнала 715
- посылка сигналов с клавиатуры 716
- поток выполнения 914
- поток Posix 915
- поток выполнения 902
- поток выполнения (нити) 50
- поток каталога 844

- потоконебезопасные функции 944
- правила обработки сигналов 721
- преднамеренные исключения 686
- предсказание
  - ветвления 230
- предсказание ветвления 499
  - штраф за неверное предсказание 500
- представление программ на машинном уровне
  - обзор 184
- представление с плавающей точкой
  - одинарной точности 73
- представление с плавающей точкой в программах 135
  - на языке C 150
  - округление 146
  - особые значения 141
  - поддержка 72
  - половинная точность 162
  - стандарт IEEE 754 139
- представление целых без знака 92
- представление чисел с плавающей точкой
  - арифметика 65
  - кодирование 65
- представления
  - в дополнительном коде 94
  - программного кода 81
  - строк 80
  - целочисленные 90
- преобразование 64-разрядного целого в число с плавающей точкой двойной точности 302
- преобразование 64-разрядного целого в число с плавающей точкой одинарной точности 302
- преобразование адресов 762
- преобразование с усечением числа с плавающей точкой двойной точности в целое 301
- преобразование с усечением числа с плавающей точкой двойной точности в 64-разрядное целое 301
- преобразование с усечением числа с плавающей точкой одинарной точности в 64-разрядное целое 301
- преобразование с усечением числа с плавающей точкой одинарной точности в целое 301
- преобразование целого в число с плавающей точкой двойной точности 302
- преобразование целого в число с плавающей точкой одинарной точности 302
- преобразование четверного слова в восьмерное слово 215
- преобразования
  - в десятичное представление 71
  - в нижний регистр 491
- препроцессор 39, 190
- препятствование оптимизации 479
- приведение типа 76
- приведение типов
  - явное 109
- привилегированные инструкции 694
- привилегированный режим 686
- привилегированный режим выполнения 684
- прикладной программный интерфейс (Application Program Interface, API) 60
- приоритет операторов сдвига 89
- проблема производитель–потребитель 931
- проблема читателей–писателей 933
- пробуксовка (thrashing) 759
- пробуксовка, кеша 593
- прогнозирование
  - в конвейерной реализации Y86-64 416
- прогнозирование ветвлений 416
- прогнозирование ветвлений
  - штраф за неверное прогнозирование 526
- программирование на машинном уровне
  - доступ к информации 198
  - историческая перспектива 187
- программируемая постоянная память 559
- программные объекты 68
- программные регистры
  - риски по данным 424
- программный код на машинном уровне 190
  - примеры 192
- программ, профилирование 538
- программы
  - из инструкций Y86-64 361
  - и процессы 709
- программы CGI 887
- продвижение данных 423
- проектирование логики
  - комбинационные цепи 369
  - принадлежность к множеству 375
- производительность
  - выражение 484
  - вытеснение регистров 525
  - кешей 602
  - низкоуровневая оптимизация 538
  - обзор 478
  - ограничивающие факторы 525
  - операций загрузки 531
  - последовательной реализации Y86-64 402
  - пример программы 487
  - стратегии повышения 537
- производитель–потребитель, модель 931
- произвольная стратегия замены 583
- промахи емкости 584
- промах кеша 455
- промах кеша DRAM 757
- промежутки между секторами диска 563
- пропускная способность 503
- пространства адресов 753
- пространственная локальность 577, 585
  - переупорядочение циклов 612
- пространство пользователя 685
- протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP) 881
- профилирование кода 480
- процедура потока выполнения 915
- процедуры 250



операции с плавающей точкой 305  
 рекурсивные 263  
 процессор 43  
 процессоры  
 многоядерные 49, 576  
 обзор 349  
 суперскалярные 59  
 тенденции 574  
 эффективное время цикла 574  
 процессы 660, 690  
 абстракция 49  
 выполнение, состояние 697  
 завершение, состояние 697  
 коды ошибок 703  
 приостановка, состояние 697  
 утилизация дочерних процессов 701  
 прямой доступ к памяти (Direct Memory Access, DMA) 45, 570  
 прямой код (величина знака) 99  
 прямой переход 223  
 прямой (тупоконечный, big endian) порядок следования байтов 74  
 пузырьки, в конвейерной обработке 422

## Р

рабочее множество 584  
 рабочий набор 759  
 равномерное приближение к нулю 141  
 разбиение свободных блоков 796  
 разблокировка мьютекса 929  
 развертывание цикла 485  
 развертывание циклов 510, 515  
 $k \times 1a$  521  
 обзор 510  
 разделение времени 692  
 раздельная компиляция 634  
 разделяемые библиотеки 53, 661  
 разделяемые объекты 661, 781  
 размер машинного слова 71  
 размер слова 43  
 размеры  
 данных 71  
 разрешение имен 638  
 разрешение ссылок  
 на повторяющиеся имена 644  
 разыменованние указателей 79  
 распространение программного обеспечения 662  
 расширение битового представления 106  
 расширение знакового разряда 106, 204  
 расширение набора переменных-аккумуляторов 545  
 расширительные слоты 570  
 реализация механизма распределения динамической памяти  
 выделения блоков 806  
 освобождение и объединение блоков 805  
 создание списка свободных блоков 803  
 реализация механизма распределения памяти  
 константы и макроопределения 802  
 метод близнецов 810

простое разделение памяти 809  
 разделение с учетом размера 809  
 раздельные списки свободных блоков 808  
 сборка мусора 811  
 реализация управляющей логики в конвейерной обработке 448  
 регистра спецификатор в Y86-64 356  
 регистров блок 355  
 регистр состояния  
 риски по данным 424  
 регистр флагов  
 риски по данным 424  
 регистры 43  
 блоки регистров 191  
 локальные 506  
 программные 352  
 только для записи 506  
 только для чтения 506  
 цикла 506  
 регистры кодов условий 191  
 регистры общего назначения 198  
 реентерабельные функции 722, 946  
 режим супервизора 694  
 режимы адресации 201  
 резидентный набор 759  
 риски конвейерной обработки 418  
 риски по данным 418  
 классы 424  
 риски по загрузке / использованию данных  
 решение 428  
 риски по управлению 418, 429  
 Ритчи, Деннис 67  
 Ричард У. Стивенс (W. Richard Stevens) 729  
 родительский каталог 832  
 родительский процесс 697  
 роль указателей в С 68  
 рычаг актуатора 565

## С

сбои 686  
 сбой страницы 688  
 сборщики мусора 787  
 Свифт, Джонатан 75  
 сдвига, операции  
 для умножения 128  
 сегмент Ethernet 856  
 сегмент данных 658  
 сегмент кода 658  
 секторы дисков 563  
 семафоры 928  
 сервер, процесс 854  
 сетевой порядок следования байтов 861  
 сетевые адаптеры 570  
 сетевые приложения  
 запросы 855  
 сети  
 всемирная сеть интернет 860  
 механизм доставки 858  
 соединения 866  
 программные порты 866  
 сигналы 681, 697, 712

прием 714  
 терминология 714  
 символические методы 451  
 символические ссылки 832  
 символьные устройства 832  
 синхронизация 376  
     в последовательной реализации SEQ 391  
 синхронизация потоков 730  
 синхронизированные регистры 389  
 система компиляции 39  
 системная шина 560  
 системные вызовы 50  
     обработка ошибок 695  
 системные прерывания 686  
 сквозная запись 600  
 скорость вращения дисков 562  
 слабое масштабирование 943  
 слабо определенные имена 644  
 следующий подходящий, стратегия поиска 796  
 слова 43, 197  
     размер 43  
 сложение  
     Y86-64 354  
     с плавающей точкой 306  
     целых в дополнительном коде 118  
     чисел с плавающей точкой 149  
 слой трансляции флеш-памяти 572  
 смещение  
     при делении 132  
     смещение в виртуальной странице 763  
     смещение в физической странице 764  
 событие 682  
 совместно используемые переменные 922  
 современные процессоры,  
     производительность 498  
 сокет 866  
 сокращение задержек путем откладывания выполнения 902  
 сопряжение с системой памяти 455  
 сортировки производительность 541  
 составные типы данных 191  
 состояние 682  
     бистабильное 554  
     видимое программисту 352  
 состояние гонки 948  
 спекулятивное выполнение 499, 527  
 сравнение  
     с плавающей точкой 309  
 сравнение байтов, инструкции 219  
 ссылки на память  
     операнды 201  
 стандартная библиотека ввода/  
     вывода 830, 849  
 стандартная библиотека C 40  
 стандартный ввод 831  
 стандартный вывод 831  
 стандартный вывод ошибок 831  
 статическая память (SRAM) 554  
 статическая память с произвольным  
     доступом (Static Random Access  
     Memory, SRAM) 47

статические компоновщики 637  
 статический контент 882  
 стек 53, 208  
     кадры переменного размера 295  
     обнаружение повреждений 292  
 стираемая программируемая постоянная  
     память 559  
 страничные блоки 754  
 стратегия вытеснения наиболее давно  
     использовавшихся блоков (Least  
     Recently Used, LRU) 583  
 стратегия размещения 584, 796  
 строго определенные имена 644  
 строка описания формата 79  
 структура программы 63  
 структуры 273  
     в программировании на машинном  
     уровне 191  
 структуры данных  
     выравнивание данных 280  
     структуры 273  
 суперскалярные микропроцессоры 498  
 суперскалярные операции 456  
 суперскалярные процессоры 59  
 суперъядейки 556  
 схема именования 858  
 счетные семафоры 929  
 счетчики инструкций  
     %rip 191  
 счетчики инструкций (PC)  
     на этапе выборки 379  
 счетчик инструкций 76  
     риски по данным 424  
 счетчик инструкций (PC)  
     архитектура набора команд Y86-64 353  
 счетчик команд 43  
 С язык  
     логические операции 87  
     операции сдвига 88  
     представление чисел с плавающей  
     точкой 150

## Т

таблица  
     векторов прерываний 683  
     исключений 683  
     процессов 694  
     страниц 694  
     файлов 694  
 таблица виртуальных узлов v-node 845  
 таблица дескрипторов 845  
 таблица заголовков в формате ELF 638  
 таблица имен 639  
 таблица связывания процедур (Procedure  
     Linkage Table, PLT) 667  
 таблица файлов 845  
 таблицы  
     заголовков сегментов 658  
 таблицы переходов 247  
 твердотельные диски 560, 572  
 текстовые строки 832



текстовые файлы 39, 832  
 текст, представление  
   ASCII 80  
   Юникод (Unicode) 81  
 текущий рабочий каталог 833  
 тестирование проекта 450  
 типы  
   имена 78  
   на машинном уровне 197  
   связь с указателями 68  
 топологическая сортировка 699  
 точки входа 659  
 точность, в представлении с плавающей точкой 140  
 транзакции шины 560  
 транзакция записи 560  
 транзисторы в законе Мура 189  
 трансляция адресов 753  
 трассировка выполнения 381  
 требования к выравниванию 280  
 тупиковые ситуации 951

## У

увеличение объема динамической памяти 797  
 узкие места  
   профилировщиков 540  
 Уильям Кахан 135  
 указатели 68, 283  
   на функции 284  
   приведение к другому типу 283  
   создание 206  
   стека 251  
 указатель кадра 296  
 уменьшение высоты дерева 545  
 умножение  
   инструкции 215  
   целых без знака на степень двойки 128  
   целых в дополнительном коде на степень двойки 129  
   чисел с плавающей точкой 149  
 умножение с плавающей точкой 306  
 умножение целых без знака 124  
 умножение целых в дополнительном коде 124  
 унарные (одноместные) операции 212  
 универсальная последовательная шина (Universal Serial Bus, USB) 569  
 управление зависимостями в конвейерной обработке 408  
 управление кешем 584  
 управление потоком  
   переходы 222  
   процедуры 250  
 управления, поток 680  
 управляемые событиями, программы 909  
 управляющая логика в конвейерной обработке 441  
   выявление особых случаев 444  
   обработка особых случаев 441  
 управляющие структуры 218  
   флаги условий 218

уровни  
   оптимизации 481  
 усечение чисел 109  
 условное ветвление 192  
   в ассемблерном коде 227  
   потока данных 229  
 установить, если равно, инструкция 220  
 установка обработчика 718  
 устройства ввода-вывода 43  
 уязвимость в getpeername 114

## Ф

файл подкачки 781  
 файлы 53, 831  
   абсолютный путь 833  
   абстракция 49  
   блочные устройства 832  
   выполняемые 40  
   выполняемые объектные 39  
   двоичные 39  
   закрытие 832  
   каталоги 832  
   обычные 832  
   открытие файла 833  
   относительный путь 833  
   позиция в файле 831  
   пути к 833  
   совместное использование 845  
   создание файла 834  
   сокеты 832  
   текстовые 39  
   тип 832  
   чтение и запись 831  
 Фибоначчи (Пизано) 64  
 физическая адресация 752  
 физические адреса 752  
 физические страницы 754  
 физическое адресное пространство 753  
 флаг знака (sign flag) 218  
 флаги доступа к памяти 294  
 флаги условий  
   Y86-64 353  
 флаг переноса (carry flag) 218  
 флаг переполнения (overflow flag) 218  
 флаг признак нуля (zero flag) 218  
 флеш-память 560  
 фоновый режим 712  
 формат выполняемых и компокуемых модулей (Executable and Linkable Format, ELF) 638  
 форматированный вывод 79  
 формат переносимых выполняемых файлов (Portable Executable, PE) 638  
 формат скалярных данных 298  
 формат элементов таблиц страниц 774  
 формула строгого масштабирования 943  
 фрагментация памяти 792  
 функции  
   в инструкциях Y86-64 355  
 функции ввода/вывода Unix 830  
 функции-обертки 672

функции системного уровня 689  
 функциональные блоки 500, 503  
 функция вывода сообщения об ошибке 696  
 функция инструкции (ifun) 379

## Х

хеш-функция 542  
 холодные промахи 584  
 хранение информации 68

## Ц

целочисленная арифметика  
   обзор 134  
 целочисленные представления 90  
 целочисленные типы 91  
   фиксированного размера 95  
 целые без знака 98  
   кодирование 65  
 целые со знаком 72, 91, 98  
   кодирование 65  
   преобразование в целые без знака 99  
 целые числа 65, 89  
   порядок следования байтов 74  
   советы по приемам работы 111  
 центральные процессорные устройства (CPU)  
   первоначальные наборы команд 359  
 циклы 235  
   do-while 235  
   неэффективности 490  
   обратное проектирование  
     (воссоздание) 236  
 цилиндры дисков 563

## Ч

четверные слова 197  
 четности флаг условия 197

## Ш

шаблон обращений с шагом 1 578  
 шаблон обращений с шагом k 578  
 шаблон последовательных обращений 578  
 шина для подключения периферийных  
   компонентов (Peripheral Component  
   Interconnect, PCI) 569  
 шины 43, 560  
   архитектуры 561, 569  
 шпиндель диска 562

## Э

эксабайты 72  
 элементы таблицы страниц 755  
 энергонезависимая память 559  
 эталонные машины 489  
 этапы  
   реализация в последовательной версии  
     Y86-64 394  
 эфемерный порт 866  
 эффективный адрес 201  
 эхо-сервер, пример 879

## Ю

Юникод (Unicode) набор символов 81

## Я

явное ожидание сигналов 732  
 ядро операционной системы 683  
 язык описания гипертекстовых  
   документов (Hypertext Markup  
   Language, HTML) 882  
 ячейки  
   DRAM 556  
   SRAM 554

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу  
в книготорговой компании «Галактика»  
(представляет интересы издательств  
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;  
Тел.: +7(499) 782-38-89. Электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),  
по которому должны быть высланы книги;  
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:  
**www.galaktika-dmk.com**

Рэндал Э. Брайант  
Дэвид Р. О'Халларон

## **Компьютерные системы: архитектура и программирование**

3-е издание

Главный редактор *Мовчан Д. А.*

*dmkpress@gmail.com*

Зам. главного редактора *Сенченкова Е. А.*

Перевод *Киселев А. Н.*

Корректор *Синяева Г. И.*

Верстка *Паранская Н. В.*

Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.

Усл. печ. л. 80.76. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Книга предназначена студентам и преподавателям, а также будет полезна разработчикам, стремящимся понять, что происходит внутри системного блока, когда выполняются написанные ими приложения. Это позволяет разрабатывать компактный, надежный и эффективный программный код и при этом облегчает поиск и устранение возможных ошибок в программе.

- Представление данных и программ на машинном уровне
- Архитектура процессора
- Приемы оптимизации программ
- Компоновка объектных модулей
- Управление потоками выполнения
- Виртуальная память и управление ею
- Ввод/вывод на системном уровне
- Сетевое и параллельное программирование

Приведенные примеры для процессоров, совместимых с Intel (x86-64), написаны на языках С и ассемблера и предназначены для выполнения в операционной системе Linux. В конце каждой главы приведено множество упражнений для закрепления пройденного материала.

Полный набор ресурсов, включая лабораторные работы и исходный код примеров, можно найти на сайте [www.csapp.cs.cmu.edu](http://www.csapp.cs.cmu.edu).



**Рэндал Э. Брайант** — доктор наук, профессор, декан факультета информатики университета Карнеги-Меллона (г. Питсбург). Лауреат многочисленных премий и наград. В течение 35 лет Брайант читает курс по архитектуре персонального компьютера, алгоритмам и программированию.



**Дэвид Р. О'Халларон** — доктор наук, профессор факультета вычислительной техники и электротехники в университете Карнеги-Меллона (г. Питсбург). Преподает курсы по компьютерной архитектуре, проектированию параллельных процессоров и др. Отмечен многочисленными наградами.

«В этой книге информация подается не так, как в других, но именно в той форме, в какой я хотел бы читать свои лекции».

Джон Грайнер,  
Университет Райс

«Это уникальный проект, имеющий все шансы радикально изменить подходы к преподаванию предмета».

Майкл Скотт,  
Университет Рочестера

PEARSON



Интернет-магазин:  
[www.dmkpress.com](http://www.dmkpress.com)

Оптовая продажа:  
КТК «Галактика»  
[books@aliens-kniga.ru](mailto:books@aliens-kniga.ru)

ISBN 978-5-97060-492-2



9 785970 604922 >