

Г. Дейтел
Введение
в операционные
системы

1

UNIX
VAX
CP/M
MVS
VM

ЧАСТЬ 1

Обзор

Глава 1

Лишь в конце работы мы обычно узнаем, с чего нужно было ее начать.

Блез Паскаль

«С чего начинать, Ваше Величество?» — спросил он. «Начни с начала,— важно ответил Король,— и продолжай, пока не дойдешь до конца. Как дойдешь — кончай!» Приключения Алисы в стране чудес Льюис Кэрролл [2\)](#)

Кусочек подлинной истории — это такая редкая вещь, что им надо очень дорожить.

Томас Джефферсон

1.1 Введение

1.2 Поколения операционных систем



1.2.1 Нулевое поколение (40-е годы)

1.2.2 Первое поколение (50-е годы)

1.2.3 Второе поколение (начало 60-х годов)

1.2.4 Третье поколение (середина 60-х — середина 70-х годов)

1.2.5 Четвертое поколение (от середины 70-х годов до настоящего времени)

1.3 Первые операционные системы

1.4 Разработки начала 60-х годов

1.5 Компьютеры семейства System/360 фирмы IBM

1.6 Реакция промышленности на появление System/360

1.7 Системы с разделением времени

1.8 Появление новой инженерной дисциплины — технологии конструирования программ

1.9 Разделение цен на аппаратуру и программное обеспечение

1.10 Перспективы

1.1 Введение

Лишь немногие из тех, кто будет читать эту книгу, могут вспомнить время, когда операционных систем вообще не существовало. Сегодня операционные системы применяются практически на всех вычислительных машинах — от гигантских супер-ЭВМ до миниатюрных персональных компьютеров. Операционные системы зачастую даже в большей степени определяют представление пользователя о машине, чем сама аппаратура этой машины. Например, в сфере персональных компьютеров де-факто стандартной стала операционная система CP/M (см. гл. 21), так что очень многие фирмы — изготовители компьютеров создают новые аппаратные средства с ориентацией на эту операционную систему. Пользователь видит аппаратуру компьютера — скажем дисплей на электроннолучевой трубке и клавиатуру, — однако функциональные возможности машины ему обеспечивает операционная система CP/M 3).

В 1960-х годах операционную систему можно было определить как «программные средства, обеспечивающие управление аппаратурой». В настоящее время, однако, наблюдается явная тенденция к миграции многих функций от традиционных программ к микропрограммам, т. е. к микрокоду (см. гл. 2). Эта тенденция сейчас становится практически общепринятой — поэтому с большой вероятностью можно полагать, что в некоторых системах микрокод по объему реализуемых функций вскоре превзойдет обычные программные средства.

Очевидно, что необходимо иметь более четкое и адекватное определение операционной системы. Мы считаем, что операционная система — это набор программ, как обычных, так и микропрограмм, которые обеспечивают возможность использования аппаратуры компьютера. При этом аппаратура компьютера предоставляет «сырую» вычислительную мощность, а задача операционных систем заключается в том, чтобы сделать аппаратуру доступной и по возможности удобной для пользователей. Операционные системы обеспечивают также рациональное и координированное управление аппаратурой для достижения высокой производительности (см. ч. 6).

Однако независимо от того, какое именно определение мы выберем для операционной системы, ясно одно: она является неотъемлемой частью вычислительного комплекса, и поэтому каждый пользователь компьютера должен в той или иной степени представлять себе ее функции.

Некоторые из существующих в настоящее время операционных систем для своей работы требуют очень большой части ресурсов компьютера. Пользователи часто рассматривают это как своего рода «тайный заговор» изготовителей аппаратуры для увеличения объема продажи выпускаемых ими изделий. В то же время большинство изготовителей аппаратуры рассматривают операционные системы как неизбежное зло — поскольку без них аппаратуру вообще нельзя было бы продавать. Эти крайние позиции сейчас несколько сближаются благодаря появлению микро-ЭВМ и персональных компьютеров с относительно простыми операционными системами. Вместо того чтобы предлагать только крупные универсальные операционные системы с максимальными возможностями, что считали для себя обязательным промышленные фирмы в 60-х и начале 70-х годов, поставщики (в том числе независимые компании) сейчас предлагают также более простые операционные системы, которые позволяют создать удобные условия для вычислений (вычислительную обстановку) в соответствии с конкретными требованиями индивидуальных пользователей.

Главное назначение операционной системы — это управление ресурсами, а главные ресурсы, которыми она управляет, — это аппаратура компьютера. Операционная система реализует множество различных функций, в том числе:

- определяет так называемый «интерфейс пользователя»;
- обеспечивает разделение аппаратных ресурсов между пользователями;
- дает возможность работать с общими данными в режиме коллективного пользования;
- планирует доступ пользователей к общим ресурсам;
- обеспечивает эффективное выполнение операций ввода-вывода;
- осуществляет восстановление информации и вычислительного процесса в случае ошибок.

Операционная система управляет следующими основными ресурсами:

- процессорами;
- памятью;
- устройствами ввода-вывода;
- данными.

Операционная система взаимодействует с

- операторами ЭВМ;
- прикладными программистами;
- системными программистами; в административным персоналом;
- программами;
- аппаратными средствами;
- пользователями (например, банковскими кассирами).

Пользователи — это абоненты вычислительного комплекса, которые применяют компьютер для выполнения полезной работы. *Операторы ЭВМ* — это специально подготовленные люди, которые следят за работой операционной системы, по запросам системы вмешиваются в работу компьютера для устранения каких-либо препятствий, устанавливают и снимают кассеты магнитных лент и пакеты магнитных дисков, ставят и вынимают колоды перфокарт, обеспечивают правильную заправку в печатающие устройства бумаги нужного формата и т. д. Операторы ЭВМ играют очень важную роль с точки зрения обеспечения нормальной работы машины. Они выполняют те функции, которые пока что не удается автоматизировать.

Системные программисты занимаются обычно сопровождением операционной системы, осуществляют ее настройку применительно к требованиям конкретной машины и при необходимости доработку для обслуживания новых типов устройств.

Администраторы систем — это люди, которые устанавливают принципы и порядок работы на ЭВМ и взаимодействуют с операционной системой, чтобы обеспечить соблюдение принятого порядка.

Программы обращаются к операционным системам при помощи специальных команд, известных под различными названиями: *вызов супервизора, вызов монитора, запрос исполнительной программы* и т. д. Эти команды дают пользователям возможность прибегать к услугам, предоставляемым операционной системой, не нарушая ее целостности и работоспособности.

Операционной системе, как правило, присваивается статус *самого*

полномочного пользователя, Она имеет возможность доступа ко всем видам аппаратных ресурсов, всем программам пользователя, данным и т. д. В некоторых случаях определенные части операционной системы получают более узкие полномочия, поскольку людям и программам, работающим с этими частями, не требуется свободного доступа к ресурсам для выполнения любых операций, и поэтому лучше им его не предоставлять (см. гл. 17).

1.2 Поколения операционных систем

Операционные системы, подобно аппаратуре компьютеров, на пути своего развития прошли через ряд радикальных изменений, так называемых *поколений*. Для аппаратных средств смена поколений связана с принципиальными достижениями в области электронных компонент: вначале вычислительные машины строились на электронных лампах (первое поколение ЭВМ), затем на транзисторах (второе поколение), интегральных микросхемах (третье поколение), а сейчас — по преимуществу на больших и сверхбольших интегральных схемах (четвертое поколение). Появление каждого из этих последовательных поколений аппаратных средств сопровождалось резким уменьшением стоимости, габаритов, потребляемой мощности и тепловыделения и столь же резким повышением быстродействия и объемов памяти компьютеров.

1.2.1 Нулевое поколение (40-е годы)

В первых вычислительных машинах операционных систем не было. Пользователи имели полный доступ к машинному языку, и все программы писали непосредственно в машинных командах.

1.2.2 Первое поколение (50-е годы)

Операционные системы 50-х годов были разработаны с целью ускорения и упрощения перехода с задачи на задачу. До создания этих операционных систем много машинного времени терялось в промежутках между завершением выполнения одной задачи и вводом в решение следующей. Это было начало *систем пакетной обработки*, которые предусматривали объединение отдельных задач в группы, или *пакеты*. Запущенная в решение задача получала в свое полное распоряжение все ресурсы машины. После завершения каждой задачи (либо нормального, либо аварийного) управление ресурсами возвращалось операционной системе, которая «очищала машину после данной задачи» и обеспечивала ввод и запуск в решение следующей задачи.

1.2.3 Второе поколение (начало 60-х годов)

Характерной особенностью операционных систем второго поколения было то, что они создавались как системы коллективного пользования с *мультипрограммным режимом работы* (см. гл. 7) и как первые *системы мультипроцессорного типа* (см. гл. 11). В мультипрограммных системах несколько пользовательских программ одновременно находятся в основной памяти компьютера, а центральный процессор быстро переключается с задачи на задачу. В случае мультипроцессорных систем единый вычислительный комплекс содержит

несколько процессоров, что повышает вычислительную мощность этого комплекса.

В то время начали появляться методы, обеспечивающие *независимость программирования от (внешних) устройств*. Если в системах первого поколения пользователю, желающему произвести запись данных на магнитную ленту, приходилось в программе задавать конкретный номер физического лентопротяжного устройства, то в системах второго поколения программа пользователя только задавала, что файл должен быть записан на устройстве, имеющем определенное число дорожек и определенную плотность записи. Операционная система сама находила свободное устройство с требуемыми характеристиками и давала оператору ЭВМ указание установить кассету магнитной ленты на это устройство.

Были разработаны системы с *разделением времени*, которые предоставляли пользователю возможность непосредственно взаимодействовать с компьютером при помощи *пультов-терминалов* телетайпного типа. С системой разделения времени пользователи работают в *диалоговом*, или *интерактивном*, режиме. Пользователь печатает запрос для компьютера на своем терминале, компьютер обрабатывает этот запрос с максимальной возможной скоростью (часто в течение секунды или менее) и выдает ответ (если требуется) на терминал пользователя. Диалоговый режим работы позволил

в значительной степени повысить эффективность процесса разработки и отладки программ. Пользователь системы разделения времени получил возможность обнаруживать и исправлять ошибки буквально за считанные секунды или минуты — вместо того чтобы ждать, зачастую несколько часов или даже дней, пока будут получены результаты пакетной обработки.

Появились первые *системы реального времени*, в которых компьютеры применялись для управления технологическими процессами производства, например на предприятиях по переработке нефти. Были созданы военные системы реального времени, которые обеспечивали постоянный контроль сразу нескольких тысяч пунктов для защиты от внезапного воздушного нападения. *Для систем реального времени характерно то, что они обеспечивают немедленную реакцию на предусмотренные события*. Если, например, от датчиков системы управления нефтеперерабатывающего предприятия поступят сигналы о том, что температура становится слишком высокой, то может потребоваться немедленное принятие соответствующих мер для предотвращения взрыва. *Системы реального времени часто работают со значительной недогрузкой — поскольку для подобных систем гораздо важнее быть в состоянии постоянной готовности и быстро реагировать на предусмотренные события, чем просто быть занятыми большую часть времени. Этот факт позволяет понять, почему такие системы обычно стоят столь дорого.*

1.2.4 Третье поколение (середина 60-х — середина 70-х годов)

Третье поколение операционных систем фактически появилось с представлением фирмой IBM в 1964 г. семейства компьютеров System/360. Эти компьютеры третьего поколения были разработаны как *машины общего назначения*. Это были громоздкие, как правило, неэффективные компьютеры, которые предназначались для решения любых задач из любых областей приложения. Подобный подход позволил продать множество компьютеров, однако имел серьезные негативные стороны. Пользователям, которые решали конкретные прикладные задачи, не требующие всех функциональных возможностей подобных компьютеров, приходилось нести большие дополнительные расходы, поскольку они по сути оплачивали увеличенные затраты машинного времени, времени обучения, времени отладки, технического обслуживания и т. д.

Операционные системы третьего поколения были *многорежимными системами*. Некоторые из них обеспечивали работу сразу во всех известных режимах: пакетную обработку, разделение времени, режим реального времени и мультипроцессорный режим. Они были громоздкими и дорогостоящими. Поскольку ранее подобные системы не создавались, многие из разработок заканчивались со значительным превышением выделенных ассигнований и значительно позже первоначально запланированных сроков. (Примечательным

исключением в этом смысле является операционная система Unix, разработанная фирмой Bell Laboratories. Системы Unix различных версий подробно рассматриваются в гл. 18.)

Операционные системы третьего поколения привели к сильному усложнению вычислительной обстановки — и поначалу пользователи оказались не готовы к работе в новых условиях. Операционные системы стали как бы *программной прослойкой* между пользователями и аппаратурой ЭВМ. Эта программная прослойка часто была настолько «толстой», что пользователь просто терял из виду аппаратуру и видел только то ее отображение, которое создавало для него программное обеспечение. Для того чтобы заставить одну из подобных систем выполнить для него простейшую полезную задачу, пользователю приходилось изучать сложные *языки управления заданиями* — чтобы уметь описывать задания и требуемые для них ресурсы. Операционные системы третьего поколения представляли собой значительный шаг вперед в развитии средств программного обеспечения, однако этот шаг оказался для многих пользователей весьма болезненным.

1.2.5 Четвертое поколение (от середины 70-х годов до настоящего времени)

Операционные системы четвертого поколения — это наиболее совершенные системы настоящего времени (см. примеры современных операционных систем, гл. 18—22). Многие разработчики и пользователи еще помнят печальный опыт операционных систем третьего поколения и весьма осторожно относятся к переходу на более сложные операционные системы.

Благодаря широкому распространению *вычислительных сетей* (см. гл. 16) и средств оперативной обработки данных (в *режиме он-лайн*) пользователи получают возможность доступа к территориально распределенным компьютерам при помощи терминалов различных типов. Появление микропроцессора создало условия для разработки персонального компьютера, который с точки зрения социальных последствий стал одним из наиболее важных достижений вычислительной техники за несколько последних десятилетий. Сейчас многие пользователи имеют у себя собственные компьютеры, на которых они могут работать в любое время дня и ночи. Сейчас менее чем за тысячу долларов можно приобрести машину, которая по своей вычислительной мощности соответствует компьютерам, стоившим в начале 60-х годов сотни тысяч долларов.

Персональные компьютеры зачастую оснащаются интерфейсными средствами приема-передачи данных и могут использоваться также в качестве терминалов мощных вычислительных систем. Возможности пользователя системы четвертого поколения больше не ограничиваются взаимодействием с одним компьютером в режиме разделения времени — он может обращаться к территориально распределенным машинам вычислительной сети. Поскольку это требует, передачи информации по линиям связи различных видов, существенно усложнились проблемы защиты информации от возможного несанкционированного доступа (см. гл. 17). Сейчас уделяется большое внимание *шифрованию* данных, поскольку

стало обязательным кодировать секретные или личные данные таким образом, чтобы даже в случае их незаконного получения никто, кроме адресатов, которым они предназначались (или обладателей *ключей дешифрования*), не мог бы их использовать.

Число людей, имеющих возможность доступа к компьютерам, в 80-х годах в процентном отношении оказалось гораздо большим, чем когда-либо ранее, и продолжает быстро увеличиваться. Сейчас часто употребляют термины *дружественная, удобная для пользователя, ориентированная на неподготовленного пользователя* — они обозначают системы, которые предоставляют пользователям со средним уровнем квалификации простой доступ к вычислительным ресурсам. Если в 60-х и 70-х годах пользователям приходилось работать на языках программирования, включающих множество символических, мнемонических обозначений и сокращений, то в 70-х годах появились системы с *управлением при помощи меню*, предоставляющие пользователю ряд различных альтернатив выбора, причем выраженных на простом английском языке.

Начала широко распространяться концепция *виртуальных машин* (см. гл. 22). Пользователь получил возможность более не задумываться о физических деталях построения вычислительных машин (или сетей), с которыми он работает. Вместо этого пользователь имеет дело с функциональным эквивалентом компьютера, создаваемым для него операционной системой и называемым виртуальной машиной. Современный пользователь хочет, чтобы машина помогала ему эффективно выполнять его работу, и, как правило, не интересуется внутренними деталями устройства этой машины и ее функционированием.

Исключительно важную роль начали играть *системы баз данных*. Наше общество зиждется на информации, так что задача систем баз данных — обеспечивать возможность удобного и управляемого доступа к информации для тех, кто имеет на это право. За последнее время появились тысячи баз данных, предусматривающих оперативный доступ при помощи терминалов через сети связи.

Широкое внедрение получила концепция *распределенной обработки данных* (см. гл. 16). Сейчас мы считаем, что гораздо целесообразнее иметь вычислительные мощности непосредственно там, где они необходимы, вместо того чтобы передавать данные для обработки в какие-либо вычислительные центры.

1.3 Первые операционные системы

Считается, что первую операционную систему создала в начале 50-х годов для своих компьютеров IBM-701 Исследовательская лаборатория фирмы General Motors. В 1955 г. эта фирма и North American Aviation совместно разработали операционную систему для компьютера IBM-704. Создание ассоциации пользователей (SHARE), работающих на машинах фирмы IBM, способствовало широкому обсуждению и выработке основных требований к операционным системам к 1957 г. появилось много операционных систем для модели 704, разработанных самими пользователями.

Эти первые операционные системы были ориентированы преимущественно на сокращение времени, которое обычно тратилось на запуск задач в решение на компьютере (так называемое *подготовительное время*) и на удаление их из машины (*заключительное время*). Пока эти операционные системы не были созданы, задачи загружались в машину индивидуально, обычно с перфокарт, причем во время своего выполнения занимали все ресурсы машины. При этом каждая задача либо выполнялась до конца, либо ее выполнение аварийно

прекращалось из-за возникновения какой-либо ошибки. Затем оператор ЭВМ загружал программу, осуществляющую дампинг основной памяти, снимал кассеты магнитных лент, вынимал колоды перфокарт и распечатывал результаты. Наконец, загружалась следующая задача и цикл начинался заново. В подготовительное и заключительное время машина фактически простаивала.

Разработчики самых первых операционных систем пытались свести к минимуму такие простои, делая переход с задачи на задачу более плавным. Эти операционные системы повышали пропускную способность машины благодаря тому, что они организовывали групповую, или пакетную, обработку задач вместо индивидуальной. Задачи загружались на магнитную ленту обычно при помощи небольшой периферийной машины (сателлита), а затем решались на более крупном компьютере. По окончании каждой задачи управление передавалось назад специальному блоку операционной системы, обеспечивающему загрузку следующей задачи. Благодаря этому время переключения с задачи на задачу значительно уменьшалось.

Уже в первых операционных системах появилась концепция *имен системных файлов* как средства достижения определенной степени независимости программ от аппаратуры. Это дало пользователю возможность не задавать непосредственно в программе конкретные номера физических устройств, а указывать *стандартный системный файл ввода* как устройство, с которого как бы считываются управляющие перфокарты, или *стандартный системный файл вывода* как устройство для распечатки результатов. Стандартные имена файлов позволили обеспечить более эффективное взаимодействие ассемблеров и компиляторов с редакторами связей и загрузчиками, работающими с их выходными модулями.

К концу 50-х годов ведущие фирмы — изготовители компьютеров поставляли операционные системы со следующими характеристиками:

- пакетная обработка одного потока задач;
- наличие стандартных подпрограмм ввода-вывода, с тем чтобы пользователю не приходилось касаться деталей программирования процессов ввода и вывода на машинном языке;
- возможности автоматического перехода от программы к программе, позволяющие сократить накладные расходы на запуск новой задачи в решение;
- средства восстановления после ошибок, обеспечивающие автоматическую «очистку» машины в случае аварийного завершения очередной задачи и позволяющие запускать следующую задачу при минимальном вмешательстве оператора ЭВМ;
- языки управления заданиями, предоставляющие пользователям возможность достаточно подробно описывать свои задания и ресурсы, требуемые для их выполнения

В то время операционные системы использовались главным образом на крупных ЭВМ. Многие из малых машин общего назначения, например компьютеры серии IBM-1400, работали без операционной системы. Пользователи подобных малых машин, как правило, сами производили загрузку собственной системы управления вводом-выводом (IOCS), небольшого пакета программ, управляющего осуществлением операций ввода-вывода. Фактически IOCS, применяемая на этих первых машинах, явилась прообразом той операционной системы, которую мы видим сегодня. Некоторые функции первых операционных систем обеспечивали более эффективное использование аппаратных ресурсов, однако в целом пакеты IOCS были нацелены на упрощение программирования и уменьшение количества времени, необходимого для написания сложных программ ввода-вывода. Разработчики современных операционных систем основное внимание уделяют тому, чтобы сделать работу пользователей и программистов более эффективной,

поскольку трудозатраты квалифицированных специалистов представляют сейчас гораздо большую долю общей стоимости вычислительных систем и комплексов, чем аппаратные средства.

1.4 Разработки начала 60-х годов

В начале 60-х годов фирмы — изготовители компьютеров вместе с аппаратурой начали поставлять операционные системы, обладающие гораздо более широкими возможностями. В то время в области вычислительной техники наибольшую роль играли следующие фирмы:

- Bendix
- Burroughs
- Control Data Corporation
- General Electric
- Honeywell
- IBM
- NCR
- Philco
- RCA
- Sperry Univac

Операционные системы этого периода были ориентированы на пакетную обработку и нацелены главным образом на повышение *пропускной способности* дорогостоящей аппаратуры (т. е. на увеличение числа работ, выполняемых в единицу времени). Эти системы предусматривали мультипрограммный режим, при котором в основной памяти машины размещается несколько программ одновременно, а центральный процессор быстро переключается с программы на программу. *Главный фактор, способствующий успешному внедрению концепции мультипрограммирования в качестве средства повышения пропускной способности машины, был связан с существенным различием в скоростных характеристиках (быстродействие) устройств ввода-вывода и центрального процессора.* Программа, которая формирует запрос ввода-вывода и не может продолжить свое выполнение, пока требуемая операция ввода-вывода не завершится, в однопрограммной машине вызывала перевод центрального процессора (ЦП) в режим ожидания результата операции ввода-вывода. Таким образом терялся самый драгоценный ресурс системы — время ЦП, причем зачастую система могла потерять 50 000 тактов процессора, ожидающего завершения операции ввода-вывода. Разработчики осознали, что, разместив в основной памяти машины сразу несколько программ, можно с пользой занять это время процессора, для этого нужно только обеспечить переключение процессора на очередную программу, готовую к выполнению (и не ожидающую завершения операции ввода-вывода).

Затем появились мультипроцессорные системы и комплексы (см. гл. 11), в которых одновременно работают несколько процессоров — иногда как независимые машины, обменивающиеся данными друг с другом, а иногда как ряд взаимодействующих процессоров с общей памятью. Подобные системы по производительности существенно превосходят машины с одним центральным процессором.

Одним из наиболее значительных достижений того периода было создание и успешное внедрение системы бронирования и предварительной продажи билетов на самолеты американских авиалиний. Эта система (под названием SABRE) стала первой крупной *системой обработки транзакций*, в которой пользователи,

находящиеся на удалении от центральной ЭВМ, взаимодействовали с ней при помощи терминалов телетайпного типа. Системы обработки транзакций предусматривают диалог между пользователем и машиной, причем пользователь задает относительно простые вопросы (например, «Есть ли место на сегодняшний рейс 404?»), а машина быстро отвечает на них. Про подобные терминалы пользователя говорят, что они работают в *оперативном режиме* (режиме он-лайн).

Обработка транзакций потребовала, чтобы при разработке операционных систем учитывался целый ряд новых соображений. Появилась необходимость обеспечить возможность диалога компьютеров со своими пользователями, а это в свою очередь привело к тому, что существенно возросло значение таких человеческих факторов, как время ответа и простота и *удобство интерфейса «человек — машина»*. Вместо того чтобы просто выполнять одно или несколько пакетных заданий одновременно, компьютерам пришлось быстро реагировать на запросы, которые поступают с, быть может, сотен или даже тысяч терминалов, расположенных в различных местах. Поэтому большое внимание начали уделять организации *подсистем внешней памяти с прямым доступом* (см. гл. 12 и 13), обеспечивающих быстрый обмен данными. Важное значение приобрело сопряжение компьютеров с телефонными линиями связи и с высокоскоростными каналами передачи данных. Самым значительным достижением того периода можно, по-видимому, считать именно стыковку компьютеров и средств передачи данных.

В 1963 г. фирма Burroughs для своих компьютеров B5000 разработала операционную систему MCP («Главная управляющая программа»), в которой были реализованы многие концепции и режимы, практически стандартные для современных операционных систем;

- мультипрограммирование;
- мультипроцессорная обработка;
- виртуальная память;
- написание операционной системы на языке высокого уровня;
- возможность отладки программ на исходном языке.

1.5 Компьютеры семейства System/360 фирмы IBM

В апреле 1964 г. фирма IBM анонсировала свое семейство компьютеров под названием System/360, *что можно, по-видимому расценивать как самое важное событие в истории операционных систем.*

У пользователей компьютеров того периода аппетиты казались ненасытными — они требовали все больших и больших ресурсов и функциональных возможностей компьютеров. Когда очередная машина переставала их удовлетворять, им приходилось переходить на более мощную машину, причем зачастую совершенно отличную по конструкции и набору команд и содержащую другую операционную систему. Переход с машины на машину осуществлялся медленно и трудно, поскольку необходимо было преодолевать проблемы несовместимости.

Учитывая эти трудности, разработчики фирмы IBM сделали компьютеры System/360 совместимыми по архитектуре, предусмотрели в них возможность применения одной и той же операционной системы (OS/360) и увеличение вычислительной мощности при переходе от младших к старшим моделям семейства. Эта *концепция семейства совместимых компьютеров* стала затем практически стандартной для всех фирм промышленности. Если пользователь выбирал серию 360, он получал возможность в течение многих лет работать с компьютерами этой серии. Однако сразу же возникла очевидная проблема —

каким образом заставить пользователя впервые выбрать серию 360, если компьютеры этого семейства были несовместимыми с другими машинами того периода.

Фирма IBM решала эту проблему, предложив самый широкий набор *имитаторов* и *эмуляторов* за всю историю вычислительной техники. Имитаторы и эмуляторы — это средства, позволяющие одному компьютеру имитировать другой компьютер. Например, пользователь компьютера IBM-1401 мог теперь легко перейти на System/360, поскольку была обеспечена возможность выполнять на System/360 программы модели 1401 без всяких доработок. Имитаторы — это, как правило, чисто программные средства моделирования, а эмуляторы на практике требуют и некоторых модификаций аппаратных средств. Имитаторы обычно работают очень медленно, однако их создание обходится относительно дешево. Эмуляторы стоят дороже, однако позволяют гораздо быстрее выполнять старые программы. Благодаря имитаторам и эмуляторам пользователи получили возможность немедленно переходить на компьютеры System/360, с тем, чтобы впоследствии перевести свои программы в формат System/360 в удобные для себя сроки.

Несмотря на то, что машины System/360 не содержали некоторых из новых возможностей, уже реализованных в то время в компьютерах других фирм, например Burroughs, эти машины оказались непревзойденными в одном очень важном аспекте, а именно в отношении удовлетворения разнообразных практических нужд пользователя. Имитаторы и эмуляторы оказались очень удобными для пользователя, а операционная система OS/360 содержала самый обширный из когда-либо поставляемых набор *программ-утилит*, что существенно упрощало применение компьютеров.

1.6 Реакция промышленности на появление System/360

Анонсирование System/360 явилось для других фирм — изготовителей компьютеров серьезным ударом, заставившим их активизировать свои усилия по разработке новых изделий. Руководители и главные конструкторы фирм сразу поняли, что им придется менять стратегические принципы своей технической политики, чтобы обеспечить для своих изделий способность конкурировать с этим семейством.

Одним из стратегических принципов стало буквальное копирование архитектуры System/360 и разработка операционных систем,

аналогичных OS/360, с последующей продажей подобных машин по гораздо более низким ценам, чем устанавливала фирма IBM. Такую стратегию выбрала фирма RCA (США) при создании своей серии компьютеров Spectra/70, а также гигантские промышленные компании Siemens (ФРГ) и Hitachi (Япония).

Эти компании считали, что таким образом им удастся привлечь на свою сторону часть заказчиков из громадного контингента пользователей фирмы IBM. Однако их стратегия потерпела неудачу. В частности, подобные попытки фирмы RCA провалились с таким треском, что ей вскоре пришлось вообще прекратить работы в области вычислительной техники; при этом она списала затраты в сумме 500 млн. долл.— это была наибольшая статья убытков в истории фирмы до того момента. Почему же на первый взгляд рациональная стратегия корпорации RCA привела к такому провалу? Эта фирма, несмотря на свои громадные размеры и прочное положение в промышленности, не завоевала достаточного доверия в кругах потенциальных пользователей, где решения об ориентации на те или иные компьютеры принимали осторожные финансисты и руководители подразделений по обработке данных. Кроме того, разработчики фирмы RCA слишком поздно

обнаружили, что задача копирования операционной системы OS/360 и связанных с ней программ является почти неразрешимой. Фирма оказалась в состоянии скопировать аппаратные средства, однако существенно недооценила, с какими трудностями и затратами ей придется столкнуться при создании операционной системы, подобной OS/360. Операционная система фирмы RCA в некоторых отношениях даже превосходила OS/360, однако надежность системы была низкой; кроме того, фирма не наладила хорошего сопровождения своих систем в эксплуатации на объектах заказчиков.

Другую стратегию в тот период приняли такие фирмы — изготовители компьютеров, как Burroughs и General Electric; они разрабатывали машины, не совместимые с компьютерами серии 360, но более мощные и более экономичные. Эта стратегия также выглядела весьма дальновидной, однако разработка новых машин оказалась гораздо более сложной задачей, чем предполагали руководители фирмы General Electric, и вскоре эта фирма вслед за RCA прекратила работы в области вычислительной техники.

Некоторые из фирм-изготовителей того времени действовали более осторожно. Они разрабатывали усовершенствованные компьютеры, напоминающие машины фирмы IBM серий 1400 и 7000. Эти поставщики надеялись привлечь тех заказчиков фирмы IBM, которые хотели бы избежать трудностей перехода на System/360. Фирмы Univac и Control Data добились определенных успехов в привлечении некоторых прежних заказчиков компьютеров фирмы IBM серии 7000, а компания Honeywell начала поставлять свои машины многим пользователям компьютеров серии 1400.

Надежды разработчиков фирмы IBM на то, что им удастся ограничиться одной операционной системой для всего своего семейства IBM/360, оказались несостоятельными из-за различных требований, предъявляемых различными пользователями. Так, пользователям малых компьютеров не нужна была сложная операционная система с бесконечным набором возможностей. Фактически производительность малых компьютеров при работе под управлением OS/360 существенно снижалась.

Пользователям более крупных компьютеров требовались операционные системы с расширенными функциональными возможностями. Кроме того, более мощные машины могли работать с большим числом операционных систем — и поэтому вместо одной операционной системы, OS/360, фирма IBM в конце концов создала четыре основные операционные системы, которые она поставляла в течение 60-х годов:

- DOS/360 (дискровая операционная система) для младших моделей системы System/360;
- OS/MFT (мультипрограммирование с фиксированным числом задач) для средних и старших моделей System/360;
- OS/MVT (мультипрограммирование с переменным числом задач) для старших моделей System/360;
- CP-67/CMS (управляющая программа-67/диалоговая мониторная система) для мощной модели 360/67, имеющей виртуальную память и предусматривающей работу в режиме разделения времени.

1.7 Системы с разделением времени

В конце 50-х и в начале 60-х годов разработчики промышленных фирм и университетов создали ряд систем с разделением времени. Эти системы дают возможность нескольким пользователям, работающим на терминалах, вести

непосредственный диалог с компьютером. От систем обработки транзакций, например системы SABRE американских авиалиний, они отличаются тем, что являются по сути вычислительными комплексами универсального типа и предоставляют своим пользователям полный набор вычислительных средств. Пользователи систем разделения времени, как правило, занимаются разработкой программ или работают со специально разработанными пакетами прикладных программ.

Быть может, наиболее известной системой разделения времени того периода стала система CTSS (совместимая система разделения времени), разработанная в Массачусетском технологическом институте (МТИ) при финансовой поддержке правительства группой научных сотрудников под названием Project MAC. Система CTSS работала на компьютере IBM-7094 со специально модифицированными аппаратными средствами. С 1961 до конца 1960-х годов в диалоговом режиме ее эксплуатировало большое число пользователей. *Она показала, насколько ценным является режим разделения времени с точки зрения повышения эффективности проектирования программ и насколько коллективное использование программ и данных способствует повышению производительности труда и творческой отдачи квалифицированных специалистов по вычислительной технике.*

Дополнительным доказательством ценности и эффективности режима разделения времени для проектирования программ явилось применение системы CTSS для разработки в МТИ системы разделения времени следующего поколения, а именно системы MULTICS (Multiplexed Information and Computing Service). Система MULTICS —• результат совместной работы группы Project MAC (МТИ), фирм Bell Laboratories и General Electric (впоследствии Honeywell) с целью создания крупного универсального вычислительного комплекса с разделением времени на основе концепции *виртуальной памяти*. Одной из примечательных особенностей системы MULTICS являлось то, что она была написана большей частью на языке высокого уровня EPL, первой версии языка ПЛ/1 фирмы IBM. Программы отрабатывались и редактировались на системе CTSS и передавались для выполнения на компьютер GE-645.

За период с середины до конца 60-х годов многие группы разработали ряд подобных систем, причем наибольшую известность получили следующие системы;

- MULTICS;
- TSS фирмы IBM для модели 360/67;
- CP-67/CMS фирмы IBM, также для модели 360/67;
- VMOS фирмы RCA;
- KRONOS фирмы CDC для ее серии компьютеров 6000.

Эти проекты помогли специалистам накопить важный опыт использования виртуальной памяти и понять основные проблемы управления ею. Однако эти проекты представляли интерес в основном для разработчиков и практически не привели к коммерческим успехам. Так, создание системы TSS оказалось весьма дорогостоящим, хотя и ценным учебным экспериментом для специалистов фирмы IBM.

1.8 Появление новой инженерной дисциплины — технологии конструирования программ

В течение 60-х годов было создано много операционных систем. Это были громадные конгломераты программ, написанных людьми, которые практически не понимали, что программы, как и аппаратуру, необходимо проектировать таким образом, чтобы они были Надежными, понятными и удобными для сопровождения.

Бесконечные часы и бесчисленные доллары затрачивались на поиск ошибок, которых вообще не должно было бы быть в системах. Ошибки,

допущенные на начальных этапах, обнаруживались лишь спустя много времени после поставки программных изделий заказчикам. Исправление этих ошибок обходилось исключительно дорого. Уход программистов, участвующих в разработке, зачастую приводил к тому, что большое количество программных модулей выбрасывали — и затем их переписывали новые люди, поскольку в существующих программах нельзя было разобраться. Этим проблемам пришлось уделять очень большое внимание, и в конце концов научные организации и промышленные фирмы начали выделять значительные ресурсы для решения задачи конструирования программных систем. Тем самым было положено начало развитию новой дисциплины — *технологии конструирования программ*.

В настоящее время технологии конструирования программ придают столь важное значение, что большинство университетов требуют включения этой дисциплины в учебные планы по вычислительным системам, причем утверждено несколько программ для получения ученых степеней в этой области. Институт инженеров по электротехнике и радиоэлектронике (ИИЭР) публикует журнал *Transactions on Software Engineering*. Должности разработчиков программного обеспечения зачастую носят такие названия, как «инженер-программист», «старший инженер-программист» и т.д. В этой области защищаются докторские диссертации (De80). Таким образом, многие неудачные разработки операционных систем в 60-х годах (Bg75) в немалой степени способствовали появлению такой дисциплины, как технология конструирования программ, и осознанию необходимости создания систематизированной методологии проектирования структурированных программных систем, надежных, понятных и удобных для сопровождения.

1.9 Разделение цен на аппаратуру и программное обеспечение

Вплоть до начала 70-х годов изготовители компьютеров, как правило, продавали изделия лишь одного вида, а именно аппаратуру. Операционные системы, вспомогательные служебные программы, пакеты прикладных программ, а также эксплуатационная документация и учебные материалы часто поставлялись пользователям бесплатно или за номинальную цену. Одним из любопытных следствий этого обстоятельства явилось то, что у поставщиков была тенденция относиться к программному обеспечению как к своего рода «подарку», и поэтому они практически отказывались нести ответственность за его качество. Кроме того, пользователи, получающие от поставщика компьютера бесплатное программное Обеспечение, редко считали целесообразным приобретать дополнительные программные средства у внешних третьих фирм или тратить время и деньги на разработку собственных версий программ.

В качестве интересного упражнения любознательный читатель может попытаться найти некоторые из контрактов на поставку компьютеров конца 60-х годов. В эти контракты обычно включали четко сформулированные пункты, в которых обязанности по эксплуатации программного обеспечения со всеми вытекающими отсюда последствиями возлагались исключительно на самих пользователей.

Поставщики компьютеров публиковали длинные перечни известных ошибок программного обеспечения и в качестве «любезности» направляли эти перечни своим заказчикам. А заказчики, обращавшиеся к поставщику компьютера с жалобой на какую-либо проблему программного обеспечения, нередко получали

сердитый ответ, что, если бы они посмотрели в список известных ошибок, они не стали бы пока что пользоваться данными возможностями программного обеспечения.

В начале 70-х годов фирма IBM разделила свои программные средства и аппаратуру. Это означало, что она установила отдельные цены на каждый вид средств, хотя продолжала поставлять то или иное базовое программное обеспечение бесплатно. Такое решение имело весьма важные последствия для всей вычислительной техники:

- Теперь поставщикам компьютеров стало гораздо труднее включать в свои контракты многословные пункты с отказом от обязательств в отношении программного обеспечения. Так или иначе, если вы получаете деньги за изделие, пользователь, по-видимому, может чувствовать большую уверенность в том, что изделие будет правильно функционировать и вы будете нести полную ответственность за это.
- Почти сразу же была создана независимая индустрия программного обеспечения как прямая реакция на разделение цен программных и аппаратных средств. Если какой-либо независимый поставщик мог предложить программные средства, которые стоили дешевле аналогичных средств фирмы IBM и при этом обладали более широкими функциональными возможностями, у него появлялись реальные шансы с выгодой продавать заказчикам фирмы IBM собственные программные изделия.
- Другие изготовители компьютеров последовали примеру фирмы IBM, так что разделение цен на программы и аппаратуру быстро стало стандартным подходом для всей промышленности.
- Пользователи получили серьезный стимул для того, чтобы тщательно анализировать многие существующие конкурирующие программные изделия при выборе программного обеспечения для своих машин.
- Важным следствием подобного разделения цен стало то, что поставщики начали более методично проводить при разработке своих программных средств модульный принцип, с тем чтобы их можно было продавать или предоставлять на них лицензии как на индивидуальные изделия.
- Появился дополнительный стимул для развития компьютеров, совместимых с машинами фирмы IBM. Многие компании, называемые изготовителями совместимых компьютеров (PCM), сейчас копируют аппаратное оборудование фирмы IBM. Они выпускают модели, обладающие увеличенной вычислительной мощностью, но при этом стоящие дешевле, чем соответствующие модели фирмы IBM. Пользователи с большей охотой приобретают компьютеры независимых компаний, совместимые с машинами фирмы IBM, поскольку они могут получать для них программное обеспечение этой фирмы.

1.10 Перспективы

В настоящее время можно выделить ряд четких тенденций, которые позволяют судить о том, как будут выглядеть машины и системы новых поколений;

- Цены на аппаратные средства компьютеров будут продолжать снижаться, в то время как скорости процессоров и емкости памяти будут увеличиваться при одновременном уменьшении габаритов этих устройств.
- Будет продолжать возрастать «степень интеграции», так что в течение следующих 10 лет произойдет переход от СБИС (сверхбольших интегральных схем) к УБИС (ультрабольшим интегральным схемам).
- Получат гораздо более широкое распространение многопроцессорные архитектуры.

- Многие функции операционных систем, реализуемые в настоящее время при помощи обычных программ, перейдут к микрокоду.
- В новых поколениях аппаратных архитектур управление будет распределено по локальным процессорам.
- Сейчас разрабатываются языки параллельного типа, а также создаются аппаратные средства и операционные системы, обеспечивающие более эффективное выполнение параллельных программ (см. гл. 3, 4, 5, 6 и 10).
- Обычным явлением станет всеобъемлющий параллелизм. Появится возможность с большой скоростью выполнять параллельные программы благодаря очень высокой степени совмещения операций.
- Компьютеры и их операционные системы будут разрабатываться таким образом, чтобы обеспечить эффективную работу виртуальных машин, а реальные машины будут скрыты от пользователя.
- Будет по-прежнему сохранять свое важное значение концепция семейства компьютеров, впервые реализованная в System/360 фирмы IBM. Пользователи смогут выполнять свои прикладные программы на многих различных моделях компьютеров одного семейства, причем видимыми для этих прикладных программ будут только виртуальные машины.
- При появлении компьютеров новых поколений существующие программы можно будет выполнять на них без всяких изменений или при минимальных доработках.
- Дальнейшее развитие технологии конструирования программ приведет к созданию операционных систем, более простых в сопровождении, надежных и понятных.
- Стоимость средств передачи данных будет продолжать снижаться, а скорости передачи — расти.
- Еще более широкие масштабы приобретет объединение компьютеров в вычислительные сети, причем может оказаться так, что работа, выполняемая для конкретного пользователя, будет делаться на компьютере, о существовании которого этот пользователь даже не подозревает. Тем самым получит дополнительное подтверждение важность концепции виртуальных машин.
- По-прежнему сохранит свое значение концепция виртуальной памяти.
- Существующее представление об операционной системе как об администраторе ресурсов сохранится, однако сами ресурсы, которыми она управляет, изменятся. В частности, данные все в большей степени будут рассматриваться как ресурс, требующий эффективного управления.
- Дальнейшее внедрение принципа распределенной обработки вызовет появление и развитие распределенных операционных систем, в которых отдельные функции будут реализовываться в многочисленных процессорах, входящих в состав крупных вычислительных сетей.
- Повсеместное распространение получат персональные компьютеры. Максимально полная загрузка подобного компьютера будет иметь гораздо меньшее значение, чем его постоянная готовность к работе, надежность, гибкость и простота использования.

Заключение

Операционная система состоит из обычных программ или микропрограмм, которые обеспечивают возможность эффективного использования аппаратуры. Операционная система — это по преимуществу администратор ресурсов; она управляет процессорами, памятью, устройствами ввода-вывода и данными.

В своем развитии операционные системы прошли через ряд поколений. Вычислительные машины нулевого поколения в 40-х годах не имели операционных систем. В 50-х годах в машинах первого поколения появились

возможности пакетной обработки. В начале 60-х годов в системах второго поколения были впервые

реализованы такие режимы вычисления, как мультипрограммирование, мультипроцессорный режим, разделение времени и режим реального времени, а также концепция независимости программ от устройств ввода-вывода. Системы третьего поколения (середина 60-х — середина 70-х годов) были по преимуществу универсальными, они предусматривали работу во многих режимах. Они стали как бы программной прослойкой между аппаратными средствами и пользователем. В настоящее время доминирующее положение занимают системы четвертого поколения, в том числе средства для вычислительных сетей, для персональных компьютеров, операционные системы виртуальных машин, системы баз данных и системы распределенной обработки данных.

В 1964 г. фирма IBM представила свое семейство компьютеров System/360. Это семейство включало ряд программно-совместимых компьютеров различной вычислительной мощности. Пользователи получили возможность начать работу с какой-либо малой моделью System/360, а затем по мере роста потребностей легко перейти на более мощные машины. Чтобы помочь пользователям в переходе на компьютеры серии 360, фирма IBM разработала обширный набор имитаторов и эмуляторов предыдущих компьютеров.

Организованная в МТИ группа Project MAC сыграла важную роль в создании систем разделения времени: в начале 60-х годов она разработала систему CTSS, а в конце 60-х годов — систему MULTICS.

Разработчики многих крупных операционных систем в 60-х годах испытывали большие трудности. Множество проблем, возникавших при работе над этими проектами, заставили ученых и специалистов промышленности создать теоретические основы конструирования надежных, простых в сопровождении и понятных программных систем. Тем самым было положено начало новой дисциплине — технологии конструирования программ.

В начале 70-х годов фирма IBM разделила цены на аппаратные средства и программное обеспечение, что вызвало ряд радикальных изменений в промышленности по выпуску компьютеров. В основном эти изменения были направлены на разработку программных изделий лучшего качества и способствовали обострению конкуренции. Наибольшую выгоду от разделения цен получили фирмы — изготовители компьютеров, совместимых с машинами фирмы IBM.

В числе будущих тенденций можно выделить следующие: продолжающееся снижение стоимости вычислений; переход от СБИС к УБИС; широкое распространение мультипроцессорных архитектур; реализация многих функций операционных систем при помощи микрокода, а не обычных программ; использование локальных процессоров; дальнейшее развитие и внедрение параллелизма; более широкое использование концепции виртуальных машин; продолжающееся использование концепции семейства компьютеров; улучшение методологии конструирования программных средств; снижение стоимости и повышение скоростей передачи данных; увеличение масштабов внедрения вычислительных сетей; продолжающееся использование концепции виртуальной памяти; дальнейшее развитие методов управления данными как ресурсами; широкомасштабное применение распределенной обработки данных, а также повсеместное распространение персональных компьютеров дома и на производстве.

Терминология

администратор системы (system administrator)

виртуальная машина (virtual machine)

виртуальная память (virtual storage)

вызов монитора (monitor call)

вызов супервизора (supervisor call, SVC)

вычислительные сети (организация) (computer networking)

Главная управляющая программа (Master Control Program (MCP))

дешифрование (decryption)

диалоговый (интерактивный) режим (conversational mode)

журнал ИИЭР «Проблемы конструирования программных средств» (IEEE Transactions of Software Engineering)

заключительное время (teardown time)

запрос исполнительной программы (executive request)

изготовитель совместимых компьютеров (plug compatible mainframe, PCM)

имена системных файлов (system file names)

имитатор (simulator)

интерактивный (диалоговый) режим (interactive mode)

исследовательская лаборатория фирмы General Motors (General Motors Research Laboratories)

компьютер-спутник (периферийная машина) (satellite computer)

крупный компьютер (mainframe)

микрокод (microcode)

микропрограммное обеспечение (firmware)

многорежимные системы (multimode systems)

мультипрограммирование (multiprogramming)

мультипроцессорная обработка (multiprocessing)

независимость от (внешних) устройств (device independence)

непосредственное указание номеров физических устройств в программе (hard coding physical device numbers)

номера физических устройств (physical device numbers)

оперативный режим (он-лайн) (on-line processing)

оператор ЭВМ (computer operator)

ориентированный на пользователя, «дружественный», удобный для пользователя (user friendly)

операционные системы первого поколения (first generation operating systems)

операционные системы второго поколения (second generation operating systems)

операционные системы третьего поколения (third generation operating systems)

операционные системы четвертого поколения (fourth generation operating systems)

организация памяти прямого доступа (direct access storage organization)

пакет заданий (batch of jobs)

пакетная обработка (batch processing)

переключение с задания на задание (job-to-job transition)

периферийная машина (компьютер-сателлит) (satellite computer)

персональный компьютер (personal computer)

подготовительное время (setup time)

поколения (generations)

пользователь (user)

программная прослойка (software layer)

пропускная способность (throughput)

разделение времени (timesharing)

разделение цен аппаратных и программных средств (unbundling)

распределенная обработка данных (distributed data processing)

самый полномочный пользователь (most trusted user)

семейство (серия) компьютеров (family of computers)

система обработки транзакций (transaction processing system)

система разделения времени TSS 360/67

система реального времени (real-time systems)

системный программист (system programmer)

системы баз данных (database systems)

системы (машины) общего назначения (универсальные машины, системы) (general-purpose systems)

стандартный системный файл ввода (standard system input file)

стандартный системный файл вывода (standard system output file)

терминал (terminal)

технология конструирования программ (software engineering)

удобный для пользователя, ориентированный на пользователя (user friendly)

управление при помощи меню (альтернатив выбора) (menudriven)

эмулятор (emulator)

шифрование (encryption)

язык управления заданиями (Job Control Language, JCL)

CP/M IOCS

CP-67/CMS KRONOS

CTSS MCP

DOS/360 MULTICS

IBM OS/MFT
System/360

 OS/MVT

IBM-701

 VMOS

Project, MAC

SABRE

SHARE

TSS 360/67

Упражнения

1.1 Почему неадекватным является определение операционной системы как «программных средств, обеспечивающих управление аппаратурой»? Дайте более удачное определение.

1.2 Перечислите различные объекты, с которыми должна взаимодействовать операционная система. Кратко опишите природу каждого из соответствующих интерфейсов.

1.3 В чем суть того, что операционной системе предоставляется статус «самого полномочного пользователя»?

1.4 Опишите различия между мультипрограммным и мультипроцессорным режимами работы.

1.5 Что такое независимость программ от (внешних) устройств? Почему это так важно для пользователей?

1.6 Дайте определения каждого из следующих терминов и укажите их сходства и различия:

- а) оперативный режим (он-лайн)
- б) режим разделения времени
- в) диалоговые вычисления
- г) режим реального времени
- д) интерактивные вычисления
- е) обработка транзакций

1.7 Что такое «программная прослойка»? Каким образом она влияет на представление пользователя о компьютере?

1.8 Что означает «дружественный по отношению к пользователю»? Почему это важно?

1.9 В чем заключались главные цели разработчиков первых операционных систем?

1.10 Что такое имена «стандартных файлов»? В чем состоит их полезность?

1.11 Какова основная причина появления мультипрограммирования?

1.12 Назовите несколько факторов, позволяющих считать появление System/360 фирмы IBM важнейшим событием в истории развития операционных систем.

1.13 Что такое имитаторы и эмуляторы? Что вы предпочли бы, чтобы обеспечить быстрый и экономичный переход с компьютера на компьютер? Что вы предпочли бы, если бы вам нужно было решать большие производственные задачи на новой машине с использованием программ старого компьютера?

Глава 2

Аппаратура, программное обеспечение и микропрограммы

Все свое время, здоровье и состояние я принес в жертву желанию завершить создание этих машин для вычислений. Я отклонил также ряд предложений, суливших для меня большие личные выгоды. Однако, несмотря на мой отказ от всех этих выгод с целью совершенствования машины, обладающей почти интеллектуальной мощностью, и после того как я истратил из моего собственного состояния больше денег, чем правительство Англии выделило на эту машину (причем только в начале работ), я не получил ни благодарности за мои труды, ни даже тех почестей и наград, которые, как правило, достаются людям, посвятившим себя чисто научным исследованиям...

□ *Если бы работа, на которую я затратил так много времени и душевных сил, была только победой над обычными техническими трудностями или просто курьезом, или если бы были сомнения в целесообразности или полезности подобных машин, то можно было бы в какой-то мере понять и оправдать такое отношение; однако я позволю себе утверждать, что ни один достойный уважения математик никогда не рискнет публично выразить мнение, что подобная машина, если она будет сделана, окажется бесполезной,— и что ни один человек, известный как грамотный инженер, не рискнет объявить построение подобных машин нецелесообразным.*

Я считаю, что в то время, когда серьезным препятствием на пути прогресса физических наук становятся громадные объемы интеллектуального и ручного труда, необходимого для дальнейшего продвижения, а именно облегчить этот труд и призвана аналитическая машина, применение машин как помощников в выполнении наиболее сложных и трудоемких вычислений нельзя более полагать проблемой, недостойной внимания страны. И действительно, нет никаких причин, не позволяющих экономить труд умственный, подобно физическому, благодаря использованию машин.

Чарльз Бэббидж Отрывок из книги «Жизнь философа»

Элементы, из которых строятся компьютеры, становятся все более малыми, они выходят за границы возможностей обычных микроскопов — в бесконечные пространства мира молекул.

Кристофер Эванс

Программы — это посредники между пользователем и машиной.

Гарлен Д. Миллс

Микропрограммирование — это реализация предположительно разумных систем путем интерпретации на неразумных машинах!

Р. Ф. Розин

Микропрограммирование неэффективного алгоритма не делает его эффективным,

Закон Роше

2.1 Введение

2.2 Аппаратура

2.2.1 Расслоение памяти

2.2.2 Регистр перемещения

2.2.3 Прерывания и опрос состояний

2.2.4 Буферизация

2.2.5 Периферийные устройства

2.2.6 Защита памяти

2.2.7 Таймеры и часы

2.2.8 Работа в режиме он-лайн и автономный режим (оф-лайн), периферийные процессоры

2.2.9 Каналы ввода-вывода

2.2.10 Захват цикла

2.2.11 Относительная адресация

2.2.12 Режим задачи, режим супервизора, привилегированные команды

2.2.13 Виртуальная память

2.2.14 Мультипроцессорная обработка

2.2.15 Прямой доступ к памяти

2.2.16 Конвейеризация

2.2.17 Иерархия памяти

2.3 Программное обеспечение

2.3.1 Программирование на машинном языке

2.3.2 Ассемблеры и макропроцессоры

2.3.3 Компиляторы

2.3.4 Системы управления вводом-выводом (IOCS)

2.3.5 Спулинг

2.3.6 Процедурно-ориентированные и проблемно-ориентированные языки

2.3.7 Быстрые компиляторы без оптимизации и оптимизирующие компиляторы

2.3.8 Интерпретаторы

2.3.9 Абсолютные и перемещающие загрузчики

2.3.10 Связывающие загрузчики и редакторы связей

2.4 Микропрограммы

2.4.1 Горизонтальный и вертикальный микрокод

2.4.2 Выбор функций для микропрограммной реализации

2.4.3 Эмуляция

2.4.4 Микродиагностика

2.4.5 Специализированные компьютеры

2.4.6 Микропрограммная поддержка

2.4.7 Микропрограммирование и операционные системы

2.4.8 Пример микропрограммирования

2.1 Введение

В данной главе мы кратко рассмотрим, что такое аппаратура, программное обеспечение и микропрограммы. Все они играют важную роль с точки зрения функционирования вычислительных машин и управления ими. *Аппаратные средства* — это устройства вычислительной машины; ее процессоры, устройства памяти и устройства ввода-вывода, а также средства приема-передачи данных. *Программное обеспечение* — это программы, содержащие команды на машинном языке и данные, которые соответствующим образом интерпретируются аппаратурой машины. В качестве примеров некоторых распространенных видов программных средств можно привести компиляторы, ассемблеры, загрузчики, редакторы связей, связывающие загрузчики, прикладные программы пользователя, системы управления базами данных, системы приема — передачи данных и операционные системы. *Микропрограммы* — это микрокодированные программы, выполняемые непосредственно из управляющей памяти очень высокого быстродействия. Наиболее часто используемые объектные программы, занесенные в постоянную память (постоянные запоминающие устройства и программируемые ПЗУ), также иногда называют микропрограммным обеспечением.

О микропрограммировании и его важной роли для современных машинных архитектур и операционных систем идет речь в последнем разделе настоящей главы.

2.2 Аппаратура

В нескольких следующих разделах рассматриваются различные виды аппаратных средств, которые имеют важное значение с точки зрения работы операционных систем. Читатель, которому требуются более подробные сведения об аппаратуре, может обратиться к таким книгам-учебникам по архитектуре компьютеров, как (Ba80), (1182) и (Le80).

2.2.1 Расслоение памяти

Метод расслоения памяти (интерливинг) применяется для увеличения скорости доступа к основной (оперативной) памяти. В обычном случае во время обращения к какой-то одной из ячеек модуля основной памяти никакие другие обращения к памяти производиться не могут. При расслоении памяти соседние по адресам ячейки размещаются в различных модулях памяти, так что появляется возможность производить несколько обращений одновременно. Например, при расслоении на два направления ячейки с нечетными адресами оказываются в одном модуле памяти, а с четными — в другом. При простых последовательных обращениях к основной памяти ячейки выбираются поочередно. Таким образом, расслоение памяти позволяет обращаться сразу к нескольким ячейкам, поскольку они относятся к различным модулям памяти.

2.2.2 Регистр перемещения

Регистр перемещения обеспечивает возможность динамического (перемещения программ в памяти. В регистр перемещения заносится базовый адрес программы, хранящейся в основной памяти. Содержимое регистра перемещения прибавляется к каждому указанному в выполняемой программе адресу. Благодаря этому пользователь может писать программу так, как если бы она начиналась с нулевой ячейки памяти. Во время выполнения программы все исполнительные адреса обращений формируются с использованием регистра перемещения — и благодаря этому программа может реально размещаться в памяти совсем не в тех местах, которые она должна была бы занимать согласно адресам, указанным при трансляции.

2.2.3 Прерывания и опрос состояний

Одним из способов, позволяющих некоторому устройству проверить состояние другого, независимо работающего устройства, является *опрос*; например, первое устройство может периодически проверять, находится ли второе устройство в определенном состоянии, и если нет, то продолжать свою работу. Опрос может быть связан с высокими накладными расходами.

Прерывания дают возможность одному устройству немедленно привлечь внимание другого устройства, с тем чтобы первое могло сообщить об изменении своего состояния. *Состояние* устройства, работа которого прерывается, должно быть сохранено, только после этого можно производить обработку данного прерывания. После завершения обработки прерывания состояние прерванного устройства восстанавливается, с тем чтобы можно было продолжить работу. Прерывания подробно обсуждаются в гл. 3.

2.2.4 Буферизация

Буфер — это область основной памяти, предназначенная для промежуточного хранения данных при выполнении операций ввода-вывода. Скорость выполнения операции ввода-вывода зависит от многих факторов, связанных с характеристиками аппаратуры ввода-вывода, однако в обычном случае ввод-вывод производится не синхронно с работой процессора. При вводе, например, данные помещаются в буфер средствами канала ввода-вывода; после занесения данных в буфер процессор получает возможность доступа к этим данным.

При вводе с *простой буферизацией* канал помещает данные в буфер, процессор обрабатывает эти данные, канал помещает следующие данные и т. д. В то время, когда канал заносит данные, обработка этих данных производиться не может, а во время обработки данных нельзя заносить дополнительные данные. Метод *двойной буферизации* позволяет совмещать выполнение операции ввода-вывода с обработкой данных; когда канал заносит данные в один буфер, процессор может обрабатывать данные другого буфера. А когда процессор заканчивает обработку данных одного буфера, канал будет заносить новые данные опять в первый буфер. Такое поочередное использование буферов иногда называют *буферизацией с переключением («триггерной» буферизацией)*. Обмен данными между каналами и процессорами будет рассмотрен ниже.

2.2.5 Периферийные устройства

Периферийные (внешние запоминающие) *устройства* позволяют /хранить громадные объемы информации вне основной памяти компьютера. Лентопротяжные устройства являются в принципе последовательными устройствами, которые предусматривают чтение и запись данных на длинной магнитной ленте. Ленты могут иметь длину до 1100 м при намотке на 30-см кассеты. Информация может фиксироваться на ленте с различными *плотностями записи*. Для первых лентопротяжных устройств плотность записи составляла восемь символов на миллиметр длины ленты, затем стандартной стала плотность записи 22 симв/мм, затем 32 симв/мм, затем 64 симв/мм, а в настоящее время — 250 симв/мм. Устройства новых поколений безусловно будут иметь еще большие плотности записи.

Самым важным устройством внешней памяти с точки зрения операционных систем является, по-видимому, накопитель на магнитных дисках. Накопители на магнитных дисках — это устройства прямого доступа, они позволяют обращаться к отдельным элементам данных, не требуя последовательного просмотра всех элементов данных, хранящихся на диске. Первые дисковые накопители могли хранить по несколько миллионов символов. Емкость памяти каждого из современных накопителей может достигать миллиарда символов. В ближайшем будущем должны быть созданы накопители еще большей емкости. Работа дисковых накопителей и их роль для операционных систем описаны в гл. 12 и 13.

2.2.6 Защита памяти

Защита памяти — важное условие для нормальной работы многоабонентских вычислительных систем (систем коллективного пользования). Защита памяти ограничивает диапазон адресов, к которым разрешается обращаться программе. Защиту памяти для программы, занимающей непрерывный блок ячеек памяти, можно реализовать при помощи так называемых *граничных регистров*, где указываются старший и младший адреса этого блока памяти. При выполнении программы все адреса обращения к памяти контролируются, чтобы убедиться в том, что они находятся в промежутке между адресами, указанными в граничных регистрах. Защиту памяти можно реализовать также при помощи *ключей защиты*

памяти, относящихся к определенным областям основной памяти; программе разрешается обращение к ячейкам памяти только тех областей, ключи которых совпадают с ключом данной программы.

2.2.7 Таймеры и часы

Интервальный таймер — эффективный способ предотвращения /монополизации процессора одним из пользователей в многоабонентских системах. По истечении заданного интервала времени таймер генерирует сигнал прерывания для привлечения внимания процессора; по этому сигналу процессор может переключиться на обслуживание другого пользователя.

Часы истинного времени дают возможность компьютеру следить за *реальным календарным временем* с точностью до миллионных долей секунды, а при необходимости даже точнее.

2.2.8 Работа в режиме он-лайн и автономный режим (оф-лайн), периферийные процессоры

Некоторые периферийные устройства обладают возможностью работать либо в *режиме он-лайн*, когда они непосредственно связаны с центральным процессором, либо в *автономном режиме (оф-лайн)*, когда ими управляют контроллеры, не связанные с центральной вычислительной машиной. *Автономные контроллеры* привлекательны тем, что они позволяют управлять периферийными устройствами, не загружая при этом непосредственно процессор. При помощи автономных устройств часто выполняются такие операции, как ввод данных с перфокарт на магнитную ленту, вывод данных с ленты на карты и с ленты на печать.

В 1959 г. фирма IBM анонсировала свою «малую» вычислительную машину 1401 (которая затем стала наиболее популярной вычислительной машиной своего времени). Модель 1401 была функционально законченной вычислительной системой, предусматривающей работу с обширным набором традиционных внешних устройств. Интересно отметить, что одним из наиболее известных применений модели 1401 явилось автономное выполнение операций ввода-вывода для более крупных машин. Компьютер, работающий в таком режиме, называется *процессором ввода-вывода*, или *автономным компьютером-сателлитом*.

2.2.9 Каналы ввода-вывода

В первых компьютерах с ростом требуемых объемов вычислений, особенно в условиях обработки экономических данных, узким местом, как правило, оказывался ввод-вывод. Во время выполнения операций ввода-вывода процессоры были заняты управлением устройствами ввода-вывода. В некоторых машинах в каждый конкретный момент времени могла выполняться всего лишь одна операция ввода-вывода. Важным шагом для решения этой проблемы явилась разработка *каналов ввода-вывода*. "Канал ввода-вывода — это специализированный процессор, предназначенный для управления вводом-выводом независимо от основного процессора вычислительной машины. Канал имеет возможность прямого доступа к основной памяти для записи или выборки информации.

В первых машинах взаимодействие между процессорами и каналами осуществлялось при помощи процессорных команд типа

- условный переход (если канал занят выполнением операции);
- ожидание (пока не закончится выполнение команды канала);
- запись (содержимого управляющих регистров канала в основную память для последующего опроса процессором).

В современных машинах с управлением по прерываниям процессор выполняет команду «начать ввод-вывод» (SIO), чтобы инициировать передачу данных ввода-вывода по каналу; после окончания операции ввода-вывода канал выдает *сигнал прерывания* по завершению операции ввода-вывода, уведомляющий процессор об этом событии.

Истинное значение каналов состоит в том, что они позволяют значительно увеличить параллелизм работы аппаратуры компьютера и освобождают процессор от подавляющей части нагрузки, связанной с управлением вводом-выводом.

Для высокоскоростного обмена данными между внешними устройствами и основной памятью используется *селекторный канал*. Селекторные каналы имеют только по одному *подканалу* и могут обслуживать в каждый момент времени только одно устройство.

Мультиплексные каналы имеют много подканалов; они могут работать сразу с многими потоками данных в режиме чередования. *Байт-мультиплексный канал* обеспечивает режим чередования байтов при одновременном обслуживании ряда таких медленных внешних устройств, как терминалы, перфокарточные устройства ввода-вывода, принтеры, а также низкоскоростные линии передачи данных. *Блок-мультиплексный канал* при обменах в режиме чередования блоков может обслуживать несколько таких высокоскоростных устройств, как лазерные принтеры и дисковые накопители.

2.2.10 Захват цикла

Узкое место, где может возникнуть конфликтная ситуация между каналами и процессором,— это доступ к основной памяти. Поскольку в каждый конкретный момент времени может осуществляться только одна операция обращения (к некоторому модулю основной памяти) и поскольку каналам и процессору может одновременно потребоваться обращение к основной памяти, в обычном случае приоритет здесь предоставляется каналам. Это называется захватом цикла памяти; канал буквально захватывает, или «крадет» циклы обращения к памяти у процессора. Каналам требуется лишь небольшой процент общего числа циклов памяти, а предоставление им приоритета в этом смысле позволяет обеспечить лучшее использование устройств ввода-вывода. Подобный подход принят и в современных операционных системах; планировщики, входящие в состав операционной системы, как правило, отдают приоритет программам с большим объемом ввода-вывода по отношению к программам с большим объемом вычислений.

2.2.11 Относительная адресация

Когда потребность в увеличении объемов основной памяти стала очевидной, архитектуры компьютеров были модифицированы для работы с очень большим диапазоном адресов. Машина, рассчитанная на работу с памятью емкостью 16 Мбайт (1 Мбайт — это 1 048 576 байт), должна иметь 24-разрядные адреса. Включение столь длинных адресов в формат каждой команды даже для машины с одноадресными командами стоило бы очень дорого, не говоря уже о машинах с многоадресными командами. Поэтому для обеспечения работы с очень большими адресными пространствами в машинах начали применять *адресацию типа база + смещение*, или *относительную адресацию*, при которой все адреса программы суммируются с содержимым *базового регистра*. Подобное решение имеет то дополнительное преимущество, что программы становятся *перемещаемыми*, или *позиционно-независимыми*; это свойство программ имеет особенно важное значение для многоабонентских систем, в которых одну и ту же программу может оказаться необходимым размещать в различных местах основной памяти при каждой загрузке.

2.2.12 Режим задачи, режим супервизора, привилегированные команды

В вычислительных машинах, как правило, предусматривается несколько различных *режимов работы*. Динамический выбор режима позволяет лучше организовать защиту программ и данных. В обычном случае, когда машина находится в конкретном режиме, работающая программа может выполнять только некоторое подмножество команд. В частности, если говорить о программах пользователя, то подмножество команд, которые пользователь может употреблять в режиме задачи, не позволяет, например, непосредственно производить операции ввода-вывода; если программе пользователя разрешить выполнение любых операций ввода-вывода, она могла бы вывести главный список паролей системы, распечатать информацию любого другого пользователя или вообще испортить операционную систему. Операционной системе обычно присваивается статус *самого полномочного пользователя* и работает она в *режиме супервизора*; она имеет доступ ко всем командам, предусмотренным в машине. Для большинства современных вычислительных машин подобного разделения на два режима — задачи и супервизора — вполне достаточно. Однако в случае машин с высокими требованиями по защите от несанкционированного доступа желательно иметь более двух режимов. Благодаря этому можно обеспечить *более высокую степень детализации защиты*. Благодаря этому можно также предоставлять доступ по *принципу минимума привилегий*: каждому конкретному пользователю следует предоставлять минимально возможный приоритет и право доступа только к тем ресурсам, которые ему действительно необходимы для выполнения предусмотренных задач.

Интересно отметить, что в процессе развития компьютерных архитектур выявилась тенденция к увеличению количества *привилегированных команд*, т. е. команд, которые не могут выполняться в режиме задачи. Это служит свидетельством определенной тенденции к реализации большего числа функций операционных систем аппаратными средствами. Некоторые микропроцессоры уже имеют целые операционные системы, реализованные микропрограммно; во многих случаях значительная часть функций операционной системы реализуется в аппаратуре.

2.2.13 Виртуальная память

Системы виртуальной памяти дают возможность указывать в программах адреса, которым не обязательно соответствуют физические адреса основной памяти. Виртуальные адреса, выдаваемые работающими программами, при помощи аппаратных средств динамически (т. е. во время выполнения программы) преобразуются в адреса команд и данных, хранящихся в основной памяти. Системы виртуальной памяти позволяют программам работать с адресными пространствами гораздо большего размера, чем адресное пространство основной памяти. Они дают пользователям возможность создавать программы, независимые (большой частью) от ограничений основной памяти, и позволяют обеспечить работу многоабонентских систем с общими ресурсами.

В системах виртуальной памяти применяются такие методы, как *страничная организация* (предусматривающая обмен между основной и внешней памятью блоками данных фиксированного размера) и *сегментация* (которая предусматривает разделение программ и данных на логические компоненты, называемые сегментами, что упрощает управление доступом и коллективное использование). Эти методы иногда реализуются порознь, а иногда в комбинации. Системы виртуальной памяти рассматриваются в гл. 8 и 9.

2.2.14 Мультипроцессорная обработка

В *мультипроцессорных* машинах несколько процессоров работают с общей

основной памятью и одной операционной системой. При мультипроцессорной работе возникает опасность конфликтных ситуаций определенных типов, которых не бывает в однопроцессорных машинах. Здесь необходимо обеспечить координированный *упорядоченный* доступ к каждой общей ячейке памяти, с тем чтобы два процессора не могли изменять ее содержимое одновременно — и в результате, быть может, портить его. Подобная координация необходима также и в случае, когда один процессор пытается изменить содержимое ячейки, которую хочет прочитать другой процессор. Более подробно эти проблемы обсуждаются в гл. И. В гл.21 при описании операционной системы MVS, выбранной в качестве одного из примеров для иллюстрации, объясняется, каким образом реализуется мультипроцессорный режим в крупных машинах фирмы IBM. Упорядочение доступа необходимо также и для однопроцессорных машин. Подробно эта проблема рассматривается в гл. 3, 4 и 5.

2.2.15 Прямой доступ к памяти

Одним из способов достижения высокой производительности вычислительных машин является минимизация количества прерываний, происходящих в процессе выполнения программы. Разработанный для этого способ *прямого доступа к памяти* (ПДП) требует лишь одного прерывания на каждый блок символов, передаваемых во время операции ввода-вывода. Благодаря этому обмен данными производится значительно быстрее, чем в случае, когда процессор прерывается при передаче каждого символа.

После начала операции ввода-вывода символы передаются в основную память по принципу захвата цикла — канал захватывает шину связи процессора с основной памятью на короткое время передачи одного символа, после чего процессор продолжает работу.

Когда внешнее устройство оказывается готовым к передаче очередного символа блока, оно «прерывает» процессор. Однако в случае ПДП состояние процессора запоминать не требуется, поскольку передача одного символа означает для процессора скорее задержку, или приостановку, чем обычное прерывание. Символ передается в основную память под управлением специальных аппаратных средств, а после завершения передачи процессор возобновляет работу.

ПДП—это способ повышения производительности, особенно необходимый для систем, в которых выполняются очень большие объемы операций ввода-вывода. Аппаратные средства, обеспечивающие захват циклов памяти и управление устройствами ввода-вывода в режиме ПДП, называются *каналом прямого доступа к памяти*.

2.2.16 Конвейеризация

Конвейеризация — это аппаратный способ, применяемый в высокопроизводительных вычислительных машинах с целью использования определенных типов параллелизма для повышения эффективности обработки команд. Упрощенно структуру конвейерного процессора можно представить очень похожей на технологическую линию производственного предприятия; на конвейере процессора на различных стадиях выполнения одновременно могут находиться

несколько команд. Такое совмещение требует несколько большего объема аппаратуры, однако позволяет существенно сократить общее время выполнения последовательности команд.

2.2.17 Иерархия памяти

Современные вычислительные машины содержат несколько видов памяти, в том числе основную (первичную, оперативную), внешнюю (вторичную, массовую) и кэш-память. В основной памяти должны размещаться команды и данные, к которым будет обращаться работающая программа. Внешняя память — это магнитные ленты, диски, карты и другие носители, предназначенные для хранения информации, которая со временем будет записана в основную память. Кэш-память — это буферная память очень высокого быстродействия, предназначенная для повышения скорости выполнения работающих программ; для программ пользователя эта память, как правило, «прозрачна». В вычислительных машинах, оснащенных кэш-памятью, текущая часть программы помещается в кэш-память, что позволяет выполнять ее гораздо быстрее, чем если бы она находилась в основной памяти. Все эти виды памяти создают единую *иерархию памяти*; переход по уровням этой иерархии от кэш-памяти к основной и затем к внешней памяти сопровождается уменьшением стоимости и скорости и увеличением емкости памяти. Память, как правило, разделяется на байты (символы) или слова (состоящие из постоянного количества байтов). Каждая ячейка памяти имеет свой адрес; множество адресов, доступных программе, называется *адресным пространством*.

2.3 Программное обеспечение

Программное обеспечение — это программы, которые содержат команды и данные и определяют для аппаратных средств алгоритмы решения задач. Существует очень много различных языков программирования.

2.3.1 Программирование на машинном языке

Машинный язык — это язык программирования, непосредственно воспринимаемый компьютером. Каждая команда машинного языка интерпретируется аппаратурой, выполняющей указанные функции. Команды машинного языка в принципе являются довольно примитивными. Только соответствующее объединение этих команд в программы на машинном языке дает возможность описывать достаточно серьезные алгоритмы. Наборы команд машинного языка современных компьютеров зачастую включают некоторые очень эффективные возможности (см, например, описание миникомпьютеров VAX в гл. 19).

Говорят, что машинный язык является *машинно-зависимым*: программа, написанная на машинном языке компьютера одного типа, как правило, не может выполняться на компьютере другого типа, если его машинный язык не идентичен машинному языку первого компьютера (или не является расширением по отношению к этому языку). Еще одним признаком машинной, или аппаратной, зависимости является характер самих команд: в командах машинного языка указываются наименования конкретных регистров компьютера и предусматривается обработка данных в той физической форме, в которой они существуют в этом компьютере. Большинство первых компьютеров программировались непосредственно на машинном языке, а в настоящее время на машинном языке пишется лишь очень небольшое число программ.

2.3.2 Ассемблеры и макропроцессоры

Программирование на машинном языке требует очень много времени и чревато ошибками. Поэтому были разработаны *языки ассемблерного типа*, позволяющие повысить скорость процесса программирования и уменьшить количество ошибок кодирования. Вместо чисел, используемых при написании программ на машинных языках, в языках ассемблерного типа применяются содержательные

мнемонические сокращения и слова естественного языка. Однако компьютеры не могут непосредственно воспринять программу на языке ассемблера, поэтому ее необходимо вначале перевести на машинный язык. Такой перевод осуществляется при помощи программы-транслятора, называемой *ассемблером*.

Языки ассемблерного типа также являются машинно-зависимыми. Их команды прямо и однозначно соответствуют командам программы на машинном языке. Чтобы ускорить процесс кодирования программы на языке ассемблера, были разработаны и включены в ассемблеры так называемые *макропроцессоры*. Программист пишет *макрокоманду* как указание необходимости выполнить действие, описываемое несколькими командами на языке ассемблера. Когда макропроцессор во время трансляции программы читает макрокоманду, он производит *макрорасширение* — т. е. генерирует ряд команд языка ассемблера, соответствующих данной макрокоманде. Таким образом, процесс программирования значительно ускоряется, поскольку программисту приходится писать меньшее число команд для определения того же самого алгоритма.

2.3.3 Компиляторы

Тенденция к повышению вычислительной мощности команд привела к разработке некоторых очень эффективных макропроцессоров и макроязыков, упрощающих программирование на языке ассемблера. Однако развитие ассемблеров путем введения макропроцессоров не решает проблемы машинной зависимости. В связи с этим были разработаны *языки высокого уровня*.

Языки высокого уровня открывают возможность *машинно-независимого* программирования. Большинству пользователей компьютер нужен только как средство реализации прикладных систем. Языки высокого уровня позволяют пользователям заниматься преимущественно задачами, специфичными для их конкретных прикладных областей, не вникая в особенности применяемых ими машин. Благодаря этому существенно повышается скорость процесса программирования, обеспечивается обмен программами между машинами различных типов, и у людей появляется возможность разрабатывать эффективные прикладные системы, не затрачивая времени и сил на полное изучение внутренней структуры машины.

Перевод с языков высокого уровня на машинный язык осуществляется при помощи программ, называемых *компиляторами*. Компиляторы и ассемблеры имеют общее название «*трансляторы*». Написанная пользователем программа, которая в процессе трансляции поступает на вход транслятора, называется *исходной программой*; программа на машинном языке, генерируемая транслятором, называется *объектной программой*, или *выходной (целевой) программой*.

2.3.4 Система управления вводом-выводом (IOCS)

Подробные канальные программы, необходимые для управления вводом-выводом, и различные подпрограммы для координации работы каналов и процессоров являются достаточно сложными. Создание супервизорной программы, учитывающей все сложности выполнения операций ввода-вывода, освобождает прикладного программиста от необходимости самому писать подобные программы. Такая супервизорная программа носит название *системы управления вводом-выводом (IOCS)*.

В 50-х годах пользователи обычно включали исходный код IOCS в свои программы на языке ассемблера. Пакет IOCS, уже написанный и отлаженный, фактически ассемблировался каждый раз заново как часть каждой индивидуальной программы, что приводило к значительному увеличению времени трансляции. Поэтому на многих машинах, как правило, использовались готовые,

заранее ассемблированные программы IOCS. Программист, работающий на языке ассемблера, писал операторы со ссылками на места расположения ключевых программ в готовом коде IOCS.

Еще одна проблема использования концепции IOCS была связана с тем, что полный пакет программ IOCS зачастую занимал значительную долю основной памяти, оставляя мало места для кода прикладных программ пользователя. Поэтому некоторые пользователи занимали в памяти место не нужных им модулей пакета IOCS своими программами. Другие пользователи разрабатывали свои собственные, более компактные пакеты. Однако в конце концов пользователи поняли, что самое рациональное — это предоставить системе IOCS возможность выполнять все функции по управлению вводом-выводом, и были просто вынуждены увеличивать объемы (дорогостоящей) основной памяти своих вычислительных машин. Такой подход стал по сути стандартным — и операционные системы начали включать все больше и больше машинно-ориентированных программ, так что разработчики прикладных систем смогли сконцентрировать свои усилия на создании программ прикладного характера. Это привело к тому, что операционным системам потребовались большие емкости основной памяти. К счастью, стоимость устройств основной памяти постоянно снижается.

2.3.5 Спулинг

При *спулинге* (вводе-выводе с буферизацией) посредником между работающей программой и низкоскоростным устройством, осуществляющим ввод-вывод данных для этой программы, становится высокоскоростное устройство, например дисковый накопитель. Вместо вывода строк данных непосредственно, скажем, на построочно печатающее устройство программа записывает их на диск. Благодаря этому текущая программа может быстрее завершиться, с тем чтобы другие программы могли быстрее начать работать. А записанные на диск строки данных можно, распечатать позже, когда освободится принтер. Для этой процедуры название «спулинг» весьма подходит, поскольку она напоминает намотку нитки на катушку, с тем чтобы ее можно было при необходимости размотать.

2.3.6 Процедурно-ориентированные и проблемно-ориентированные языки

Языки высокого уровня бывают либо *процедурно-ориентированными*, либо *проблемно-ориентированными*. Процедурно-ориентированные языки высокого уровня — это универсальные языки программирования, которые можно использовать для решения самых разнообразных задач. Проблемно-ориентированные языки предназначаются специально для решения задач конкретных типов. Такие языки, как Паскаль, Кобол, Фортран, Бейсик и ПЛ/1 обычно считаются процедурно-ориентированными, а такие языки, как GPSS (язык моделирования) и SPSS (язык для выполнения статистических вычислений), — проблемно-ориентированными. ¹⁾

2.3.7 Быстрые компиляторы без оптимизации и оптимизирующие компиляторы

Когда программы разрабатываются и отлаживаются, компиляции производятся часто, а выполняются программы обычно недолго — пока не обнаружится очередная ошибка. В этих условиях вполне приемлемы *быстрые компиляторы без оптимизации*. Они позволяют быстро получить объектную программу, однако ее код может оказаться совершенно неэффективным с точки зрения как занимаемой памяти, так и скорости выполнения. После того как программа отлажена и готова для производственной эксплуатации, при помощи *оптимизирующего компилятора* для нее генерируется эффективный машинный код. Оптимизирующий компилятор работает гораздо медленнее, однако обеспечивает получение объектного кода

очень высокого качества.

В 70-х годах господствовало мнение о том, что хороший программист, работающий на языке ассемблера, может создать гораздо лучший код программы, чем оптимизирующий компилятор. В настоящее время оптимизирующие компиляторы настолько эффективны, что генерируемый ими код по качеству не уступает или даже превосходит код, созданный высококвалифицированным программистом на языке ассемблера. Благодаря этому программы, которые должны быть исключительно эффективными (например, операционные системы), больше не приходится писать на языке ассемблера. Сегодня крупные операционные системы в большинстве случаев пишутся на языках высокого уровня и при помощи оптимизирующих компиляторов очень высокого качества транслируются в эффективный машинный код.

2.3.8 Интерпретаторы

Существует один интересный и распространенный вид трансляторов, *интерпретаторы*, которые не генерируют объектную программу, а фактически обеспечивают непосредственное выполнение исходной программы. Интерпретаторы особенно распространены в системах проектирования программ, где программы идут, как правило, лишь небольшое время до момента обнаружения очередной ошибки. Интерпретаторы распространены также в сфере персональных компьютеров. Они свободны от накладных затрат, свойственных ассемблированию или компиляции. Однако выполнение программы в режиме интерпретации идет медленно по сравнению с скомпилированным кодом, поскольку интерпретаторам надо транслировать каждую команду при каждом ее выполнении.

2.3.9 Абсолютные и перемещающие загрузчики

Программы для выполнения должны размещаться в основной памяти. Распределение команд и элементов данных по конкретным ячейкам основной памяти является исключительно важной задачей. Решение этой задачи иногда осуществляет сам пользователь, иногда транслятор, иногда системная программа, называемая *загрузчиком*, а иногда — операционная система. Сопоставление команд и элементов данных с конкретными ячейками памяти называется *привязкой к памяти*. При программировании на машинном языке привязка к памяти производится в момент кодирования. Уже давно наблюдается тенденция откладывать привязку программы к памяти на как можно более поздний срок, и в современных системах виртуальной памяти привязка осуществляется динамически в процессе выполнения программы. Отсрочка привязки программы к памяти обеспечивает увеличение гибкости как для пользователя, так и для системы, однако это связано с существенным повышением сложности трансляторов, загрузчиков, аппаратных средств и операционных систем.

Загрузчик — это системная программа, которая размещает команды и данные программы в ячейках основной памяти. *Абсолютный загрузчик* размещает эти элементы именно в те ячейки, адреса которых указаны в программе на машинном языке. *Перемещающий загрузчик* может загружать программу в различные места основной памяти в зависимости, например, от наличия свободного участка основной памяти в момент загрузки (называемое *временем загрузки*).

2.3.10 Связывающие загрузчики и редакторы связей

В первых вычислительных машинах программист писал на машинном языке программу, которая содержала все команды, необходимые для решения конкретной задачи. Даже сложные и в определенном смысле опасные процедуры управления вводом-выводом в каждой программе на машинном языке приходилось кодировать вручную.

В настоящее время программы пользователя зачастую содержат лишь незначительную часть команд и данных, необходимых для решения поставленной задачи. В составе системного программного обеспечения поставляются большие *библиотеки подпрограмм*, так что программист, которому необходимо выполнять определенные стандартные операции, может воспользоваться для этого готовыми подпрограммами. В частности, операции ввода-вывода обычно выполняются подпрограммами, находящимися вне программы пользователя. Поэтому программу на машинном языке, полученную в результате трансляции, приходится, как правило, комбинировать с другими программами на машинном языке, чтобы сформировать необходимый выполняемый модуль. Эту процедуру *объединения программ* выполняют *связывающие загрузчики* и *редакторы связей* до начала выполнения программы.

Связывающий загрузчик во время загрузки объединяет необходимые программы и загружает их непосредственно в основную память. Редактор связей также осуществляет подобное объединение программ, однако он создает *загрузочный модуль*, который записывается во внешнюю память для будущего использования. Редактор связей играет особенно важную роль для производственных систем; когда программу необходимо выполнять, ее загрузочный модуль, сформированный при помощи редактора связей, может быть загружен немедленно — без накладных затрат времени (часто весьма больших) на повторное объединение отдельных частей программы.

2.4 Микропрограммы

Принято считать, что автором концепции микропрограммирования является профессор Морис Уилкс. В своей статье в 1951 г. (Wi 51) он предложил принципы, которые легли в основу современных методов микропрограммирования. Однако начало реального внедрения микропрограммирования связано с появлением System/360 в середине 60-х годов. В течение 60-х годов изготовители компьютеров применяли микропрограммирование для реализации наборов команд машинного языка (Ни 70).

В конце 60-х — начале 70-х годов появилось *динамическое микропрограммирование*, предусматривающее возможность легкой загрузки новых микропрограммных модулей в управляющую память, служащую для выполнения микропрограмм. Благодаря этому стало возможным динамично и часто менять наборы команд вычислительной машины. И сейчас уже никого не удивит, если в мультипрограммных системах новых поколений будут предусматриваться возможности предоставления различных наборов команд различным пользователям, с тем чтобы в процессе переключения процессора с программы на программу можно было также осуществлять переход с одного набора машинных команд на другой, необходимый следующему пользователю.

Микропрограммирование вводит дополнительный уровень средств программирования, ниже лежащий по отношению к машинному языку компьютера, и тем самым оно позволяет определять конкретные команды машинного языка. Подобные возможности являются неотъемлемой частью архитектуры современных компьютеров и имеют громадное значение с точки зрения обеспечения высоких скоростных характеристик и защиты операционных систем.

Микропрограммы размещаются в специальной управляющей памяти очень высокого быстродействия. Они состоят из индивидуальных микрокоманд, которые гораздо более элементарны по своей природе и более рассредоточены по функциям, чем обычные команды машинного языка. В компьютерах, где набор команд машинного языка реализуется при помощи микропрограммирования, каждой команде машинного языка соответствует целая и, быть может, большая микропрограмма. Тем самым сразу же становится очевидным, что

микропрограммирование окажется эффективным только в том случае, если управляющая память будет обладать гораздо большим быстродействием; чем основная.

2.4.1 Горизонтальный и вертикальный микрокод

Команды микрокода можно разделить на *горизонтальные* и *вертикальные*. Выполнение вертикальных микрокоманд очень похоже на выполнение обычных команд машинного языка. Типичная вертикальная микрокоманда задает пересылку одного или нескольких элементов данных между регистрами.

Горизонтальный микрокод действует совсем по-другому. Каждая его команда содержит гораздо большее число бит, поскольку может задавать параллельную операцию пересылки данных для многих или даже всех регистров данных устройства управления. Горизонтальные микрокоманды являются более мощными, чем вертикальные, однако может оказаться, что соответствующие микропрограммы гораздо сложнее кодировать и отлаживать.

2.4.2 Выбор функций для микропрограммной реализации

Для разработчика очень важно правильно решить, какие именно функции вычислительной машины реализовать при помощи микрокода. Микрокод предоставляет реальную возможность повысить быстродействие вычислительной машины. Реализуя часто используемые последовательности команд микропрограммно, а не обычным программным способом, разработчики добиваются существенного повышения показателей быстродействия. Читателям, которые будут знакомиться с функциями операционных систем по мере проработки настоящей книги, рекомендуется тщательно анализировать, для реализации каких из этих функций может быть эффективно использован микрокод.

2.4.3 Эмуляция

Эмуляция — метод, позволяющий сделать одну вычислительную машину функциональным эквивалентом другой. Набор команд машинного языка эмулируемого компьютера микропрограммируется на *эмулирующем компьютере* — и благодаря этому программы, представленные на машинном языке первого компьютера, могут непосредственно выполняться на втором. Фирмы-разработчики компьютеров широко применяют эмуляцию при внедрении новых машин, и пользователи, привязанные к старым компьютерам, получают, например, возможность без всяких изменений выполнять свои ранее отлаженные программы на новых машинах. Тем самым процесс перехода с машины на машину становится менее сложным и болезненным.

2.4.4 Микродиагностика

Микропрограммы значительно теснее связаны с аппаратурой, чем программы, написанные на машинном языке. Благодаря этому появляется возможность в гораздо более широких масштабах осуществлять контроль и исправление ошибок, причем выполнять эти операции с большей степенью детализации. В некоторые машины вводятся средства микродиагностики, «переплетающиеся» с командами программ на машинном языке. Это позволяет избежать многих потенциальных проблем и повысить надежность работы машины.

2.4.5 Специализированные компьютеры

Поскольку разработка, производство и организация сбыта вычислительной машины обходится дорого, фирмы-изготовители сосредотачивают свои усилия обычно на выпуске машин общего назначения. Громадные капитальные вложения,

без которых невозможно создание новой машины, требуют больших объемов сбыта, поскольку только в этом случае можно будет возместить затраты и получить необходимые прибыли. Поэтому промышленные фирмы обычно стараются не изготавливать специализированных, единственных в своем роде машин; этим занимаются университеты, в которых подобные машины создаются главным образом для выполнения научных исследований.

В связи с этим пользователям компьютеров приходится решать проблему специализации своих машин применительно к собственным конкретным требованиям; такая специализация традиционно осуществляется при помощи соответствующего программного обеспечения. Аппаратура машины представляет собой некую универсальную среду для выполнения системных программ, при помощи которых машина приспособляется к конкретным требованиям пользователей.

В некоторых машинах подобную специализацию пользователи могут осуществлять при помощи микрокода. При этом они могут либо применить микрокод, поставляемый фирмой-изготовителем, либо написать собственный микрокод; сейчас широко распространены оба этих подхода.

2.4.6 Микропрограммная поддержка

Фирмы-изготовители зачастую по отдельному заказу поставляют факультативные микрокодовые средства, обеспечивающие повышение производительности вычислительных машин. Фирме IBM удалось довольно успешно сделать это для своих компьютеров семейства System/370 в рамках операционной системы VM (см. гл. 22). Как будет показано при описании этой операционной системы в гл. 22, она реализует концепцию многих виртуальных машин благодаря эффективному использованию механизма прерываний. Для этого ряд наиболее часто используемых программ обработки прерываний реализуется микрокодом; такая *микропрограммная поддержка* позволяет достигнуть существенного повышения скоростных характеристик.

2.4.7 Микропрограммирование и операционные системы

К числу наиболее часто выполняемых последовательностей команд в большинстве вычислительных машин относятся определенные части операционной системы. Например, в системе с диалоговой обработкой транзакций имеется механизм диспетчирования, обеспечивающий выбор очередной единицы работы, которую должен будет выполнять процессор; поскольку подобный механизм диспетчирования может действовать сотни раз в секунду, он должен работать очень эффективно, и одним из способов сделать его более высокоскоростным и эффективным является именно микрокодовая реализация.

К числу функций, чаще всего реализуемых при помощи микрокода, относятся следующие:

- обработка прерываний;
- управление различными типами структур данных;
- примитивы синхронизации, координирующие доступ к общим данным и другим ресурсам;
- операции обработки частей слова, позволяющие эффективно выполнять манипуляции с битами;
- переключения контекста, т. е. быстрые переключения процессора с программы на программу в многоабонентской системе;
- последовательности вызова процедур и возврата.

Реализация функций операционных систем при помощи микрокода позволяет повысить эффективность и скоростные характеристики, снизить затраты на разработку программ и обеспечить более надежную защиту систем (см. гл. 17).

Читателям, желающим более подробно познакомиться с применением микропрограммирования в операционных системах, рекомендуется обратиться к работам (Br77), (Bu81) и (So75).

2.4.8 Пример микропрограммирования

В настоящем разделе рассматривается небольшая гипотетическая микропрограммируемая вычислительная машина. Наша цель заключается в том, чтобы попытаться передать определенные нюансы микропрограммирования и, в частности, показать, каким образом оно может использоваться для реализации набора команд машинного языка компьютера. В основу настоящего раздела положен пример, который представили Роше и Адаме в своей отличной статье для изучающих микропрограммирование (Ra80).

Простой гипотетический небольшой компьютер ITSIAС имеет набор команд машинного языка, показанный на рис. 2.1.

Компьютер ITSIAС имеет накапливающий регистр-аккумулятор АКК, который участвует в выполнении всех арифметических операций. Каждая команда машинного языка содержит два поля

Команда	Описание
ADD (Сложить)	АКК (АКК + (А)
SUB (Вычесть)	АКК (АКК - (А)
LOAD (Загрузить)	АКК ((А)
STORE (Записать)	(А) (АКК
BRANCH (Переход)	Переход на А
COND BRANCH (Условный переход)	Если АКК=0, то переход на А

Рис. 2.1 Набор команд машинного языка для гипотетического компьютера ITSIAС.

по 8 бит — код операции (КОП) и адрес памяти А. В состав процессора входит арифметико-логическое устройство (АЛУ) для выполнения некоторых арифметических действий. Регистры компьютера ITSIAС и их функции показаны на рис. 2.2.

Регистр	Функция
АКК	Аккумулятор. Этот накапливающий регистр участвует в выполнении всех арифметических операций. При выполнении каждой арифметической операции один из операндов должен находиться в аккумуляторе, а другой — в

	основной памяти.
РАКОП	Регистр адреса команды основной памяти. Этот регистр указывает ячейку основной памяти, где находится следующая команда машинного языка, подлежащая выполнению.
РАП	Регистр адреса памяти. Этот регистр участвует во всех обращениях к основной памяти. Он содержит адрес ячейки памяти, к которой производится обращение для чтения или записи.
РДП	Регистр данных памяти. Этот регистр также участвует во всех обращениях к основной памяти. Он содержит данные, которые записываются, или принимает данные, которые считываются из ячейки основной памяти, указанной в РАП.
РР	Рабочий регистр. Этот регистр используется для выделения поля адреса (8 бит справа) машинной команды, хранящейся в регистре РДП, чтобы его можно было поместить в РАП (в машине прямая пересылка данных из РДП в РАП невозможна).
РАКУП	Регистр адреса команд управляющей памяти. Этот регистр указывает адрес следующей микрокоманды (в управляющей памяти), подлежащей выполнению.
РМК	Регистр микрокоманды. Этот регистр содержит текущую выполняемую микрокоманду.

Рис. 2.2 Регистры компьютера ITSIAС.

Машина ITSIAС работает следующим образом. Прежде всего в управляющую память загружается микропрограмма. Команда машинного языка декодируется, и управление передается соответствующей подпрограмме микропрограммы для ее интерпретации. Каждая команда микропрограммы занимает одну ячейку управляющей памяти. Регистр адреса команд управляющей памяти РАКУП указывает на следующую выполняемую микрокоманду. Эта микрокоманда выбирается из управляющей памяти и помещается в регистр микрокоманд РМК.

Затем содержимое регистра РАКУП увеличивается на 1 (теперь он указывает на следующую выполняемую микрокоманду), и весь процесс повторяется. Микропрограмма принимает из регистра адреса команды основной памяти РАКОП адрес ячейки, где хранится следующая команда машинного языка, подлежащая интерпретации. После интерпретации очередной команды машинного языка микропрограмма меняет содержимое регистра РАКОП — теперь он указывает на ячейку основной памяти, где хранится следующая выполняемая команда машинного языка.

Декодированные микрокоманды непосредственно соответствуют тем элементарным операциям, которые могут выполняться аппаратными средствами;

они гораздо проще, чем команды машинного языка

Микрооперации, которые может выполнять машина, приведены на рис. 2.3.

Межрегистровые передачи (РЕГ — это АКК, РАКОП или РР):

$\text{РДП} \leftarrow \text{РЕГ}$

$\text{РЕГ} \leftarrow \text{РДП}$

$\text{РАП} \leftarrow \text{РДП}$

Операции с основной памятью:

READ (чтение ячейки основной памяти в РДП)

WRITE (запись РДП в ячейку основной памяти)

Операции управления последовательностью:

$\text{РАКУП} \leftarrow \text{РАКУП} + 1$ (обычный случай)

$\text{РАКУП} \leftarrow \text{декодированный РДП}$

$\text{РАКУП} \leftarrow \text{константа}$

SKIP (перескок, т. е. прибавление 2 к РАКУП, если АКК = 0; в противном случае прибавляется 1)

Операции с участием аккумулятора:

$\text{АКК} \leftarrow \text{АКК} + \text{РЕГ}$

$\text{АКК} \leftarrow \text{АКК} - \text{РЕГ}$

$\text{АКК} \leftarrow \text{РЕГ}$

$\text{РЕГ} \leftarrow \text{АКК}$

$\text{АКК} \leftarrow \text{РЕГ} + 1$

Рис. 2.3 Микрооперации машины ITSIAС.

Из перечисленных микроопераций составляются последовательности, которые реализуют команды машинного языка нашей простой машины. Микропрограмма, выполняющая команды машинного языка, показана на рис. 2.4. Выполнение микропрограммы начинается с ячейки 00, с подпрограммы, которая выбирает следующую выполняемую команду машинного языка.

Рассматриваемая машина с микропрограммным управлением работает следующим образом. Вначале регистр РАКУП устанавливается в нуль и тем самым указывает на микрокод подпрограммы, осуществляющей выборку команды машинного языка. Следующая команда машинного языка для выполнения выбирается из ячейки, адрес которой находится в РАКОП. Загрузка этой команды из основной

Выбор команды:

(00) $PAП \leftarrow PAКОП$

(01) READ

(02) $PAКУП \leftarrow \text{декодированный РДП}$
ADD:
(10) $AKK \leftarrow PAКОП - 1$

(11) $PAКОП \leftarrow AKK$

(12) $PP \leftarrow РДП$

(13) $PAП \leftarrow PP$

(14) READ

(15) $PP \leftarrow РДП$

(16) $AKK \leftarrow AKK + PP$

(17) $PAКУП \leftarrow 0$
SUB:
(20) $AKK \leftarrow PAКОП + 1$

(21) $PAКОП \leftarrow AKK$

(22) $PP \leftarrow РДП$

(23) $PAП \leftarrow PP$

(24) READ

(25) $PP \leftarrow РДП$

(26) $AKK \leftarrow AKK - PP$

(27) $PAКУП \leftarrow 0$
LOAD:
(30) $AKK \leftarrow PAКОП + 1$

(31) $PAКОП \leftarrow AKK$

(32) $PP \leftarrow РДП$

(33) $PAП \leftarrow PP$

(34) READ

(35) $PP \leftarrow РДП$

(36) $AKK \leftarrow PP$

(37) $PAКУП \leftarrow 0$
STORE:

(40) АКК \leftarrow РАКОП + 1

(41) РАКОП \leftarrow АКК

(42) РР \leftarrow РДП

(43) РАП \leftarrow РР

(44) РДП \leftarrow АКК

(45) WRITE

(46) РАКУП \leftarrow 0

 BRANCH:

(50) РАКОП \leftarrow РДП

(51) РАКУП \leftarrow 0

 COND BRANCH:

(60) SKIP

(61) РАКУП \leftarrow 0

(62) РАКОП \leftarrow РДП

(63) РАКУП \leftarrow 0

Рис. 2.4 Микропрограмма, интерпретирующая команды машинного языка компьютера ITSIAС.

памяти в регистр РДП осуществляется по команде чтения READ. Команда

РАКУП (декодированный РДП

устанавливает в РАКУП адрес соответствующей микрокодированной подпрограммы управляющей памяти для интерпретации данной команды машинного языка; при этом просто анализируется код операции и осуществляется как бы табличный поиск с использованием кода операции в качестве ключа поиска. Следующий микрокомандный цикл вызывает передачу управления на микрокод подпрограммы.

Если, например, интерпретируемая машинная команда имеет вид

ADD 50

то нужно содержимое ячейки 50 основной памяти сложить с содержимым аккумулятора. Рассмотрим микрокод, который выполняет эту операцию.

(10) АКК \leftarrow РАКОП + 1

(11) РАКОП \leftarrow АКК

(12) РР \leftarrow РДП

(13) РАП \leftarrow РР

(14) READ

(15) PP ← РДП

(16) АКК ← АКК + PP

(17) РАКУП ← 0

Микрокоманды (10) и (11) обеспечивают установку в РАКОП адреса следующей по порядку ячейки основной памяти. Микрокоманды (12) и (13) выделяют адрес основной памяти команды, находящейся в РДП, и передают его в РАП. (Две микрокоманды с использованием PP для этого необходимы опять-таки потому, что машина не позволяет производить непосредственную передачу содержимого РДП в РАП.) После выполнения микрокоманды (13) в РАП оказывается адрес ячейки 50. Команда чтения READ (14) вызывает загрузку содержимого ячейки, указанной в РАП, в регистр РДП. По команде (15) эти данные заносятся в PP, а по команде (16) — суммируются с содержимым аккумулятора. Команда (17) устанавливает в РАКУП нулевой адрес подпрограммы выборки микрокоманд, так что следующий микрокомандный цикл начнет процесс выборки следующей команды машинного языка для выполнения.

Заключение

Аппаратура — это устройства вычислительной машины. Программное обеспечение составляют команды, интерпретируемые аппаратурой, а микропрограммы включают микрокодированные команды, размещаемые в высокоскоростной управляющей памяти.

Расслоение памяти обеспечивает возможность одновременного доступа к последовательным ячейкам основной памяти, поскольку ячейки с соседними адресами размещаются к различным модулям памяти. Механизм прерываний играет важную роль для режимов работы, при которых много операций могут выполняться асинхронно, но в определенных случаях требуют синхронизации. Буферизация с несколькими буферами позволяет эффективно совмещать операции ввода-вывода с вычислениями.

Спулинг (ввод-вывод с буферизацией) позволяет отделить работающую программу от низкоскоростных устройств ввода-вывода, таких, как принтеры и устройства ввода данных с перфокарт. При спулинге ввод-вывод данных для программы осуществляется при посредстве высокоскоростного внешнего запоминающего устройства, например накопителя на магнитных дисках, а фактическое чтение или распечатка данных производится в то время, когда устройства ввода перфокарт и принтеры свободны. Для изоляции пользователей друг от друга в многоабонентских системах необходимо предусматривать защиту памяти; защиту можно реализовывать несколькими различными способами, в том числе при помощи граничных регистров или ключей защиты.

Применение стандартного интерфейса ввода-вывода существенно упрощает подключение к машине новых внешних устройств. Внешние устройства могут работать под непосредственным управлением центрального процессора в режиме он-лайн, или автономно (оф-лайн), под управлением отдельных контроллеров, независимых от процессора. Функционально законченные вычислительные машины, которые выполняют операции ввода данных с перфокарт на магнитную

ленту, вывода данных с магнитной ленты на печать и т. д. для более крупных машин, называются процессорами ввода-вывода, или компьютерами-сателлитами.

Канал — специализированная вычислительная машина для выполнения операций ввода-вывода без участия центрального процессора. Для координации взаимодействия между центральным процессором и каналом применяется, как правило, способ регулярного опроса или механизм прерываний. Наиболее известные типы каналов — это селекторные, байт-мультиплексные и блок-мультиплексные каналы.

Система управления вводом-выводом (IOCS) — это пакет программ, назначение которого заключается в том, чтобы освободить пользователя от необходимости детального управления вводом-выводом. Пакеты IOCS являются важной частью современных операционных систем.

Метод относительной адресации дает возможность работать с очень большим адресным пространством без необходимости увеличивать размер машинного слова; во время выполнения программы все адреса формируются путем прибавления смещения к содержимому базового регистра. Благодаря этому упрощается также перемещение программ по памяти.

Наличие в машине нескольких режимов работы обеспечивает защиту программ и данных. В режиме супервизора могут выполняться любые команды (включая привилегированные), а в режиме задачи — только непривилегированные. Эти режимы определяют

границу между возможностями пользователя и возможностями операционной системы. В некоторых машинах предусматриваются более двух подобных режимов работы.

Системы виртуальной памяти в обычном случае дают возможность программам работать с гораздо более широким диапазоном адресов, чем адресное пространство имеющейся основной памяти. Это позволяет освободить программиста от ограничений, связанных с емкостью основной памяти.

Метод прямого доступа к памяти исключает необходимость прерывать работу центрального процессора при передаче каждого байта блока данных во время выполнения операций ввода-вывода — в этом случае требуется только один сигнал прерывания, вырабатываемый при завершении передачи всего блока. Каждый символ записывается в основную память и читается из нее с захватом цикла памяти — здесь каналу предоставляется приоритет, поскольку центральный процессор может и подождать.

В архитектурах компьютеров высокого быстродействия применяются конвейеры, которые позволяют совмещать работу нескольких команд, в каждый конкретный момент времени находящихся на различных стадиях выполнения.

В современных вычислительных машинах реализуется иерархическая структура памяти, включающая кэш-память, основную (первичную, оперативную) память и внешнюю (вторичную, массовую) память; при переходе с уровня на уровень иерархии в указанном порядке емкости памяти увеличиваются, а стоимость в расчете на байт уменьшается.

Программное обеспечение — это программы, состоящие из команд, которые интерпретируются аппаратными средствами; команды определяют алгоритмы, обеспечивающие решение задач. Программы для компьютера можно писать на машинном языке, языке ассемблера или языках высокого уровня. Программисты редко работают непосредственно на машинных языках — программы в машинных

кодах генерируются при помощи ассемблеров и компиляторов. Макропроцессоры дают возможность программистам, работающим на языке ассемблера, писать макрокоманды, которые порождают много команд на языке ассемблера. Компиляторы обеспечивают трансляцию программ, написанных на языках высокого уровня, на машинный язык. Интерпретаторы непосредственно выполняют исходные программы, не генерируя при этом объектные модули.

Процедурно-ориентированные языки являются универсальными, а проблемно-ориентированные языки специализируются для эффективного решения задач определенных классов. Компиляторы без оптимизации работают быстро, однако генерируют относительно неэффективные коды; оптимизирующие компиляторы позволяют получать эффективные коды, но работают гораздо медленнее, чем без оптимизации.

Абсолютные загрузчики осуществляют загрузку программ в конкретные ячейки, адреса которых указываются при компиляции; перемещающие загрузчики могут размещать программы в различных свободных участках памяти. Привязка программы к памяти по абсолютным адресам осуществляется во время трансляции, а привязка перемещаемых программ — во время загрузки или даже во время выполнения.

Связывающие загрузчики объединяют отдельные блоки программы, создавая единый модуль, готовый к выполнению; этот выполняемый модуль размещается в основной памяти. Редакторы связей также осуществляют объединение программ, однако сформированный ими готовый к выполнению модуль записывается во внешнюю память для последующего использования.

Микропрограммирование — это написание программ, которые управляют элементарными операциями аппаратуры; микропрограммирование играет исключительно важную роль в современных архитектурах компьютеров и операционных системах. Динамическое микропрограммирование предусматривает возможность простой загрузки новых микропрограмм в управляющую память для непосредственного выполнения.

Команды вертикального микрокода очень похожи на команды машинного языка; типичная вертикальная микрокоманда указывает, что необходимо выполнить определенную операцию с определенными данными. Горизонтальный микрокод содержит команды с более широкими возможностями — они позволяют задавать одновременное выполнение многих операций над многими элементами данных.

Микропрограммирование зачастую применяется при эмуляции, позволяя сделать один компьютер функциональным эквивалентом другого компьютера. Эмуляция особенно необходима и полезна в тех случаях, когда пользователям приходится переводить свои программы с машины на машину.

Микропрограммирование позволяет реализовать микродиагностику, т. е. контроль ошибок с гораздо большей степенью детализации, чем это возможно для команд машинного языка.

Благодаря микропрограммированию можно специализировать компьютер применительно к требованиям конкретных пользователей. Поставщики компьютеров обычно предлагают факультативные средства микропрограммной поддержки, позволяющие повысить скоростные характеристики; реализация часто выполняемых последовательностей команд при помощи микрокода сокращает время их выполнения. Многие функции операционных систем в современных вычислительных машинах реализуются не обычными программами, а микропрограммами, и благодаря этому, как правило, обеспечивается более высокая скорость выполнения и более надежная защита.

Терминология

абсолютный загрузчик (absolute loader)

автономный режим (оф-лайн) (off-line)

адресное пространство (диапазон адресов) (address space)

аппаратура, аппаратные средства (hardware)

ассемблер (assembler)

базовый регистр, регистр базы (base register)

байт-мультиплексный канал (byte-multiplex or channel)

библиотека подпрограмм (subroutine library)

блок-мультиплексный канал (block-multiplexor channel)

быстрый компилятор без оптимизации (quick-and-dirty compiler)

буфер (buffer)

буферизация с переключением («триггерная» буферизация) (flip-flop buffering)

вертикальный микрокод (vertical microcode)

виртуальная память (virtual storage)

внешняя (вторичная, массовая)

память (secondary storage)

время загрузки (load time)

входной, фронтальный процессор (front end processor)

выходная, целевая программа (target program)

горизонтальный микрокод (horizontal microcode)

готовая, заранее асSEMBЛИРОВАННАЯ IOCS (preassembled IOCS)

граничные регистры (bounds registers)

двойная буферизация (double buffering)

двухадресные команды (two-address instructions)

загрузочный модуль, модуль загрузки (load module)

загрузчик (loader)

захват цикла (памяти) (cycle stealing)

защита памяти (storage protection)

иерархия памяти (storage hierarchy)

интервальный таймер (interval timer)

интерпретатор (interpreter)

интерфейс ввода-вывода (I/O interface)

исходная программа (source program)

канал (channel)

канал прямого доступа к памяти (DMA channel)

ключи защиты памяти (storage protect keys)

компиляторы (compilers)

конвейеризация (pipelining)

контроллер устройств ввода-вывода (I/O device controller)

координированный (упорядоченный) доступ (sequentialization of access)

кэш-память (cache storage)

макрокоманда (macro instruction)

макропроцессор (macro processor)

макрорасширение (macro expansion)

машинно-зависимый (machine-dependent)

машинно-независимый (machine-independent)

машинный язык (machine language)

микродиагностика (microdiagnostics)

микрокод (microcode)

микрокоманда (microinstruction)

микропрограмма (microprogram)

микропрограммирование (microprogramming)

микропрограммная поддержка (microcode assists)

микропрограммное обеспечение (firmware)

мультиплексные каналы (multiplexor channels)

мультипроцессорная обработка (multiprocessing)

объединение программ (program combination)

объектная программа (object program)

одноадресные команды (single-address instructions)

опрос (регулярный, упорядоченный) (polling)

оптимизирующий компилятор (optimizing compiler)

основная (первичная, оперативная) память (primary storage)

относительная адресация, адресация «база + смещение» (base-plus-displacement addressing)

перемещаемая (позиционно-независимая) программа (location-independent program)

перемещающий загрузчик (relocating loader)

плотность записи (recording density)

подканал (subchannel)

последовательное внешнее запоминающее устройство (sequential device)

прерывание по завершению (операции) ввода-вывода (I/O completion interrupt)

прерывания (interrupts)

привилегированные команды (privileged instructions)

привязка (программы к памяти) (binding)

принцип минимума привилегий (при предоставлении доступа к ресурсам и информации) (principle of least privilege)

проблемно-ориентированный (problem-oriented)

программное обеспечение (software)

простая (одионочная) буферизация (single buffering)

процедурно-ориентированный (procedure-oriented)

процессор ввода-вывода (stand-alone processor)

процессор-спутник (satellite processor)

прямой доступ к памяти (ПДП) (Direct Memory Access, DMA)

расслоение памяти (интерливинг) (storage interleaving)

регистр перемещения (relocation register)

редактор связей (linkage editor)

режим задачи (пользователя) (problem state)

режим выполнения программ (execution states)

режим супервизора (supervisor state)

режим он-лайн (on-line)

связывающий загрузчик (linking loader)

сегментация (segmentation)

селекторный канал (selector channel)

спулинг (ввод-вывод с буферизацией) (spooling)

стандартный интерфейс ввода-вывода (standard I/O interface)

степень (глубина) детализации защиты (granularity of protection)

страничная организация памяти (paging)

система управления вводом-выводом (input-output control system IOCS)

транслятор (translator)

трехадресные команды (three-address instruction)

управляющая память (control storage)

устройство прямого (непосредственного) доступа (direct access device)

часы истинного (реального, календарного) времени (time-of-day clock)

эмулятор (emulator)

эмуляция (emulation)

язык ассемблера, язык ассемблерного типа (assembly language)

DMA

IOCS

Упражнения

2.1 Укажите различия между аппаратурой, программным обеспечением и микропрограммами.

2.2 Объясните, в чем заключается концепция расслоения памяти.

2.3 Что такое двойная буферизация? Подробно опишите, каким образом могла бы работать схема тройной буферизации. При каких условиях могла бы быть целесообразной тройная буферизация?

2.4 Что такое спулинг? Как должна работать система входного спулинга, предназначенная для чтения перфокарт с устройства ввода?

2.5 Объясните, что такое «прямой доступ к памяти» и «захват цикла».

2.6 Опишите несколько способов реализации защиты памяти.

2.7 Опишите два различных способа организации взаимодействия между центральным процессором и каналом.

2.8 Приведите несколько причин, обусловивших появление концепции относительной адресации.

2.9 Сопоставьте принцип минимума привилегий с такими концепциями, как режим задачи, режим супервизора и привилегированные команды.

2.10 Укажите различия между селекторными, байт-мультиплексными и блок-мультиплексными каналами.

2.11 Укажите различия между машинным языком, языком ассемблера и языками высокого уровня.

2.12 Что такое макропроцессор?

2.13 Укажите различия между проблемно-ориентированными и процедурно-ориентированными языками.

2.14 При каких обстоятельствах целесообразно использовать быстрый компилятор без оптимизации? Когда следует применять оптимизирующий компилятор?

2.15 Чем интерпретаторы отличаются от ассемблеров и компиляторов? Когда более целесообразно использовать интерпретатор, чем компилятор?

2.16 Укажите сходства и различия абсолютных загрузчиков и перемещающих загрузчиков.

2.17 Что такое «привязка программы к памяти»? Почему и у пользователя и у системы появляются более гибкие возможности, если привязку программ к памяти выполнять как можно позднее?

2.18 Чем редакторы связей отличаются от связывающих загрузчиков?

2.19 Что такое микропрограммирование? Почему термин «микропрограммное обеспечение» уместен для обозначения микрокода, резидентно размещающегося в управляющей памяти?

2.20 Укажите различия между горизонтальным и вертикальным микрокодом.

ЧАСТЬ 2

Управление процессами

Глава 3

Концепции процесса

Просто удивительно, что Природе удалось сравнительно спокойно пройти свой счастливый процесс развития среди такой дьявольской кутерьмы.

«Алый знак доблести» Стивен Крейн

Жить — значит действовать.



«Уолден, или жизнь в лесу» Генри Дейвид Торо

3.1 Введение

3.2 Определения термина «процесс»

3.3 Состояния процесса

3.4 Переход процесса из состояния в состояние

3.5 Блок управления процессом

3.6 Операции над процессами

3.7 Приостановка и возобновление

3.8 Обработка прерываний

3.8.1 Типы прерываний

3.8.2 Переключения контекста

3.9 Ядро операционной системы

3.9.1 Основные функции ядра

3.9.2 Разрешение и запрещение прерываний

3.9.3 Иерархическая структура системы

3.9.4 Реализация ядра при помощи микрокода

3.1 Введение

В этой главе мы вводим понятие *процесса*, наиболее важное для получения представления о работе современных многоабонентских вычислительных машин. Здесь приведено несколько наиболее распространенных определений процесса, однако следует отметить, что «идеального» определения в литературе пока еще нет.

Мы представим концепцию дискретных *состояний процесса*, а также рассмотрим, каким образом процессы переходят из состояния в состояние. Кроме того, будет описан ряд основных операций, которые могут выполняться над процессами.

Приведенные здесь определения и концепции послужат основой для обсуждения асинхронных параллельных процессов и планирования процессов в последующих главах.

3.2 Определения термина «процесс»

Термин «процесс» впервые начали применять разработчики системы MULTICS в 60-х годах. За прошедшее время термин «процесс», используемый в ряде случаев как синоним «задачи», получил много различных определений. Мы приведем здесь некоторые из них:

- программа в стадии выполнения;
- асинхронная работа;
- «живая душа» процедуры;
- «концентрация средств управления» для выполняемой процедуры;
- нечто, представленное в виде «блока управления процессом» в операционной системе;
- объект, которому выделяются процессоры;
- «диспетчируемый» модуль.

Встречаются и многие другие определения. Таким образом, общепринятого определения пока нет, однако чаще всего, по-видимому, под процессом понимается «программа во время выполнения».

3.3 Состояния процесса

В период своего существования процесс проходит через ряд дискретных состояний. Смену состояний процесса могут вызывать различные события.

Говорят, что процесс выполняется (т. е. находится в *состоянии выполнения*), если в данный момент ему выделен центральный процессор (ЦП). Говорят, что процесс *готов* (т. е. находится в *состоянии готовности*), если он мог бы сразу использовать ЦП, предоставленный в его распоряжение. Говорят, что процесс заблокирован (т. е. находится в *состоянии блокировки*), если он ожидает появления некоторого события (например, завершения операции ввода-вывода), чтобы получить возможность продолжать выполнение. Существуют и другие состояния процесса, однако для первоначального обсуждения можно ограничиться этими тремя.

Для простоты рассмотрим машину с одним центральным процессором, хотя все приведенные ниже рассуждения нетрудно распространить и на мультипроцессорную систему. В однопроцессорной машине в каждый конкретный момент времени может выполняться только один процесс, однако несколько процессов могут находиться в состоянии готовности, а несколько — в состоянии блокировки. Поэтому мы можем создать *список готовых к выполнению процессов* и *список заблокированных процессов*. Список готовых процессов упорядочен по приоритету, так что следующим процессом, получающим в свое распоряжение ЦП, будет первый процесс этого списка. Список заблокированных процессов не упорядочен — не предусматривается никакого приоритетного порядка разблокировки процессов (т. е. их перевода в состояние готовности), разблокировка процессов осуществляется в том порядке, в котором происходят ожидаемые ими события.

3.4 Переход процесса из состояния в состояние

Когда в систему поступает некоторое задание, она создает соответствующий процесс, который затем устанавливается в конец списка готовых процессов. Этот процесс постепенно продвигается к головной части списка — по мере завершения выполнения предыдущих процессов. Когда процесс оказывается первым в списке готовых и когда освобождается центральный процессор, этому процессу выделяется ЦП и говорят, что происходит *смена состояния процесса* — он переходит из состояния готовности в состояние выполнения (рис. 3.1). Предоставление центрального процессора первому процессу списка готовых процессов называется запуском, или

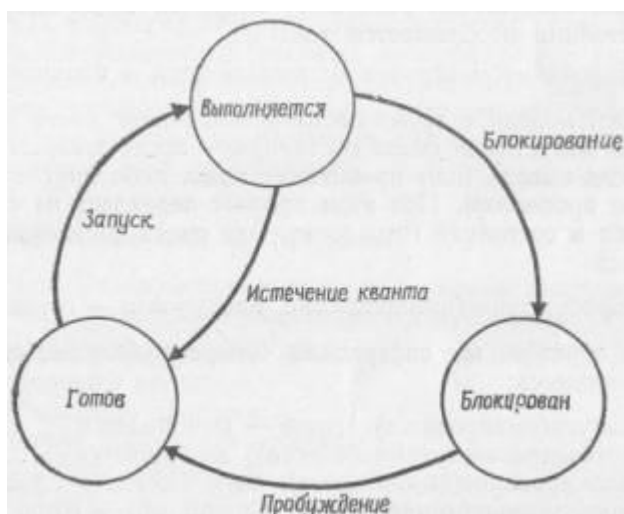


Рис. 3.1 Диаграмма переходов процесса из состояния в состояние.

выбором процесса для выполнения, и это делается при помощи системной программы, называемой диспетчером. Подобную смену состояния мы обозначаем следующим образом:

запуск (имя процесса): готов > выполняется

Про процесс, имеющий в своем распоряжении ЦП, говорят, что он выполняется. Чтобы предотвратить либо случайный, либо умышленный монопольный захват ресурсов машины каким-то одним процессом, операционная система устанавливает в специальном аппаратном таймере прерываний некоторое значение, определяющее временной интервал, или квант времени, в течение которого данному процессу пользователя разрешается занимать ЦП. Если процесс добровольно не освободит ЦП до истечения указанного временного интервала, таймер выработает сигнал прерывания, по которому управление будет передано операционной системе. После этого операционная система переведет ранее выполнявшийся процесс в состояние готовности, а первый процесс списка готовых — в состояние выполнения. Эти смены состояний обозначаются следующим образом:

истечение кванта (имя процесса): выполняется > готов

и

запуск (имя процесса): готов > выполняется

Если выполняющийся процесс до истечения отпущенного ему кванта времени инициирует операцию ввода-вывода, этот процесс тем самым добровольно освобождает ЦП (т. е. сам себя блокирует в ожидании завершения указанной операции ввода-вывода). Эта смена состояния изображается так:

блокирование (имя процесса): выполняется > блокирован

В нашей модели с тремя состояниями может иметь место еще лишь одна допустимая смена состояния — когда завершается операция ввода-вывода (или происходит какое-либо другое событие, ожидаемое процессом). При этом процесс переходит из состояния блокировки в состояние готовности. Эта смена состояния обозначается так:

пробуждение (имя процесса); блокирован > готов

Таким образом, мы определили четыре возможные смены состояния процесса:

запуск (имя процесса): готов > выполняется

истечение кванта (имя процесса): выполняется > готов

блокирование (имя процесса): выполняется > блокирован

пробуждение (имя процесса): блокирован > готов

Отметим, что единственная смена состояния, инициируемая самим процессом пользователя,— это блокирование, остальные три смены состояния инициируются объектами, внешними по отношению к данному процессу.

3.5 Блок управления процессом

Представителем процесса в операционной системе является *блок управления процессом* (PCB). Это структура данных, содержащая определенную важную информацию о процессе, в том числе:

- текущее состояние процесса;
- уникальный идентификатор процесса;
- приоритет процесса;
- указатели памяти процесса;
- указатели выделенных процессу ресурсов;
- область сохранения регистров.

Блок управления процессом является центральным пунктом, в котором операционная система может сосредоточить всю ключевую информацию о процессе. Когда операционная система переключает ЦП с процесса на процесс, она использует области сохранения регистров, предусмотренные в PCB, чтобы запомнить информацию, необходимую для рестарта (повторного запуска) каждого процесса, когда этот процесс в следующий раз получит в свое распоряжение ЦП.

Таким образом, PCB — это объект, который определяет процесс для операционной системы. Поскольку операционная система должна иметь возможность быстро выполнять операции с различными PCB, во многих вычислительных машинах предусматривается специальный аппаратный регистр, который всегда указывает на PCB текущего выполняемого процесса. Зачастую имеются также аппаратно-реализованные команды, которые обеспечивают быструю загрузку информации состояния в PCB и последующее быстрое восстановление этой информации.

3.6 Операции над процессами

Системы, управляющие процессами, должны иметь возможность выполнять определенные операции над процессами, в том числе:

- создание (образование) процесса;
- уничтожение процесса;
- возобновление процесса;
- изменение приоритета процесса;
- блокирование процесса;
- пробуждение процесса;

- запуск (выбор) процесса.

Создание процесса состоит из многих операций, включая:

- присвоение имени процессу;
- включение этого имени в список имен процессов, известных системе;
- определение начального приоритета процесса;
- формирование блока управления процессом PCB;
- выделение процессу начальных ресурсов.

Процесс может породить новый процесс. В этом случае первый, порождающий процесс называется *родительским процессом*, а второй, созданный процесс - *дочерним процессом*. Для создания дочернего процесса необходим. Только один родительский процесс. При таком подходе создается *иерархическая структура процессов*, подобная показанной на рис. 3.2, в которой у дочернего процесса есть только один родительский процесс, но у каждого родительского процесса может быть много дочерних процессов.

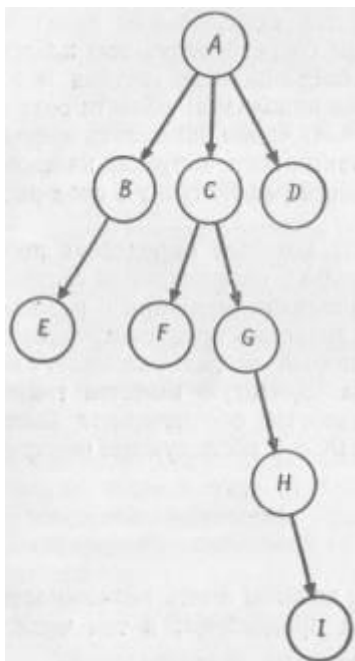


Рис. 3.2 Иерархия создания процессов.

Уничтожение процесса означает его удаление из системы. Ресурсы, выделенные этому процессу, возвращаются системе, имя процесса в любых системных списках или таблицах стирается, и блок управления процессом освобождается.

Приостановленный процесс не может продолжить свое выполнение до тех пор, пока его не активизирует какой-либо другой процесс. Приостановка — важная операция, которая реализуется во многих различных системах по-разному. Приостановки, как правило, длятся лишь небольшое время. Зачастую система прибегает к приостановкам для кратковременного исключения определенных

процессов в периоды пиковой нагрузки. В случае длительной приостановки процесса его ресурсы должны быть освобождены. Решение о необходимости освобождения конкретного ресурса в значительной степени зависит от природы этого ресурса. Основная память при приостановке процесса должна освобождаться немедленно. Закрепленное за процессом лентопротяжное устройство в случае кратковременной приостановки процесса может быть за ним сохранено, однако его необходимо освобождать, если процесс приостанавливается на длительный или неопределенный период времени.

Возобновление (или активация) процесса — операция подготовки процесса к повторному запуску с той точки, в которой он был приостановлен.

Уничтожение процесса усложняется, если это родительский процесс. В некоторых системах дочерний процесс уничтожается автоматически, когда уничтожается его родительский процесс, в других системах порожденные процессы начинают существовать независимо от своих родительских процессов, так что уничтожение родительского процесса не оказывает влияния на его потомков.

Изменение приоритета процесса, как правило, означает просто модификацию значения приоритета в блоке управления данным процессом.

3.7 Приостановка и возобновление

В предыдущем разделе были введены понятия о приостановке и возобновлении процессов. Эти операции играют важную роль по нескольким причинам.

- Если система работает ненадежно и есть признаки, что она может отказаться, то текущие процессы можно приостановить, чтобы вновь активизировать впоследствии после исправления ошибки.
- Пользователь, у которого отдельные промежуточные результаты работы процесса вызвали сомнение, может приостановить его (а не выбросить совсем), пока он не выяснит, правильно ли работает этот процесс.
- Некоторые процессы можно приостанавливать в моменты кратковременных пиков нагрузки системы, с тем чтобы возобновлять их выполнение после того, как нагрузка возвратится к обычному уровню.

На рис. 3.3 показана диаграмма состояний процесса, модифицированная с учетом операций приостановки и возобновления. В диаграмму введены два новых состояния, а именно «приостановлен готов» и «приостановлен блокирован», состояния «приостановлен выполняет» не бывает. На рисунке выше штриховой линии изображены *активные состояния*, а ниже — состояния приостановки.

Инициатором приостановки может быть либо сам процесс, либо другой процесс. В однопроцессорной машине выполняющийся процесс может приостановить только сам себя, ни один другой процесс не может работать одновременно с ним, чтобы выдать сигнал приостановки. В мультипроцессорной машине выполняющийся процесс может быть приостановлен и другим процессом, выполняющимся на другом процессоре.

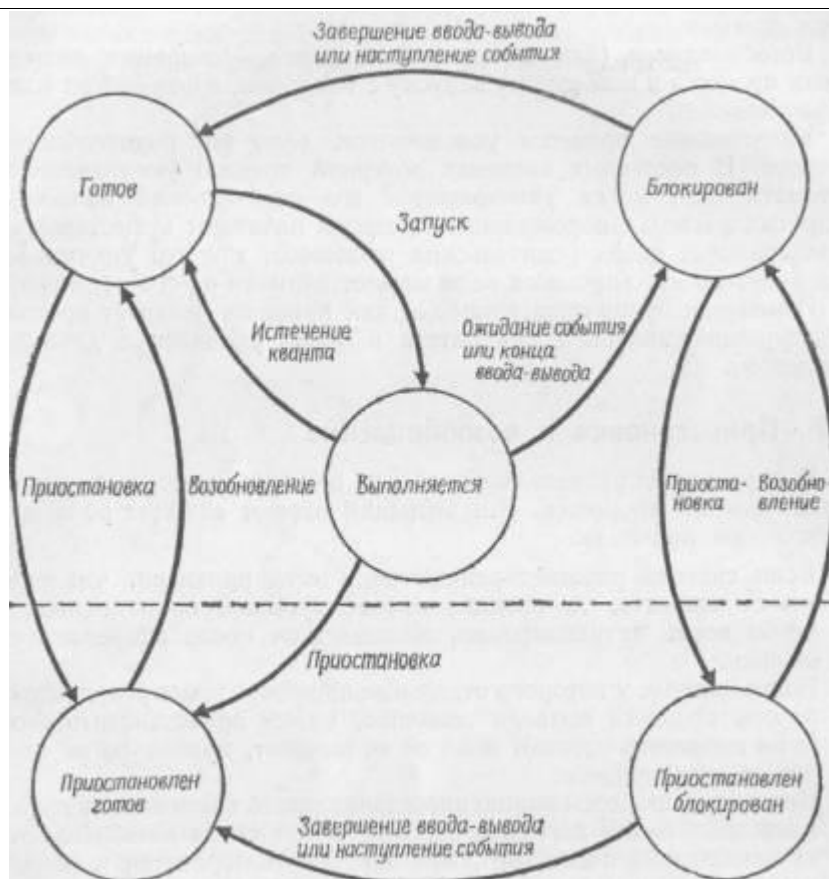


Рис. 3.3 Диаграмма состояний процесса с операциями приостановки и возобновления.

Процесс, находящийся в состоянии готовности, может быть приостановлен только другим процессом. При этом происходит следующая смена состояния:

приостановка (имя процесса): готов > приостановлен готов

Процесс, находящийся в состоянии «приостановлен готов», может быть переведен в состояние готовности другим процессом. Состояния меняются следующим образом:

возобновление (имя процесса): приостановлен готов > готов

Заблокированный процесс может быть переведен в состояние приостановки другим процессом. При этом состояния меняются:

приостановка (имя процесса): блокирован > приостановлен блокирован

Процесс, находящийся в состоянии «приостановлен блокирован», может быть активизирован другим процессом. При этом состояния меняются:

возобновление (имя процесса): приостановлен блокирован > блокирован

У читателя может возникнуть вопрос, не лучше ли вместо перевода заблокированного процесса в состояние приостановки подождать, пока не завершится выполнение операции ввода-вывода или не произойдет другое событие, необходимое для того, чтобы данный процесс стал готовым к выполнению. К сожалению, завершение операции ввода-вывода или ожидаемое событие может никогда не произойти или может задержаться на неопределенно долгий срок. Поэтому перед разработчиком возникает дилемма: либо выполнять приостановку заблокированного процесса, либо предусмотреть механизм, который позволял бы переводить процесс из состояния готовности в состояние приостановки, когда завершится операция ввода-вывода или наступит другое ожидаемое событие. Поскольку приостановка часто является операцией высокого приоритета, ее следует выполнять немедленно. Когда ожидаемое событие в конце концов происходит (если оно все-таки происходит), процесс, находящийся в состоянии «приостановлен блокирован», меняет свое состояние следующим образом:

наступление события (имя процесса): приостановлен блокирован > приостановлен готов

Использование операций приостановки и возобновления процессов операционной системой с целью регулирования нагрузки системы обсуждается в гл. 10.

3.8 Обработка прерываний

На рис. 3.4 показан пример «обработки прерывания» из реальной жизни, который безусловно знаком и понятен каждому. В вычислительной машине *прерывание* — это событие, при котором меняется нормальная последовательность команд, выполняемых процессором. Сигнал «прерывание» отрабатывается аппаратурой вычислительной машины. Если произошло прерывание, то

- управление передается операционной системе;
- операционная система запоминает состояние прерванного процесса. Во многих машинах эта информация запоминается в блоке управления процессом для прерванного процесса;
- операционная система анализирует тип прерывания и передает управление соответствующей программе обработки этого прерывания.



Рис. 3.4 Обработка «прерывания» в реальной жизни.

Инициатором прерывания, в частности, может оказаться выполняющийся процесс — или оно может быть вызвано некоторым событием, связанным или даже не связанным с этим процессом.

3.8.1 Типы прерываний

В этом разделе мы рассмотрим схему прерываний, применяемую в крупных компьютерах фирмы IBM. Эта информация будет особенно полезна для понимания описаний операционных систем MVS и VM фирмы IBM, приведенных в конце книги. Здесь предусматриваются шесть типов прерываний:

- *SVC-прерывания (по вызову супервизора).* Инициатором этих прерываний является работающий процесс, который выполняет команду SVC. Команда SVC — это генерируемый программой пользователя запрос на предоставление конкретной системной услуги, например на выполнение операции ввода-вывода, увеличение размера выделенной памяти или взаимодействие с оператором машины. Механизм SVC помогает защитить операционную систему от пользователей. Пользователю не разрешается самому произвольно входить в операционную систему — он должен запросить требуемую ему услугу при помощи команды SVC. Благодаря этому операционная система всегда знает обо всех попытках пользователя пересечь ее границы и может отказать от выполнения определенных запросов, если данный пользователь не имеет соответствующих полномочий.
- *Прерывания ввода-вывода.* Эти прерывания инициируются аппаратурой ввода-вывода. Они сигнализируют центральному процессору о том, что произошло изменение состояния канала или устройства ввода-вывода. Прерывания ввода-вывода происходят, например, когда завершается выполнение операции ввода-вывода, возникает ошибка или устройство переходит в состояние готовности.
- *Внешние прерывания.* Причинами этих прерываний могут быть различные события, в том числе истечение кванта времени, заданного на таймере прерываний, нажатие оператором клавиши прерывания на пульте управления или прием сигнала прерывания от другого процессора в мультипроцессорной системе.
- *Прерывания по рестарту.* Эти прерывания происходят, когда оператор

нажимает на пульте управления кнопку рестарта или когда от другого процессора в мультипроцессорной системе поступает команда рестарта *SIGP* (сигнал процессору).

- *Прерывания по контролю (ошибке) программы.* Эти прерывания вызываются различными видами ошибок, обнаруженных в выполняющемся процессе, таких, как попытка деления на ноль, попытка процесса пользователя выполнить привилегированную команду, попытка выполнить операцию с неправильным кодом и т. д.
- *Прерывания по контролю (ошибке) машины.* Эти прерывания вызываются аппаратными ошибками.

3.8.2 Переключения контекста

Для обработки каждого из этих различных типов прерываний в составе операционной системы предусмотрены программы, называемые *обработчиками прерываний* (*IH* — Interrupt Handler). Таким образом, в системе имеется шесть обработчиков прерываний: это обработчики прерываний *SVC*, прерываний ввода-вывода, внешних прерываний, прерываний по рестарту, прерываний по контролю программы и прерываний по контролю машины. Когда происходит прерывание, операционная система запоминает состояние прерванного процесса и передает управление соответствующему обработчику прерываний. Это делается способом, получившим название *переключение контекста*.

При реализации этого способа используются *слова состояния программы (PSW)*, которые управляют порядком выполнения команд и содержат различную информацию относительно состояния процесса. Существуют три типа слов состояния программы, а именно *текущее PSW*, *новое PSW* и *старое PSW*.

Адрес следующей команды, подлежащей выполнению, содержится в текущем *PSW*, в котором указываются также типы прерываний, *разрешенных* и *запрещенных* в данный момент. Центральный процессор реагирует только на разрешенные прерывания; обработка запрещенных прерываний либо задерживается, либо в некоторых случаях они игнорируются. Процессору нельзя запретить реагировать на прерывания по вызову супервизора, по рестарту или на некоторые виды программных прерываний. Причины, по которым разрешаются или запрещаются те или иные виды прерываний, будут объяснены ниже.

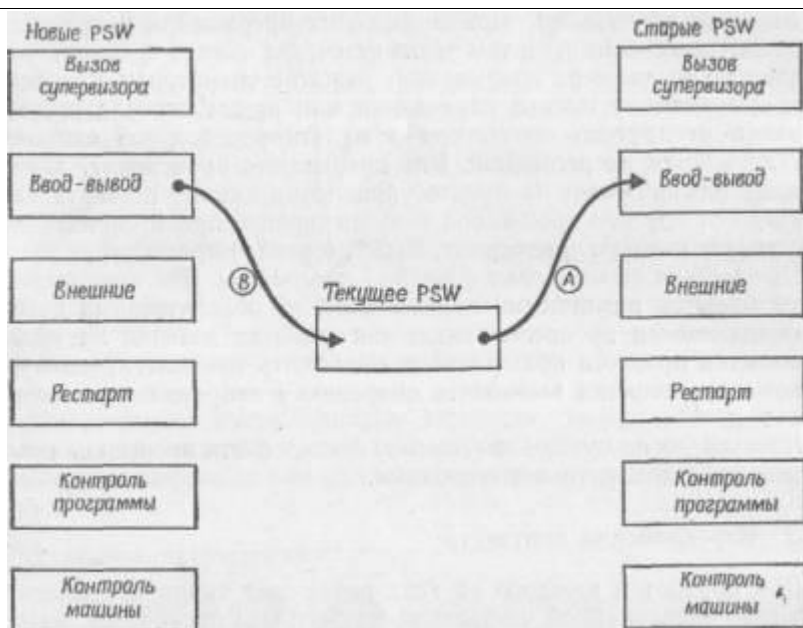


Рис. 3.5 Замещение PSW при обработке прерывания.

В однопроцессорной машине имеется только одно текущее PSW, но шесть новых PSW (по одному для каждого типа прерывания) и шесть старых PSW (опять-таки по одному для каждого типа прерывания). Новое PSW для данного типа прерывания содержит постоянный адрес, по которому резидентно размещается обработчик прерываний этого типа. Когда происходит прерывание (рис. 3.5) (если процессору не запрещено обрабатывать прерывания этого типа), производится автоматическое (выполняемое аппаратурой) переключение слов состояния программы следующим образом:

- текущее PSW становится старым PSW для прерывания этого типа;
- новое PSW для прерывания этого типа становится текущим PSW.

После такого замещения слов состояния текущее PSW содержит адрес соответствующего обработчика прерываний, который затем начинает обрабатывать данное прерывание.

Когда обработка прерывания завершается, центральный процессор начинает обслуживать либо тот процесс, который выполнялся в момент прерывания, либо готовый процесс с наивысшим приоритетом. Это зависит от того, допускает ли прерванный процесс перехват ЦП или нет. Если процесс не допускает перехвата у него ЦП, он снова получает в свое распоряжение ЦП. Если процесс допускает перехват ЦП, он получает ЦП только в том случае, если нет других процессов, готовых к выполнению.

Существует много различных интересных схем прерывания, отличных от описанной выше. Читателю, желающему познакомиться с ними более подробно, мы рекомендуем обратиться к книгам, где рассматриваются принципы организации и архитектуры компьютеров, например (Ba80), (Le80), (Ha78), (Hy78), (St75) и (Ce74).

3.9 Ядро операционной системы

Все операции, связанные с процессами, выполняются под управлением той части операционной системы, которая называется ее ядром (Lo81, Br70, Sc77, Sh75, Wu74). Ядро представляет собой лишь небольшую часть кода операционной системы в целом, однако оно относится к числу наиболее интенсивно используемых компонент системы. По этой причине ядро обычно резидентно размещается в основной памяти, в то время как другие части операционной системы перемещаются во внешнюю память и обратно по мере необходимости.

Одной из самых важных функций, реализованных в ядре, является обработка прерываний. В больших многоабонентских системах в процессор поступает постоянный поток прерываний. Быстрая реакция на эти прерывания играет весьма важную роль с точки зрения полноты использования ресурсов системы и обеспечения приемлемых значений времени ответа для пользователей, работающих в диалоговом режиме.

Когда ядро обрабатывает текущее прерывание, оно запрещает другие прерывания и разрешает их снова только после завершения обработки текущего прерывания. При постоянном потоке прерываний может сложиться такая ситуация, что ядро будет блокировать прерывания в течение значительной части времени, т. е. не будет иметь возможности эффективно реагировать на прерывания. Поэтому ядро обычно разрабатывается таким образом, чтобы оно осуществляло лишь минимально возможную предварительную обработку каждого прерывания, а затем передавало это прерывание на дальнейшую обработку соответствующему системному процессу, после начала работы которого ядро могло бы разрешить последующие прерывания. Подобный подход означает, что в конечном итоге прерывания могут быть разрешены в течение гораздо большей части общего времени, так что средняя скорость реакции системы существенно возрастает.

3.9.1 Основные функции ядра

Ядро операционной системы, как правило, содержит программы для реализации следующих функций:

- обработка прерываний;
- создание и уничтожение процессов;
- переключение процессов из состояния в состояние;
- диспетчирование (см. также гл. 10);
- приостановка и активизация процессов (см. также гл. 10);
- синхронизация процессов (см. гл. 4 и 5);
- организация взаимодействия между процессами (см. гл. 4 и 5);
- манипулирование блоками управления процессами;
- поддержка операций ввода-вывода (см. примеры операционных систем в гл. 18—22. В вычислительных машинах различных типов ввода-вывод обычно осуществляется по-разному);
- поддержка распределения и перераспределения памяти (см. гл. 7, 8 и 9);
- поддержка работы файловой системы (см. гл. 13);
- поддержка механизма вызова-возврата при обращении к процедурам;
- поддержка определенных функций по ведению учета работы машины (см. гл. 21).

Читателю, который хочет подробно познакомиться со структурой и работой

ядра, мы рекомендуем обратиться к гл. 6 книги Лорина и Дейтела (Lo81).

3.9.2 Разрешение и запрещение прерываний

Вход в ядро обычно осуществляется по прерыванию. Когда ядро реагирует на данное прерывание, оно запрещает другие прерывания. После определения причины данного прерывания ядро передает его обработку специальному системному процессу, предназначенному для работы с прерываниями этого типа.

В некоторых машинах вся обработка каждого прерывания производится большой, единой операционной системой. В подобных машинах прерывания запрещаются в течение большей части общего времени, однако операционные системы получаются в принципе более простыми. Подобный подход оправдан для небольших машин, которые предусматривают работу с ограниченным числом процессов. В крупных многоабонентских машинах выделение ядра операционной системы обеспечивает гораздо лучшие скоростные характеристики.

3.9.3 Иерархическая структура системы

Приведенный ниже список литературы включает статью (Di68), в которой описаны преимущества иерархического подхода к проектированию операционных систем. В основе иерархии находится сама аппаратура компьютера, называемая иногда «чистой машиной» или «голым железом». На следующем уровне иерархии (или в некоторых случаях на нескольких следующих уровнях) находятся различные функции ядра. В совокупности с этими функциями ядра компьютер становится *расширенной машиной*, т. е. машиной, которая предоставляет для операционной системы и своих пользователей не только свой машинный язык, но также и ряд дополнительных возможностей. Эти дополнительные функции, реализуемые при помощи ядра, часто называются *примитивами*.

Над ядром в иерархии находятся различные процессы операционной системы, которые обеспечивают поддержку процессов пользователя — например процессы управления внешними устройствами, фактически реализующие супервизорные функции для системных устройств при выполнении операций ввода-вывода по запросам различных пользователей. На вершине иерархии располагаются сами процессы пользователей.

Опыт показал, что подобные иерархические системы легче отлаживать, модифицировать и тестировать (Di68). В системах, где само ядро распределяется по нескольким уровням иерархии, необходимо тщательно продумывать, какие именно функции ядра на каком уровне размещать. В подобных иерархических системах, как правило, идут на следующее ограничение: допускают только обращения сверху вниз в иерархии, т. е. средства каждого уровня могут обращаться только к тем функциям, которые находятся на ближайшем нижележащем уровне.

3.9.4 Реализация ядра при помощи микрокода

В последнее время наметилась одна четкая тенденция: разработчики систем значительную часть функций ядра реализуют при помощи микрокода. Это является эффективным способом защиты ядра, причем тщательная отработка

микропрограмм позволяет обеспечить более высокую скорость выполнения ядром своих функций.

Заключение

Наиболее распространенное определение процесса — это «программа в стадии выполнения». Процесс находится в состоянии выполнения, если в его распоряжение выделен процессор. Процесс находится в состоянии готовности, если он мог бы выполняться при условии выделения ему процессора. Процесс находится в состоянии блокирования, если он ожидает наступления некоторого события, чтобы получить возможность продолжить свою работу.

Выполняющийся процесс переводится в состояние готовности, если отпущенный ему квант времени истекает до того, как он добровольно освободит процессор. Выполняющийся процесс переходит в состояние блокирования, если он инициирует операцию ввода-вывода и ждет ее завершения (или наступления некоторого другого события), чтобы продолжить свою работу. Заблокированный процесс переводится в состояние готовности, когда происходит ожидаемое им событие. Готовый процесс начинает выполняться, когда диспетчер выделяет в его распоряжение процессор.

Блок управления процессом (PCB) — это структура данных, в которой операционная система может сосредоточить всю ключевую информацию относительно процесса, включая его текущее состояние, идентификатор, приоритет, сведения о выделенной ему памяти и других ресурсах, значения регистров и т. д. Блок управления процессом определяет конкретный процесс для операционной системы.

Операционная система содержит механизмы, позволяющие выполнять над процессами различные операции, такие, как создание, уничтожение, приостановка, возобновление, изменение приоритета, блокирование, пробуждение и запуск (выбор для выполнения). Процесс может породить другой процесс, и в этом случае порождающий процесс называется родительским процессом, а порожденный процесс — дочерним процессом. Когда для создания дочернего процесса требуется только один родительский процесс, образуется иерархическая структура процессов.

Возможность приостанавливать и возобновлять процессы играет для операционной системы особенно важную роль, позволяя ей эффективно регулировать нагрузку машины.

Для вычислительных машин, предусматривающих параллельную и совмещенную работу различных устройств и процессов, важное значение имеют средства обработки прерываний. Когда происходит прерывание, операционная система немедленно реагирует на него. После завершения обработки прерывания запускается следующий процесс. Это может быть ранее прерванный процесс, если он не допускает перехвата у него ЦП другим процессом.

Схема обработки прерываний, принятая в крупных компьютерах фирмы IBM, предусматривает шесть типов прерываний: SVC, ввод-вывод, внешние, по рестарту, контролю программы и контролю машины. Для каждого типа прерываний в операционной системе имеется специальная программа, называемая обработчиком прерываний (IH) и обеспечивающая обработку прерываний соответствующего типа. Когда возникает прерывание, происходит замещение слова состояния программы PSW или переключение контекста; управление от выполняющегося процесса передается операционной системе. Схема прерываний,

реализованная в процессорах фирмы IBM, включает текущее PSW, где содержится адрес следующей выполняемой команды, набор старых PSW (по одному на каждый тип прерываний), где можно сохранять значения текущих PSW, когда происходит прерывание данного типа, и набор новых PSW (также по одному PSW на тип прерываний), где содержатся адреса соответствующих обработчиков прерываний.

Ядро — это небольшая часть операционной системы, которая используется наиболее интенсивно и код которой резидентно размещается в основной памяти. Как правило, ядро включает такие функции, как обработка прерываний, манипулирование процессами, манипулирование блоками управления процессами, диспетчирование, синхронизация процессов, обеспечение взаимодействия между процессами, поддержка выполнения операций ввода-вывода, распределение и перераспределение памяти, поддержка файловой системы, механизм вызова-возврата при обращении к процедурам, а также определенные учетные функции.

В иерархических системных структурах ядро занимает уровень, расположенный непосредственно над базовым уровнем аппаратуры самой вычислительной машины, а на лежащих выше уровнях размещаются процессы системы и процессы пользователя. Опыт показывает, что иерархические структуры проще отлаживать, модифицировать и тестировать.

Сейчас наблюдается явная тенденция к реализации основных частей ядра операционной системы при помощи микрокода, это обеспечивает более надежную защиту и более высокую скорость работы.

Терминология

активные состояния (active states)

блокирование (процесса) (block process)

блок управления процессом (process control block, PCB)

внешнее прерывание (external interrupt)

возобновление процесса (resume, activate a process)

выполняющийся процесс (running process)

«голое железо» («naked iron»)

диспетчер (dispatcher)

диспетчеризация (dispatching)

дочерний процесс (child process)

завершившийся процесс (terminated process)

запрещенные прерывания (desabled interrupts)

замещение PSW (PSW swapping)

задача (task)

запуск процесса (выбор процесса для выполнения) (dispatch)

иерархическая структура процессов (hierarchical process structure)

иерархическая структура системы (hierarchical system structure)

команда вызова супервизора SVC (supervisor call instruction)

новое PSW (new PSW)

обработчик прерываний (interrupt handler, IH)

очередь готовых (к выполнению) процессов (ready queue)

перевод процесса в состояние готовности (ready a process)

переключение контекста (context switching)

переключение PSW (PSW switching)

порождение процесса (spawn a process)

прерывание (interrupt)

прерывание ввода-вывода (I/O interrupt)

прерывание по контролю машины (machine check interrupt)

прерывание по контролю программы (program check interrupt)

прерывание по вызову супервизора (supervisor call (SVC) interrupt)

прерывание по рестарту (перезапуску) (restart interrupt)

примитивы (primitives)

приостановка процесса (suspend a process)

приостановленный процесс (suspended process)

пробуждение процесса (wakeup a process)

процесс (process)

разрешенные прерывания (enabled interrupts)

расширенная машина (extended machine)

родительский процесс (parent process)

слово состояния программы (program status word, PSW)

смены состояний процесса (process state transitions)

создание (образование) процесса (create a process)

состояние «выполняется» (running state)

состояние «готов» (ready state)

состояние «приостановлен готов» (suspended ready state)

состояние «приостановлен блокирован» (suspended block state)

состояния «приостановлен» (suspended states)

состояния процесса (process states)

список заблокированных (процессов) (blocked list)

список готовых к выполнению (процессов) (ready list)

старое PSW (old PSW)

таймер прерываний (interrupting clock)

текущее PSW (current PSW)

уничтожение процесса (destroy a process)

чистая (сырая) машина (raw machine)

ядро (nucleus, core, kernel)

Упражнения

3.1 Приведите несколько определений процесса. Как вы думаете, почему пока не существует единого общепринятого определения?

3.2 Дайте определение для каждого из следующих терминов: программа, процедура, процессор, процесс, пользователь, задача, задание.

3.3 Иногда термины «пользователь», «задание» и «процесс» применяются как

синонимы. Дайте определение для каждого из этих терминов.,, В каких случаях эти термины действительно имеют аналогичные смысловые значения?

3.4 Почему обычно нецелесообразно устанавливать приоритетный порядок для списка заблокированных процессов? При каких обстоятельствах, однако, это могло бы оказаться полезным?

3.5 Способность одного процесса порождать новые процессы имеет весьма важное значение, однако в этом кроются определенные опасности. Рассмотрим, к каким последствиям может привести то, что пользователю будет позволено выполнить

следующий процесс:

гангстер: process;

do всегда;

породить новый процесс точную мою копию;

end;

а) Если предположить, что система разрешила выполнение подобного процесса, к каким последствиям это могло бы привести?

б) Предположим, что перед вами как перед разработчиком операционной системы поставлена задача предусмотреть встроенные средства защиты против подобных процессов. Мы знаем (из «проблемы останова», рассматриваемой в *теории вычислимости*), что в общем случае невозможно предсказать, какой путь выполнения выберет программа. Каким образом этот фундаментальный теоретический результат повлияет на вашу способность предотвратить выполнение процессов, подобных приведенному выше?

в) Предположим, вы решили, что нецелесообразно просто отвергать определенные процессы и что наилучший выход в этом случае — установить за ними некоторый контроль во время выполнения. Какие средства контроля во время выполнения могла бы использовать операционная система для обнаружения процессов подобных вышеприведенному? Не могут ли предложенные вами средства контроля помешать способности процесса порождать новые процессы? Каким образом реализация предложенных вами средств контроля может повлиять на построение системных механизмов для манипулирования процессами?

3.6 В системах индивидуального пользования, как правило, очевидно, когда выполняемая программа закикливается. А в системах коллективного пользования, где одновременно выполняются десятки или сотни процессов, далеко не просто определить, что какой-то индивидуальный процесс не продвигается в своем выполнении.

а) Может ли операционная система определить, что процесс находится в бесконечном цикле?

б) Какие рациональные средства защиты можно было бы встроить в операционную систему, чтобы предотвратить бесконечное выполнение закиклившихся процессов?

3.7 Выбор оптимальной величины кванта времени имеет важное значение для эффективной работы операционной системы. В дальнейшем в данной книге мы подробно рассмотрим проблему определения размера кванта времени. А сейчас попытаемся предварительно указать некоторые из факторов, связанных с этой проблемой.

Рассмотрим однопроцессорную систему с разделением времени, которая обслуживает в диалоговом режиме большое количество пользователей. Каждый раз, когда процессор выделяется в распоряжение некоторого процесса, в интервальном таймере устанавливается значение кванта времени, по истечении которого должно произойти прерывание. Предположим, что для всех процессов системы задается один и тот же квант времени.

а) Что будет, если выбрать очень большую величину кванта времени, например десятки минут?

б) Что будет, если задать очень небольшую величину кванта времени, например несколько тактов процессора?

в) Очевидно, что величину кванта времени необходимо выбирать в диапазоне между значениями, указанными в а) и б). Предположим, что вам предоставлена возможность менять величину кванта времени. Каким образом вы узнаете, когда вам удалось выбрать «оптимальную» величину? Какие факторы делают эту величину оптимальной с точки зрения пользователя? Какие факторы делают ее оптимальной с точки зрения системы?

3.8 Механизм блокирования/возобновления процессов предусматривает, что процесс может заблокировать себя в ожидании некоторого события. Другой процесс должен обнаружить, когда это событие произошло, и активизировать заблокированный процесс. Возможна ситуация, когда процесс заблокирует себя в ожидании события, которое никогда не произойдет.

а) Может ли операционная система обнаружить, что заблокированный процесс ожидает события, которое никогда не произойдет?

б) Какие рациональные средства защиты можно было бы встроить в операционную систему, чтобы предотвратить бесконечное ожидание процессами «несбыточного» события?

3.9 Операционная система А работает строго по принципу «один пользователь — один процесс», а система В — по принципу «один пользователь — много процессов». Обсудите организационные различия между операционными системами А и В с точки зрения поддержки процессов.

3.10 В конечном счете пользователи должны платить за все ресурсы. Системе во время работы приходится выполнять различные операции, которые трудно отнести на счет конкретного пользователя. Каким образом могла бы система распределить подобные системные накладные затраты по процессам пользователя?

3.11 Одна из причин прерывания выполняющегося процесса после истечения некоторого «разумного» интервала времени (кванта времени) заключается в том, чтобы позволить операционной системе взять на себя управление процессором и запустить следующий процесс. Предположим, что в машине нет таймера прерываний и что процесс может потерять возможность управления процессором только в одном случае, если он освободит его добровольно. Предположим также, что в операционной системе машины механизм диспетчирования отсутствует. Опишите, каким образом можно было бы организовать взаимодействие группы процессов пользователя, чтобы создать управляемый пользователями механизм диспетчирования. Какие потенциальные опасности присущи подобной схеме? В чем заключаются ее преимущества для пользователей по сравнению с механизмом диспетчирования, управляемым системой?

3.12 В некоторых системах порожденный процесс автоматически уничтожается, когда уничтожается его родительский процесс; в других системах порожденные процессы в дальнейшем существуют независимо от своих родительских процессов, так что уничтожение порождающего процесса не отражается на его потомках. Обсудите достоинства и недостатки каждого подхода.

3.13 Если прерывания запрещены, на большинстве типов устройств они остаются в состоянии ожидания — возможность их обработки появится только после того как снова будут разрешены прерывания. При этом последующие прерывания не

допускаются, а функционирование самих устройств временно прекращается. Однако в системах реального времени оборудование, которое вырабатывает сигналы прерываний, зачастую непосредственно не связано с вычислительной машиной. Это оборудование продолжает в любом случае вырабатывать сигналы прерываний, даже если прерывания на вычислительной машине запрещены. Такие прерывания часто теряются. Обсудите последствия потери прерываний. Что более целесообразно для системы реального времени; терять некоторые прерывания или временно останавливать систему, пока не появится снова разрешение прерываний?

3.14 Приведите пример каждого из следующих типов прерываний!

- а) по SVC;
- б) по вводу-выводу;
- в) внешнее;
- г) по рестарту;
- д) по контролю программы;
- е) по контролю машины.

3.15 Что означает, что процесс не допускает перехвата ЦП? Может ли процессор обрабатывать прерывания в то время, когда выполняется процесс, не допускающий перехвата ЦП? Приведите пример процесса, не допускающего перехвата ЦП. Процесс какого типа предположительно допускает перехват ЦП?

3.16 Обсудите схему прерываний, отличную от описанной в настоящей главе. Сравните обе схемы.

3.17 Что такое ядро операционной системы? Почему ядро обычно размещается в основной памяти? Какие функции, как правило, выполняет ядро?

3.18 Почему ядро обычно работает при запрещенных прерываниях?

3.19 Что такое расширенная машина? Что такое примитивы?

3.20 Приведите несколько причин, обуславливающих целесообразность микропрограммной реализации основных частей ядра операционной системы.

Глава 4

Асинхронные параллельные процессы

Человек, имеющий одни часы, всегда может с уверенностью сказать, сколько времени, — и этим выгодно отличается от того, у кого двое часов.

Пословица

4.1 Введение

4.2 Параллельная обработка

4.3 Управляющая конструкция для указания параллелизма: Parbegin/Parend

4.4 Взаимоисключение

4.5 Критические участки

4.6 Примитивы взаимного исключения

4.7 Реализация примитивов взаимного исключения

▣ 4.8 Алгоритм Деккера

4.9 Взаимоисключение для N процессов

4.10 Аппаратная реализация взаимного исключения: команда testandset

4.11 Семафоры

4.12 Синхронизация процессов при помощи семафоров

4.13 Пара «производитель—потребитель»

4.14 Считающие семафоры

4.15 Реализация семафоров, операция P и V

4.1 Введение

Процессы называются *параллельными*, если они существуют одновременно. Параллельные процессы могут работать совершенно независимо друг от друга или они могут быть *асинхронными* — это значит, что им необходимо периодически синхронизироваться и взаимодействовать. Асинхронность — весьма сложная тема, и в настоящей главе и в гл. 5 обсуждается организация и управление системами, осуществляющими поддержку асинхронных параллельных процессов.

Здесь будут рассмотрены многие серьезные проблемы асинхронной работы.

Решение этих проблем иллюстрируется на примерах параллельных программ, написанных с применением нотации, аналогичной, хотя и не идентичной, нотации параллельного Паскаля, языка программирования, который разработал Бринк Хансен (Bg75, Bg77), Еще один популярный язык параллельного программирования, созданный на базе Паскаля,— это язык Модуля, разработанный Никлаусом Виртом (Wi77). В гл. 5 обсуждаются средства параллельного программирования, реализованные в новом языке Ада (Ad82), который безусловно станет основным языком для создания параллельных программ в 80-х годах.

4.2 Параллельная обработка

Поскольку габариты и стоимость аппаратуры компьютеров продолжают уменьшаться, следует ожидать дальнейшего развития мультипроцессорных систем и в конечном итоге реализации максимального параллелизма на всех уровнях. Если определенные операции логически можно выполнять параллельно, то компьютеры новых поколений будут физически выполнять их параллельно, даже если требуемая *степень параллелизма* будет при этом выражаться в тысячах или, быть может, даже миллионах параллельных процессов.

Параллельная обработка вызывает активный интерес по ряду причин. Люди, как правило, скорее способны концентрировать свое внимание на выполнении в каждый момент времени лишь одной операции, чем думать о параллельных вещах. (Чтобы убедиться в этом, читатель может попытаться читать сразу две книги — строку из одной, строку из другой, затем вторую строку из первой и т. д.).

Обычно весьма трудно определить, какие операции можно и какие нельзя выполнять параллельно. Отлаживать параллельные программы гораздо сложнее, чем последовательные; после того как выявленная ошибка предположительно исправлена, может оказаться, что восстановить последовательность событий, на которой эта ошибка проявилась впервые, не удастся, поэтому, вообще говоря, просто нельзя утверждать с уверенностью, что данная ошибка устранена.

Асинхронные процессы должны периодически взаимодействовать друг с другом, причем эти взаимодействия могут быть достаточно сложными. В настоящей главе и в нескольких последующих мы рассмотрим много примеров взаимодействий между процессами.

Наконец, доказывать корректность для параллельных программ гораздо труднее, чем для последовательных, а сейчас широко распространено мнение о том, что со временем должны появиться эффективные методы доказательства корректности программ, заменяющие исчерпывающее тестирование; только это позволит добиться заметных реальных успехов в разработке исключительно надежных программных систем.

4.3 Управляющая конструкция для указания параллелизма: **Parbegin/Parend**

В последнее время в литературе появляются сообщения о том, что во многих языках программирования предусматриваются специальные конструкции для указания параллелизма. Эти конструкции, как правило, представляют следующие парные операторы:

- Один оператор, указывающий, что последовательное течение программы должно быть разделено на несколько параллельно выполняемых последовательностей (цепочек управления).
- Один оператор, указывающий, что определенные параллельно выполняемые последовательности должны слиться воедино и должно возобновиться последовательное выполнение программы.

Эти операторы всегда встречаются парами и обычно носят такие названия, как *parbegin/parend* («начало/конец параллельного выполнения») или *cobegin/coend* («начало/конец совмещенного выполнения»). В настоящей книге мы используем пару операторов *parbegin/parend*, как рекомендует Дейкстра (Di65). В общем виде конструкция для указания параллелизма приведена на рис. 4.1.

```
parbegin
оператор1;
оператор2;
.
.
.
оператор n
parend
```

Рис. 4.1 Конструкция *parbegin/parend* для указания параллелизма.

Предположим, что в программе, где в данный момент выполняется единая последовательность команд, встречается первый оператор управляющей конструкции *parbegin*. Это приводит к тому, что единая цепочка управления разделяется на *n* отдельных самостоятельных цепочек — по одной для каждого внутреннего оператора данной конструкции. Это могут быть простые операторы, вызовы процедур, блоки последовательных операторов с ограничителями *begin/end* или какие-либо комбинации этих элементов. Каждая из отдельных цепочек управления со временем завершается и приходит к конечному оператору *parend*. Когда, наконец, завершается последняя из всех этих параллельных цепочек, они снова сливаются в единую цепочку управления и программа проходит точку *parend*.

В качестве примера рассмотрим следующее вычисление одного корня квадратного уравнения:

$$x := (-b + (b^2 - 4ac)^{.5}) / (2a)$$

Этот оператор присваивания можно вычислить при помощи последовательного процессора (имеющего команду возведения в степень) следующим образом:

1. b^2
2. $4a$

3. $(4*a)*c$
4. $(b**2)-(4*a*c)$
5. $(b**2-4*a*c)**.5$
6. $-b$
7. $(-b)+((b**2-4*a*c)**.5)$
8. $2*a$
9. $(-b+(b**2-4*a*c)**.5)/(2*a)$

Здесь каждая из девяти указанных операций выполняется в последовательности, определяемой принятыми в системе правилами предшествования операторов.

А в системе, предусматривающей параллельную обработку, данное выражение может быть вычислено следующим образом:

```

1 parbegin

temp1 := -b;

temp2 := b**2;

temp3 := 4*a;

temp4 := 2*a

parend;

2 temp5 := temp3*c;

3 temp5 := temp2-temp5;

4 temp5 := temp5**.5;

5 temp5 := temp1+temp5;

6 x := temp5/temp4

```

Здесь четыре операции, входящие в конструкцию parbegin/ parend, выполняются параллельно, а остальные пять операций по-прежнему приходится выполнять последовательно. Параллельное выполнение вычислений дает возможность значительно уменьшить реальное время решения задачи.

4.4 Взаимоисключение

Рассмотрим систему, обслуживающую в режиме разделения времени много терминалов. Предположим, что пользователи заканчивают каждую строку, которую они вводят в машину, сигналом возврата каретки, что необходимо вести постоянный учет общего количества строк, введенных пользователями с начала рабочего дня, и что контроль каждого терминала пользователя осуществляется при помощи отдельного процесса. Каждый раз, когда один из этих процессов принимает строку с соответствующего терминала пользователя, он прибавляет единицу к глобальной разделяемой переменной системы под названием

СТРОКВВЕДЕННЫХ. Рассмотрим, что произойдет, когда два процесса попытаются одновременно произвести прибавление 1 к этой переменной. Предположим, что у каждого процесса имеется собственная копия кода

LOAD СТРОКВВЕДЕННЫХ

ADD 1

STORE СТРОКВВЕДЕННЫХ

Пусть в данный момент переменная СТРОКВВЕДЕННЫХ имеет значение 21687. Предположим теперь, что первый процесс выполняет команды загрузки LOAD и сложения ADD, после чего в аккумуляторе оказывается значение 21688. Затем этот процесс (ввиду истечения выделенного ему кванта времени) уступает процессор второму процессу. Второй процесс выполняет теперь все три команды, устанавливая значение СТРОКВВЕДЕННЫХ равным 21688. Затем второй процесс возвращает управление процессором первому процессу, который продолжает свое выполнение с команды записи STORE и также помещает значение 21688 в поле СТРОКВВЕДЕННЫХ. Таким образом, из-за некоординированного доступа к разделяемой переменной СТРОКВВЕДЕННЫХ система в сущности теряет одну строку — правильная общая сумма сейчас должна была бы составить 21689.

Эту задачу можно решить, если каждому процессу предоставлять монопольное, исключительное право доступа к переменной СТРОКВВЕДЕННЫХ. Когда один процесс увеличивает эту разделяемую переменную, всем остальным процессам, которым нужно было бы также произвести приращение в то же самое время, придется ждать, а когда данный процесс закончит свое обращение к переменной, будет разрешено продолжить работу одному из процессов, находящихся в состоянии ожидания. Таким образом, каждый процесс, обращающийся к разделяемым данным, исключает для всех других процессов возможность одновременного с ним обращения к этим данным. Это называется *взаимоисключением*.

4.5 Критические участки

Взаимоисключение необходимо только в том случае, когда процессы обращаются к разделяемым, общим данным — если же они выполняют операции, которые не приводят к конфликтным ситуациям, они должны иметь возможность работать параллельно. Когда процесс производит обращение к разделяемым данным, то говорят, что он находится в своем *критическом участке* (или в своей критической области (Di65)). Очевидно, что для решения задачи, о которой говорилось в предыдущем разделе, необходимо, в случае если один процесс находится в своем критическом участке, исключить возможность вхождения для всех других процессов (по крайней мере для тех, которые обращаются к тем же самым разделяемым данным) в их критические участки.

Когда какой-либо процесс находится в своем критическом участке, другие процессы могут, конечно, продолжать выполнение, но без входа в их критические участки. Когда процесс выходит из своего критического участка, то одному из остальных процессов, ожидающих входа в свои критические участки, должно быть разрешено продолжить работу (если в этот момент действительно есть процесс в состоянии ожидания). Обеспечение взаимного исключения является одной из ключевых проблем параллельного программирования. Было предложено много способов решения этой проблемы — программные и аппаратные; частные, низкоуровневые и глобальные, высокоуровневые; одни из предложенных способов предусматривают свободное взаимодействие между процессами, а другие требуют

строгого соблюдения жестких протоколов.

Когда процесс находится в своем критическом участке, это его ко многому обязывает. В этом особом режиме процесс имеет исключительное право доступа к разделяемым данным, а всем остальным процессам, которым в то же самое время требуется доступ к этим данным, приходится ждать. Поэтому процессы должны как можно быстрее проходить свои критические участки и не должны в этот период блокироваться, так что критические участки необходимо кодировать очень тщательно (чтобы, например, не допустить возможности заикливания внутри критического участка).

Если процесс, находящийся в своем критическом участке, завершается, либо естественным, либо аварийным путем, то операционная система при выполнении служебных функций по завершению процессов должна отменить режим взаимного исключения, с тем чтобы другие процессы получили возможность входить в свои критические участки.

4.6 Прimitives взаимного исключения

Параллельная программа, представленная на рис. 4.2, правильно реализует механизм подсчета строк, описанный в разд. 4.4. Для простоты предположим, что в программах, приведенных в настоящем и нескольких следующих разделах, действуют только два параллельных процесса. Управлять n параллельными процессами значительно сложнее.

Представленные в программе рис. 4.2 конструкции *входвзаимного исключения* и *выходвзаимного исключения* в каждом процессе содержат участок программы, который обеспечивает доступ к разделяемой переменной СТРОКВВЕДЕННЫХ, т. е. эти конструкции обрамляют критические участки. Иногда подобные операторы называют примитивами взаимного исключения, другими словами, они вызывают выполнение самых фундаментальных операций, обеспечивающих реализацию взаимного исключения.

В случае двух процессов эти примитивы работают следующим образом. Когда «процессодин» выполняет примитив «входвзаимного исключения» (и если «процессдва» в это время находится вне своего критического участка) он входит в свой критический участок, обращается к разделяемой переменной, а затем выполняет «выходвзаимного исключения», показывая тем самым, что он вышел из критического участка.

Если «процессодин» выполняет «входвзаимного исключения», в то время как «процессдва» находится в своем критическом участке, «процессодин» переводится в состояние ожидания, пока «процессдва» не выполнит «выходвзаимного исключения». Только после этого «процессодин» сможет войти в свой критический участок.

program взаимное исключение;

var строквведенных: целое;

procedure процессодин;

while истина **do**

begin

взятиеследующейстрокистерминала;

вход взаимоиключения;

строковведенных := строковведенных + 1;

выходвзаимоисключения;

обработкастроки

end;

procedure процесс два;

while истина **do**

begin

взятиеследующейстрокистерминала;

входвзаимоисключения;

строковведенных := строковведенных + 1;

выходвзаимоисключения;

обработка строки

end;

begin

строковведенных := 0;

parbegin

процессодин;

процессдва

parend

end;

Рис. 4.2 Применение примитивов взаимоиключения.

Если «процессодин» и «процессдва» выполняют «входвзаимоисключения» одновременно, то одному из них будет разрешено продолжить работу, а другому придется ждать, причем мы будем предполагать, что «победитель» выбирается

случайным образом.

4.7 Реализация примитивов взаимного исключения

Мы хотим найти реализации примитивов «входвзаимного исключения» (код входа взаимного исключения) и «выходвзаимного исключения» (код выхода взаимного исключения) с соблюдением следующих четырех ограничений:

- Задача должна быть решена чисто программным способом на машине, не имеющей специальных команд взаимного исключения. Каждая команда машинного языка выполняется как *неделимая* операция, т. е. каждая начатая операция завершается без прерывания. Мы будем считать, что в случае одновременных попыток нескольких процессоров обратиться к одному и тому же элементу данных возможные конфликты разрешаются аппаратно при помощи схемы *защитной блокировки памяти*. Эта схема защитной блокировки распараллеливает конфликтующие обращения к памяти со стороны отдельных процессоров, выстраивая их в очередь, т. е. разрешая производить только одно обращение в каждый конкретный момент времени. Предположим, что эти отдельные обращения обслуживаются в случайном порядке.
- Не должно быть никаких предположений об относительных скоростях выполнения асинхронных параллельных процессов.
- Процессы, находящиеся вне своих критических участков, не могут препятствовать другим процессам входить в их собственные критические участки.
- Не должно быть бесконечного откладывания момента входа процессов в их критические участки.

Изящную программную реализацию механизма взаимного исключения впервые предложил голландский математик Деккер. В следующем разделе мы рассмотрим усовершенствованный вариант; *алгоритма Деккера*, разработанный Дейкстрой (Di65).

4.8 Алгоритм Деккера

На рис. 4.3 показана первая версия программного кода для реализации взаимного исключения в контексте параллельной программы с двумя процессами. Конструкция `parbegin/parend` позволяет организовать параллельную работу процессов «процессодин» и «процессдва». Каждый из этих процессов представляет собой бесконечный цикл с многократным входом в свой критический участок. В программе рис. 4.3 примитив «входвзаимного исключения» реализуется как один цикл `while`, который повторяется до тех пор, пока переменная «номерпроцесса» не станет равной номеру данного процесса, примитив «выходвзаимного исключения» реализуется как одна команда, которая устанавливает для переменной «номерпроцесса» значение, равное номеру другого процесса.

«Процессодин» выполняет свой цикл. Поскольку первоначально переменная «номерпроцесса» имеет значение 1, «процессодин» входит в свой критический участок. «Процессдва» обнаруживает, что «номерпроцесса» равен 1 и остается заблокированным на своем цикле `while do`. Если «процессдва» получает в свое распоряжение процессор, он просто выполняет цикл ожидания момента, когда для переменной «номерпроцесса» будет установлено значение 2, т. е. «процессдва» не может войти в свой критический участок и тем самым гарантируется

взаимоисключение.

В конце концов «процессодин» закончит работу в своем критическом участке (мы должны предполагать, что здесь нет бесконечных циклов) и установит «номерпроцесса» равным 2, так что «процессдва» получит возможность входа в свой критический участок.

program версияодин,

var номер процесса: целое;

procedure процессодин;

begin

while истина **do**

begin

while номерпроцесса = 2 **do**;

критическийучасткодин;

номерпроцесса := 2;

прочиеоператорыодин

end

end;

procedure процессдва;

begin

while истина **do**

begin

while номерпроцесса = 1 **do**;

критическийучастокдва;

номерпроцесса := 1;

прочиеоператорыдва

end

end;

begin

```
номерпроцесса := 1;
```

```
parbegin
```

```
    процессодин;
```

```
    процессдва
```

```
parend
```

```
end;
```

Рис. 4.3 Программная реализация примитивов взаимного исключения (версия 1),

Программа рис. 4.3 гарантирует взаимное исключение, однако весьма дорогой ценой. «Процессодин» должен выполняться первым, так что если «процессдва» готов к входу в свой критический участок, он может получить разрешение на это со значительной задержкой. После того как «процессодин» войдет в свой критический участок и затем выйдет из него, должен будет выполняться «процессдва», даже если «процессодин» хочет вновь войти в свой критический участок, а «процессдва» еще не готов. Таким образом, процессы должны входить и выходить из своих критических участков строго поочередно. Если одному процессу приходится это делать во много

раз чаще, чем другому, то он вынужден работать с гораздо меньшей скоростью, чем это необходимо. Подобная программа не может оказаться в состоянии полного тупика; если оба процесса одновременно пытаются войти в свои критические участки, то по крайней мере одному из них удастся продолжить работу. А если один из процессов завершится, то со временем другой окажется не в состоянии продолжать выполнение.

```
program версиядва;
```

```
var п1внутри, п2внутри: логический;
```

```
procedure процессодин;
```

```
begin
```

```
    while истина do
```

```
        begin
```

```
            while п2внутри do;
```

```
            п1внутри := истина;
```

```
            критическийучастоккодин;
```

```
            п1внутри := ложь;
```

прочиеоператорыодин

end

end;

procedure процессдва;

begin

while истина **do**

begin

while п1внутри **do**;

п2внутри := истина;

критическийучастокдва;

п2внутри := ложь;

прочиеоператорыдва

end

end;

begin

п1внутри := ложь;

п2внутри := ложь;

parbegin

процессодин;

процессдва

parend

end;

Рис. 4.4 Программная реализация примитивов взаимного исключения (версия 2).

В первой версии программной реализации взаимного исключения имеется только одна глобальная переменная и, таким образом» возникает проблема *жесткой синхронизации*. Поэтому во второй версии (рис. 4.4) мы используем две переменные — «п1внутри» и «п2внутри», которые имеют истинное значение, если «процессодин» и «процессдва» соответственно находятся внутри своих критических участков.

В этой версии программы «процессодин» остается в состоянии активного ожидания (busy wait) до тех пор, пока «п2внутри» имеет значение «истина». В конце концов «процессдва» выходит из своего критического участка и выполняет

собственный код «выходизаимоисключения», устанавливая для переменной «п2внутри» значение «ложь». После этого «процессодин» устанавливает для переменной «п1внутри» значение «истина» и входит в свой критический участок. Когда переменная «п1внутри» имеет значение «истина», «процессдва» в свой критический участок войти не может.

Здесь опять-таки выявляются некоторые нюансы, связанные со спецификой параллельного программирования. Поскольку «процессодин» и «процессдва» являются параллельными процессами, они оба могут одновременно попытаться начать выполнять свои входные последовательности взаимоисключения. Вначале «п1внутри» и «п2внутри» имеют значение «ложь». «Процессодин» может проверить переменную «п2внутри» и обнаружить, что она имеет значение «ложь», а затем, еще до того, как «процессодин» успеет установить для переменной «п1внутри» значение «истина», «процессдва» может проверить переменную «п1внутри» и обнаружить «ложь». В этот момент «процессодин» установит истинное значение для «п1внутри» и войдет в свой критический участок, и «процессдва» установит истинное значение для «п2внутри» и войдет в свой критический участок. При этом оба процесса оказываются в своих критических участках одновременно, так что программа версии 2 даже не гарантирует взаимоисключения.

Слабым местом программы версии 2 является то, что между моментом, когда процесс, находящийся в цикле ожидания, определяет, что он может идти дальше, и моментом, когда этот процесс устанавливает флаг-признак, говорящий о том, что он вошел в свой критический участок, проходит достаточно времени, чтобы другой процесс успел проверить еще не установленный флаг и войти в свой критический участок. Поэтому необходимо, чтобы, в то время как один процесс выполняет свой цикл ожидания, другой процесс не мог выйти из своего собственного цикла ожидания. В программе версии 3 (рис. 4.5) для решения этой проблемы предусматривается установка каждым процессом своего собственного флага перед выполнением цикла ожидания.

Программа версии 3 позволила решить одну задачу, однако сразу же появилась другая. Если каждый процесс перед переходом на цикл проверки будет устанавливать свой флаг, то каждый процесс будет обнаруживать, что флаг другого процесса установлен и будет бесконечно оставаться в цикле while. Эта версия программы может служить примером тупика для двух процессов.

Главный недостаток программы версии 3 заключается в том, что каждый из процессов может заблокироваться в своем цикле ожидания while. Нам нужен способ «разорвать» эти циклы. В версии 4

program версиятри;

var п1хочетвойги, п2хочетвойти: логический;

procedure процессодин;

begin

while истина **do**

begin


```

    п1хочетвойти := истина;

    while п2хочетвойти do;
        критическийучастокдин;
        п1хочетвойти := ложь;
        прочиеоператорыодин
    end

end;

procedure процессдва;
begin
    while истина do;
        begin
            п2хочетвойти := истина;

            while п1хочетвойти do;
                критическийучастокдва;
                п2хочетвойти := ложь;
                прочиеоператорыдва
            end

        end;

    begin
        п1хочетвойти := ложь;
        п2хочетвойти := ложь;

        parbegin
            процессодин;

            процессдва;

        parend

    end;

end;

```

Рис. 4.5 Программная реализация примитивов взаимного исключения (версия 3).

(рис. 4.6) для этого предусматривается периодическая кратковременная установка ложного значения флага каждым вошедшим в цикл процессом. Благодаря этому другой процесс получает возможность выйти из своего цикла ожидания при по-прежнему установленном собственном флаге.

В программе версии 4 гарантируется взаимное исключение и отсутствие тупика, однако возникает другая потенциальная проблема, также очень неприятная, а именно бесконечное откладывание. Рассмотрим, каким образом это происходит. Поскольку мы не можем делать никаких предположений об относительных скоростях

```
program версиячетыре;
```

```
var п1хочетвойти, п2хочетвойти: логический;
```

```
procedure процессодин;
```

```
begin
```

```
while истина do
```

```
begin
```

```
п1хочетвойти := истина;
```

```
while п2хочетвойти do
```

```
begin
```

```
п1хочетвойти := ложь;
```

```
задержка (случайная, несколькотактов);
```

```
п1хочетвойти := истина
```

```
end
```

```
критическийучасткодин;
```

```
п1хочетвойти := ложь;
```

```
прочиеоператорыодин
```

```
end
```

```
end;
```

```
procedure процессдва;
```

```
begin
```

while истина **do**

begin

п2хочетвойти := истина;

while п1хочетвойти **do**

begin

п2хочетвойти := ложь;

задержка (случайная, несколько тактов);

п2хочетвойти := истина

end

критический участок два;

п2хочетвойти := ложь;

прочие операторы два

end

end;

begin

п1хочетвойти := ложь;

п2хочетвойти := ложь;

parbegin

процесс один;

процесс два

parend

end;

Рис. 4.6 Программная реализация примитивов взаимного исключения (версия 4).

program алгоритм Деккера;

var избранный процесс: (первый, второй);

п1хочетвойти, п2хочетвойти: логический;

```

procedure процессодин;

begin

while истина do

begin

    п1хочетвойти := истина;

    while п2хочетвойти do

    if избранныйпроцесс = второй then

    begin

        п1хочетвойти := ложь;

        while избранныйпроцесс = второй do;

        п1хочет войти := истина

    end

    критическийучастокодин;

    избранный процесс := второй;

    п1хочетвойти := ложь;

    прочиеоператорыюдин

    end

end;

procedure процессдва;

begin

while истина do

begin

    п2хочетвойти := истина

    while п1хочетвойти do

    if избранныйпроцесс = первый then

    begin

        п2хочет войти := ложь;

```

```
while избранныйпроцесс = первый do;
```

```
    п2хочетвойти := истина
```

```
end
```

```
критическийучастокдва;
```

```
избранныйпроцесс := первый;
```

```
    п2хрчетвойти := ложь;
```

```
    прочиеоператорыдва
```

```
end
```

```
end;
```

```
begin
```

```
    п1хочетвойти := ложь;
```

```
    п2хочетвойти := ложь;
```

```
    избранныйпроцесс := первый;
```

```
parbegin
```

```
    процессодин;
```

```
    процессдва
```

```
parend
```

```
end;
```

Рис. 4.7 Реализация примитивов взаимного исключения согласно алгоритму Деккера.

асинхронных параллельных процессов, мы должны проанализировать все возможные последовательности выполнения программы. Процессы здесь могут, например, выполняться «тандемом», друг за другом в следующей последовательности: каждый процесс может установить истинное значение своего флага; произвести проверку в начале цикла; войти в тело цикла; установить ложное значение своего флага; снова установить истинное значение флага; повторить всю эту последовательность, начиная с проверки при входе в цикл. Когда процессы будут выполнять все эти действия, условия проверки будут оставаться истинными. Естественно, подобный режим работы весьма маловероятен — однако все же в принципе возможен. Поэтому версия 4 также оказывается неприемлемой. Программу, реализующую взаимное исключение таким способом, нельзя применять, например, в системе управления космическими полетами, управления воздушным движением или в водителе ритма сердца человека, где даже малая вероятность бесконечного откладывания процесса и последующий отказ системы в целом категорически недопустимы.

Алгоритм, предложенный Деккером, позволяет при помощи небольшого по объему программного кода (рис. 4.7) изящно решить проблему взаимного исключения для двух процессов, не требуя при этом никаких специальных аппаратно-реализованных команд.

Алгоритм Деккера исключает возможность бесконечного откладывания процессов, из-за которых программа версии 4 неприемлема. Рассмотрим, каким образом это делается. Процесс «p1» уведомляет о желании войти в свой критический участок, устанавливая свой флаг. Затем он переходит к циклу, в котором проверяет, не хочет ли также войти в свой критический участок и «p2». Если флаг «p2» не установлен, то «p1» пропускает тело цикла ожидания и входит в свой критический участок.

Предположим, однако, что «p1» при выполнении цикла проверки обнаруживает, что флаг «p2» установлен. Это заставляет «p1» войти в тело своего цикла ожидания. Здесь он анализирует значение переменной «избранный процесс», которая используется для разрешения конфликтов, возникающих в случае, когда оба процесса одновременно хотят войти в свой критический участок. Если избранным процессом является «p1», он пропускает тело своего цикла if и повторно выполняет цикл проверки в ожидании момента, когда «p2» сбросит свой флаг. (Мы вскоре увидим, что «p2» со временем должен это сделать.)

Если процесс «p1» определяет, что преимущественное право принадлежит процессу «p2», он входит в тело своего цикла if, где сбрасывает свой собственный флаг, а затем блокируется в цикле ожидания, пока избранным процессом остается «p2». Сбрасывая свой флаг, «p1» дает возможность «p2» войти в свой критический участок.

Со временем «p2» выйдет из своего критического участка и выполнит свой код «выход из взаимного исключения». Операторы этого кода обеспечат возврат преимущественного права процессу «p1» и сброс флага «p2». Теперь у «p1» появляется возможность выйти из внутреннего цикла ожидания while и установить собственный флаг. Затем «p1» выполняет внешний цикл проверки. Если флаг «p2» (недавно сброшенный) по-прежнему сброшен, то «p1» входит в свой критический участок. Если, однако, «p2» сразу же пытается вновь войти в свой критический участок, то его флаг будет установлен, и «p1» снова придется войти в тело внешнего цикла while. Однако на этот раз «бразды правления» находятся уже у процесса «p1», поскольку сейчас именно он является избранным процессом (напомним, что «p2», выходя из своего критического участка, установил для переменной «избранный процесс» значение «первый»). Поэтому «p1» пропускает тело условной конструкции if и многократно выполняет внешний цикл проверки, пока «p2» «смирно» не сбросит собственный флаг, позволяя процессу «p1» войти в свой критический участок.

Рассмотрим сейчас следующую интересную возможность. Когда «p1» выходит из внутреннего цикла активного ожидания, он может потерять центральный процессор, а «p2» в это время пройдет цикл и вновь попытается войти в свой критический участок. При этом «p2» первым установит свой флаг и вновь войдет в свой критический участок. Когда «p1» снова захватит процессор, он установит свой флаг. Поскольку сейчас будет очередь процесса «p1», то «p2», если он попытается вновь войти в свой критический участок, должен будет сбросить свой флаг и перейти на внутренний цикл активного ожидания, так что «p1» получит возможность входа в свой критический участок. Этот прием позволяет решить проблему бесконечного откладывания.

4.9 Взаимоисключение для N процессов

Программное решение проблемы реализации примитивов взаимного исключения для n процессов первым предложил Дейкстра (Di65a). Затем Кнут (Kn66) усовершенствовал алгоритм Дейкстры, исключив возможность бесконечного откладывания процессов, однако в варианте Кнута некоторый процесс по-прежнему мог испытывать (потенциально) длительную задержку. В связи с этим многие ученые начали искать алгоритмы, обеспечивающие более короткие задержки. Так, Эйзенберг и Макгайр (Ei72) предложили решение, гарантирующее, что процесс будет входить в свой критический участок не более чем за $n-1$ попыток. Лэмпорт (La74) разработал алгоритм, применимый, в частности, для распределенных систем обработки данных. Алгоритм Лэмпорта предусматривает «предварительное получение номерка», подобно тому как это делается в модных кондитерских! и поэтому получил наименование алгоритма кондитера (Bakery Algorithm). Бринк Хансен (Br78a) также предлагает возможные варианты управления параллельными распределенными процессами.

4.10 Аппаратная реализация взаимного исключения: команда *testandset*

Алгоритм Деккера — это программное решение проблемы взаимного исключения. В данном разделе мы приводим аппаратное решение данной проблемы.

Главным фактором, обеспечивающим успех в этом случае, является наличие одной аппаратной команды, которая осуществляет чтение переменной, запись ее значения в область сохранения и установку нужного конкретного значения этой переменной. Подобная команда, обычно называемая *testandset* (проверить-и-установить), после запуска выполняет все эти действия до конца без прерывания. Неделимая команда

testandset (a, b)

читает значение логической переменной b , копирует его в a , а затем устанавливает для b значение «истина» — и все это в рамках одной непрерываемой операции. Пример использования команды проверки и установки для реализации взаимного исключения приведен на рис. 4.8.

Переменная «активный» логического типа имеет значение «истина», когда любой из процессов находится в своем критическом участке, и значение «ложь» в противном случае. «Процессодин» принимает решение о входе в критический участок в зависимости от значения своей локальной логической переменной «первоувходитьнельзя». Он устанавливает для переменной «первоувходитьнельзя» значение «истина», а затем многократно выполняет команду проверки и установки для глобальной логической переменной «активный». Если «процессдва» находится вне критического участка, переменная «активный» будет иметь значение «ложь». Команда проверки и установки запишет это значение в «первоувходитьнельзя» и установит значение «истина» для переменной «активный». При проверке в цикле *while* будет получен результат «ложь», и «процессодин» войдет в свой критический участок. Поскольку для переменной «активный» установлено значение «истина», «процессдва» в свой критический участок войти не может.

Предположим теперь, что «процессдва» уже находится в своем критическом участке, когда «процессодин» хочет войти в критический участок. «Процессодин»

устанавливает значение «истина» для переменной «первомувходитьнельзя», а затем многократно проверяет значение переменной «активный» по команде testandset. Поскольку «процессдва» находится в своем критическом участке, это значение остается истинным. Каждая команда проверки и установки

program примепtestandset

var активный: логический;

procedure процессодин;

var первомувходитьнельзя: логический;

begin

while истина **do**

begin

первомувходитьнельзя: истина;

while первомувходитьнельзя **do**

testandset (первомувходитьнельзя, активный);

критическийучасткодин;

активный := ложь;

прочиеоператорыодин

end

end;

procedure процессдва;

var второмувходитьнельзя: логический;

begin

while истина **do**

begin

второмувходитьнельзя := истина;

while второмувходитьнельзя **do**

testandset (второмувходитьнельзя, активный);

критическийучастокдва;


```

активный := ложь;

прочиеоператорыдва

end

end;

begin

активный := ложь;

parbegin

процессодин;

процессдва

parend

end;

```

Рис 4.8 Реализация взаимного исключения при помощи команды testandset (ПРОВЕРИТЬ И УСТАНОВИТЬ).

обнаруживает, что «активный» имеет значение «истина», и устанавливает это значение для переменных «первомувходитьнельзя» и «активный». Таким образом, «процессодин» продолжает находиться в цикле активного ожидания, пока «процессдва» в конце концов не выйдет из своего критического участка и не установит значение «ложь» для переменной «активный». В этот момент команда проверки и установки, обнаружив это значение переменной «активный» (и установив для нее истинное значение, чтобы «процессдва» не мог больше войти в свой критический участок), установит значение «ложь» для переменной «первомувходитьнельзя», что позволит, чтобы «процессодин» вошел в свой критический участок.

Этот способ реализации взаимного исключения не исключает бесконечного откладывания, однако здесь вероятность такой ситуации весьма мала, особенно если в системе имеется несколько процессоров. Когда процесс, выходя из своего критического участка, устанавливает значение «ложь» переменной «активный», команда проверки и установки testandset другого процесса, вероятнее всего, сможет «перехватить» переменную «активный» (установив для нее значение «истина») до того, как первый процесс успеет пройти цикл, чтобы снова установить значение «истина» для этой переменной.

4.11 Семафоры

Все описанные выше ключевые понятия, относящиеся к взаимному исключению, Дейкстра суммировал в своей концепции семафоров (Di65). Семафор — это защищенная переменная, значение которой можно опрашивать и менять только при помощи специальных операций *P* и *V* и операции инициализации, которую мы будем называть «инициализациясемафора». Двоичные семафоры могут принимать только значения 0 и 1. Считывающие семафоры (семафоры со счетчиками) могут

принимать неотрицательные целые значения.

Операция P над семафором записывается как P(S) и выполняется следующим образом:

если $S > 0$

то $S := S - 1$

иначе (ожидать на S)

Операция V над семафором S записывается как V(S) и выполняется следующим образом:

если (один или более процессов ожидают на S)

то (разрешить одному из этих процессов продолжить работу)

иначе $S := S + 1$

Мы будем предполагать, что очередь процессов, ожидающих на S, обслуживается в соответствии с дисциплиной «первый пришел первый» (FIFO).

Подобно операции проверки и установки testandset, операции P и V являются неделимыми. Участки взаимоисключения по семафору S в процессах обрамляются операциями P(S) и V(S). Если одновременно несколько процессов попытаются выполнить операцию P(S), это будет разрешено только одному из них, а остальным придется ждать.

Семафоры и операции над ними могут быть реализованы как программно, так и аппаратно. Как правило, они реализуются в ядре операционной системы, где осуществляется управление сменой состояния процессов.

На рис. 4.9 приводится пример того, каким образом можно обеспечить взаимоисключение при помощи семафоров. Здесь примитив P (активный) — эквивалент для «входвзаимоисключения», а примитив V (активный) — для «выходвзаимоисключения».

program примерсемафораодин;

var активный: семафор;

procedure процессодин;

begin

while истина **do**

begin

```

    предшествующие операторы один;

    P(активный);

    критический участок один;

    V(активный);

    прочие операторы один;

end

end;

procedure процесс два;

begin

while истина do

begin

    предшествующие операторы два;

    P(активный);

    критический участок два;

    V(активный);

    прочие операторы два

end

end;

begin

    инициализация семафора (активный, 1);

parbegin

    процесс один;

    процесс два

parend

end;

```

Рис. 4.9 Обеспечение взаимного исключения при помощи семафора и примитивов P и V.

4.12 Синхронизация процессов при помощи семафоров

Когда процесс выдает запрос ввода-вывода, он блокирует себя в ожидании завершения соответствующей операции ввода-вывода. Заблокированный процесс должен быть активизирован каким-либо другим процессом. Подобное взаимодействие может служить примером функций, относящихся к протоколу блокирования/возобновления.

Рассмотрим более общий случай, когда одному процессу необходимо, например, чтобы он получал уведомление о наступлении некоторого события. Предположим, что какой-либо другой процесс может обнаружить, что данное событие произошло. Программа рис. 4.10 показывает, каким образом при помощи семафора можно реализовать простой механизм синхронизации (блокирования/возобновления) для двух процессов.

program блокированиевозобновления;

var событие: семафор;

procedure процессодин;

begin

предшествующиеоператорыодин;

P(событие);

прочиеоператорыодин

end;

procedure процессдва;

begin

предшествующиеоператорыдва;

V(событие);

прочиеоператорыдва

end;

begin

инициализациясемафора (событие,0);

parbegin

процессодин;

процессдва

parend

end;

Рис. 4.10 Синхронизация блокирования/возобновление процессов при помощи семафоров.

Здесь «процесссдин» выполняет некоторые «предшествующиеоператорыюдин», а затем операцию P(событие). Ранее при инициализации семафор был установлен в нуль, так что «процесссдин» будет ждать. Со временем «процесссдва» выполнит операцию V(событие), сигнализируя о том, что данное событие произошло. Тем самым «процесссдин» получает возможность продолжить свое выполнение.

Отметим, что подобный механизм будет выполнять свои функции даже в том случае, если «процесссдва» обнаружит наступление события и просигнализирует об этом еще до того, как «процесссдин» выполнит операцию P (событие); при этом семафор переключится из 0 в 1, так что операция P (событие) просто произведет обратное переключение, из 1 в 0, и «процесссдин» продолжит свое выполнение без ожидания.

4.13 Пара «производитель — потребитель»

Когда в последовательной программе одна процедура вызывает другую и передает ей данные, обе эти процедуры являются частями единого процесса — они не выполняются параллельно. Если, однако, один процесс передает данные другому процессу, возникают определенные проблемы. Подобная передача может служить примером взаимодействия, или обмена информацией между процессами.

program парaproизводительпотребитель;

var исключительныйдоступ: семафор;

числозанесено: семафор;

буферчисла: целое;

procedure процесспроизводитель;

var следующийрезультат: целое;

begin

while истина **do**

begin

вычислениеследующегорезультата;

```

P(исключительныйдоступ);

буферчисла := следующийрезультат;

V(исключительныйдоступ);

V(числозанесено)

end

end;

procedure процесспотребитель;

var следующийрезультат: целое;

begin

while истина do

begin

P(числозанесено);

P(исключительныйдоступ);

следующийрезультат := буферчисла;

V(исключительныйдоступ);

записать (следующий результат)

end

end;

begin

инициализациясемафора (исключительныйдоступ, 1);

инициализациясемафора (числозанесено, 0);

parbegin

процесспроизводитель:

процесспотребитель

parend

end;

```

Рис. 4.11 Реализация взаимодействия в паре «производитель—потребитель» при помощи семафоров.

Рассмотрим следующую пару (отношение) «производитель— потребитель». Предположим, что один процесс, *источник*, или *производитель*, генерирует информацию, которую другой процесс, *получатель*, или *потребитель*, использует. Предположим, что они взаимодействуют при помощи одной разделяемой целой переменной с именем «буферчисла». Процесс-производитель производит некоторые вычисления, а затем заносит результат в «буферчисла»; процесс-потребитель читает «буферчисла» и печатает результат.

Возможно, что эти процессы, производитель и потребитель, работают в достаточно близком темпе либо резко различаются по скоростям. Если каждый раз, когда процесс-производитель помещает свой результат в «буферчисла», процесс-потребитель будет немедленно считывать и печатать его, то на печать будет верно выдаваться та последовательность чисел, которую формировал процесс-производитель.

Предположим теперь, что скорости обоих процессов резко различны. Если процесс-потребитель работает быстрее, чем процесс-производитель, он может прочитать и напечатать одно и то же число дважды (или в общем случае много раз), прежде чем процесс-производитель выдаст следующее число. Если же процесс-производитель работает быстрее, чем потребитель, он может записать новый результат на место предыдущего до того, как процесс-потребитель успеет прочитать и напечатать этот предшествующий результат; процесс-производитель, работающий с очень высокой скоростью, фактически может по несколько раз перезаписывать результат, так что будет потеряно много результатов.

Очевидно, что здесь мы хотели бы обеспечить такое взаимодействие процесса-производителя и процесса-потребителя, при котором данные, заносимые в «буферчисла», никогда не терялись бы и не дублировались. Создание подобного режима взаимодействия является примером синхронизации процессов.

На рис. 4.11 показана параллельная программа, в которой для реализации взаимодействия в паре «производитель — потребитель» применяются операции над семафорами.

В этой программе мы ввели два семафора: «исключительный доступ» используется для обеспечения доступа к разделяемой переменной в режиме взаимного исключения, а «числозанесено» — для обеспечения синхронизации процессов.

4.14 Считающие семафоры

Считающие семафоры особенно полезны в случае, если некоторый ресурс выделяется из пула идентичных ресурсов. При инициализации подобного семафора в его счетчике указывается количественный показатель объема ресурсов пула. Каждая операция P вызывает уменьшение значения счетчика семафора на 1, показывая, что некоторому процессу для использования выделен еще один ресурс

из пула. Каждая операция V вызывает увеличение значения счетчика семафора на 1, показывая, что процесс возвратил в пул ресурс и этот ресурс может быть выделен теперь другому процессу. Если делается попытка выполнить операцию P, когда в счетчике семафора уже нуль, то соответствующему процессу придется ждать момента, пока в пул не будет возвращен освободившийся ресурс, т. е. пока не будет выполнена операция V.

4.15 Реализация семафоров, операции P и V

При помощи алгоритма Деккера и/или команды проверки и установки `testandset`, если она предусмотрена в машине, можно достаточно просто реализовать операции P и V с циклом активного ожидания. Однако активное ожидание может приводить к потере эффективности.

В главе 3 мы рассмотрели реализуемые в ядре операционной системы механизмы для переключения процессов из состояния в состояние. Было отмечено, что процесс, выдавший запрос на операцию ввода-вывода, тем самым добровольно блокирует себя в ожидании завершения данной операции. Заблокированный процесс — это не тот процесс, который находится в состоянии активного ожидания. Он просто освобождает процессор, а ядро операционной системы включает блок управления этого процесса (PCB) в список заблокированных. Тем самым этот процесс «засыпает» — до того момента, пока не будет активизирован ядром, которое переведет этот процесс из списка заблокированных в список готовых к выполнению.

Операции над семафорами можно также реализовать в ядре операционной системы, чтобы избежать режима активного ожидания для процессов. Семафор реализуется как защищенная переменная и очередь, в которой процессы могут ожидать выполнения V-операций. Когда некоторый процесс пытается выполнить P-операцию для семафора, имеющего нулевое текущее значение, этот процесс освобождает процессор и блокирует себя в ожидании V-операции для данного семафора. При этом ядро операционной системы включает PCB процесса в очередь процессов, ожидающих разрешающего значения данного семафора. (Здесь мы предполагаем, что реализуется дисциплина обслуживания очереди «первый пришедший обслуживается первым». Были исследованы и другие дисциплины обслуживания, в том числе очереди с различными приоритетами.) Затем ядро предоставляет центральный процессор следующему процессу, готовому к выполнению.

Процесс, находящийся в очереди данного семафора, со временем становится первым в этой очереди. Затем при выполнении следующей V-операции этот процесс переходит из очереди семафора в список готовых к выполнению. Естественно, что для процессов, пытающихся одновременно выполнить операции P и V для семафора, ядро гарантирует взаимоисключающий доступ к этому семафору.

Отметим, что в особом случае однопроцессорной машины неделимость операций P и V может быть обеспечена просто путем запрета прерываний в то время, когда при помощи этих операций осуществляются манипуляции с семафором. Тем самым предотвращается перехват процессора до завершения операции (после чего прерывания разрешаются снова).

Заключение

Процессы называются параллельными, если они существуют одновременно. Асинхронным параллельным процессам требуется периодически синхронизироваться и взаимодействовать.

Системы, которые ориентированы на параллельные вычисления, интересно и важно исследовать, поскольку сейчас существуют явные тенденции перехода на мультипроцессорные архитектуры с максимальным параллелизмом на всех

уровнях; поскольку трудно в принципе определить, какие операции можно и какие нельзя выполнять параллельно; поскольку параллельные программы отлаживать гораздо сложнее, чем последовательные; поскольку взаимодействия между асинхронными процессами могут быть самыми разнообразными и поскольку производить доказательство корректности для параллельных программ гораздо сложнее, чем для последовательных.

Конструкция `parbegin/parend` применяется для указания того, что единая последовательность управления разделяется на несколько параллельных цепочек и что в какой-то момент все эти цепочки снова сливаются в одну. В общем случае эта конструкция имеет вид

`parbegin`

`оператор1;`

`оператор2;`

`.`

`.`

`.`

`оператор n`

`parend`

и показывает, что все эти операторы могут выполняться параллельно.

Способ взаимодействия, при котором во время обращения одного процесса к разделяемым данным всем другим процессам это запрещается, называется взаимным исключением. Если процесс обращается к разделяемым данным, то говорят, что этот процесс находится в своем критическом участке (критической области). Когда процессы взаимодействуют с использованием общих переменных и один из процессов находится в своем критическом участке, для всех остальных процессов возможность вхождения в критические участки должна исключаться.

Процесс должен проходить свой критический участок как можно быстрее, он не должен блокироваться в критическом участке; критические участки необходимо программировать особенно тщательно (чтобы исключить, например, возможность закливания).

Процесс, которому нужно войти в свой критический участок, выполняет примитив «входвзаимногоисключения», который заставляет данный процесс подождать, если в этот момент другой процесс находится в своем критическом участке. Процесс, выходящий из своего критического участка, выполняет примитив «выходвзаимногоисключения», что позволяет ожидающему процессу продолжить свое выполнение.

Алгоритм Деккера, обеспечивающий программную реализацию примитивов взаимного исключения, обладает следующими свойствами:

- Он не требует специальных аппаратных команд.
- Процесс, работающий вне своего критического участка, не может помешать другому процессу войти в его критический участок.
- Процесс, желающий войти в свой критический участок, получает такую возможность без бесконечного откладывания.

Алгоритм Деккера предусматривает реализацию взаимного исключения для двух процессов. Разработаны программные способы реализации взаимного исключения и для n процессов; как правило, эти способы достаточно сложны.

Неделимая команда проверки и установки `testandset(a, b)` считывает значение логической переменной b , копирует его в a , а затем устанавливает для b истинное значение. Эта команда может обеспечить реализацию взаимного исключения.

Семафор — это защищенная переменная, значение которой можно читать и менять только при помощи операций P и V , а также операции инициализации (в тексте она носит название «инициализация семафора»). Двоичные семафоры могут принимать только значения 0 и 1. Считывающие семафоры могут принимать неотрицательные целые значения.

Операция P над семафором записывается как $P(S)$ и выполняется следующим образом:

если $S > 0$

то $S := S - 1$

иначе (ожидать на S)

Операция V над семафором S записывается как $V(S)$ и выполняется следующим образом:

если (один или более процессов ожидают на S)

то (разрешить одному из этих процессов продолжить работу)

иначе $S := S + 1$

Семафоры можно использовать для реализации механизма синхронизации процессов путем блокирования/возобновления: один процесс блокирует себя (выполняя операцию $P(S)$ с начальным значением $S=0$), чтобы подождать наступления некоторого события; другой процесс обнаруживает, что ожидаемое событие произошло, и возобновляет заблокированный процесс (при помощи операции $V(S)$).

В паре «производитель—потребитель» один процесс, источник, или производитель, генерирует информацию, которую использует другой процесс, получатель, или потребитель. Это пример взаимодействия между процессами. Если процессы взаимодействуют при помощи общего буфера, то источник не должен выдавать информацию, когда буфер заполнен, а получатель не должен пытаться принять информацию, когда буфер пустой. Создание подобного режима взаимодействия является примером синхронизации процессов.

Считывающие семафоры особенно полезны в случае, если некоторый ресурс выделяется из пула идентичных ресурсов. Каждая P -операция показывает, что

ресурс выделяется некоторому процессу, а V-операция — что ресурс возвращается в общий пул.

Операции над семафорами можно реализовать с использованием режима активного ожидания, однако это сопряжено с потерей эффективности. Чтобы избежать этого, подобные операции можно реализовать в ядре операционной системы.

Терминология

активное ожидание (в цикле) (busy wait)

алгоритм Деккера (Dekker's Algorithm)

алгоритм Лэмпорта, алгоритм кондитера (Lamport's Bakery Algorithm)

асинхронность (asynchronism)

асинхронные параллельные процессы (asynchronous concurrent processes)

бесконечное откладывание (процесса) (indefinite postponement)

блокировка памяти (защитная) (storage interlock)

взаимоисключение (mutual exclusion)

взаимоисключение для двух процессов (two-process mutual exclusion)

взаимоисключение для n процессов (n -process mutual exclusion)

входвзаимоисключения (entermutualexclusion)

выделение (в самостоятельный элемент), форматирование, обрамление ограничителями (encapsulation)

выходвзаимоисключения (exitmutualexclusion)

двоичный семафор (binary semaphore)

Дейкстра (Dijkstra)

жесткая (пошаговая) синхронизация (lockstep synchronization)

защищенная переменная (protected variable)

код входа взаимодействия (mutual exclusion entry code)

код выхода взаимодействия (mutual exclusion exit code)

команда проверки и установки (testandset instruction)

координированный (последовательный) доступ (sequentialization)

критическая область (critical section)

критический участок (critical region)

неделимые операции (indivisible operations)

общие, разделяемые, совместно (коллективно) используемые данные (shared data)

общие, разделяемые, совместно (коллективно) используемые ресурсы (shared resource)

операция P (P operation)

операция V (V operation)

параллелизм (совмещение) (concurrency)

параллельные вычисления (параллельная обработка) (parallel processing)

параллельное программирование (concurrent programming)

примитивы взаимного исключения (mutual exclusion primitives)

протокол блокирования/пробуждения (block/wakeup protocol)

семафор (semaphore)

синхронизация процессов (process synchronization)

степень параллелизма (level of parallelism)

читающий семафор (семафор со счетчиком) (counting semaphore)

тупик, дедлок (deadlock)

цепочка (нить, последовательность) управления (thread of control)

cobegin/coend

parbegin/parend

Упражнения

4.1 Назовите несколько причин, обуславливающих важность и актуальность исследования параллелизма для изучающих операционные системы.

4.2 Преобразуйте следующее выражение с использованием конструкции parbegin/parend для обеспечения максимального параллелизма:

$$3*a*b+4/(c+d)**(e-f).$$

4.3 Преобразуйте следующую программу параллельных вычислений в простую последовательность операций:

```
a := b+c;
```

```
parbegin
```

```
d := b*c-x;
```

```
e := a/6+n**2
```

```
parend
```

4.4 Почему может оказаться неприемлемой следующая программа:

```
parbegin
```

```
a := b+c;
```

```
d := b*c-x;
```

```
e := a/6+n**2
```

```
parend
```

4.5 Приведите несколько причин, по которым можно считать ложным следующее утверждение: «Когда несколько процессов работают с общей, разделяемой информацией, хранящейся в основной памяти, во избежание получения неопределенных результатов необходимо обеспечивать взаимоисключение»

4.6 Для реализации взаимоисключения можно использовать алгоритм Деккера, команду проверки и установки testandset и операции P и V над семафорами. Укажите сходства и различия всех этих способов, а также рассмотрите их относительные достоинства и недостатки.

4.7 Мы предполагали, что, когда два процесса одновременно пытаются выполнить примитив «входвзаимоисключения», «победитель» выбирается случайным образом. Обсудите следствия этого предположения.

4.8 Прокомментируйте следующий пример использования примитивов взаимоисключения:

```
какиетооператорыодин;
```

```
выходвзаимоисключения;
```

```
какиетооператорыдва;
```

```
выходвзаимоисключения;
```

```
какиетооператорытри;
```

```
выходвзаимоисключения;
```

```
какиетооператорычетыре;
```

```
выходвзаимоисключения;
```

```
какиетооператорыпять
```

4.9 В чем заключается действительный смысл алгоритма Деккера?

4.10 Проведите исчерпывающий временной анализ алгоритма Деккера. Имеются ли в этом алгоритме какие-либо слабости?

4.11 Не ссылаясь на известные варианты реализации примитивов взаимного исключения для n процессов, разработайте собственный способ решения этой проблемы. (Это очень сложная проблема.)

4.12 Способ реализации взаимного исключения для n процессов, который предложили Эйзенберг и Макгайр (Ei 72), гарантирует, что процесс войдет в свой критический участок не более чем за $n-1$ попыток. Можно ли надеяться, что удастся получить лучшее решение для n процессов?

4.13 Примитивы взаимного исключения можно реализовать с использованием ре« жима активного ожидания или блокирования процессов. Обсудите возможности применения и относительные достоинства обоих способов.

4.14 Объясните достаточно подробно, каким образом можно реализовать в ядре операционной системы семафоры и операции над семафорами.

4.15 Объясните, каким образом при помощи запрещения и разрешения прерываний можно достаточно эффективно реализовать примитивы взаимного исключения в однопроцессорной машине.

4.16 Почему V-операцию необходимо выполнять как неделимую?

4.17 Что произойдет в программе, приведенной на рис. 4.11, если в процессе-потребителе поменять местами P-операторы? Что произойдет, если в процессе-производителе поменять местами V-операторы?

4.18 Перепишите программу, приведенную на рис. 4.9, с использованием считающих семафоров, с тем чтобы обеспечить управление доступом к пулу из пяти идентичных ресурсов.

4.19 Как упоминалось в тексте, критические участки, в которых производятся обращения к непересекающимся множествам разделяемых переменных, фактически могут выполняться одновременно. Предположим, что каждый из примитивов взаимного исключения модифицирован таким образом, что включает в качестве параметров список конкретных разделяемых переменных, к которым будет производиться обращение в критическом участке.

а) Прокомментируйте следующий вариант использования этих новых примитивов взаимного исключения:

какietoоператорыюдин;

входвзаимногоисключения (a);

какietoоперацииис«a»;

входвзаимногоисключения (b);

какietoоперацииис«a»и«b»;

выходвзаимногоисключения (b);

ещекакietoоперацииис«a»;

выходвзаимногоисключения (a);

Глава 5

Параллельное программирование: мониторы; рандеву в языке Ада 1)

Минотавр, сущ.— 1: В древнегреческой мифологии — чудовище, рожденное Пасифаей от быка и имеющее туловищ/! человека и голову быка; обитало в лабиринте на Крите и пожирало людей, пока Тесей с помощью Ариадны не убил его. 2: Живое существо или неодушевленный предмет, главные свойства которого — уничтожение и разрушение.

Толковый словарь английского языка издательства Random House

Вы смотрите, но не видите.

Сэр Артур Конан Доил «Записки Шерлока Холмса»

■ *Мы вполне можем сказать, что Аналитическая машина позволяет обрабатывать алгебраические выражения, подобно тому как ткацкий станок Джаккарда позволяет получать ткань с рисунком в виде цветов и листьев.*

Ада Лавлейс

5.1 Введение

5.2 Мониторы

5.3 Простое распределение ресурсов при помощи мониторов

5.4 Пример монитора: кольцевой буфер

5.5 Пример монитора: читатели и писатели

5.6 Ада — язык параллельного программирования 80-х годов

5.7 Мотивы для реализации многозадачного режима в языке Ада

5.8 Корректность параллельных программ

5.9 Рандеву в языке Ада

5.10 Команда приема ACCEPT

5.11 Пример на языке Ада; пара «производитель—потребитель»

5.12 Команда выбора SELECT

5.13 Пример на языке Ада: кольцевой буфер

5.14 Пример на языке Ада: читатели и писатели

5.1 Введение

В предыдущей главе мы представили алгоритм Деккера для реализации примитивов взаимoisключения и описали семафоры, которые предложил Дейкстра. Однако эти средства координации взаимодействия процессов имеют ряд слабостей. Они настолько элементарны, что не позволяют достаточно легко описывать решение сравнительно серьезных проблем параллельных вычислений, а включение этих средств в параллельные программы осложняют доказательства корректности подобных программ. Неправильное использование этих примитивов, либо умышленное, либо непреднамеренное, может привести к нарушению работоспособности системы параллельной обработки. Поэтому для реализации взаимoisключения разработчики вынуждены искать механизмы более высокого уровня, которые

- упрощали бы описание решений сложных проблем параллельных вычислений,
- упрощали бы доказательство корректности программы,
- было бы трудно (если не невозможно) испортить или неправильно использовать.

Самым важным из подобных механизмов является, по-видимому, монитор, первый вариант которого предложил Дейкстра (Di 71), затем переработал Бринк Хансен (Br72, Br73), а впоследствии усовершенствовал Хоор (Ho74).

5.2 Мониторы

Монитор — это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для реализации динамического распределения конкретного общего ресурса или группы общих ресурсов. Чтобы обеспечить выделение нужного ему ресурса, процесс должен обратиться к конкретной процедуре монитора. Необходимость *входа в монитор* в различные моменты времени может возникать у многих процессов. Однако вход в монитор находится под жестким контролем — здесь осуществляется взаимoisключение процессов, так что в каждый момент времени только одному процессу разрешается войти в монитор. Процессам, которые хотят войти в монитор, когда он уже занят, приходится ждать, причем режимом ожидания автоматически управляет сам монитор. Поскольку механизм монитора гарантирует взаимoisключение процессов, исключаются серьезные проблемы, связанные с параллельным режимом работы (например, проблема неоднозначности результатов); эти проблемы обсуждались в гл. 4.

Внутренние данные монитора могут быть либо глобальными (относящимися ко всем процедурам монитора), либо локальными (относящимися только к одной конкретной процедуре). Ко всем этим данным можно обращаться только изнутри монитора; процессы, находящиеся вне монитора, просто не могут получить доступа к данным монитора. Принимая такое структурное решение, значительно упрощающее разработку программных систем повышенной надежности, говорят, что информация спрятана.

Если процесс обращается к некоторой процедуре монитора и обнаруживается, что соответствующий ресурс уже занят, эта процедура монитора выдает *команду ожидания WAIT*. Процесс мог бы оставаться внутри монитора, однако это привело бы к нарушению принципа взаимоисключения, если в монитор затем вошел бы другой процесс. Поэтому процесс, переводящийся в режим ожидания, должен вне монитора ждать того момента, когда необходимый ему ресурс освободится.

Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы вернуть ресурс системе. Соответствующая процедура монитора при этом может просто принять уведомление о возвращении ресурса, а затем ждать, пока не поступит запрос от другого процесса, которому потребуется этот ресурс. Однако может оказаться, что уже имеются процессы, ожидающие освобождения данного ресурса. В этом случае монитор выполняет *примитив оповещения (сигнализации) SIGNAL*, чтобы один из ожидающих процессов мог занять данный ресурс и покинуть монитор. Если процесс сигнализирует о возвращении (иногда называемом освобождением) ресурса и в это время нет процессов, ожидающих данного ресурса, то подобное оповещение не вызывает никаких других последствий, кроме того, что монитор естественно вновь вносит ресурс в список свободных. Очевидно, что процесс, ожидающий освобождения некоторого ресурса, должен находиться вне монитора, чтобы другой процесс имел возможность войти в монитор и вернуть ему этот ресурс.

Чтобы гарантировать, что процесс, находящийся в ожидании некоторого ресурса, со временем действительно получит этот ресурс, считается, что ожидающий процесс имеет более высокий приоритет, чем новый процесс, пытающийся войти в монитор. В противном случае новый процесс мог бы перехватить ожидаемый ресурс до того, как ожидающий процесс вновь войдет в монитор. Если допустить многократное повторение подобной нежелательной ситуации, то ожидающий процесс мог бы откладываться бесконечно.

Фактически, как мы покажем ниже на примерах, у процессов может возникнуть желание (и необходимость) находиться в режиме ожидания вне монитора по многим различным причинам. Поэтому было введено понятие *переменной-условия*. Для каждой отдельно взятой причины, по которой процесс может быть переведен в состояние ожидания, назначается свое условие. В связи с этим команды ожидания и сигнализации модифицируются — в них включаются имена условий:

wait (имя условия)

signal (имя условия)

Переменные-условия совершенно не похожи на «обычные» переменные, с которыми мы привыкли иметь дело. Когда определяется *такая* переменная, заводится очередь. Процесс, выдавший команду

ожидания, включается в эту очередь, а процесс, выдавший команду сигнализации, тем самым позволяет ожидающему процессу выйти из очереди и войти в монитор. Мы можем предполагать, что используется дисциплина обслуживания очереди «первый пришедший обслуживается первым», однако в некоторых случаях могут оказаться более полезными различные приоритетные схемы.

5.3 Простое распределение ресурсов при помощи мониторов

Предположим, что нескольким процессам необходим доступ к определенному ресурсу, который может быть использован в каждый конкретный момент времени только одним процессом. На рис. 5.1 приведен простой монитор-распределитель ресурсов, обеспечивающий выделение и освобождение подобного ресурса:

monitor распределительресурсов;

var ресурсзанят: логический;

ресурссвободен: условие;

procedure захватитьресурс;

begin

if ресурсзанят **then**

wait (ресурссвободен);

ресурсзанят := истина

end;

procedure вернутьресурс;

begin

ресурсзанят := ложь;

signal (ресурссвободен)

end;

begin

ресурсзанят := ложь

end;

Рис. 5.1 Простое распределение ресурса при помощи монитора.

Достоинство подобного распределителя ресурсов заключается в том, что он работает точно так же, как двоичный семафор: команда «захватитьресурс» выполняет функцию Р-оператора, а команда «возвратитьресурс» — V-оператора. Поскольку с помощью такого простого монитора с одним ресурсом можно реализовать семафор, мониторы по меньшей мере обладают той же логической мощностью, что и семафоры.

5.4 Пример монитора: кольцевой буфер

В настоящее время операционные системы строятся, как правило, в виде множества асинхронных параллельных процессов, работающих под управлением ядра. Эти процессы служат для организации параллельных работ, выполняются достаточно независимо, однако требуют периодического взаимодействия. В этом разделе мы рассмотрим так называемый *кольцевой буфер* и покажем, каким образом он может применяться в случаях, когда процесс-производитель должен передать данные процессу-потребителю.

Производителю иногда требуется передать данные, в то время как потребитель еще не готов их принять, а потребитель иногда пытается принять данные, которые производитель еще не выдал. Поэтому необходимо иметь соответствующие средства синхронизации работы процесса-производителя и процесса-потребителя.

В операционных системах часто предусматривается выделение некоторого фиксированного количества ячеек памяти для использования в качестве буфера передач между процессом-производителем и процессом-потребителем. Этот буфер можно представить в виде массива заданного размера. Процесс-производитель помещает передаваемые данные в последовательные элементы этого массива. Процесс-потребитель считывает данные в том порядке, в котором они помещались. Производитель может опережать потребителя на несколько шагов. Со временем процесс-производитель займет последний элемент массива. Когда он сформирует после этого очередные данные для передачи, он должен будет «возвратиться к исходной точке» и снова начать с записи данных в первый элемент массива (при этом естественно предполагается, что потребитель уже прочитал те данные, которые ранее сюда поместил производитель). Такой массив по сути работает как замкнутое кольцо, поэтому и носит название кольцевого буфера.

Поскольку размер кольцевого буфера ограничен, процесс-производитель в какой-то момент может столкнуться с тем, что все элементы массива окажутся занятыми; в этом случае процессу-производителю необходимо будет подождать, пока потребитель не прочитает и тем самым не освободит хотя бы один элемент массива. Аналогично может возникнуть ситуация, когда потребитель хотел бы прочитать данные, а массив оказывается пустым; в этом случае потребитель должен будет ждать, пока процесс-производитель не поместит данные в массив. На рис. 5.2 приведен монитор под названием «мониторкольцевогобуфера», который реализует кольцевой буфер и соответствующий механизм синхронизации для обеспечения взаимодействия в паре «производитель-потребитель» (базовую версию этого монитора предложил Хоор (Ho 74)).

Мы будем предполагать, что массив содержит определенное количество позиций (задаваемое константой «размербуфера»), рассчитанных на занесение данных некоторого также задаваемого типа. Переменные «очереднаязаполняемаяпозиция» и «очереднаяосвобождаемаяпозиция» указывают соответственно, в какой сегмент необходимо поместить очередной элемент данных и откуда необходимо будет выбирать очередной элемент данных. Условие «буфернезаполнен» является признаком, которого ждет процесс-производитель, если обнаружит, что кольцевой буфер целиком заполнен;

monitor мониторкольцевогобуфера;

var кольцевойбуфер: **array** [0..размербуфера - 1] **of** тип;

```

текущая позиция: 0..размербуфера;

очереднаязаполняемаяпозиция: 0..размербуфера - 1;

очереднаяосвобождаемаяпозиция: 0..размербуфера - 1;

буфернепуст, буфернезаполнен: условие;

procedure заполнитьпозицию(данные:тип);

begin

if текущая позиция = размербуфера then wait(буфернезаполнен);

кольцевойбуфер [очереднаязаполняемаяпозиция] := данные;

текущаяпозиция := текущаяпозиция + 1;

очереднаязаполняемаяпозиция := (очереднаязаполняемаяпозиция + 1) mod
размербуфера;

signal(буфернепуст)

end;

procedure освободитьпозицию(var данные:тип);

begin

if текущаяпозиция = 0 then wait(буфернепуст);

данные := кольцевойбуфер[очереднаяосвобождаемаяпозиция];

текущаяпозиция := текущаяпозиция - 1;

очереднаяосвобождаемаяпозиция := (очереднаяосвобождаемаяпозиция + 1) mod
размербуфера;

signal(буфернезаполнен)

end;

begin

текущаяпозиция := 0;

очереднаязаполняемаяпозиция := 0;

очереднаяосвобождаемаяпозиция := 0

end;

```

Рис. 5.2 Реализация кольцевого буфера при помощи монитора.

этот признак устанавливается по сигналу процесса-потребителя о том, что он только что освободил позицию. Условие «буфернепуст» является признаком, которого ждет процесс-потребитель, если обнаружит, что кольцевой буфер пуст; этот признак устанавливается по сигналу процесса-производителя о том, что он только что поместил данные в некоторую позицию.

Механизм кольцевого буфера весьма удобен для реализации управления спулингом в операционных системах. В качестве одного из наиболее распространенных примеров спулинга можно привести ситуацию, когда процесс формирует строки данных, подлежащие выводу на такое относительно низкоскоростное внешнее устройство, как построчный принтер. Поскольку процесс может формировать строки данных с гораздо более высокой скоростью, чем принтер способен их распечатывать, и поскольку желательно создать для процесса такой режим, при котором он мог бы работать с максимальной высокой скоростью, выходные строки данных процесса направляются в кольцевой буфер. Кольцевой буфер может размещаться в основной памяти, но чаще находится на диске. Первый процесс, записывающий данные в буфер, обычно называется спулером. Другой процесс читает строки данных из кольцевого буфера и выдает их на принтер. Однако этот второй процесс, обычно называемый деспулером, работает с гораздо меньшей скоростью — со скоростью принтера. Кольцевой буфер должен иметь достаточно большой размер, чтобы «выбирать слабину», которая возникает из-за несоответствия скоростей работы спулера и деспулера.

5.5 Пример монитора: читатели и писатели

В вычислительных системах обычно имеются некоторые процессы, которые читают данные (и называются «читателями»), и другие процессы, которые записывают данные (и называются «писателями»). Так, в системе предварительной продажи авиационных билетов может быть гораздо больше читателей, чем писателей, поскольку к базе данных, содержащей всю существующую информацию о рейсах самолетов, будет произведено много обращений по запросам кассира-оператора, прежде чем пассажир реально выберет и купит билет на конкретный рейс.

Процессы-читатели не меняют содержимого базы данных, поэтому несколько таких процессов могут обращаться к ней одновременно. А процесс-писатель может изменять данные, поэтому он должен иметь монопольный, исключительный доступ к базе данных. Когда работает процесс-писатель, никакие другие писатели и читатели работать не должны. Конечно, такой режим монопольного доступа следует предусматривать только на уровне работы с индивидуальными записями — не обязательно предоставлять процессу-писателю монопольный доступ к целой базе данных.

Проблему проектирования параллельной программы для управления доступом процессов-читателей и писателей к базе данных впервые сформулировали и решили Куртуа, Хейманс и Парнас (Co71). На рис. 5.3 представлен монитор, решающий задачу читателей и писателей и разработанный на основе алгоритма, который предложили Хоор и Горман (Ho74).

Этот монитор под названием «читателииписатели» может быть использован как средство управления доступом для целой базы данных, для некоторого подмножества базы данных, содержащего много или лишь несколько записей, и даже для отдельно взятой записи. Приведенные ниже рассуждения справедливы для любого из этих случаев. В каждый конкретный момент времени работать может

только один писатель; когда какой-либо писатель работает,

monitor читателииписатели;

var читатели: целое;

ктопишет: логический;

читатьразрешается, писатьразрешается: условие;

procedure началочтения;

begin

if ктопишет **or** очередь(писатьразрешается) **then** wait(читатьразрешается);

читатели := читатели + 1;

signal(читатьразрешается)

end;

procedure конецчтения;

begin

читатели := читатели — 1;

if читатели = 0 **then** signal(писатьразрешается)

end;

procedure началозаписи;

begin

if читатели > 0 **or** ктопишет **then** wait(писатьразрешается);

ктопишет := истина

end;

procedure конецзаписи

begin

ктопишет := ложь;

if очередь(читатьразрешается) **then** signal(читатьразрешается)

else signal(писатьразрешается)

end

begin

читатели := 0;

ктотопишет := ложь

end;

Рис. 5.3 Монитор, решающий задачу читателей и писателей

логическая переменная «ктотопишет» имеет истинное значение. Ни один из читателей не может работать, во время работы какого-либо писателя. Переменная «читатели» указывает количество активных читателей. Когда число читателей оказывается равным нулю, ожидающий процесс-писатель получает возможность начать работу. Новый процесс-читатель не может продолжить свое выполнение, пока не появится истинное значение условия «читатьразрешается», а новый процесс-писатель — пока не появится истинное значение условия «писатьразрешается».

Когда процессу-читателю нужно произвести чтение, он вызывает процедуру монитора под названием «началочтения», а после завершения чтения — процедуру «конецчтения». После входа в процедуру «началочтения» новый процесс-читатель сможет продолжить свою работу, если нет процесса-писателя, производящего в данный момент запись или ожидающего очереди на запись. Второе из этих условий необходимо для того, чтобы предотвратить возможность бесконечного откладывания ожидающих процессов-писателей. Отметим, что процедура «началочтения» завершается выдачей сигнала «читатьразрешается», чтобы следующий ожидающий очереди читатель мог начать чтение. В этом случае следующий процесс-читатель активизируется и выдает находящемуся после него в очереди читателю сигнал о возможности продолжить выполнение. Фактически подобная «цепная реакция» будет идти до тех пор, пока не активизируются все ожидающие процессы-читатели. Во время выполнения такой цепочки действий все вновь приходящие процессы включаются в очередь ожидания. Цепная активизация процессов-читателей — весьма рациональное решение. Поскольку процессы-читатели не мешают друг другу и поскольку они могут выполняться в мультипроцессорных системах параллельно, это эффективный способ обслуживания подобных процессов. Отметим, что во время выполнения цепочки активизации процессов даже вновь приходящие процессы-читатели не могут войти в монитор, поскольку мы соблюдаем правило, согласно которому процессы, уже получившие сигнал активизации, обслуживаются раньше новых процессов.

Когда процесс завершает операцию чтения, он вызывает процедуру «конецчтения», которая уменьшает число читателей на единицу. В конце концов, в результате многократного выполнения этой процедуры количество процессов-читателей становится равным нулю; в этот момент вырабатывается сигнал «писатьразрешается», так что ожидающий процесс-писатель получает возможность продолжить свою работу.

Когда процессу-писателю нужно произвести запись, он вызывает процедуру монитора под названием «началозаписи». Процесс-писатель должен иметь совершенно монополярный доступ к базе данных, поэтому, если в настоящий момент уже есть работающие процессы-читатели или какой-либо активизированный процесс-писатель, данному писателю придется подождать, пока не будет установлено истинное значение условия «писатьразрешается». Когда

писатель получает возможность продолжить работу, устанавливается истинное значение логической переменной «ктотопишет». Тем самым для всех остальных процессов-писателей и читателей доступ к базе данных или ее частям блокируется.

Когда процесс-писатель заканчивает свою работу, он устанавливает для переменной «ктотопишет» ложное значение, тем самым, открывая вход в монитор для других процессов. Затем он должен выдать очередному ожидающему процессу сигнал о том, что можно продолжить работу. Должен ли он при этом отдавать предпочтение ожидающему процессу-читателю или ожидающему процессу-писателю? Если он отдаст предпочтение ожидающему писателю, то может возникнуть такая ситуация, когда постоянный поток приходящих писателей приведет к бесконечному откладыванию ожидающих процессов-читателей. Поэтому писатель, завершивший свою работу, прежде всего, проверяет, нет ли ожидающего читателя. Если есть, то выдается сигнал «читатьразрешается», так что этот ожидающий читатель получает возможность продолжить работу. Если ни одного ожидающего читателя нет, то выдается сигнал «писатьразрешается» и получает возможность продолжить выполнение ожидающий писатель.

5.6 Ада—язык параллельного программирования 80-х годов

Далее в этой главе обсуждаются вопросы параллельного программирования с использованием нового и перспективного языка Ада (Ad79a, Ad79b, Ad82). Большинство популярных в настоящее время языков программирования разрабатывалось с ориентацией на написание последовательных программ — в них не предусматриваются специальные средства, необходимые для написания параллельных программ.

Язык Ада предоставляет индивидуальному пользователю возможность создавать много отдельных цепочек (последовательностей) управления для выполнения различных функций, необходимых этому пользователю. Такой режим называется *многозадачным режимом*.

5.7 Мотивы для реализации многозадачного режима в языке Ада

В военных системах управления и контроля часто требуется контролировать и координировать многие параллельные действия и процессы. В качестве примеров систем, которые должны контролировать многие параллельные процессы и оперативно реагировать на непредвиденные изменения в ходе подобных процессов, можно привести системы слежения за положением военно-морских судов различных стран мира, системы управления воздушным движением и системы наблюдения за воздухом с целью своевременного обнаружения вражеских ракет. Написание эффективного программного обеспечения для компьютеров, входящих в состав этих систем, существенно упрощается, если использовать язык параллельного программирования. Именно по этой причине правительство Соединенных Штатов пошло на финансирование разработки языка Ада.

Параллельные программы обычно гораздо сложнее писать, отлаживать и проверять на корректность, чем последовательные. Однако существует много ситуаций, когда в действительности их создавать проще, поскольку они позволяют более естественно выразить взаимосвязи, присущие моделируемым системам с высокой степенью параллелизма.

Если иметь в виду однопроцессорные компьютеры, то здесь нет особых мотивов

к разработке и применению языков параллельного программирования, не учитывая, быть может, чисто научных интересов. Однако в случае мультипроцессорных машин параллельные программы могут выполняться гораздо более эффективно. Поэтому есть все основания полагать, что в ближайшем будущем интерес к языкам параллельного программирования резко возрастет.

5.8 Корректность параллельных программ

Говорят, что программа является *корректной*, если она удовлетворяет своей спецификации, т. е. делает то, что задумано. Ранее разработчики устанавливали корректность своих программ путем исчерпывающего тестирования. Однако в последние годы термин «корректность» приобрел новое значение, а именно появилось понятие программы, корректной в математическом смысле. Исследователи, работающие в области технологии программирования, сосредоточивают свое внимание на следующем вопросе: каким образом продемонстрировать, что программа корректна (еще даже не пытаясь ее выполнять или тестировать)? Установить, что параллельная программа корректна, в общем случае гораздо сложнее, чем установить корректность последовательной программы (La77, La79, Ow76, Pu81). Перед разработчиками языка Ада была поставлена задача создать язык, который позволял бы легче убеждать в корректности параллельных программ.

5.9 Рандеву в языке Ада 2)

Разработчики языка Ада для обеспечения взаимоисключения и синхронизации задач предпочли встроить в него механизм под названием *рандеву*. Чтобы в языке Ада состоялось рандеву, необходимы две задачи — *вызывающая* и *обслуживающая*. Вызывающая задача обращается к определенному входу обслуживающей задачи. Обслуживающая задача, чтобы принять вызов, когда она готова это сделать, выдает команду приема ACCEPT. Если вызывающая задача производит вызов входа, для которого обслуживающая пока еще не выдала команды приема, то вызывающей задаче приходится ждать. Если обслуживающая задача выдает команду ACCEPT для входа, к которому вызывающая задача пока еще не обратилась, то обслуживающая будет ждать (на этой команде) обращения вызывающей к данному входу.

Рандеву происходит в момент, когда вызов принимается. Вызывающая задача передает данные обслуживающей как параметры вызова. Эти данные обрабатываются при помощи операторов, входящих в тело команды ACCEPT. Результаты, если они имеются, возвращаются вызывающей задаче так же, как параметры вызова.

Вызывающая задача ждет, пока обслуживающая выполнит действия, предусмотренные в команде ACCEPT. Когда эти действия завершатся, параметры-результаты возвращаются вызывающей задаче, рандеву заканчивается и вызывающая и вызываемая задачи возобновляют независимую работу.

Одна интересная особенность механизма рандеву в языке Ада заключается в том, что вызывающая задача должна знать о существовании обслуживающей задачи и ее различных входов. В то же время обслуживающая задача принимает вызовы от любой вызывающей задачи, причем много вызывающих задач могут пытаться обращаться к одной обслуживающей. В этом смысле механизм рандеву является несимметричным.

В каждый конкретный момент времени возможность рандеву с данной обслуживающей задачей предоставляется только одной вызывающей задаче — причем взаимоисключение обеспечивается при помощи встроенных системных механизмов. Все остальные вызывающие задачи, одновременно пытающиеся

получить randevу, переводятся в режим ожидания. Синхронизация задач в процессе randevу осуществляется неявно. После завершения randevу, ожидающие своей очереди вызывающие задачи обслуживаются по принципу «первая пришедшая обслуживается первой».

5.10 Команда приема АСCEPT

На рис. 5.4 приведена написанная на языке Ада задача под названием КОНТРОЛЛЕРРЕСУРСА, предназначенная для управления доступом к одному совместно используемому ресурсу. Эта задача представляет собой бесконечный цикл с попеременным приемом вызовов по входам ВЗЯТЬУПРАВЛЕНИЕ и ОТДАТЬУПРАВЛЕНИЕ.

task КОНТРОЛЛЕРРЕСУРСА **is**

entry ВЗЯТЬУПРАВЛЕНИЕ;

entry ОТДАТЬУПРАВЛЕНИЕ;

end КОНТРОЛЛЕРРЕСУРСА;

task body КОНТРОЛЛЕРРЕСУРСА **is**

begin

loop

accept ВЗЯТЬУПРАВЛЕНИЕ;

accept ОТДАТЬУПРАВЛЕНИЕ;

end loop;

end КОНТРОЛЛЕРРЕСУРСА;

▪

▪

▪

КОНТРОЛЛЕРРЕСУРСА.ВЗЯТЬУПРАВЛЕНИЕ;

- использование данного ресурса

КОНТРОЛЛЕРРЕСУРСА.ОТДАТЬУПРАВЛЕНИЕ;

Рис. 5.4 Управление доступом к одному разделяемому ресурсу при помощи механизма randevу языка Ада.

Здесь задачи произвольно взаимодействуют, используя КОНТРОЛЛЕРРЕСУРСА и обеспечивая взаимоисключение. Если несколько задач одновременно вызовут вход ВЗЯТЬУПРАВЛЕНИЕ, то будет принят только один вызов, а остальным придется ждать в очереди «первый пришедший обслуживается первым».

Это по существу точно такой же механизм, как двоичный семафор. Если одна задача будет игнорировать данный протокол взаимодействия или неправильно использовать его, то взаимоисключение нельзя будет гарантировать. Например, некая «коварная» задача, желающая захватить управление данным ресурсом, может сделать это следующим образом:

-
-
-

КОНТРОЛЛЕРРЕСУРСА.ВЗЯТЬУПРАВЛЕНИЕ;

КОНТРОЛЛЕРРЕСУРСА.ОТДАТЬУПРАВЛЕНИЕ;

Тем самым задача получит доступ к ресурсу (если в этот момент нет другой задачи, уже ожидающей освобождения входа ВЗЯТЬ-УПР ДЕЛЕНИЕ).

5.11 Пример на языке Ада: пара «производитель — потребитель»

В качестве простого примера пары «производитель—потребитель» рассмотрим задачу-производитель, которая помещает образ 80-литерной карты в буфер, и задачу-потребитель, которая по одному извлекает литеры из этого буфера, пока буфер не окажется пустым. Производитель не может поместить очередную строку в буфер до тех пор, пока потребитель не обработает полностью все 80 литер строки; потребитель не может начать извлекать литеры из буфера до тех пор, пока в него не будет записана строка. Кроме того, после того как потребитель обработает все литеры одной строки, он должен ждать, чтобы производитель записал в буфер следующую строку. В противном случае могло бы случиться так, что потребитель обработает одну и ту же строку более одного раза. На рис. 5.5 (а) приведена задача, обеспечивающая синхронизацию задачи-производителя и задачи-потребителя, которые приведены на рис. 5.5(6).

5.12 Команда выбора SELECT

Вызовы входа не обязательно должны приниматься в некотором заранее оговоренном и жестком порядке. В языке Ада предусматривается команда выбора SELECT, позволяющая задачам принимать вызовы входа более гибким образом. Команда SELECT имеет следующую общую форму:

select

when УСЛОВИЕ1 => **accept** ВХОД1

последовательность операторов;

or when УСЛОВИЕ2 => **accept** ВХОД2

последовательность операторов;

or ...

else

последовательность операторов;

end select;

Команда SELECT выполняется следующим образом:

task ПРЕОБРАЗОВАНИЕОБРАЗАКАРТЫ **is**

type ОБРАЗКАРТЫ **is array** (1..80) **of character;**

entry ПОМЕСТИТЬ КАРТУ (КАРТА: **in** ОБРАЗКАРТЫ);

entry ПРОЧИТАТЬЛИТЕРУ (СЛЕДУЮЩАЯЛИТЕРА: **out character**);

end;

task body ПРЕОБРАЗОВАНИЕОБРАЗАКАРТЫ **is** БУФЕРКАРТЫ: ОБРАЗКАРТЫ;

begin

loop

accept ПОМЕСТИТЬ КАРТУ (КАРТА: **in** ОБРАЗКАРТЫ) **do** БУФЕРКАРТЫ := КАРТА;

end ПОМЕСТИТЬ КАРТУ;

for ПОЗИЦИЯ **in** 1..80 **loop**

accept ПРОЧИТАТЬЛИТЕРУ(СЛЕДУЮЩАЯЛИТЕРА: **out character**) **do**
СЛЕДУЮЩАЯЛИТЕРА := БУФЕРКАРТЫЦПОЗИ-ЦИЯ);

end ПРОЧИТАТЬЛИТЕРУ;

end loop;

end loop;

end;

(a)

task ПРОИЗВОДИТЕЛЬ;

task body ПРОИЗВОДИТЕЛЬ **is use** ПРЕОБРАЗОВАНИЕОБРАЗАКАРТЫ;

НОВАЯКАРТА: ОБРАЗАКАРТЫ;

begin

loop

- создать НОВАЯКАРТА

ПОМЕСТИТЬ КАРТУ (НОВАЯКАРТА);

end loop;

end;

task ПОТРЕБИТЕЛЬ;

task body ПОТРЕБИТЕЛЬ **is use** ПРЕОБРАЗОВАНИЕОБРАЗАКАРТЫ;

НОВАЯЛИТЕРА: **character**;

begin

loop

ПРОЧИТАТЬЛИТЕРУ (НОВАЯЛИТЕРА);

- обработать НОВАЯЛИТЕРА

end loop;

end;

(6)

Рис. 5.5 (a) Задача для синхронизации задачи-производителя и задачи-потребителя, (6) Задача-производитель и задача-потребитель.

- Каждое из условий (они называются также охраной или предохранителями (см. Di 75)) оценивается на истинность или ложность. Каждое условие, если оно оказывается истинным, позволяет следующую за ним последовательность операторов приема рассматривать как открытую. Если перед оператором приема условие опущено, этот вход считается всегда

открытым.

- Допускается существование нескольких открытых операторов приема. Из этих операторов несколько в свою очередь могут быть готовыми к randevу, т. е. соответствующие входы могут быть уже вызваны другими задачами. Один из этих входов, выбирается произвольно, т. е. язык Ада не регламентирует, какой именно, после чего происходит соответствующее randevу.
- Если в конструкции .SELECT имеется раздел else и нет вызовов входа для открытых операторов приема, то выполняется этот раздел. Если такого раздела нет, задача ожидает появления какого-нибудь вызова входа.
- Если открытых операторов приема нет, выполняется завершающий раздел else, а если такого раздела также нет, то вырабатывается сигнал ошибки оператора выбора SELECT-ERROR.

5.13 Пример на языке Ада: кольцевой буфер

На рис. 5.6 показано, каким образом может быть реализован кольцевой буфер на языке Ада. Оператор SELECT позволяет задаче кольцевого буфера обслуживать соответствующие вызовы. Охрана ТЕКУЩИЙБУФЕР<ЧИСЛОБУФЕРОВ разрешает принять вызов ЗАПИСАТЬ ПАКЕТ, когда имеется свободное место, а охрана ТЕКУЩИЙБУФЕР>0 позволяет принять вызов ПРОЧИТАТЬ-ПАКЕТ, когда буфер содержит данные.

5.14 Пример на языке Ада: читатели и писатели

В данном разделе представлены два варианта решения задачи читателей и писателей на языке Ада.

В первом варианте (рис. 5.7) ПИСАТЕЛЬ производит запись значения параметра РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ прежде, чем какой-либо ЧИТАТЕЛЬ сможет обратиться к ней. Тем самым гарантируется, что значение параметра РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ определяется до первого чтения.

После того как первый ПИСАТЕЛЬ закончит работу, задача первого варианта входит в бесконечный цикл. Если ни один ПИСАТЕЛЬ не активизирован, то возможно инициирование все новых и новых ЧИТАТЕЛЕЙ. Естественно, что это может привести к бесконечному откладыванию доступа для ПИСАТЕЛЕЙ. Очередной ПИСАТЕЛЬ может быть инициирован только в случае, когда нет активных ЧИТАТЕЛЕЙ, однако даже, при наличии ожидающего ПИСАТЕЛЯ случайный закон выбора, предусмотренный в операторе

task КОЛЬЦЕВОЙБУФЕР **is**

type ПАКЕТДАННЫХ **is array** (1..80) **of character**;

entry ПРОЧИТАТЬПАКЕТ (ПАКЕТ: **out** ПАКЕТДАННЫХ);

entry ЗАПИСАТЬПАКЕТ (ПАКЕТ: **in** ПАКЕТДАННЫХ);

end;

```

task body КОЛЬЦЕВОЙБУФЕР is ЧИСЛОБУФЕРОВ: constant integer := 20;

КОЛЬЦО: array (1..ЧИСЛОБУФЕРОВ) of ПАКЕТДАННЫХ;

ТЕКУЩИЙБУФЕР: ЦЕЛОЕ range 0..ЧИСЛОБУФЕРОВ := 0;

СЛЕДУЮЩИЙВВ, СЛЕДУЮЩИЙВЫВ: integerrange 1..ЧИСЛОБУФЕРОВ := 1;

begin

loop

select

when ТЕКУЩИЙБУФЕР < ЧИСЛОБУФЕРОВ =>

accept ЗАПИСАТЬПАКЕТ(ПАКЕТ: in ПАКЕТДАННЫХ) do

КОЛЬЦО(СЛЕДУЮЩИЙВВ) := ПАКЕТ;

end;

ТЕКУЩИЙБУФЕР := ТЕКУЩИЙБУФЕР + 1;

СЛЕДУЮЩИЙВВ := СЛЕДУЮЩИЙВВ mod ЧИСЛО БУФЕРОВ + 1;

or when ТЕКУЩИЙБУФЕР > 0 =>

accept ПРОЧИТАТЬПАКЕТ(ПАКЕТ: out ПАКЕТ-ДАННЫХ) do

ПАКЕТ := КОЛЬЦО(СЛЕДУЮЩИЙВЫВ);

end;

ТЕКУЩИЙБУФЕР := ТЕКУЩИЙБУФЕР — 1;

СЛЕДУЮЩИЙВЫВ := СЛЕДУЮЩИЙВЫВ mod ЧИСЛОБУФЕРОВ + 1;

end select;

end loop;

end КОЛЬЦЕВОЙБУФЕР;

```

Рис. 5.6 Реализация кольцевого буфера как задачи на языке Ада.

SELECT, может привести к тому, что первым продолжит свое выполнение ожидающий ЧИТАТЕЛЬ. Такой вариант решения задачи читателей и писателей обеспечивает взаимоисключение, однако не исключает бесконечного откладывания процессов.

Отметим, что ЧИТАТЕЛЬ — это процедура, а не вход. Благодаря этому много

задач могут читать одновременно.

Во втором варианте решения задачи читателей и писателей (рис. 5.8) также предусматривается, что ПИСАТЕЛЬ должен сделать минимум одну запись, чтобы ЧИТАТЕЛИ могли продолжить работу. Рассмотрим теперь основной бесконечный цикл задачи.

Сначала в первом охранном условии по значению функции

task ЧИТАТЕЛИИПИСАТЕЛИ **is**

procedure ЧИТАТЕЛЬ (ЧИТАЕМОЕЗНАЧЕНИЕ: **out integer**);

entry ПИСАТЕЛЬ (ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ: **in integer**);

end;

task body ЧИТАТЕЛИИПИСАТЕЛИ **is**

РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ: **integer**;

ЧИТАТЕЛИ: **integer** := 0;

entry НАЧАЛОЧТЕНИЯ;

entry КОНЕЦЧТЕНИЯ;

procedure ЧИТАТЕЛЬ (ЧИТАЕМОЕЗНАЧЕНИЕ: **out integer**) **is**

begin

НАЧАЛОЧТЕНИЯ;

ЧИТАЕМОЕЗНАЧЕНИЕ := РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ;

КОНЕЦЧТЕНИЯ;

end;

begin

accept ПИСАТЕЛЬ (ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ: **in integer**) **do**

РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ \wedge ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ;

end;

loop

select

accept НАЧАЛОЧТЕНИЯ; ЧИТАТЕЛИ := ЧИТАТЕЛИ + 1;


```

or

accept КОНЕЦЧТЕНИЯ; ЧИТАТЕЛИ := ЧИТАТЕЛИ — 1;

or

when ЧИТАТЕЛИ = 0 =>

accept ПИСАТЕЛЬ (ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ: in integer) do

    РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ := ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ;

end

end select;

end loop;

end ЧИТАТЕЛИИПИСАТЕЛИ;

```

Рис. 5.7 Решение задачи читателей и писателей на языке Ада (вариант первый).

ПИСАТЕЛЬ'COUNT1) проверяется, нет ли ожидающих очереди ПИСАТЕЛЕЙ. Если нет, то будет инициирован новый ЧИТАТЕЛЬ.

```

task ЧИТАТЕЛИИПИСАТЕЛИ is

    procedure ЧИТАТЕЛЬ (ЧИТАЕМОЕЗНАЧЕНИЕ: out integer);

    entry ПИСАТЕЛЬ (ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ: in integer);

    end;

task body ЧИТАТЕЛИИПИСАТЕЛИ is

    РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ: integer;

    ЧИТАТЕЛЕЙ: integer := 0;

    entry НАЧАЛОЧТЕНИЯ;

    entry КОНЕЦЧТЕНИЯ;

    procedure ЧИТАТЕЛЬ (ЧИТАЕМОЕЗНАЧЕНИЕ: out integer) is

        begin

            НАЧАЛОЧТЕНИЯ;

            ЧИТАЕМОЕЗНАЧЕНИЕ := РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ;

```

КОНЕЦЧТЕНИЯ;

end;

begin

accept ПИСАТЕЛЬ (ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ: **in integer**) **do**

РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ := ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ;

end;

loop

select

when ПИСАТЕЛЬ'count = 0 =>

accept НАЧАЛОЧТЕНИЯ;

ЧИТАТЕЛИ = ЧИТАТЕЛИ + 1;

or

accept КОНЕЦЧТЕНИЯ;

ЧИТАТЕЛИ := ЧИТАТЕЛИ — 1;

or

when ЧИТАТЕЛИ = 0 =>

accept ПИСАТЕЛЬ (ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ: **in integer**) **do**

РАЗДЕЛЯЕМАЯПЕРЕМЕННАЯ := ЗАПИСЫВАЕМОЕЗНАЧЕНИЕ;

end;

loop

select

accept НАЧАЛОЧТЕНИЯ;

ЧИТАТЕЛИ := ЧИТАТЕЛИ+1;

else

exit;

end select;

end loop;

end select;

end loop;

end ЧИТАТЕЛИИПИСАТЕЛИ;

Рис. 5.8 Решение задачи читателей и писателей на языке Ада (вариант второй).

И действительно, когда ожидающих ПИСАТЕЛЕЙ нет, можно продолжать инициировать ЧИТАТЕЛЕЙ.

Второй оператор принятия в основном цикле, оператор КОНЕЦ-ЧТЕНИЯ, не имеет охраны, так что он всегда открыт. Поэтому заканчивающий ЧИТАТЕЛЬ будет всегда приниматься, что естественно способствует уменьшению числа ЧИТАТЕЛЕЙ до нуля, с тем, чтобы со временем мог активизироваться очередной ПИСАТЕЛЬ.

Когда число ЧИТАТЕЛЕЙ, наконец, уменьшается до нуля и имеется ожидающий своей очереди ПИСАТЕЛЬ, этот ПИСАТЕЛЬ иницируется. Непосредственно после завершения работы этого ПИСАТЕЛЯ все ЧИТАТЕЛИ, пытавшиеся работать, получают возможность это сделать.

Фактически они получают randevу всегда в момент, когда не работает ПИСАТЕЛЬ, так что взаимoisключение гарантируется. Поэтому реальное различие между обоими вариантами решения заключается в том, каким образом организуется работа ЧИТАТЕЛЕЙ во втором варианте.

Ни один ЧИТАТЕЛЬ не может работать, когда работает ПИСАТЕЛЬ, поскольку randevу гарантирует взаимoisключение. Поэтому рассмотрим, что происходит в случае, когда нет ни одного работающего ПИСАТЕЛЯ.

Если нет ни одного ПИСАТЕЛЯ в режиме ожидания, то будет продолжаться инициирование ЧИТАТЕЛЕЙ. Как только появится ПИСАТЕЛЬ, ожидающий возможности продолжить работу, инициирование ЧИТАТЕЛЕЙ прекратится. Все вызовы входа КО-НЕЦЧТЕНИЯ будут приниматься, пока число ЧИТАТЕЛЕЙ не уменьшится до нуля. В этот момент охранный условие ЧИТАТЕ-ЛЕЙ=0 станет истинным и первый ожидающий ПИСАТЕЛЬ будет принят. Когда этот ПИСАТЕЛЬ закончит работу, все ЧИТАТЕЛИ, пришедшие с момента начала выполнения ПИСАТЕЛЯ, иницируются во внутреннем цикле. Такое групповое инициирование ЧИТАТЕЛЕЙ прекратится, как только при очередном проходе цикла и выполнении оператора приема НАЧАЛОЧТЕНИЯ не окажется ни одного ЧИТАТЕЛЯ, готового к работе. При этом произойдет переход на завершающий раздел else и выход из цикла.

Второй вариант решения не свободен от недостатков. Во внутреннем цикле может сложиться ситуация, когда ожидающие ПИСАТЕЛИ будут бесконечно откладываться из-за быстрого наплыва ЧИТАТЕЛЕЙ. Когда ЧИТАТЕЛИ приходят быстрее, чем цикл может принять их, работа ПИСАТЕЛЕЙ задерживается. При помощи языка Ада можно разработать более качественное решение проблемы ЧИТАТЕЛЕЙ и ПИСАТЕЛЕЙ. Оно оставляется для упражнений.

Заключение

Монитор — это конструкция параллелизма, которая содержит как данные, так и процедуры, необходимые для управления распределением общего ресурса или группы общих ресурсов.

Информация спрятана внутри монитора, процессы, обращающиеся к монитору,

не знают, какие данные находятся внутри монитора, и не получают к ним доступа.

Взаимоисключение процессов жестко обеспечивается на границе монитора, в каждый конкретный момент времени войти в монитор разрешается только одному процессу.

Если процесс, находящийся внутри монитора, не может продолжить свое выполнение, пока не будет удовлетворяться определенное условие, этот процесс вызывает команду `wait` с именем переменной-условия в качестве параметра и ждет вне монитора в очереди, пока другой процесс не выдаст команду `signal` с тем же именем.

Чтобы гарантировать, что процесс, уже находящийся в состоянии ожидания ресурса, со временем действительно получит доступ к ресурсу, монитор отдает ожидающему процессу предпочтение перед новым процессом, пытающимся войти в монитор.

Каждой отдельно взятой причине, по которой процесс может быть переведен в состояние ожидания, соответствует своя переменная-условие. Когда определяется такая переменная, тем самым устанавливается очередь. Процесс, выдающий команду ожидания `wait`, включается в эту очередь; процесс, выдающий команду сигнализации `signal`, позволяет ожидающему процессу выйти из очереди и войти в монитор.

Кольцевой буфер — это структура данных, широко применяемая в операционных системах для буферизации обменов между процессом-производителем и процессом-потребителем. В случае реализации кольцевого буфера при помощи монитора процесс-производитель, обнаруживший, что буфер заполнен, вызывает вход **`wait`** (буфернезаполнен); процесс-потребитель, обнаруживший, что буфер пуст, вызывает вход **`wait`** (буфернепуст). Производитель, помещающий данные в буфер, выдает **`signal`** (буфернепуст); потребитель, извлекающий данные из буфера, выдает **`signal`** (буфернезаполнен).

Рассмотрена задача читателей и писателей и представлено ее решение при помощи монитора. Процесс-писатель должен получать исключительный, монопольный доступ к базе данных; допускается одновременная работа любого числа процессов-читателей. Читатель, желающий получить данные, может сделать это только в том случае, если в настоящий момент нет работающего писателя и нет ожидающих писателей (последнее условие предотвращает бесконечное откладывание процессов-писателей из-за наплыва читателей). Когда читатель заканчивает работу, число активных читателей уменьшается на единицу; когда больше не остается активных читателей, последний закончивший работу читатель сигнализирует ожидающему писателю (если таковой имеется), что можно продолжить работу. Писатель, которому нужно произвести запись, вынужден ждать, если есть активные читатели или активный писатель; в противном случае он сразу активизируется. Когда писатель заканчивает работу, предпочтение отдается ожидающим читателям, а не ожидающим писателям; тем самым предотвращается бесконечное откладывание процессов-читателей из-за наплыва писателей.

Новый и перспективный язык Ада специально разрабатывался с ориентацией на параллельное программирование. Этот язык дает индивидуальному пользователю возможность устанавливать много отдельных цепочек (последовательностей) управления — организовывать так называемый многозадачный режим. Каждая цепочка управления в языке Ада называется задачей.

Необходимость разработки языков параллельного программирования, подобных языку Ада, обуславливается быстрым развитием мультипроцессорных архитектур и

вычислительных сетей. Говорят, что программа корректна, если она удовлетворяет своей спецификации; при разработке языка Ада ставилось целью упростить доказательство корректности программ.

В языке Ада для реализации взаимоисключения и синхронизации задач используется механизм под названием *рандеву*. Вызывающая задача может обратиться к какому-то входу другой вызываемой задачи; вызываемая, обслуживающая задача выдает команду приема ACCEPT, чтобы принять вызов по одному из своих входов. Когда вызов принимается, происходит рандеву. Вызывающая задача передает данные обслуживающей задаче как параметры вызова; результаты возвращаются вызывающей задаче также как параметры вызова. Пока обслуживающая задача осуществляет обработку данных, вызывающая задача находится в режиме ожидания.

В языке Ада предусматривается команда выбора SELECT, дающая возможность задачам гибко управлять порядком приема вызовов. Команда выбора обычно имеет следующую форму:

select

when УСЛОВИЕ1 => **accept** ВХОД1

последовательность операторов;

or when УСЛОВИЕ2 => **accept** ВХОД2

последовательность операторов;

or ...

else

последовательность операторов;

end select;

Производится проверка каждого из охранных условий. Если условие удовлетворяется, то следующая за ним последовательность команд считается открытой. Если перед командой приема условие не указывается, эта команда считается открытой всегда. Среди открытых и вызванных входов выбирается один случайным образом — и происходит соответствующее рандеву. Если ни один из открытых входов не готов к рандеву, выполняется раздел *else*. Если такого раздела нет, задача ожидает новые вызовы.

Терминология

бесконечное откладывание (indefinite postponement)

Бринк Хансен (Brinch Hansen)

вход монитора; (конкретная) процедура монитора (monitor entry)

вызывающая задача в языке Ада (caller task in Ada)

глобальные данные в мониторе (global data in a monitor)

Дейкстра (Dijkstra)

ждать (имяусловия) (wait (conditionvariablename))

задача (task)

кольцевой буфер (ring buffer)

команда выбора SELECT в языке Ада (select statement in Ada)

команда приема АСCEPT в языке Ада (accept statement in Ada)

корректность (программы) (correctness)

локальные данные в мониторе (local data in a monitor)

многозадачный режим (multitasking)

монитор (monitor)

обслуживающая задача в языке Ада (server task in Ada)

открытая команда приема в языке Ада (open accept statement in Ada)

отношение «производитель-потребитель» (producer-consumer relationship)

охрана, предохранители в языке Ада (guards in Ada)

переменная-условие (condition variable)

процесс деспулинга (despooler process)

процесс спулинга (spooler process)

рандеву (rendezvous)

сигнал (имяусловия) (signal (conditionvariablename))

спрятанность информации (защита внутренней информационной структуры) (information hiding)

Хoop (Hoare)

цепочка (последовательность) управления (thread of control)

читатели и писатели (readers and writers)

язык Ада (Ada)

язык параллельного программирования (concurrent programming language)

Упражнения

5.1 Укажите сходства и различия в применении мониторов и операций над семафорами.

5.2 Когда процесс, вызвавший монитор, возвращает ему ресурс, монитор при выделении этого ресурса, отдает приоритет ожидающему процессу, а не новому процессу, выдавшему запрос. Почему?

5.3 Чем переменные-условия отличаются от обычных переменных? Имеет ли смысл инициализация переменных-условий?

5.4 В данной книге неоднократно указывается, что не следует делать никаких предположений об относительных скоростях асинхронных параллельных процессов. Почему?

5.5 Какие факторы, по вашему мнению, должны определять решение разработчика о том, сколько позиций будет в массиве, представляющем кольцевой буфер?

5.6 Укажите сходства и различия многозадачного и мультипрограммного режимов.

5.7 Почему существенно сложнее тестировать, отлаживать и доказывать корректность параллельных программ по сравнению с последовательными?

5.8 Объясните, каким образом механизм рандеву, предусмотренный в языке Ада, обеспечивает синхронизацию задач, взаимодействие между задачами и взаимоисключение.

5.9 Почему говорят, что механизм рандеву в языке Ада является асимметричным?

5.10 Для пары «производитель—потребитель», описанной в разд. 5.11, объясните, почему синхронизация задачи-производителя и задачи-потребителя осуществляется при помощи третьей задачи.

5.11 Приведите общую форму команды выбора SELECT языка Ада. Объясните, каким образом выполняются такие команды.

5.12 Изучив литературу по языку Ада, предложите! более удачное решение задачи читателей и писателей, чем представлено в данной книге.

5.13 Философы за обеденным столом. Одна из наиболее интересных задач, предложенных Дейкстрой,— это задача об обедающих философях (D171). Она иллюстрирует многие нюансы, присущие параллельному программированию. Задача формулируется следующим образом.

Пять философов сидят за круглым столом. Каждый из этих пяти философов ведет простую жизнь — некоторое время они предаются размышлениям, а некоторое время едят спагетти. Перед каждым философом находится блюдо спагетти, которое постоянно пополняет специальный слуга. На столе лежат ровно пять вилок, по одной между каждыми двумя философами-соседями. Чтобы есть спагетти (с полным соблюдением правил хорошего тона), философ должен использовать обе вилки (одновременно), которые лежат рядом с его тарелкой.

Ваша задача заключается в том, чтобы разработать параллельную программу (с монитором), моделирующую поведение философов. В вашей программе должна

исключаться возможность тупиков и бесконечного откладывания, в противном случае один или несколько философов вскоре погибнут голодной смертью. Ваша программа должна естественно обеспечивать взаимоисключение — два философа не могут одновременно пользоваться одной и той же вилок.

Типичный философ ведет себя следующим образом:

procedure типичныйфилософ;

begin

while истина **do**

begin

мыслить;

есть

end

end;

Прокомментируйте каждый из следующих программных фрагментов, реализующих поведение типичного философа.

а) **procedure** типичныйфилософ;

begin

while истина **do**

begin

мыслитьнекотсроевремя;

взятьлевуювилку;

взятьправуювилку;

естьнекотороевремя;

положитьлевуювилку;

положитьправуювилку

end

end;

б) **procedure** типичныйфилософ;

begin

while истина **do**

begin

мыслить некоторое время;

взять обе вилки сразу;

есть некоторое время;

положить обе вилки сразу

end

end;

в) **procedure** типичный философ

begin

while истина **do**

begin

мыслить некоторое время;

repeat

взять левую вилку;

if правой вилки нет

then положить левую вилку

else взять правую вилку

until в руках обе вилки;

есть некоторое время;

положить левую вилку;

положить правую вилку

end

end;

Глава 6

Тупики

Оттяжки и проволочки — самая ужасная форма отказа.

К. Норткот Паркинсон

Разорвите цепи объятий — и гуляйте и пляшите, пока можете.

Вистан Хью Оден

Ссора быстро прекращается, когда один из участников покидает поле боя,— не может быть столкновения, если нет двух спорящих сторон.

□

Сенека

6.1 Введение

6.2 Примеры тупиков

6.2.1 Транспортная пробка как пример тупика

6.2.2 Простой пример тупика при распределении ресурсов

6.2.3 Тупики в системах спулинга

6.3 Аналогичная проблема — бесконечное откладывание

6.4 Концепции ресурсов

6.5 Четыре необходимых условия для возникновения тупика

6.6 Основные направления исследований по проблеме тупиков

6.7 Предотвращение тупиков

6.7.1 Нарушение условия «ожидания дополнительных ресурсов»

6.7.2 Нарушение условия неперераспределяемости

6.7.3 Нарушение условия «кругового ожидания»

6.8 Предотвращение тупиков и алгоритм банкира

6.8.1 Алгоритм банкира, предложенный Дейкстрой

6.8.2 Пример надежного состояния

6.8.3 Пример ненадежного состояния

6.8.4 Пример перехода из надежного состояния в ненадежное

6.8.5 Распределение ресурсов согласно алгоритму банкира

6.8.6 Недостатки алгоритма банкира

6.9 Обнаружение тупиков

6.9.1 Графы распределения ресурсов

6.9.2 Приведение графов распределения ресурсов

6.10 Восстановление после тупиков

6.11 Тупики как критический фактор для будущих систем

6.1 Введение

Говорят, что в мультипрограммной системе процесс находится в состоянии *тупика*, *дедлока*, или *клинча*, если он ожидает некоторого события, которое никогда не произойдет. Системная тупиковая ситуация, или «зависание» системы — это ситуация, когда один или более процессов оказываются в состоянии тупика.

В мультипрограммных вычислительных машинах одной из главных функций операционной системы является распределение ресурсов. Когда ресурсы разделяются между многими пользователями, каждому из которых предоставляется право исключительного управления выделенными ему конкретными ресурсами, вполне возможно возникновение тупиков, из-за которых процессы некоторых пользователей никогда не смогут завершиться.

В настоящей главе обсуждается проблема тупиков и приводятся основные результаты научных исследований, посвященных вопросам *предотвращения*, *обхода*, *обнаружения* тупиков и вопросам *восстановления* после них. Рассматривается также тесно связанная проблема *бесконечного откладывания*, когда процесс, даже не находящийся в состоянии тупика, ожидает события, которое может никогда не произойти из-за «необъективных» принципов, заложенных в системе планирования ресурсов.

Рассматриваются возможные компромиссные решения с точки зрения накладных затрат на включение средств борьбы с тупиками и ожидаемых от этого выгод. В некоторых случаях цена, которую приходится платить за то, чтобы сделать систему свободной от тупиков, слишком высока. В других случаях, например в системах управления процессами реального времени, просто нет иного выбора, поскольку возникновение тупика может привести к катастрофическим последствиям.

6.2 Примеры тупиков

По-видимому, простейший способ создания тупика проиллюстрировал Холт (Ho72) в программе, написанной на языке PL/I и предназначенной для того, чтобы привести процесс к тупику при работе под управлением операционной системы OS/360.

```
REVENGE: PROCEDURE OPTIONS(MAINJASK);
```

```
WAIT(EVENT);
```

```
END REVENGE;
```

Процесс, связанный с приведенной программой, будет все время ждать, чтобы произошло событие EVENT; однако в этой программе не предусмотрена сигнализация о наступлении ожидаемого события. Системе придется обнаруживать, что данный процесс «завис», а затем аннулировать задание, чтобы выйти из тупиковой ситуации. Тупики подобного типа обнаруживать исключительно сложно.

6.2.1 Транспортная пробка как пример тупика

На рис. 6.1 показана транспортная пробка, которая периодически возникает в наших городах. Ряд автомобилей пытаются проехать через перегруженный транспортом участок города, однако движение оказывается полностью парализованным. Здесь образовался абсолютный затор, так что необходимо вмешательство полиции, чтобы постепенно и осторожно подать автомобили назад из зоны затора и тем самым ликвидировать «пробку». В конце концов

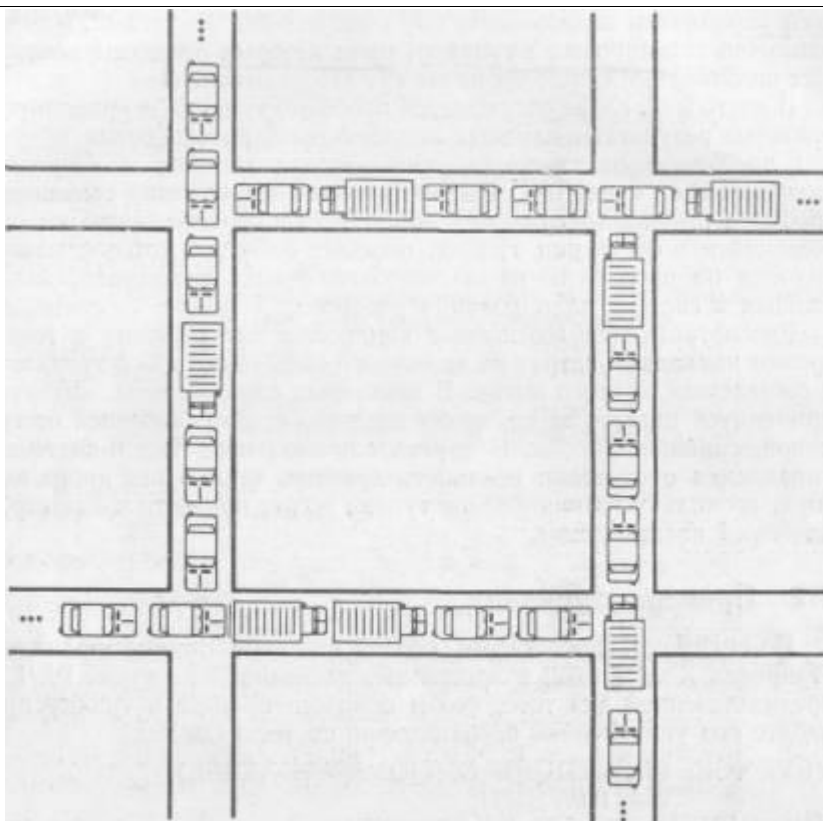


Рис. 6.1 Тупиковая ситуация—транспортная пробка.

нормальное движение машин восстанавливается, однако подобные ситуации связаны с большими неприятностями, затратами нервных усилий и значительными потерями времени.

6.2.2 Простой пример тупика при распределении ресурсов

В операционных системах тупики в большинстве случаев возникают в результате обычной конкуренции за обладание выделяемыми, или закрепляемыми ресурсами (т. е. ресурсами, которые в каждый момент времени отводятся только одному пользователю и которые поэтому иногда называются *ресурсами последовательного использования*). На рис. 6.2 показан простой пример тупика подобного рода. На этом *графе распределения ресурсов* представлены два процесса (в виде прямоугольников) и два ресурса (в виде окружностей). Стрелка, направленная от ресурса к процессу, показывает,

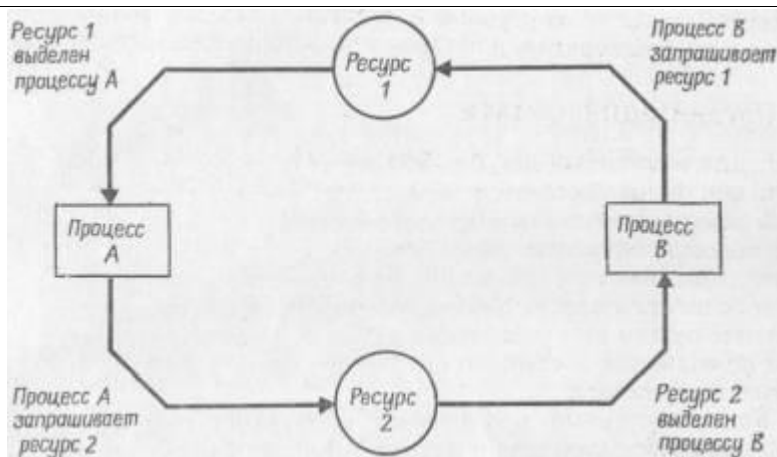


Рис. 6.2 Простая тупиковая ситуация. Данная система оказалась в тупиковой ситуации, поскольку каждый процесс удерживает ресурс, запрашиваемый другим процессом, причем ни один из процессов не хочет освободить принадлежащий ему ресурс.

что данный ресурс принадлежит или был выделен данному процессу. Стрелка, направленная от процесса к ресурсу, показывает, что данный процесс требует, но пока еще не получил в свое распоряжение данный ресурс. На рисунке изображена система в состоянии тупика; процесс А удерживает в своем распоряжении ресурс 1, а для продолжения выполнения ему необходим ресурс 2. Процесс В удерживает ресурс 2, а для продолжения работы ему нужен ресурс 1. Каждый процесс ждет, чтобы другой процесс освободил нужный ему ресурс, причем каждый не освобождает свой ресурс до тех пор, пока другой не освободит свой ресурс, и т. д. Такое состояние кругового ожидания характерно для систем в тупиковом состоянии.

6.2.3 Тупики в системах спулинга

Системы спулинга часто оказываются подвержены тупикам. Режим спулинга (ввода-вывода с буферизацией) применяется для повышения производительности системы путем изолирования программы от такого низкоскоростного периферийного оборудования, как устройства ввода данных с перфокарт и принтеры. Если, например, программе, выдающей строки данных на принтер, приходится ждать распечатки каждой строки перед передачей следующей строки, то такая программа будет выполняться медленно. Чтобы повысить скорость выполнения программы, строки данных, предназначенные для печати, вначале записываются на более высокоскоростное устройство, например дисковый накопитель, где они временно хранятся до момента распечатки. В некоторых системах спулинга программа должна сформировать все выходные данные — только после этого начинается реальная распечатка. В связи с этим несколько незавершенных заданий, формирующих строки данных и записывающих их в файл спулинга для печати, могут оказаться в тупиковой ситуации, если предусмотренная емкость буфера будет заполнена до того, как завершится выполнение какого-либо задания. Для восстановления, или выхода из подобной тупиковой ситуации, мог бы потребоваться перезапуск, рестарт системы с потерей всей работы, выполненной до этого момента. Если система попадает в тупиковую ситуацию таким образом, что у оператора ЭВМ остаются возможности управления, то в качестве менее радикального способа восстановления работоспособности можно предложить уничтожение одного или нескольких заданий, пока у остающихся заданий не окажется достаточно свободного места в буфере, чтобы они могли завершиться.

Когда системный программист производит генерацию операционной системы, он задает размер буферных файлов для спулинга. Один из способов уменьшить вероятность тупиков при спулинге заключается в том, чтобы предусмотреть значительно больше места для файлов спулинга, чем потребуется согласно предварительной оценке. Подобное решение не всегда осуществимо, если память дефицитна. Более распространенное решение состоит в том, что для процессов входного спулинга устанавливаются ограничения, с тем чтобы они не могли принимать дополнительные задания, когда файлы спулинга начинают приближаться к некоторому порогу насыщения, например, оказываются заполненными на 75 процентов. Такой подход может привести к некоторому снижению производительности системы, однако это — та цена, которую приходится платить за уменьшение вероятности тупика.

Современные системы являются в этом смысле гораздо более совершенными. Они могут позволять начинать распечатку до того, как завершится очередное задание, с тем чтобы полный или почти полный файл спулинга опустошался полностью или частично уже в процессе выполнения задания. Во многих системах предусматривается динамическое распределение буферной памяти, так что, если отведенного места в памяти оказывается мало, для файлов спулинга выделяется дополнительная память. Преимущества спулинга в общем случае оказываются гораздо более весомыми, чем те проблемы, о которых шла речь выше, однако мы хотели бы, чтобы читатель имел представление о потенциальных сложностях проектирования операционных систем.

6.3 Аналогичная проблема — бесконечное откладывание

В системе, где процессам приходится ждать, пока она принимает решения по распределению ресурсов и планированию, возможно возникновение ситуации, при которой предоставление процессора некоторому процессу будет откладываться на неопределенно долгий срок, в то время как система будет уделять внимание другим процессам. Такая ситуация, называемая бесконечным откладыванием, часто может приводить к столь же неприятным последствиям, как и тупик.

Бесконечное откладывание процесса может происходить из-за «дискриминационной» политики планировщика ресурсов системы. Когда ресурсы распределяются по приоритетному принципу, может случиться, что данный процесс будет бесконечно долго ожидать выделения нужного ему ресурса, поскольку будут постоянно приходить процессы с более высокими приоритетами. Ожидание — неперенный факт нашей жизни и безусловно важный аспект внутренней работы вычислительных машин. При разработке операционных систем необходимо предусматривать справедливое, а также эффективное управление процессами, находящимися в состоянии ожидания. В некоторых системах бесконечное откладывание предотвращается благодаря тому, что приоритет процесса увеличивается по мере того, как он ожидает выделения нужного ему ресурса. Это называется *старением* процесса. В конце концов, ожидающий процесс получит более высокое значение приоритета, чем у всех приходящих процессов, и будет обслужен.

Для описания и анализа систем, которые имеют дело с управлением ожидающими объектами, создан формальный аппарат — математическая теория очередей. Основы теории очередей представлены в гл. 15 «Аналитическое моделирование».

6.4 Концепции ресурсов

Операционная система выполняет по преимуществу функции администратора ресурсов. Она отвечает за распределение обширных наборов ресурсов различных типов. Отчасти именно благодаря такому разнообразию типов ресурсов разработка операционных систем является столь интересной темой. Рассмотрим ресурсы, которые являются «оперативно перераспределяемыми»¹⁾, такие, как центральный процессор и основная память; программа, занимающая в данный момент некоторую область основной памяти, может быть вытеснена или передана другой программе. Программа пользователя, которая выдала запрос на ввод-вывод, практически не может работать с основной памятью в течение длительного периода ожидания завершения данной операции ввода-вывода. В вычислительной машине самым динамичным ресурсом является, по-видимому, центральный процессор. Центральный процессор должен работать в режиме быстрого переключения (мультиплексирования), обслуживая большое число конкурирующих процессов, чтобы все эти процессы могли продвигаться с приемлемой скоростью. Если конкретный процесс достигает точки, когда он не может эффективно использовать центральный процессор (как это бывает в течение длительного периода ожидания завершения операций ввода-вывода), право управления центральным процессором отбирается у этого процесса и предоставляется другому процессу. *Таким образом, организация динамического переключения ресурсов является исключительно важным фактором для обеспечения эффективной работы мультипрограммных вычислительных систем.*

Определенные виды ресурсов являются оперативно неперераспределяемыми в том смысле, что их нельзя произвольно отбирать у процессов, за которыми они закреплены, например, лентопротяжные устройства, как правило, выделяются конкретным процессам на периоды от нескольких минут до нескольких часов. Лентопротяжный механизм, закрепленный за одним процессом, нельзя просто отобрать у этого процесса и передать другому.

Некоторые виды ресурсов допускают разделение между несколькими процессами, а некоторые предусматривают лишь монопольное использование индивидуальными процессами. Накопители на магнитных дисках иногда закрепляются за индивидуальными процессами, однако чаще содержат файлы, принадлежащие многим процессам. Основная память и центральный процессор коллективно используются многими процессами.

Данные и программы — это также ресурсы, которые требуют соответствующей организации управления и распределения. Так, в мультипрограммных системах многим пользователям может одновременно понадобиться прибегнуть к услугам программы-редактора. Однако иметь собственную копию редактора в основной памяти для каждого пользователя было бы неэкономично. Поэтому в память записывается одна копия кода подобной программы и несколько копий соответствующих данных, по одной для каждого пользователя. Поскольку с таким кодом могут одновременно работать многие пользователи, он не должен изменяться. Код программы, который не может изменяться во время выполнения, называется *реентерабельным*, или *многократно входимым*. Код, который может изменяться, но предусматривает повторную инициализацию при каждом выполнении, называется кодом *последовательного (многократного) использования*. С реентерабельным кодом могут коллективно работать несколько процессов одновременно, а с кодом последовательного использования — только один процесс в каждый конкретный момент времени.

Когда речь идет о конкретных разделяемых ресурсах, мы должны четко представлять себе, могут ли они реально использоваться несколькими процессами одновременно или несколькими процессами, но только одним из них в каждый момент времени. Именно с ресурсами второго вида чаще всего оказывается связанным возникновение тупиков.

6.5 Четыре необходимых условия возникновения тупика

Коффман, Элфик и Шощани (Co71) сформулировали следующие четыре необходимых условия наличия тупика.

- Процессы требуют предоставления им права монопольного управления ресурсами, которые им выделяются (*условие взаимоисключения*) .
- Процессы удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (*условие ожидания ресурсов*).
- Ресурсы нельзя отобрать у процессов, удерживающих их, пока эти ресурсы не будут использованы для завершения работы (*условие неперераспределяемости*).
- Существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся следующему процессу цепи (*условие кругового ожидания*).

6.6 Основные направления исследований по проблеме тупиков

В связи с проблемой тупиков были проведены одни из наиболее интересных исследований в области информатики и операционных систем. Результаты этих исследований оказались весьма эффективными в том смысле, что позволили разработать четкие методы решения многих распространенных проблем. В исследованиях по проблеме тупиков можно выделить следующие четыре основных направления:

- предотвращение тупиков;
- обход тупиков;
- обнаружение тупиков;
- восстановление после тупиков.

При *предотвращении тупиков* целью является обеспечение условий, исключающих возможность возникновения тупиковых ситуаций. Такой подход является вполне корректным решением в том, что касается самого тупика, однако он часто приводит к нерациональному использованию ресурсов. Тем не менее различные методы предотвращения тупиков широко применяются в практике разработчиков.

Цель средств обхода тупиков заключается в том, чтобы можно было предусматривать менее жесткие ограничения, чем в случае предотвращения тупиков, и тем самым обеспечить лучшее использование ресурсов. При наличии средств обхода тупиков не требуется такой реализации системы, при которой опасность тупиковых ситуаций даже не возникает. Напротив, методы обхода тупиков учитывают подобную возможность, однако в случае увеличения вероятности конкретной тупиковой ситуации здесь принимаются меры по аккуратному обходу тупика. (См. обсуждение алгоритма банкира, который предложил Дейкстра, ниже в настоящей главе.)

Методы *обнаружения тупиков* применяются в системах, которые допускают возможность возникновения тупиковых ситуаций как следствие либо умышленных,

либо неумышленных действий программистов. Цель средств обнаружения тупиков — установить сам факт возникновения тупиковой ситуации, причем точно определить те процессы и ресурсы, которые оказались вовлеченными в данную тупиковую ситуацию. После того как все это сделано, данную тупиковую ситуацию в системе можно будет устранить.

Методы *восстановления* после тупиков применяются для устранения тупиковых ситуаций, с тем чтобы система могла продолжать работать, а процессы, попавшие в тупиковую ситуацию, могли завершиться с освобождением занимаемых ими ресурсов. Восстановление — это, мягко говоря, весьма серьезная проблема, так что в большинстве систем оно осуществляется путем полного выведения из решения одного или более процессов, попавших в тупиковую ситуацию. Затем выведенные процессы обычно перезапускаются с самого начала, так что почти вся или даже вся работа, проделанная этими процессами, теряется.

6.7 Предотвращение тупиков

До сих пор разработчики систем при решении проблемы тупиков чаще всего шли по пути исключения самих возможностей тупиков. В настоящем разделе рассматриваются различные методы предотвращения тупиков, а также оцениваются различные последствия их реализации как для пользователя, так и для систем, особенно с точки зрения эксплуатационных характеристик и эффективности работы.

Хавендер (Hv68) показал, что возникновение тупика невозможно, если нарушено хотя бы одно из указанных выше четырех необходимых условий. Он предложил следующую стратегию.

- Каждый процесс должен запрашивать все требуемые ему ресурсы сразу, причем не может начать выполняться до тех пор, пока все они не будут ему предоставлены.
- Если процесс, удерживающий определенные ресурсы, получает отказ в удовлетворении запроса на дополнительные ресурсы, этот процесс должен освободить свои первоначальные ресурсы и при необходимости запросить их снова вместе с дополнительными.
- Введение линейной упорядоченности по типам ресурсов для всех процессов; другими словами, если процессу выделены ресурсы данного типа, в дальнейшем он может запросить только ресурсы более далеких по порядку типов.

Отметим, что Хавендер предлагает три стратегических принципа, а не четыре. Каждый из указанных принципов, как мы увидим в следующих разделах, имеет целью нарушить какое-нибудь одно из необходимых условий существования тупика. Первое необходимое условие, а именно условие взаимоисключения, согласно которому процессы получают право на монопольное управление выделяемыми им ресурсами, мы не хотим нарушать, поскольку нам, в частности, нужно предусмотреть возможность работы с *закрепленными* ресурсами.

6.7.1 Нарушение условия «ожидания дополнительных ресурсов»

Первый стратегический принцип Хавендера требует, чтобы процесс сразу же запрашивал все ресурсы, которые ему понадобятся. Система должна предоставлять эти ресурсы по принципу «все или ничего». Если набор ресурсов, необходимый процессу, имеется, то система может предоставить все эти ресурсы данному процессу сразу же, так что он получит возможность продолжить работу. Если в текущий момент полного набора ресурсов, необходимых процессу, нет, этому процессу придется ждать, пока они не освободятся. Однако, когда процесс находится в состоянии ожидания, он не должен удерживать какие-либо ресурсы. Благодаря этому предотвращается возникновение условия «ожидания дополнительных ресурсов» и тупиковые ситуации просто невозможны.

На первый взгляд это звучит неплохо, однако подобный подход может привести к серьезному снижению эффективности использования ресурсов. Например, программа, которой в какой-то момент работы требуются 10 лентопротяжных устройств, должна запросить — и получить — все 10 устройств, прежде чем начать выполнение. Если все 10 устройств необходимы программе в течение всего времени выполнения, то это не приведет к серьезным потерям. Рассмотрим, однако, случай, когда программе для начала работы требуется только одно устройство (или, что еще хуже, ни одного), а оставшиеся устройства потребуются лишь через несколько часов. Таким образом, требование о том, что программа должна запросить

и получить все эти устройства, чтобы начать выполнение, на практике означает, что значительная часть ресурсов компьютера будет использоваться вхолостую в течение нескольких часов.

Один из подходов, к которому часто прибегают разработчики систем с целью улучшения использования ресурсов в подобных обстоятельствах, заключается в том, чтобы разделять программу на несколько программных шагов, выполняемых относительно независимо друг от друга. Благодаря этому можно осуществлять выделение ресурсов для каждого шага программы, а не для всего процесса. Это позволяет уменьшить непроизводительное использование ресурсов, однако связано с увеличением накладных расходов на этапе проектирования прикладных программ, а также на этапе их выполнения.

Такая стратегия существенно повышает вероятность бесконечного откладывания, поскольку не все требуемые ресурсы могут оказаться свободными в нужный момент. Вычислительная система должна позволить завершиться достаточно большому числу заданий и освободить их ресурсы, чтобы ожидающее задание смогло продолжить выполнение. В течение периода, когда требуемые ресурсы накапливаются, их нельзя выделять другим заданиям, так что эти ресурсы простаивают. В настоящее время существуют противоречивые мнения о том, кто должен платить за эти неиспользуемые ресурсы. Некоторые разработчики считают, что, поскольку ресурсы накапливаются для конкретного пользователя, этот пользователь должен платить за них, даже в течение периода простоя ресурсов. Однако другие разработчики считают, что это привело бы к нарушению принципа *предсказуемости платы за ресурсы*, если пользователь попытается выполнить свою работу в период пиковой загрузки машины, ему придется платить гораздо больше, чем в случае, когда машина относительно свободна.

6.7.2 Нарушение условия неперераспределяемости

Второй стратегический принцип Хавендера, рассматриваемый здесь независимо от других, предотвращает возникновение условия неперераспределяемости. Предположим, что система позволяет процессам, запрашивающим дополнительные ресурсы, удерживать за собой ранее выделенные ресурсы. Если у системы

достаточно свободных ресурсов, чтобы удовлетворять все запросы, тупиковая ситуация не возникнет. Посмотрим, однако, что произойдет, когда какой-то запрос удовлетворить не удастся. В этом случае один процесс удерживает ресурсы, которые могут потребоваться второму процессу для продолжения работы, а этот второй процесс может удерживать ресурсы, необходимые первому. Это и есть тупик.

Второй стратегический принцип Хавендера требует, чтобы процесс, имеющий в своем распоряжении некоторые ресурсы, если он получает отказ на запрос о выделении дополнительных ресурсов,

должен освобождать свои ресурсы и при необходимости запрашивать их снова вместе с дополнительными. Такая стратегия действительно ведет к нарушению условия неперераспределяемости, подобным образом можно забирать ресурсы у удерживающих их процессов до завершения этих процессов.

Но этот способ предотвращения тупиковых ситуаций также не свободен от недостатков. Если процесс в течение некоторого времени использует определенные ресурсы, а затем освобождает эти ресурсы, он может потерять всю работу, сделанную до данного момента. На первый взгляд такая цена может показаться слишком высокой, однако необходим реальный ответ на вопрос «насколько часто приходится платить такую цену?» Если такая ситуация встречается редко, то можно считать, что в нашем распоряжении имеется относительно недорогой способ предотвращения тупиков. Если, однако, такая ситуация встречается часто, то подобный способ обходится дорого, причем приводит к печальным результатам, особенно когда не удается вовремя завершить высокоприоритетные или срочные процессы.

Одним из серьезных недостатков такой стратегии является возможность бесконечного откладывания процессов. Выполнение процесса, который многократно запрашивает и освобождает одни и те же ресурсы, может откладываться на неопределенно долгий срок. В этом случае у системы может возникнуть необходимость вывести данный процесс из решения, с тем чтобы другие процессы получили возможность продолжения работы. Нельзя игнорировать и вероятный случай, когда процесс, откладываемый бесконечно, может оказаться не очень «заметным» для системы. В такой ситуации возможно поглощение процессом значительных вычислительных ресурсов и деградация производительности системы.

6.7.3 Нарушение условия «кругового ожидания»

Третий стратегический принцип Хавендера исключает круговое ожидание. Поскольку всем ресурсам присваиваются уникальные номера и поскольку процессы должны запрашивать ресурсы в порядке возрастания номеров, круговое ожидание возникнуть не „может“.

Такой стратегический принцип реализован в ряде операционных систем, однако это оказалось сопряжено с определенными трудностями.

- Поскольку запрос ресурсов осуществляется в порядке возрастания их номеров и поскольку номера ресурсов назначаются при установке машины и должны «жить» в течение длительных периодов времени (месяцев или даже лет), то в случае введения в машину новых типов ресурсов может возникнуть необходимость переработки существующих программ и систем.
- Очевидно, что назначаемые номера ресурсов должны отражать нормальный порядок, в котором большинство заданий фактически используют ресурсы.

Для заданий, выполнение которых соответствует этому порядку, можно ожидать высокой эффективности. Если, однако, заданию требуются ресурсы не в том порядке, который предполагает вычислительная система, то оно должно будет захватывать и удерживать ресурсы, быть может, достаточно долго еще до того, как они будут фактически использоваться. А это означает потерю эффективности.

- Одной из самых важных задач современных операционных систем является ^предоставление пользователю максимальных удобств для работы («дружественной» обстановки). Пользователи должны иметь возможность разрабатывать свои прикладные программы при минимальном «загрязнении своей окружающей среды» из-за ограничений, накладываемых недостаточно удачными аппаратными и программными средствами. Последовательный порядок заказа ресурсов, предложенный Хавендером, действительно исключает возможность возникновения кругового ожидания, однако безусловно отрицательно сказывается при этом на возможности пользователя свободно и легко писать прикладные программы.

6.8 Предотвращение тупиков и алгоритм банкира

Если даже необходимые условия возникновения тупиков не нарушены, то все же можно избежать тупиковой ситуации, если рационально распределять ресурсы, придерживаясь определенных правил. По-видимому, наиболее известным алгоритмом обхода тупиковых ситуаций является алгоритм банкира, который предложил Дейкстра (Di 65); алгоритм получил такое любопытное наименование потому, что он как бы имитирует действия банкира, который, располагая определенным источником капитала, выдает ссуды и принимает платежи. Ниже мы изложим этот алгоритм применительно к проблеме распределения ресурсов в операционных системах.

6.8.1 Алгоритм банкира, предложенный Дейкстрой

Когда при описании алгоритма банкира мы будем говорить о ресурсах, мы будем подразумевать ресурсы одного и того же типа, однако этот алгоритм можно легко распространить на пулы ресурсов нескольких различных типов. Рассмотрим, например, проблему распределения некоторого количества t идентичных лентопротяжных устройств.

Операционная система должна обеспечить распределение некоторого фиксированного числа t одинаковых лентопротяжных устройств между некоторым фиксированным числом пользователей i . Каждый пользователь заранее указывает максимальное число устройств, которые ему потребуются во время выполнения своей программы на машине. Операционная система примет запрос пользователя в случае, если максимальная потребность этого пользователя в лентопротяжных устройствах не превышает t .

Пользователь может занимать или освобождать устройства по одному. Возможно, что иногда пользователю придется ждать выделения дополнительного устройства, однако операционная система гарантирует, что ожидание будет конечным. Текущее число устройств, выделенных пользователю, никогда не превысит указанную максимальную потребность этого пользователя. Если операционная система в состоянии удовлетворить максимальную потребность пользователя в устройствах, то пользователь гарантирует, что эти устройства будут использованы

и возвращены операционной системе в течение конечного периода времени.

Текущее состояние вычислительной машины называется *надежным*, если операционная система может обеспечить всем текущим пользователям завершение их заданий в течение конечного времени. (Здесь опять-таки предполагается, что лентопротяжные устройства являются единственными ресурсами, которые запрашивают пользователи.) В противном случае текущее состояние системы называется ненадежным.

Предположим теперь, что работают n пользователей.

Пусть $l(i)$ представляет текущее количество лентопротяжных устройств, выделенных i пользователю. Если, например, пользователю 5 выделены четыре устройства, то $l(5)=4$. Пусть $t(i)$ — максимальная потребность пользователя i , так что если пользователь 3 имеет максимальную потребность в двух устройствах, то $t(3)=2$. Пусть $c(t)$ — текущая потребность пользователя, равная его максимальной потребности минус текущее число выделенных ему ресурсов. Если, например, пользователь 7 имеет максимальную потребность в шести лентопротяжных устройствах, а текущее количество выделенных ему устройств составляет четыре, то мы получаем

$$c(7) = t(7) - l(7) = 6 - 4 = 2.$$

В распоряжении операционной системы имеются t лентопротяжных устройств. Число устройств, остающихся для распределения, обозначим через a . Тогда a равно t минус суммарное число устройств, выделенных различным пользователям.

Алгоритм банкира, который предложил Дейкстра, говорит о том, что выделять лентопротяжные устройства пользователям можно только в случае, когда после очередного выделения устройств состояние системы остается надежным. Надежное состояние — это состояние, при котором общая ситуация с ресурсами такова, что все пользователи имеют возможность со временем завершить свою работу. Ненадежное состояние — это состояние, которое может со временем привести к тупику.

6.8.2 Пример надежного состояния

Предположим, что в системе имеются двенадцать одинаковых лентопротяжных устройств, причем эти накопители распределяются между тремя пользователями, как показывает состояние I.

Состояние I

	Текущее количество	Максимальная потребность
--	-----------------------	-----------------------------

	выделенных устройств		
Пользователь (1)	1		4
Пользователь (2)	4		6
Пользователь (3)	5		8
Резерв		2	

Это состояние «надежное», поскольку оно дает возможность всем трем пользователям закончить работу. Отметим, что в текущий момент пользователь (2) имеет четыре выделенных ему устройства и со временем потребует максимум шесть, т. е. два дополнительных устройства. В распоряжении системы имеются двенадцать устройств, из которых десять в настоящий момент в работе, а два — в резерве. Если эти два резервных устройства, имеющихся в наличии, выделить пользователю (2), удовлетворяя тем самым максимальную потребность этого пользователя, то он сможет продолжать работу до завершения. После завершения пользователь (2) освободит все шесть своих устройств так что система сможет выделить их пользователю (1) и пользователю (3). Пользователь (1) имеет одно устройство и со временем ему потребуется еще три. У пользователя (3) — пять устройств и со временем ему потребуется еще три. Если пользователь (2) возвращает шесть накопителей, то три из них можно выделить пользователю (1), который получит таким образом возможность закончить работу и затем вернуть четыре накопителя системе. После этого система может выделить три накопителя пользователю (3), который тем самым также получает возможность закончить работу. Таким образом, основной критерий надежного состояния — это существование последовательности действий, позволяющей всем пользователям закончить работу.

6.8.3 Пример ненадежного состояния

Предположим, что двенадцать лентопротяжных устройств, имеющихся в системе, распределены согласно состоянию II.

Состояние II

	Текущее количество выделенных устройств		Максимальная потребность
Пользователь (1)	8		10
Пользователь (2)	2		5
Пользователь (3)	1		3
Резерв		1	

Здесь из двенадцати устройств системы одиннадцать в настоящий момент находятся в работе и только одно остается в резерве. В подобной ситуации независимо от того, какой пользователь запросит это резервное устройство, мы не можем гарантировать, что все три пользователя закончат работу. И действительно, предположим, что пользователь (1) запрашивает и получает последнее оставшееся устройство. При этом тупиковая ситуация может возникнуть в трех случаях, когда каждому из трех процессов потребуется запросить по крайней мере одно дополнительное устройство, не возвратив некоторое количество устройств в общий пул ресурсов.

Здесь важно отметить, что термин «ненадежное состояние'!) не предполагает, что в данный момент существует или в какое-то время обязательно возникнет тупиковая ситуация. Он просто говорит о том, что в случае некоторой неблагоприятной последовательности событий система может зайти в тупик.

6.8.4 Пример перехода из надежного состояния в ненадежное

Если известно, что данное состояние надежно, это вовсе не означает, что все последующие состояния также будут надежными. Наш механизм распределения ресурсов должен тщательно анализировать все запросы на выделение ресурсов, прежде чем удовлетворять их. Рассмотрим, например, случай, когда система находится в текущем состоянии, показанном как состояние III.

Предположим теперь, что пользователь (3) запрашивает дополнительный ресурс. Если бы система удовлетворила этот запрос, она перешла бы в новое состояние, состояние IV.

Конечно, состояние IV не обязательно приведет к тупиковой ситуации. Если, однако, состояние III было надежным, то состояние IV стало ненадежным. Состояние IV характеризует систему,

Состояние III

	Текущее количество выделенных устройств		Максимальная потребность
Пользователь (1)	1		4
Пользователь (2)	4		6
Пользователь (3)	5		8
Резерв		2	

Состояние IV

	Текущее количество выделенных устройств		Максимальная потребность
Пользователь (1)	1		4
Пользователь (2)	4		6
Пользователь (3)	6		8
Резерв		1	

в которой нельзя гарантировать успешное завершение всех процессов пользователей. В резерве здесь остается только один ресурс, в то время как в наличии должно быть минимум два ресурса, чтобы либо пользователь (2), либо пользователь (3) могли завершить свою работу, возвратить занимаемые ими ресурсы системе и тем самым позволить остальным пользователям закончить выполнение.

6.8.5 Распределение ресурсов согласно алгоритму банкира

Сейчас уже должно быть ясно, каким образом осуществляется распределение ресурсов согласно алгоритму банкира, который предложил Дейкстра. Условия «взаимоисключения», «ожидания дополнительных ресурсов» и «неперераспределяемости» выполняются. Процессы могут претендовать на монопольное использование ресурсов, которые им требуются. Процессам реально разрешается удерживать за собой ресурсы, запрашивая и ожидая выделения дополнительных ресурсов, причем ресурсы нельзя отбирать у процесса, которому они выделены. Пользователи не ставят перед системой особенно сложных задач, запрашивая в каждый момент времени только один ресурс. Система может либо удовлетворить, либо отклонить каждый запрос. Если запрос отклоняется, пользователь удерживает за собой уже выделенные ему ресурсы и ждет определенный конечный период времени, пока этот запрос в конце концов не будет удовлетворен. Система удовлетворяет только те запросы, при которых ее состояние остается надежным. Запрос пользователя, приводящий к переходу системы в ненадежное состояние, откладывается до момента, когда его все же можно будет выполнить. Таким образом, поскольку система всегда поддерживается в надежном состоянии, рано или поздно (т. е. в течение конечного времени) все запросы можно будет удовлетворить и все пользователи смогут завершить свою работу.

6.8.6 Недостатки алгоритма банкира

Алгоритм банкира представляет для нас интерес потому, что он дает возможность распределять ресурсы таким образом, чтобы обходить тупиковые ситуации. Он позволяет продолжать выполнение таких процессов, которым в случае системы с предотвращением тупиков пришлось бы ждать. Однако у этого алгоритма имеется ряд серьезных недостатков, из-за которых разработчик системы может оказаться вынужденным выбрать другой подход к решению проблемы тупиков.

- Алгоритм банкира исходит из фиксированного количества распределяемых ресурсов. Поскольку устройства, представляющие ресурсы, зачастую требуют технического обслуживания либо из-за возникновения неисправностей, либо в целях профилактики, мы не можем считать, что количество ресурсов всегда остается фиксированным.
- Алгоритм требует, чтобы число работающих пользователей оставалось постоянным. Подобное требование также является нереалистичным. В современных мультипрограммных системах количество работающих пользователей все время меняется. Например, большая система с разделением времени вполне может обслуживать 100 или более пользователей одновременно. Однако текущее число обслуживаемых пользователей непрерывно меняется, быть может, очень часто, каждые несколько секунд.
- Алгоритм требует, чтобы банкир—распределитель ресурсов гарантированно удовлетворял все запросы за некоторый конечный период времени. Очевидно, что для реальных систем необходимы гораздо более конкретные гарантии.
- Аналогично, алгоритм требует, чтобы клиенты (т. е. задания или процессы) гарантированно «платили долги» (т. е. возвращали выделенные им ресурсы) в течение некоторого конечного времени. И опять-таки для реальных систем требуются гораздо более конкретные гарантии.
- Алгоритм требует, чтобы пользователи заранее указывали свои максимальные потребности в ресурсах. По мере того как распределение ресурсов становится все более динамичным, все труднее оценивать максимальные потребности пользователя. Вообще говоря, поскольку компьютеры становятся более «дружественными» по отношению к пользователю, все чаще встречаются пользователи, которые не имеют ни малейшего представления о том, какие ресурсы им потребуются.

6.9 Обнаружение тупиков

Обнаружение тупика — это установление факта, что возникла тупиковая ситуация, и определение процессов и ресурсов, вовлеченных в эту тупиковую ситуацию. Алгоритмы обнаружения тупиков, как правило, применяются в системах, где выполняются первые три необходимых условия возникновения тупиковой ситуации. Эти алгоритмы затем определяют, не создан ли режим кругового ожидания.

Применение алгоритмов обнаружения тупиков сопряжено с определенными дополнительными затратами машинного времени. Таким образом, здесь мы снова сталкиваемся с необходимостью прибегать к компромиссным решениям, столь распространенным в операционных системах. Будут ли накладные расходы, связанные с реализацией алгоритмов обнаружения тупиковых ситуаций, в достаточной степени оправданы потенциальными выгодами от локализации и устранения тупиков? В данный момент мы этот вопрос рассматривать не будем, а лучше сосредоточим свое внимание на описании алгоритмов, позволяющих обнаруживать тупики.

6.9.1 Графы распределения ресурсов

При рассмотрении задачи обнаружения тупиков применяется весьма распространенная нотация, согласно которой распределение ресурсов и запросы изображаются в виде направленного графа. Квадраты обозначают процессы, а большие кружки — классы идентичных ресурсов. Малые кружки, находящиеся

внутри больших, обозначают количество идентичных ресурсов каждого класса. Например, если в большом кружке с меткой R1 содержатся три малых кружка, это говорит о том, что система *имеет в наличии* три эквивалентных ресурса типа R1.

На рис. 6.3 показаны отношения, которые могут быть представлены на графе запросов и распределения ресурсов. На рис. 6.3(а) показано, что процесс P1 запрашивает ресурс типа R1. Стрелка от квадрата P1 доходит только до большого кружка — это говорит о том, что в текущий момент запрос от процесса на выделение данного ресурса находится в стадии рассмотрения.

На рис. 6.3(б) показано, что процессу P2 выделен ресурс типа R2 (один из двух идентичных ресурсов), — здесь стрелка прочерчена

от малого кружка, находящегося внутри большого кружка R2, до квадрата P2.

На рис. 6.3(в) показана ситуация, в некоторой степени приближающаяся к потенциальному тупику: процесс P3 запрашивает ресурс R3, выделенный процессу P4.

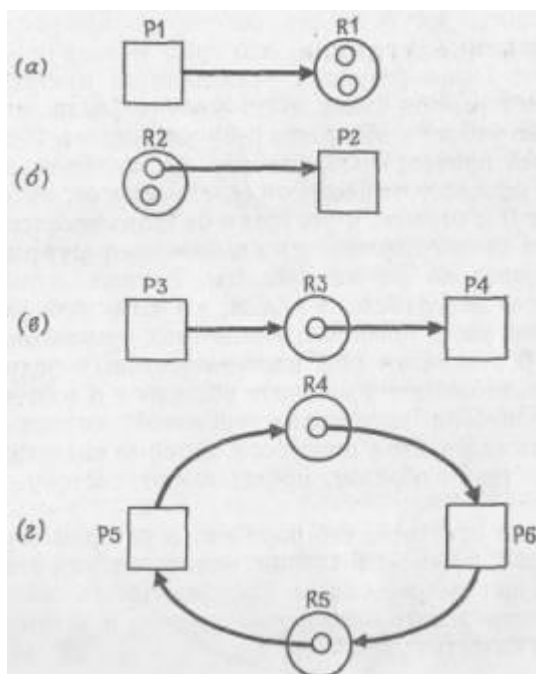


Рис. 6.3 Граф запросов и распределения ресурсов: .а) процесс P1 запрашивает ресурс типа R1;

б) ресурс типа R2 выделен процессу P2;

в) процесс P3 запрашивает ресурс R3, который выделен процессу P4;

г) процессу P5 выделен ресурс R5, необходимый процессу P6, которому выделен ресурс R4, необходимый процессу P5 («круговое ожидание»).

На рис. 6.3(г) показана небольшая система в тупиковой ситуации, когда процесс P5 запрашивает ресурс R4, который выделен процессу P6, который запрашивает ресурс R5, который выделен процессу P5, который в свою очередь запрашивает ресурс R4 — это пример «кругового ожидания», типичного для

системы в состоянии тупика.

Графы запросов и распределения ресурсов динамично меняются по мере того, как процессы запрашивают ресурсы, получают их в свое распоряжение, а через какое-то время возвращают операционной системе.

6.9.2 Приведение графов распределения ресурсов

Задача механизма обнаружения тупиков — определить, не возникла ли в системе тупиковая ситуация. Одним из способов обнаружения тупиков является приведение, или редукция графа,— это позволяет определять процессы, которые могут завершиться, и процессы, которые будут оставаться в тупиковой ситуации.

Если запросы ресурсов для некоторого процесса могут быть удовлетворены, то мы говорим, что граф можно *редуцировать на этот процесс*. Такая редукция эквивалентна изображению графа в том виде, который он будет иметь в случае, если данный процесс завершит свою работу и возвратит ресурсы системе. Редукция графа на конкретный процесс изображается исключением стрелок, идущих к этому процессу от ресурсов (т. е. ресурсов, выделенных данному процессу) и стрелок к ресурсам от этого процесса (т. е. текущих запросов данного процесса на выделение ресурсов). *Если граф можно редуцировать на все процессы, значит, тупиковой ситуации нет, а если этого сделать нельзя, то все «нередуцируемые» процессы образуют набор процессов, вовлеченных в тупиковую ситуацию.*

На рис. 6.4 показан ряд последовательных редукций графа, которые в конце концов позволяют убедиться в том, что для этого конкретного набора процессов тупиковой ситуации нет. На рис. 6.3(г) показана пара процессов, которые являются «нередуцируемыми» и, таким образом, представляют систему в тупиковой ситуации.

Здесь важно отметить, что порядок, в котором осуществляется редукция графа, не имеет значения: окончательный результат в любом случае будет тем же самым. Справедливость такого утверждения продемонстрировать относительно легко, и читатель может сам это сделать в качестве упражнения.

6.10 Восстановление после тупиков

Систему, оказавшуюся в тупике, необходимо вывести из него, нарушив одно или несколько необходимых условий его существования. При этом, как правило, несколько процессов потеряют частично или полностью уже проделанную ими работу. Однако это обойдется гораздо дешевле, чем оставлять систему в тупиковой ситуации. Сложность восстановления системы, т. е. ее вывода из тупика, обуславливается рядом факторов.

- Прежде всего, в первый момент вообще может быть неочевидно, что система попала в тупиковую ситуацию.
- В большинстве систем нет достаточно эффективных средств, позволяющих приостановить процесс на неопределенно долгое время, вывести его из системы и возобновить его выполнение впоследствии. В действительности некоторые процессы, например процессы реального времени, должны работать непрерывно и не допускают приостановки и последующего возобновления.

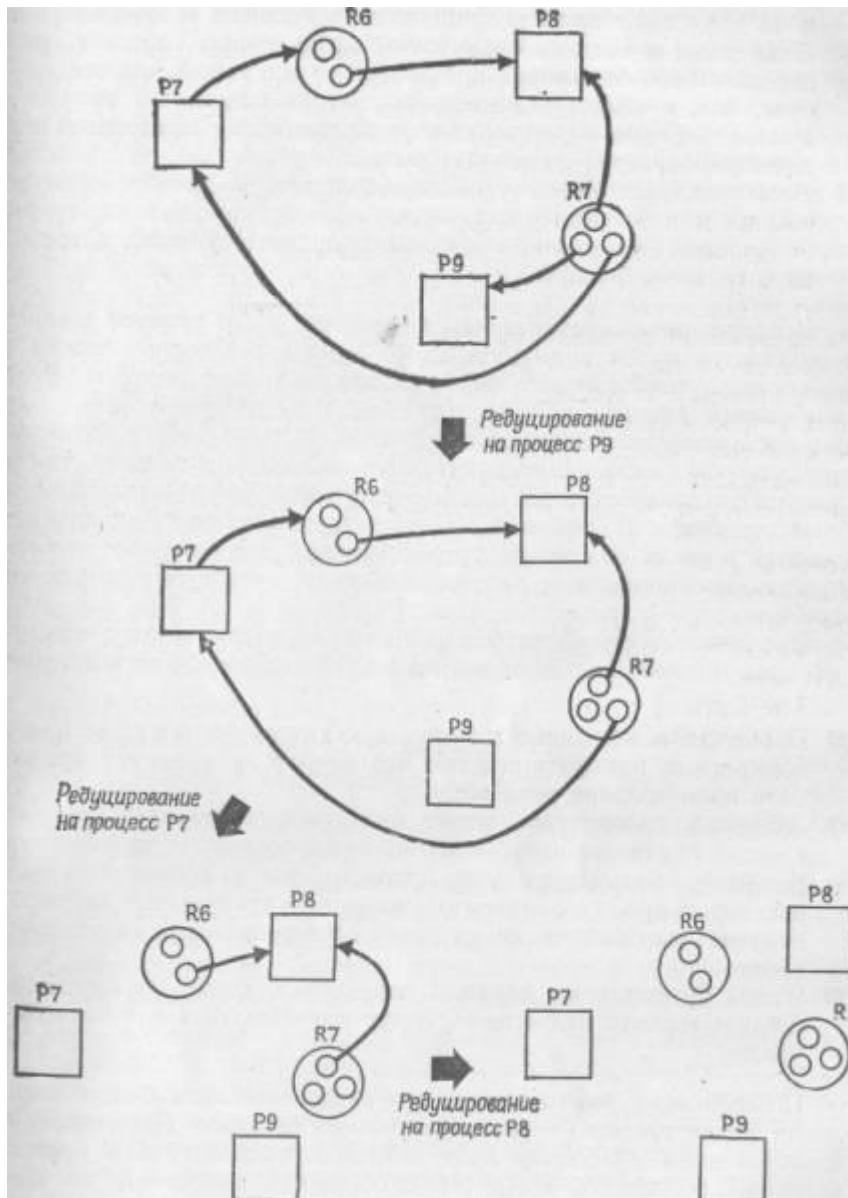


Рис. 6.4 Редукция графа.

- Если даже в системе существуют эффективные средства приостановки/возобновления процессов, то их использование требует, как правило, значительных затрат машинного времени, а также внимания высококвалифицированного оператора. А подобного оператора может и не быть.
- Восстановление после тупика небольших масштабов требует обычно и небольшого количества работы; крупный же тупик (с участием десятков или даже сотен процессов) может потребовать громадной работы.

В современных системах восстановление после тупиков обычно выполняется путем принудительного вывода некоторого процесса из системы, чтобы можно было использовать его ресурсы. Этот выведенный процесс обычно теряется, однако теперь остальные процессы получают возможность завершить свою работу. В некоторых случаях необходимо уничтожить несколько процессов, чтобы

освободить количество ресурсов, достаточное для завершения работы остающихся процессов. Здесь сам термин «восстановление» кажется в какой-то степени неправомерным. Это все равно что говорить о восстановлении работоспособности руки путем ампутации пальца.

Процессы могут выводиться из системы в соответствии с некоторой приоритетностью. Здесь мы также сталкиваемся с несколькими трудностями.

- Процессы, вовлеченные в тупиковую ситуацию, могут не иметь конкретных приоритетов, так что оператору придется принимать произвольное решение.
- Значения приоритетов могут оказаться неправильными или в какой-то степени нарушаться из-за определенных особых соображений, например в случае планирования по конечному сроку некоторый процесс с относительно низким приоритетом временно получает высокий приоритет ввиду приближающегося конечного срока.
- Чтобы оптимальным образом определить, какой из процессов следует вывести из системы, могут потребоваться значительные усилия.

По всей вероятности, самый целесообразный способ восстановления после тупиков — это эффективный механизм приостановки/ возобновления процессов. Этот механизм позволяет нам кратковременно переводить процессы в состояние ожидания, а затем активизировать ожидающие процессы, причем без потери результатов работы. Исследования в этой области имеют важное значение не только в связи с проблемой выхода из тупиковых ситуаций. Так, например, может возникнуть необходимость на какое-то время выключить систему, а затем запустить ее снова с той же точки без

потери промежуточных результатов. В подобном случае механизм приостановки/возобновления может оказаться весьма эффективным. Средства рестарта с контрольной точки, реализованные во многих системах, обеспечивают приостановку/возобновление вычислений с потерей результатов только после последней контрольной точки (от момента, когда запоминалось состояние системы). Однако в конструкции многих систем такой достаточно эффективный механизм контрольных точек с рестартом не предусмотрен. Поэтому разработчикам прикладных программ обычно приходится тратить усилия на включение контрольных точек с возможностью рестарта в свою программу, так что если речь не идет о программах, выполнение которых требует много часов машинного времени, подобный механизм применяется редко.

Тупики могут приводить к катастрофическим последствиям в некоторых системах реального времени. Так, система управления процессами реального времени, ответственная за нормальную работу нефтеперерабатывающего предприятия, просто обязана непрерывно функционировать, поскольку от этого зависит безопасность и нормальная деятельность предприятия. В подобных условиях малейшего риска оказаться в тупике быть не должно. Что же делать, если тупиковая ситуация все же возникнет? Очевидно, что ее необходимо мгновенно обнаруживать и устранять. Однако всегда ли это возможно? Таковы некоторые из проблем, не позволяющие спокойно спать разработчикам операционных систем.

6.11 Тупики как критический фактор для будущих систем

В современных вычислительных системах тупиковые ситуации обычно рассматриваются как неприятность ограниченного характера. В большинстве

систем реализуются основные методы предотвращения тупиков, предложенные Хавендером, причем эти методы кажутся вполне удовлетворительными.

Однако в будущих системах тупики станут в гораздо большей мере критическим фактором — по нескольким причинам.

- Эти системы будут в гораздо большей степени ориентированы на асинхронную параллельную работу, чем прежние системы с преобладанием последовательных режимов. Всеобщее распространение получают мультипроцессорные архитектуры, а параллельные вычисления займут доминирующее положение. Попросту говоря, в системах будет гораздо больше операций, выполняемых одновременно.
- В этих системах будет реализовано преимущественно динамическое распределение ресурсов. Процессы получают возможность свободно захватывать и освобождать ресурсы по мере необходимости. Пользователям не придется заблаговременно, до начала выполнения своих программ, достаточно точно оценивать свои потребности в ресурсах. В действительности с появлением «дружественных» интерфейсов будущего, о которых сейчас много говорят, большинство пользователей сможет практически не задумываться о том, какие ресурсы потребуются для выполнения их процессов.
- Среди разработчиков операционных систем растет тенденция рассматривать данные как ресурс, и в связи с этим количество ресурсов, которыми должны будут управлять операционные системы, резко увеличится.

Таким образом, в будущих вычислительных машинах ответственность за распределение ресурсов с недопущением тупиков будет ложиться на операционную систему. Если учесть, что вычислительные ресурсы становятся дешевле, то вполне целесообразно, чтобы компьютер и его операционная система приняли на себя подобные расширенные функции.

Заключение

Тупики (дедлоки) и близкая к ним проблема бесконечного откладывания — важные факторы, которые должны учитывать разработчики операционных систем. Один процесс может оказаться в тупиковой ситуации, если он будет ждать наступления события, которое никогда не произойдет. Два или более процессов могут попасть в тупик, при котором каждый процесс будет удерживать ресурсы, запрашиваемые другими процессами, в то время как самому ему требуются ресурсы, удерживаемые другими. Системы со спулингом оказываются чреваты тупиковыми ситуациями, когда выделенная область буферной памяти заполняется до того, как процессы завершат свою работу.

Динамически перераспределяемые ресурсы у процесса можно отобрать, а динамически неперераспределяемые — нельзя. Выделенные, или закрепленные ресурсы в каждый конкретный момент времени может монопольно использовать только один процесс. Реентерабельный код в процессе своего выполнения не изменяется; несколько выполняющихся процессов могут использовать его коллективно, что позволяет сэкономить основную память. Код многократного последовательного использования в период с начала до конца выполнения может использовать только один процесс; этот код во время работы может меняться, однако для каждого нового процесса он сам повторно инициализирует себя.

Для возникновения тупиковой ситуации должны существовать четыре

необходимых условия: «взаимоисключение», «ожидание дополнительных ресурсов», «неперераспределяемость» и «круговое ожидание». «Взаимоисключение» означает, что процессы заявляют исключительные права на управление своими ресурсами; «ожидание дополнительных ресурсов» означает, что процессы могут удерживать за собой ресурсы, ожидая выделения им дополнительных запрошенных ресурсов; «неперераспределяемость» означает, что ресурсы нельзя принудительно отнимать у процессов; а «круговое ожидание» означает, что существует цепочка процессов, в которой каждый процесс удерживает ресурс, запрашиваемый другим процессом, который в свою очередь удерживает ресурс, запрашиваемый следующим процессом, и т. д.

Основные направления научных исследований в области тупиков — это предотвращение тупиков (если обеспечить нарушение хотя бы одного необходимого условия, то в системе полностью исключается всякая возможность возникновения тупика); обход тупиков (здесь тупиковая ситуация в принципе допускается, однако в случае ее приближения принимаются соответствующие предупредительные меры — предполагается, что такое принципиальное допущение тупиковой ситуации позволяет достичь более рационального использования ресурсов); обнаружение тупиков (здесь возникающие тупики локализуются, с выдачей соответствующей информации для привлечения внимания операторов и системы) и восстановление после тупиков (здесь обеспечивается выход из тупиковых ситуаций — почти всегда с некоторой потерей результатов текущей работы).

Предотвращение тупиков можно обеспечить, если нарушается одно или более необходимых условий тупиковых ситуаций. Только условие взаимоисключения нарушать нельзя, поскольку мы должны предусматривать взаимоисключающее использование многих видов системных ресурсов. Условие ожидания дополнительных ресурсов можно нарушить, если потребовать, чтобы пользователи заблаговременно запрашивали все необходимые ресурсы; система либо удовлетворяет такой запрос в целом, либо отвергает его и заставляет пользователя ждать до тех пор, пока не освободятся все нужные ему ресурсы. Нарушение условия неперераспределяемости можно обеспечить, если потребовать, чтобы пользователь, который удерживает некоторые ресурсы и не получает удовлетворения своего запроса на дополнительные ресурсы, вначале освободил все удерживаемые ресурсы, а затем запросил их все сразу, что достаточно неприятно. Кругового ожидания можно не допустить, если потребовать, чтобы пользователи запрашивали ресурсы в некотором заранее определенном в системе порядке; легко показать, что таким образом предотвращается возможность возникновения циклов в графе распределения ресурсов.

Основу для реализации механизмов обхода тупиковых ситуаций составляет алгоритм банкира, который предложил Дейкстра. Согласно этому алгоритму считается, что состояние системы надежно, если имеется возможность завершить выполнение всех пользовательских программ и вернуть ресурсы системе; говорят, что состояние ненадежно, если нельзя гарантировать завершение работы всех пользователей. Ненадежное состояние не обязательно приводит к тупиковой ситуации; система, находящаяся в ненадежном состоянии, может успешно выйти из него, если события сложатся достаточно благоприятно. Алгоритм банкира содержит несколько серьезных недостатков, которые не позволяют использовать его в современных операционных системах, однако он представляет несомненный теоретический интерес, быть может, как база для создания более подходящих алгоритмов обхода тупиковых ситуаций в будущих операционных системах.

Графы распределения ресурсов удобны тем, что позволяют видеть «состояние ресурсов» системы. Метод редукции графов позволяет обнаруживать тупики.

Восстановление после обнаружения тупика обычно осуществляется путем

вывода из системы одного или нескольких процессов, вовлеченных в тупиковую ситуацию; при этом их ресурсы освобождаются, с тем чтобы их можно было предоставить другим процессам. Восстановление в современных системах производится сложно и неэффективно. Усовершенствованные средства приостановки/возобновления процессов должны существенно улучшить возможности выхода из тупиковых ситуаций.

Тупики в современных системах обычно рассматриваются как достаточно редкая неприятность. В будущих системах тупиковые ситуации могут стать гораздо более критическим фактором, поскольку эти системы будут работать при гораздо более динамичном распределении ресурсов и большем количестве одновременно выполняемых процессов.

Терминология

алгоритм банкира, который предложил Дейкстра (Dijkstra's Banker's Algorithm)

бесконечное откладывание (indefinite postponement)

восстановление после тупика (deadlock recovery)

выделенный, или закрепленный, или монопольно используемый ресурс (dedicated resource)

граф распределения ресурсов (resource allocation graph)

динамическое распределение ресурсов (dynamic resource allocation)

контрольная точка/рестарт (checkpoint/restart)

надежное состояние (safe state)

ненадежное состояние (unsafe state)

необходимые условия для существования тупиков (necessary conditions for deadlock)

неперераспределяемый ресурс (nonpreemptible resource)

«нередуцируемый» процесс (irreducible process)

обнаружение тупиков (deadlock detection)

обход тупиков (deadlock avoidance)

перераспределяемый ресурс (preemptible resource)

предотвращение тупиков (deadlock prevention)

предсказуемость платы за ресурсы (predictability of resource charges)

редукция, или приведение графа распределения ресурсов (reduction of a resource allocation graph)

реентерабельная процедура (reentrant procedure)

«редуцируемый» процесс (reducible process)

ресурс последовательного повторного использования (serially reusable resource)

стандартный способ распределения Хавендера (Standard Allocation Pattern of Havender)

старение процесса (aging)

тупик, тупиковая ситуация, дедлок (deadlock)

условие взаимного исключения (mutual exclusion condition)

условие «кругового взаиможидания» («circular wait» condition)

условие «неперераспределяемости» («no-preemption» condition)

условие «ожидания дополнительных ресурсов» («wait for» condition)

Упражнения

6.1 Дайте определение понятия тупика.

6.2 Приведите "пример тупика с участием всего лишь одного процесса.

6.3 Приведите пример простого тупика с участием трех процессов и трех ресурсов. Начертите соответствующий граф распределения ресурсов.

6.4 Рассмотрите предпосылки для создания систем со спулингом. Почему эти системы зачастую чреваты тупиковыми ситуациями?

6.5 Предположим, что система со спулингом подвержена тупикам. Как разработчик операционных систем, какие средства могли бы вы предусмотреть в данной операционной системе, чтобы помочь оператору обеспечивать выход из тупиковых ситуаций без потери работы, проделанной до этого момента любым из процессов пользователя, вовлеченных в тупиковую ситуацию?

6.6 Современные системы со спулингом зачастую допускают, чтобы распечатка данных начиналась еще до того, как процесс закончит свое выполнение. Предположим, что процесс сформировал 1000 строк для печати и что эти строки были приняты механизмами спулинга и помещены на диск. Предположим, что этот процесс будет продолжать выполняться и со временем сформирует 19 000 оставшихся строк для печати. После формирования первых 1000 строк освободился принтер и началась распечатка этой первой тысячи строк. Что, по вашему мнению, произойдет, когда завершится распечатка этих строк, в каждом из следующих случаев:

а) процесс закончил выдачу оставшихся 19 000 строк на диск.

б) процесс еще продолжает выполняться.

В своих рассуждениях вы должны учитывать, что отпечатанный листинг не должен иметь каких-либо пропусков — он должен выглядеть так, как если бы все 20 000 строк печатались непрерывно.

6.7 Многие из первых систем со спулингом работали, имея фиксированный объем

буферной памяти для спулинга. Современные системы являются более динамичными в том смысле, что операционная система может увеличивать объем буферной памяти для спулинга в процессе работы. Каким образом это помогает уменьшить вероятность тупиковых ситуаций? Однако подобная система все же может попасть в тупиковую ситуацию. Как?

6.8 Что такое «бесконечное откладывание»? Чем оно отличается от тупика? Что у них общего?

6.9 Предположим, что данная система допускает бесконечное откладывание для определенных категорий объектов. Каким образом вы как разработчик систем можете обеспечить предотвращение бесконечного откладывания?

6.10 Обсудите последствия бесконечного откладывания для каждого из следующих типов систем:

а) системы пакетной обработки;

б) системы разделения времени;

в) системы реального времени.

6.11 Почему возможность динамического перераспределения ресурсов является критическим фактором для успешной работы мультипрограммных вычислительных систем?

6.12 Обсудите каждый из следующих видов ресурсов:

а) перераспределяемый ресурс;

б) неперераспределяемый ресурс;

в) разделяемый ресурс (ресурс, коллективного пользования);

г) выделенный, или закрепленный ресурс;

д) реентерабельный код;

е) код многократного последовательного использования.

6.13 Сформулируйте четыре необходимых условия существования тупика. Приведите краткое интуитивное обоснование причин, по которым каждое отдельно взятое условие необходимо.

6.14 Рассмотрите транспортную пробку, показанную на рис. 6.1. Обсудите каждое из необходимых условий существования тупика применительно к транспортной пробке.

6.15 Какие четыре направления научных исследований по проблеме тупиков упомянуты в тексте? Кратко рассмотрите каждое из них.

6.16 Чтобы условие «ожидание дополнительных ресурсов» было нарушено, по методу Хавендера процессы должны сразу запрашивать все ресурсы, которые им понадобятся, только после этого система может разрешить их выполнение. То есть система предоставляет им ресурсы по принципу «все или ничего». Это один из наиболее широко распространенных способов предотвращения тупиков. Обсудите его достоинства и недостатки. , ;

6.17 Предложенное Хавендером нарушение условия «неперераспределяемости» не стало популярным для исключения возможности тупиковых ситуаций. Почему?

6.18 Предложенный Хавендером способ нарушения условия «кругового ожидания»

реализован во многих системах. Обсудите его достоинства и недостатки.

6.19 Покажите, что стандартный способ распределения Хавендера, нарушающий условие «кругового ожидания», действительно предотвращает образование циклов в графах распределения ресурсов,

6.20 Объясните, почему средства обхода тупиков интуитивно кажутся более привлекательными по сравнению со средствами предотвращения тупиков.

6.21 В контексте алгоритма банкира определите, является ли каждое из приведенных ниже состояний надежным или ненадежным. Если состояние надежно, то покажите, каким образом могут завершиться все процессы. Если состояние ненадежно, покажите, каким образом может возникнуть тупик.

Состояние А

	Текущее количество выделенных устройств		Максимальная потребность
Пользователь (1)	2		6
Пользователь (2)	4		7
Пользователь (3)	5		6
Пользователь (4)	0		2
Резерв		1	

Состояние Б

	Текущее количество выделенных устройств		Максимальная потребность
Пользователь (1)	4		8
Пользователь (2)	3		9
Пользователь (3)	5		8
Пользователь (4)	0		0
Резерв		2	

6.22 Сам факт, что состояние является ненадежным, не обязательно говорит о том, что в системе возникнет тупик. Объясните, почему это так. Приведите пример ненадежного состояния и покажите, каким образом все процессы могут завершиться без тупика.

6.23 Алгоритм банкира, который предложил Дейкстра, имеет ряд недостатков, препятствующих его эффективному применению в реальных системах. Прокомментируйте, почему каждое из указанных ниже ограничений может рассматриваться как недостаток

алгоритма банкира.

а) Количество распределяемых ресурсов считается постоянным.

б) Число работающих пользователей считается постоянным.

в) Операционная система гарантирует, что запросы на ресурсы будут удовлетворяться за конечное время.

г) Пользователи гарантируют, что они возвратят системе занимаемые ими ресурсы в течение конечного времени.

д) Пользователи должны заранее указывать свои максимальные потребности в ресурсах.

6.24 Если система не исключает тупиков, то при каких обстоятельствах будете вы обращаться к алгоритму их обнаружения?

6.25 Для алгоритма обнаружения тупиков, основанного на редукции графа, покажите, что порядок редукций не имеет значения, в любом случае будет достигнут один и тот же конечный результат.

6.26 Почему восстановление после тупиков является столь трудной проблемой?

6.27 Почему так сложно выбирать, какие из процессов следует «выбросить» при восстановлении после тупика?

6.28 В данной главе мы говорили о том, что эффективные средства приостановки/возобновления процессов играют важную роль при выходе из тупиковых ситуаций. Проблема «приостановки/возобновления» представляет большой интерес для разработчиков операционных систем, причем пока что она еще не имеет удовлетворительного решения.

а) Объясните, почему так важно обеспечивать приостановку/возобновление процессов.

б) Объясните, почему так сложно реализовать эффективный механизм приостановки/возобновления процессов.

6.29 Почему проблема тупиков станет, по всей вероятности, более критическим фактором для будущих операционных систем?

6.30 В предположении, что первые три необходимых условия существования тупика выполнены, прокомментируйте правомерность следующей стратегии: всем процессам присваиваются уникальные значения приоритетов. Когда имеется более чем один процесс, ожидающий освобождения некоторого ресурса, и этот ресурс освобождается, он выделяется ожидающему процессу с наивысшим приоритетом.

6.31 Предположим, что все ресурсы идентичны, они могут захватываться и освобождаться строго по одному в каждый конкретный момент времени, причем ни одному процессу никогда не требуется больше ресурсов, чем имеется в системе: укажите, сможет ли возникнуть тупик в каждой из следующих систем:

	Число процессов	Число ресурсов
а)	1	1
б)	1	2
в)	2	1
г)	2	2
д)	2	3

Предположим теперь, что ни одному из процессов никогда не потребуется более двух ресурсов. Укажите, может ли возникнуть тупик в каждой из следующих систем:

	Число процессов	Число ресурсов
е)	1	2
ж)	2	2
з)	2	3
и)	3	3
к)	3	4

6.32 Иногда информации относительно максимальных потребностей процессов в ресурсах достаточно для того, чтобы обеспечить обход тупиков без значительных затрат рабочего времени. Рассмотрим, например, следующую ситуацию (Холт): набор из t процессов коллективно использует r идентичных ресурсов. Ресурсы могут захватываться и освобождаться строго по одному в каждый момент времени. Ни одному процессу никогда не требуется более n ресурсов. Сумма максимальных потребностей для всех процессов строго меньше $t+n$. Покажите, что в подобной системе тупики невозможны.

ЧАСТЬ 3

Управление памятью

Глава 7

Физическая память

Ничто и никогда не становится реально осознаваемым, пока не будет испытано и прочувствовано, даже пословица не станет для вас пословицей, пока сама ваша жизнь не докажет ее справедливость.

Джон Кито

7.1 Введение

7.2 Организация памяти



7.3 Управление памятью

7.4 Иерархия памяти

7.5 Стратегии управления памятью

7.6 Связное и несвязное распределение памяти

7.7 Связное распределение для одного пользователя

7.7.1 Защита памяти в однопрограммных системах

7.7.2 Однопоточковая пакетная обработка

7.8 Мультипрограммирование с фиксированными разделами

7.8.1 Мультипрограммирование с фиксированными разделами? трансляция и загрузка модулей в абсолютных адресах

7.8.2 Мультипрограммирование с фиксированными разделами: трансляция и загрузка перемещаемых модулей

7.8.3 Защита в мультипрограммных системах

7.8.4 Фрагментация при мультипрограммировании с фиксированными разделами

7.9 Мультипрограммирование с переменными разделами

7.9.1 Объединение соседних свободных участков памяти

7.9.2 Уплотнение памяти

7.9.3 Стратегии размещения информации в памяти

7.10 Мультипрограммирование со свопингом

7.1 Введение

Организация и управление *основной* или *первичной* или *физической (реальной) памятью* вычислительной машины — один из самых важных факторов, определяющих построение операционных систем. В английской технической литературе память обозначается синонимами *memory* и *storage*, причем из этих двух терминов в настоящее время чаще употребляется *storage*. Для непосредственного выполнения программ или обращения к данным необходимо, чтобы они размещались в основной памяти. Вторичная, или внешняя память — это, как правило, накопители на магнитных дисках, магнитных барабанах и магнитных лентах — имеет гораздо большую емкость, стоит дешевле и позволяет хранить множество программ и данных, которые должны быть наготове для обработки,

В этой и последующих главах рассматривается много распространенных схем организации и управления памятью компьютера. В этой главе речь идет о физической, или реальной памяти, а в нескольких следующих главах о виртуальной памяти. Схемы описаны приблизительно в том порядке, в котором они исторически появлялись.

7.2 Организация памяти

В прошлом основная память представляла собой самый дорогостоящий ресурс. В связи с этим она требовала особого внимания со стороны разработчиков систем — необходимо было обеспечить как можно более эффективное использование столь дорогого ресурса. В первых машинах главной задачей считалось оптимальное использование основной памяти благодаря рациональной организации и управлению ею.

Под организацией памяти мы понимаем то, каким образом представляется и используется основная память. Будем ли мы помещать в основную память только одну программу пользователя или несколько программ одновременно? Если в основной памяти размещается несколько пользовательских программ сразу, будем ли мы предоставлять каждой из них одинаковое количество ячеек или разобьем основную память на части, так называемые разделы, различных размеров? Будем ли мы разбивать основную память жестким образом, когда разделы определяются на достаточно длительные периоды времени, либо предусмотрим более динамичное разбиение, позволяющее вычислительной машине быстро реагировать на изменения потребностей программ пользователя в ресурсах? Будем ли мы требовать такого построения программ пользователя, чтобы они выполнялись только в конкретном разделе, либо предусмотрим возможность выполнения программ с занятием любых подходящих для них разделов? Будем ли мы требовать, чтобы каждая программа помещалась в одном непрерывном, сплошном блоке ячеек памяти, либо допустим возможность разбиения программ на отдельные блоки, размещаемые в любых свободных участках (дырах) основной

памяти?

Существуют системы, построенные с ориентацией на каждую из указанных схем, и в настоящей главе мы будем рассматривать способы реализации каждой из этих схем.

7.3 Управление памятью

Независимо от того, какую схему организации памяти мы примем для конкретной системы, необходимо решить, какие стратегии следует применять для достижения оптимальных характеристик. Стратегии управления памятью определяют, каким образом будет работать память конкретной организации при различных подходах к решению следующих вопросов: когда мы должны помещать новую

программу в память? Будем ли мы помещать ее, когда система специально попросит об этом, либо будем пытаться предупреждать запросы системы? В какое место основной памяти мы будем помещать очередную программу для выполнения? Будем ли мы размещать программы как можно более плотно с занятием имеющихся свободных участков, чтобы свести к минимуму потери памяти, либо будем стремиться к возможно более быстрому размещению программ, чтобы свести к минимуму потери машинного времени?

Если в основную память необходимо поместить новую программу и если в настоящий момент основная память уже заполнена целиком, то какую из находящихся в ней других программ следует вывести из памяти? Должны ли мы замещать программы, которые находятся в памяти дольше других, или программы, используемые наименее часто, или те, которые дольше всего не использовались? Существуют системы, ориентированные на использование каждой из этих стратегий управления памятью.

7.4 Иерархия памяти

В 50-х и 60-х годах основная память — это была, как правило, память на магнитных сердечниках — стоила очень дорого. Выбор размера основной памяти каждой вычислительной машины приходилось осуществлять весьма продуманно. Заказчик не хотел покупать больше, чем он мог себе позволить, однако ему приходилось платить за память такого объема, который был бы достаточен для обеспечения работы операционной системы и запланированного количества пользователей. Задача заключалась в том, чтобы приобрести только тот необходимый минимум памяти, который мог бы удовлетворить предполагаемые нужды пользователей и в то же время не выходил за пределы финансовых возможностей заказчика.

Чтобы программы можно было выполнять, а к данным можно было обращаться, они должны размещаться в основной памяти. Программы и данные, которые в настоящий момент не нужны можно хранить во внешней памяти, а затем, когда в этом возникает необходимость, переписывать в основную память для выполнения или использования. Накопители внешней памяти, например магнитные ленты или диски, стоят, как правило, дешевле, чем основная память, и имеют гораздо большую емкость. Однако доступ к основной памяти может осуществляться, вообще говоря, гораздо быстрее, чем к внешней памяти.

Системам с несколькими уровнями иерархии памяти свойственна высокая интенсивность челночных обменов программами и данными между устройствами памяти различных уровней. Такие обмены отнимают системные ресурсы, например дорогое время центрального процессора, которое в противном случае можно было бы использовать продуктивно.

В 60-х годах стало ясно, что иерархию памяти можно расширить

еще на один уровень, что позволит получить громадный выигрыш в скоростных характеристиках и эффективности использования памяти. Этот дополнительный уровень — так называемая *кэш-память*, которая обладает гораздо большим быстродействием, чем основная память: Однако по сравнению с основной памятью кэш-память стоит очень дорого, и поэтому в реальных системах применяются только кэш-памяти относительно небольшого объема. Отношения между кэш-памятью, основной (первичной) памятью и Внешней (вторичной) памятью иллюстрируются рис. 7.1.

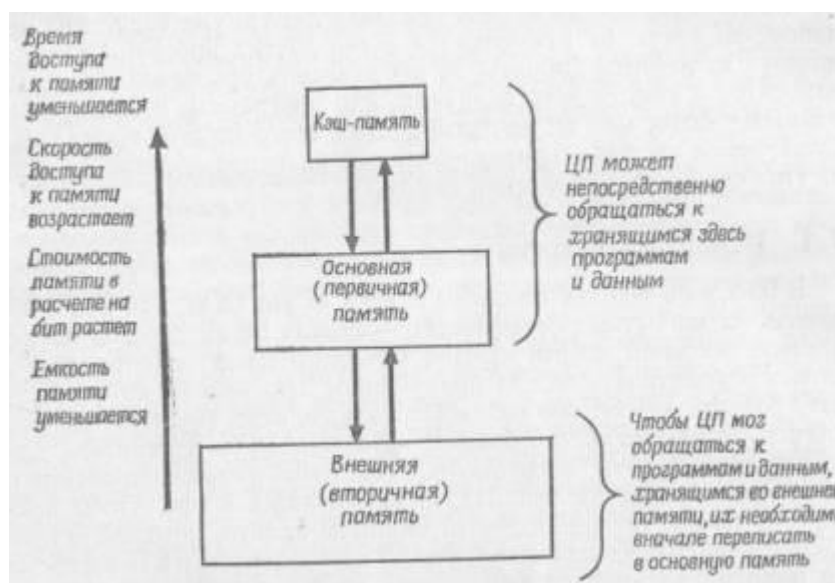


Рис. 7.1 Иерархическая организация памяти.

Кэш-память вносит в систему еще один уровень обменов. Программы, хранящиеся в основной памяти, для своего выполнения передаются в сверхскоростную кэш-память, поскольку это обеспечивает значительный выигрыш во времени выполнения. Разработчики систем, применяющие концепцию кэш-памяти, надеются на то, что затраты, связанные с переписью программ из основной памяти в кэш-память и обратно, окажутся гораздо меньше того выигрыша в быстродействии, который может быть получен благодаря сокращению времени выполнения программ, находящихся в кэш-памяти.

7.5 Стратегии управления памятью

Для того чтобы обеспечить интенсивное использование дорогостоящих ресурсов, ими нужно эффективно управлять. Стратегии управления памятью направлены на то, чтобы обеспечить наилучшее

возможное использование ресурсов основной памяти. Стратегии управления памятью делятся на следующие категории.

1. Стратегии выборки:

- а) стратегии выборки по запросу (по требованию),
- б) стратегии упреждающей выборки.

2. Стратегии размещения.

3. Стратегии замещения.

Стратегии выборки ставят своей целью определять, когда следует «втолкнуть» очередной блок программы или данных в основную память. В течение многих лет полагали, что наиболее целесообразно осуществлять *выборку по запросу*: согласно этому принципу, очередной блок программы или данных загружается в основную память, когда его запрашивает работающая программа. Считалось, что, поскольку в общем случае мы не можем предсказать, куда будет передаваться управление по программе, дополнительные затраты, связанные с прогнозированием дальнейшего хода программы и упреждающей выборкой, будут значительно превышать ожидаемые выгоды. А сегодня многие специалисты уверены в том, что упреждающая выборка вполне может обеспечить повышение быстродействия системы.

Стратегии *размещения* ставят своей целью определить, в какое место основной памяти следует помещать поступающую программу. В этой главе мы рассмотрим стратегии размещения, реализующие принципы занятия «*первого подходящего*», «*наиболее подходящего*» и «*наименее подходящего*» по размерам свободного участка памяти.

Стратегии *замещения* ставят своей целью определить, какой блок программы или данных следует вывести («вытолкнуть») из основной памяти, чтобы освободить место для записи поступающих программ или данных.

7.6 Связное и несвязное распределение памяти

Самые первые вычислительные машины требовали *связного* распределения памяти — каждая программа должна была занимать один сплошной блок ячеек памяти. Только после появления мультипрограммирования с разделами переменного размера стало ясно, что гораздо более эффективным может быть несвязное распределение памяти.

При *несвязном распределении памяти* программа разбивается на ряд блоков, или *сегментов*, которые могут размещаться в основной памяти в участках, не обязательно соседствующих друг с другом. Операционной системе гораздо сложнее обеспечить несвязное распределение памяти, однако подобный подход обладает важным преимуществом: если основная память имеет ряд небольших свободных участков вместо одного большого, то, как правило, операционная система все же может загрузить и выполнить программу, которой в противном случае пришлось бы ждать.

В нескольких следующих главах мы рассмотрим системы виртуальной памяти,

получившие широкую популярность в настоящее время. Все они предусматривают несвязное распределение памяти.

7.7 Связное распределение памяти для одного пользователя

На самых первых вычислительных машинах в каждый момент времени мог работать только один человек, и все ресурсы машины ¹ оказывались в распоряжении этого пользователя. Предъявление счетов за использование машинного времени производилось по самому простому принципу — поскольку пользователю машина предоставлялась целиком, ему приходилось платить за все ее ресурсы независимо от того, занимала ли реально его программа эти ресурсы. В связи с этим обычные механизмы выписки счетов ориентировались на чисто календарное время. Пользователю машина выделялась на определенное время, а затем предъявлялся счет на почасовую оплату этого времени. В некоторых современных вычислительных системах, работающих с разделением времени, для расчетов с пользователями применяются гораздо более сложные алгоритмы.

Первоначально каждый пользователь сам писал всю программу, необходимую для реализации конкретной прикладной задачи, включая подробные процедуры ввода-вывода в машинных кодах. Но очень скоро коды программ, необходимых для реализации базовых функций ввода-вывода, начали объединяться в так называемые *системы управления вводом-выводом (IOCS)*. Благодаря этому у пользователей, которым требовались определенные операции ввода-вывода, отпала необходимость самим непосредственно кодировать эти операции, — они получили возможность вызывать соответствующие подпрограммы ввода-вывода, эффективно выполняющие \ реальную работу. Тем самым было обеспечено существенное упрощение и ускорение процесса кодирования программ. Создание систем управления вводом-выводом можно, по-видимому, считать началом развития концепции современных операционных систем. Организация памяти в типичном случае *связного распределения* для одного пользователя показана на рис. 7.2.

Размер программ в обычном случае ограничивается емкостью имеющейся основной памяти, однако существует возможность выполнения программ, превышающих по размеру основную память, благодаря использованию так называемых *оверлейных* сегментов. Эта концепция иллюстрируется рис. 7.3. Если какой-то конкретный модуль программы не работает в течение всего периода выполнения программы, то из внешней памяти можно выбрать

другой модуль и записать его в основную память на место уже не нужного модуля.

Оверлейный режим предоставляет программисту возможность как бы расширить ограниченные рамки основной памяти. Однако ручная реализация оверлейного режима требует продуманного



Рис. 7.2 Распределение памяти с выделением непрерывного сегмента одному пользователю.

и трудоемкого планирования. Программа со сложной оверлейной структурой может весьма трудно поддаваться модификации. Как мы увидим в последующих главах, проблема оверлейных сегментов, контролируемых самим программистом, отпадает благодаря появлению систем виртуальной памяти.

7.7.1 Защита памяти в однопрограммных системах

В однопрограммных системах пользователю выделяется связный (непрерывный) сегмент памяти и предоставляются полные возможности управления практически всей основной памятью. При этом память разделяется на три части, первая из которых служит для размещения подпрограмм операционной системы, вторая содержит программу пользователя, а третья остается свободной. Проблема защиты в этом случае достаточно проста. Каким образом следует защищать операционную систему, чтобы программа пользователя не могла ее испортить?

Если программа пользователя начинает работать неправильно, это может привести к порче операционной системы. Если операционная система будет испорчена настолько, что пользователь не сможет продолжать работу, то он по крайней мере поймет, что что-то не в порядке, прекратит выполнение своей программы, исправит ошибку и сделает повторную попытку запустить задачу. При такой ситуации необходимость специально защищать операционную систему не слишком очевидна.

Предположим, однако, что программа пользователя портит операционную систему более «нежно». Например, она случайно изменяет определенные служебные программы ввода-вывода. При этом программа может продолжать выполняться, но все выходные

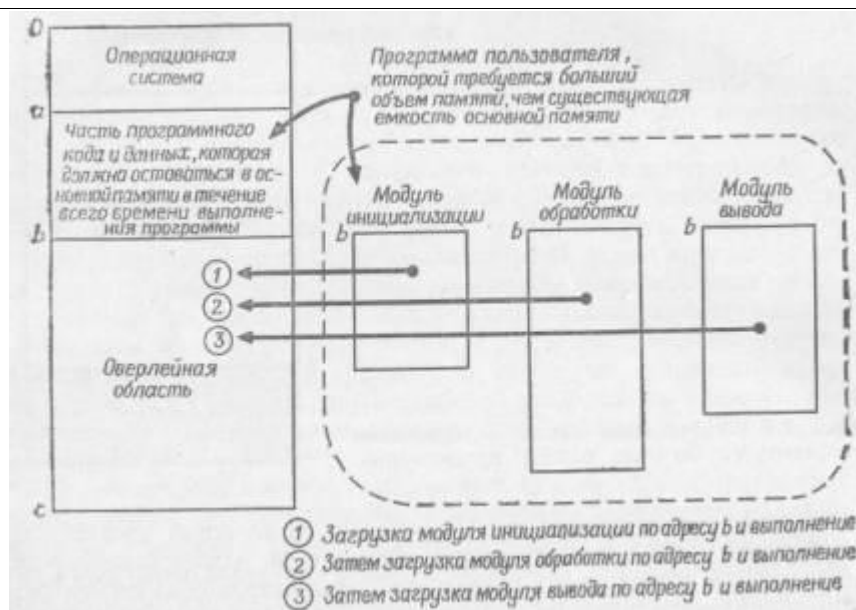


Рис. 7.3 Типичная оверлейная структура.

записи будут потеряны. Если результаты работы программы не контролировать до момента завершения, то окажется, что машинные ресурсы расходовались впустую. Еще более неприятна ситуация, когда испорченная система выдает результаты, неправильность которых установить весьма нелегко.

Таким образом, ясно, что операционную систему однопрограммной вычислительной машины следует защищать от пользователя. Защиту можно реализовать при помощи одного *граничного регистра*, встроенного в ЦП, как показано на рис. 7.4. Граничный регистр содержит самый старший адрес команды, относящийся к операционной системе. Если пользователь попытается войти в операционную систему, его команда будет перехвачена, а задание аварийно завершено с выдачей соответствующего сообщения об ошибке.

Естественно, что у пользователя время от времени возникает необходимость обращения к операционной системе, например, за услугами по выполнению операций ввода-вывода. Для этого в распоряжение пользователя предоставляется специальная команда,

при помощи которой он может запрашивать определенные услуги операционной системы (например, команда *вызова супервизора SVC*). Пользователь, если ему нужно прочитать данные с магнитной ленты, выдает команду, которая для операционной системы будет просьбой выполнить эту операцию для пользователя. Операционная система выполняет требуемую операцию, а затем возвращает управление программе пользователя.

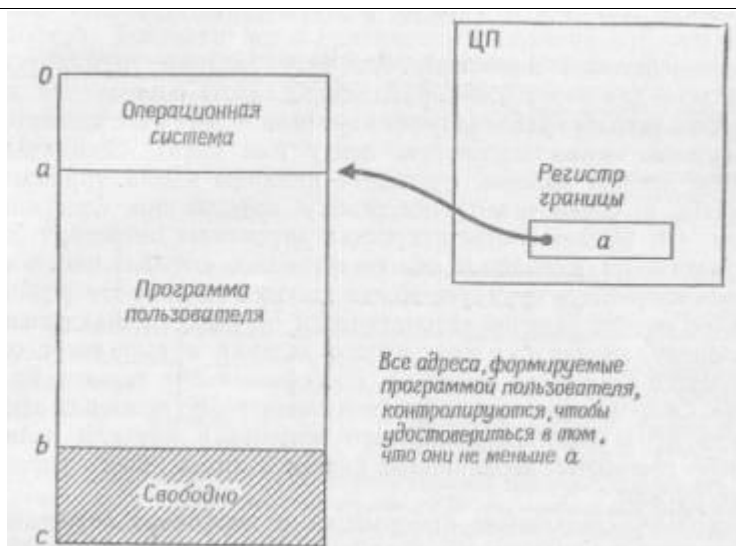


Рис. 7.4 Защита памяти с выделением непрерывного сегмента одному пользователю.

С ростом сложности операционных систем появляется необходимость создания гораздо более серьезных и эффективных механизмов, защищающих операционную систему от пользователей, а пользователей — друг от друга. Ниже мы обсудим эти механизмы подробнее.

7.7.2 Однопоточковая пакетная обработка

Каждое задание занимает однопрограммные системы в течение большого периода времени, чем требуется для непосредственного выполнения этого задания. Как правило, перед выполнением задания приходится затрачивать значительное *подготовительное время*, в течение которого загружается операционная система, устанавливаются кассеты магнитных лент и дисковые пакеты, вкладываются в устройство ввода колоды перфокарт, заправляются в принтер соответствующие бланки, «пробивается» начальное время на управляющих картах и т. д. После завершения выполнения заданий требуется значительное *заключительное время*, когда снимаются магнитные ленты, колоды перфокарт и пакеты дисков, забираются распечатки, на картах контроля времени «пробивается» конечное время и т. д. В течение подготовительного и заключительного периодов компьютер фактически простаивает.

Разработчики сразу же поняли, что, если бы им удалось автоматизировать *переход с задания на задание*, то это позволило бы существенно сократить количество времени, теряемого между заданиями. Это привело к созданию систем пакетной обработки. При *однопоточковой пакетной обработке* задания группируются в пакеты — для этого их управляющие карты помещаются друг за другом в устройство ввода перфокарт (или задания последовательно загружаются на магнитную ленту или диск). Специальный *процессор потока заданий* считывает команды языка управления заданиями и обеспечивает подготовку выполнения следующего задания. Он выдает соответствующие директивы оператору ЭВМ и автоматически выполняет многие функции, которые ранее приходилось выполнять вручную. Когда текущее задание завершается, процессор потока заданий автоматически считывает команды языка управления для запуска следующего задания и выполняет соответствующие *служебные действия*, обеспечивающие

переход на это задание. Системы пакетной обработки значительно повысили эффективность использования машинного времени и помогли оценить истинную важность операционных систем и интенсивного управления ресурсами.

Поскольку считывание информации с перфокарт осуществляется относительно медленно, в крупных вычислительных центрах начали применять небольшие *компьютеры-спутники* с целью подготовки пакетов заданий для выполнения на крупной машине. Когда колоды перфокарт заданий поступали в вычислительный центр, они группировались в пакеты и загружались на магнитную ленту на компьютерах-спутниках. Магнитные ленты затем переносились на крупную машину. Это был эффективный способ сокращения времени переключения с задания на задание на крупной машине, он позволял повысить эффективность использования этого компьютера. Системы пакетной обработки с одним потоком заданий были последним словом вычислительной техники в начале 60-х годов.

7.8 Мультипрограммирование с фиксированными разделами

Даже при наличии операционных систем пакетной обработки однопрограммные машины по-прежнему непроизводительно теряют значительное количество вычислительных ресурсов. На рис. 7.5 показано, что программа занимает ресурсы центрального процессора только до тех пор, пока ей не потребуется произвести

операцию ввода или вывода. После выдачи запроса ввода-вывода программа зачастую не может продолжаться, пока не будут либо переданы, либо приняты соответствующие данные. Скорости операций ввода и вывода крайне низки по сравнению с быстродействием ЦП. Например, устройству ввода перфокарт, считывающему 1000 карт в минуту, требуется 60 000 микросекунд, чтобы прочитать одну карту и передать ее данные для обработки. В течение всего этого времени ЦП простаивает.

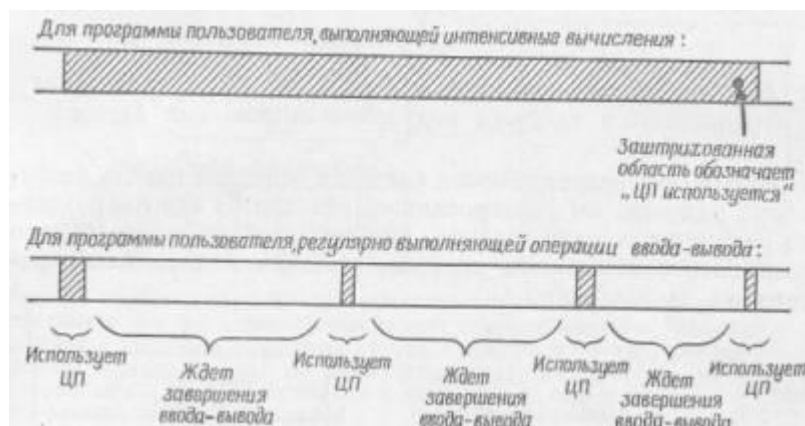


Рис. 7.5 Использование ЦП в однопрограммной машине. Примечание: при выполнении многих заданий в однопрограммной машине длительность периодов ожидания завершения ввода-вывода в гораздо большей степени превышает длительность периодов использования ЦП, чем показано на этой диаграмме.

Разработчики увидели, что здесь у них опять-таки имеется возможность значительного увеличения коэффициента использования ЦП благодаря активному управлению заданиями. На этот раз они пошли по пути реализации *мультипрограммных* систем, в которых несколько пользователей одновременно

«состязаются» за обладание машинными ресурсами. При этом задание, ожидающее в текущий момент завершения операции ввода-вывода, будет уступать ЦП другому заданию, готовому для выполнения вычислений, если, естественно, таковое имеется. Таким образом, обеспечивается возможность одновременного выполнения и операций ввода-вывода, и вычислений на ЦП. Благодаря этому существенно повышается коэффициент использования ЦП и производительность системы.

Потенциальные преимущества мультипрограммирования можно использовать в максимальной степени только в том случае, если в основной памяти компьютера размещается сразу несколько заданий. Благодаря этому в случае, когда одно задание запрашивает ввод-вывод, ЦП может немедленно переключиться на другое задание и выполнять вычисления без всяких задержек. Когда это новое задание освобождает ЦП, другое задание уже может быть готовым для его использования.

Мультипрограммирование зачастую требует большего объема памяти, чем однопрограммный режим. Однако затраты на дополнительную память с лихвой окупаются благодаря улучшенному использованию ресурсов ЦП и периферийных устройств. Было реализовано много различных схем мультипрограммирования, которые мы обсудим в нескольких следующих разделах.

7.8.1 Мультипрограммирование с фиксированными разделами: трансляция и загрузка модулей в абсолютных адресах

В первых мультипрограммных системах основная память разбивалась на ряд *разделов* фиксированного размера. В каждом разделе могло размещаться одно задание. Центральный процессор быстро переключался с задания на задание, создавая иллюзию одновременного их выполнения.

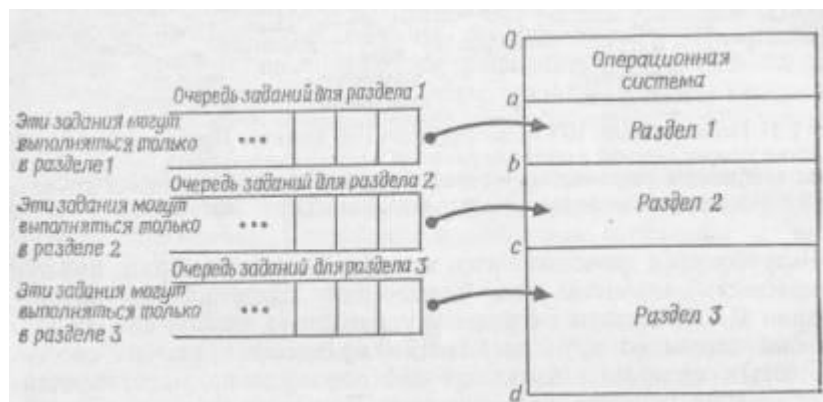


Рис. 7.6 Мультипрограммирование с фиксированными разделами, с трансляцией и загрузкой в абсолютных адресах.

Трансляция заданий производилась при помощи ассемблеров и компиляторов в абсолютных адресах с расчетом на выполнение только в конкретном разделе (рис. 7.6). Если задание было готово для выполнения, а его раздел в это время был занят, то заданию приходилось ждать, несмотря на то что другие разделы были свободны (рис. 7.7).

Это приводило к неэффективному использованию ресурсов памяти, однако

позволяло относительно просто реализовать операционную систему.

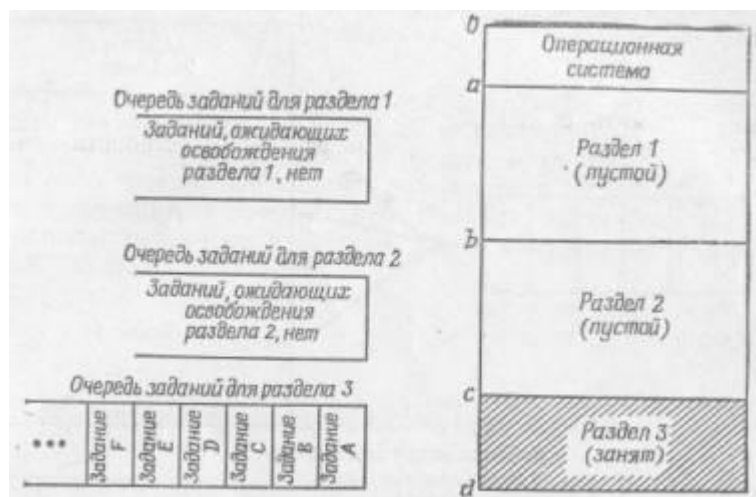


Рис. 7.7 Крайний случай исключительно неэффективного использования ресурсов памяти при мультипрограммировании с фиксированными разделами и трансляцией и загрузкой в абсолютных адресах. Задания, ожидающие освобождения раздела 3, малы и могли бы разместиться в других разделах. Однако их абсолютные адреса требуют, чтобы они выполнялись только в разделе 3. Остальные два раздела остаются пустыми.

7.8.2 Мультипрограммирование с фиксированными разделами; трансляция и загрузка перемещаемых модулей

Перемещающие компиляторы, ассемблеры и загрузчики применяются для трансляции и загрузки перемещаемых модулей, которые могут работать в любом свободном разделе, достаточно большом для их размещения (рис. 7.8). Такой подход свободен от некоторых принципиальных недостатков в использовании памяти, присущих мультипрограммированию с трансляцией и загрузкой в абсолютных адресах. Трансляторы и загрузчики перемещаемых модулей гораздо сложнее, чем трансляторы и загрузчики в абсолютных адресах.

7.8.3 Защита в мультипрограммных системах

В мультипрограммных системах со связным распределением памяти защита чаще всего реализуется при помощи нескольких *граничных регистров*. Два регистра позволяют указывать нижнюю и верхнюю границы раздела пользователя (рис. 7.9), либо нижнюю границу (верхнюю границу) и размер раздела. Программа, которой требуется обратиться к услугам операционной системы,

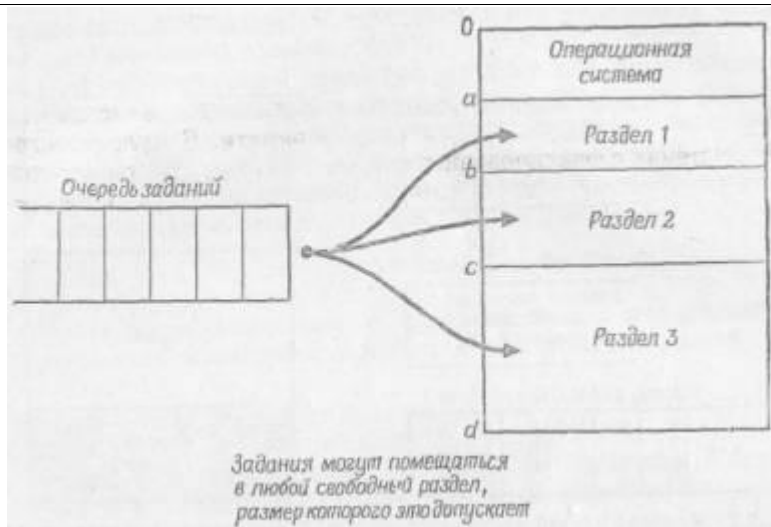


Рис. 7.8 Мультипрограммирование с фиксированными разделами, с трансляцией и загрузкой перемещаемых модулей.

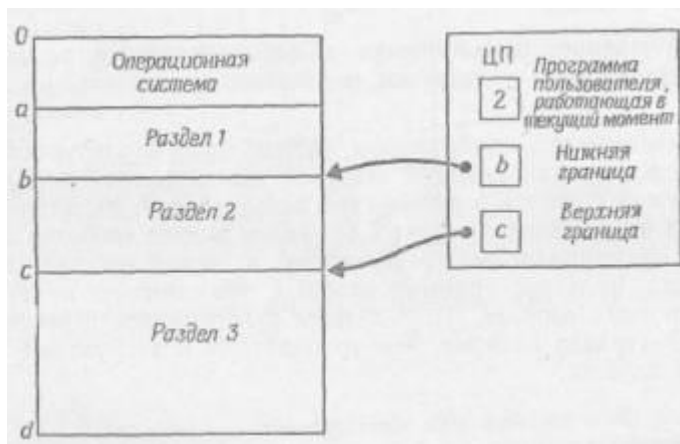


Рис. 7.9 Защита памяти в мультипрограммных системах со связным распределением. Когда программа пользователя в разделе 2 работает, все формируемые ею адреса памяти контролируются, чтобы удостовериться, что они находятся в диапазоне между *b* и *c*.

использует для этого команду вызова супервизора. Тем самым программа пользователя получает возможность пересекать границу операционной системы и обращаться к ее услугам.

7.8.4 Фрагментация при мультипрограммировании с фиксированными разделами

Фрагментация памяти имеет место в любой вычислительной машине независимо от организации ее памяти. В мультипрограммных системах с фиксированными разделами фрагментация происходит либо потому, что задания пользователя не полностью занимают выделенные им разделы, либо в случае, когда некоторый раздел остается незанятым из-за того, что он слишком мал для размещения

ожидающего задания.

7.9 Мультипрограммирование с переменными разделами

Анализируя проблемы, присущие мультипрограммированию с фиксированными разделами, разработчики операционных систем пришли к выводу, что явно лучше было бы позволить заданиям занимать столько места (в пределах всей физической памяти), сколько им требуется. Не нужно соблюдать никаких фиксированных границ — напротив, заданиям следует предоставлять столько памяти, сколько им необходимо. Такой подход называется мультипрограммированием с переменными разделами. Начальное разделение памяти при мультипрограммировании с переменными разделами показано на рис. 7.10.

Мы продолжаем обсуждать здесь только схемы связного распределения памяти, при котором каждое задание должно занимать смежные ячейки. При мультипрограммировании с переменными разделами мы не делаем никаких предположений относительно величины заданий (если не считать того, что задания не должны превышать размер имеющейся основной памяти компьютера). Заданиям, когда они поступают и если механизмы планирования решают, что их необходимо выполнять, выделяется такой объем памяти, который они требуют. Здесь не происходит никакого «перерасхода» памяти — раздел каждого задания в точности соответствует размеру этого задания.

Однако любая схема организации памяти сопряжена с определенными потерями. В случае мультипрограммирования с переменными разделами эти потери становятся очевидными только тогда, когда задания начинают завершаться, а в основной памяти остаются свободные участки, или «дыры», как показано на рис. 7.11. Эти участки можно использовать для размещения других заданий, однако даже если это происходит, то все равно остаются «дыры» только меньшего размера. Таким образом, и в мультипрограммировании с переменными разделами без потерь памяти не обойтись.

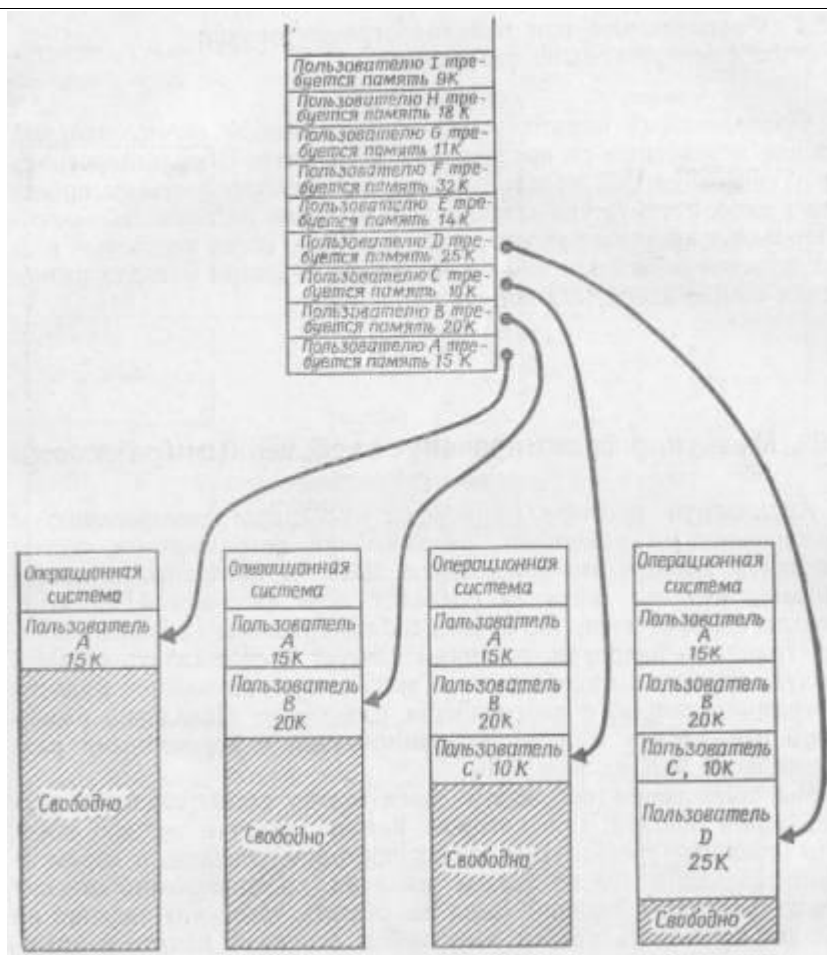


Рис. 7.10 Начальное распределение разделов при мультипрограммировании с переменными разделами.

7.9.1 Объединение соседних свободных участков памяти

Когда в мультипрограммной системе с переменными разделами некоторое задание завершается, мы можем проверить, не соседствует ли освобожденный участок памяти с другими свободными участками («дырами»). Если соседствует, то мы можем в список свободной памяти занести либо (1) новый свободный участок, либо (2) единый свободный участок, полученный объединением новой «дыры» с соседними.

Процесс объединения соседних «дыр» с образованием единого большого свободного участка называется слиянием и иллюстрируется рис. 7.12. Благодаря объединению «дыр» мы формируем непрерывные блоки памяти максимально возможного размера.

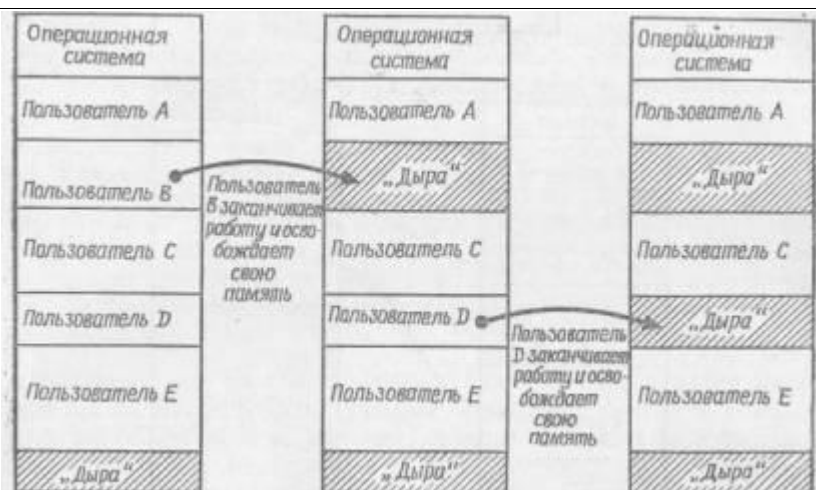


Рис. 7.11 Образование «дыр» в памяти при мультипрограммировании с переменными разделами.

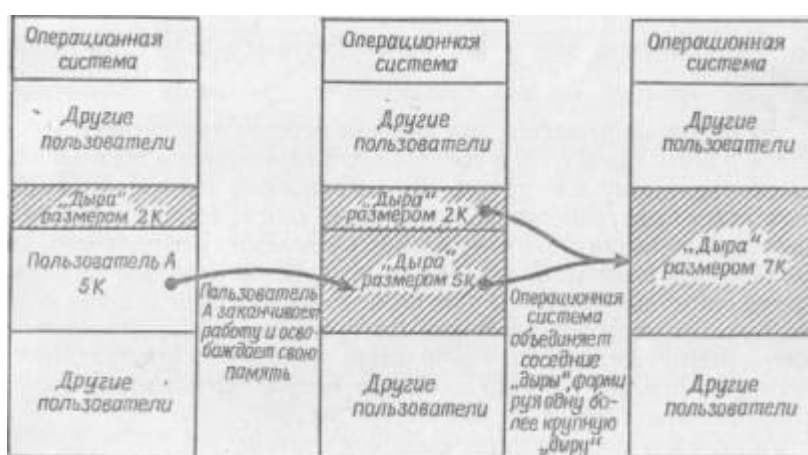


Рис. 7.12 Объединение «дыр» в памяти при мультипрограммировании с переменными разделами.

7.9.2 Уплотнение памяти

Часто, даже после объединения соседних свободных участков бывает так, что во всей основной памяти остаются разбросанными отдельные «дыры», составляющие в целом значительный объем памяти. Иногда очередное задание требует определенного объема основной памяти, но оказывается, что нет ни одного индивидуального свободного участка, достаточно большого для размещения этого задания, несмотря на то, что сумма всех свободных участков превышает общий объем требуемой памяти.



Рис. 7.13 Уплотнение памяти при мультипрограммировании с переменными разделами.

Эта проблема решается при помощи метода, называемого уплотнением памяти (рис. 7.13) и состоящего в перемещении всех занятых участков к одному или другому краю основной памяти. Благодаря этому вместо многочисленных небольших «дыр», обычных для мультипрограммирования с переменными разделами, мы получаем единый большой свободный участок памяти. А уж если вся свободная память представлена одним участком, то ожидающее задание может выполняться, если ему требуется память, не превышающая размера этого единого полученного в результате уплотнения участка. Иногда уплотнение памяти образно называют «утряской памяти», а среди программистов бытует термин «сбор мусора».

Уплотнение памяти имеет свои недостатки:

- Оно отнимает ресурсы системы, которые можно было бы использовать продуктивно.
- Во время уплотнения памяти система должна прекращать любые другие работы. Результатом этого могут стать непрогнозируемые времена ответа для пользователей диалогового режима, и это может оказаться неприемлемым для систем реального времени.
- Уплотнение предполагает перемещение заданий в памяти. Это означает, что информация, связанная с размещением программы и обычно теряющаяся после загрузки программы, теперь должна сохраняться в легкодоступной форме.
- В типичном случае быстро меняющейся смеси заданий возникает необходимость частого уплотнения памяти. Затрачиваемые на это системные ресурсы могут не оправдываться получаемыми при этом выгодами.

7.9.3 Стратегии размещения информации в памяти

Те или иные стратегии размещения информации в памяти применяются для того, чтобы определить, в какое место основной памяти следует помещать поступающие программы и данные. В технической литературе чаще всего описываются три стратегии, приведенные на рис. 7.14.

- *Стратегия наиболее подходящего.* Поступающее задание помещается в тот свободный участок основной памяти, в котором ему наиболее «тесно», так что остается минимально возможное неиспользуемое пространство. Для многих людей выбор наиболее подходящего кажется интуитивно самой рациональной стратегией.
- *Стратегия первого подходящего.* Поступающее задание помещается в первый встретившийся свободный участок основной памяти достаточного размера. Выбор первого подходящего по размеру свободного участка также кажется интуитивно рациональным, поскольку он позволяет быстро принять решение о размещении задания.
- *Стратегия наименее подходящего.* На первый взгляд подобный подход кажется весьма странным. Однако более тщательное рассмотрение показывает, что выбор наименее подходящего по размеру также имеет сильные интуитивные аргументы в свою пользу. Этот принцип говорит о том, что при помещении программы в основную память нужно занимать свободный участок, имеющий наиболее далекий размер, т. е. максимальный возможный свободный участок. Интуитивный аргумент в пользу такого подхода достаточно прост: после помещения программы в большой свободный участок остающийся свободный участок зачастую также оказывается большим и, таким образом, в нем можно разместить относительно большую новую программу.

7.10 Мультипрограммирование со свопингом

Каждая из систем мультипрограммирования, обсуждавшихся выше, предполагает, что программы пользователя остаются в основной памяти до момента завершения. Существует схема, называемая *свопингом*, которая не накладывает подобного требования.

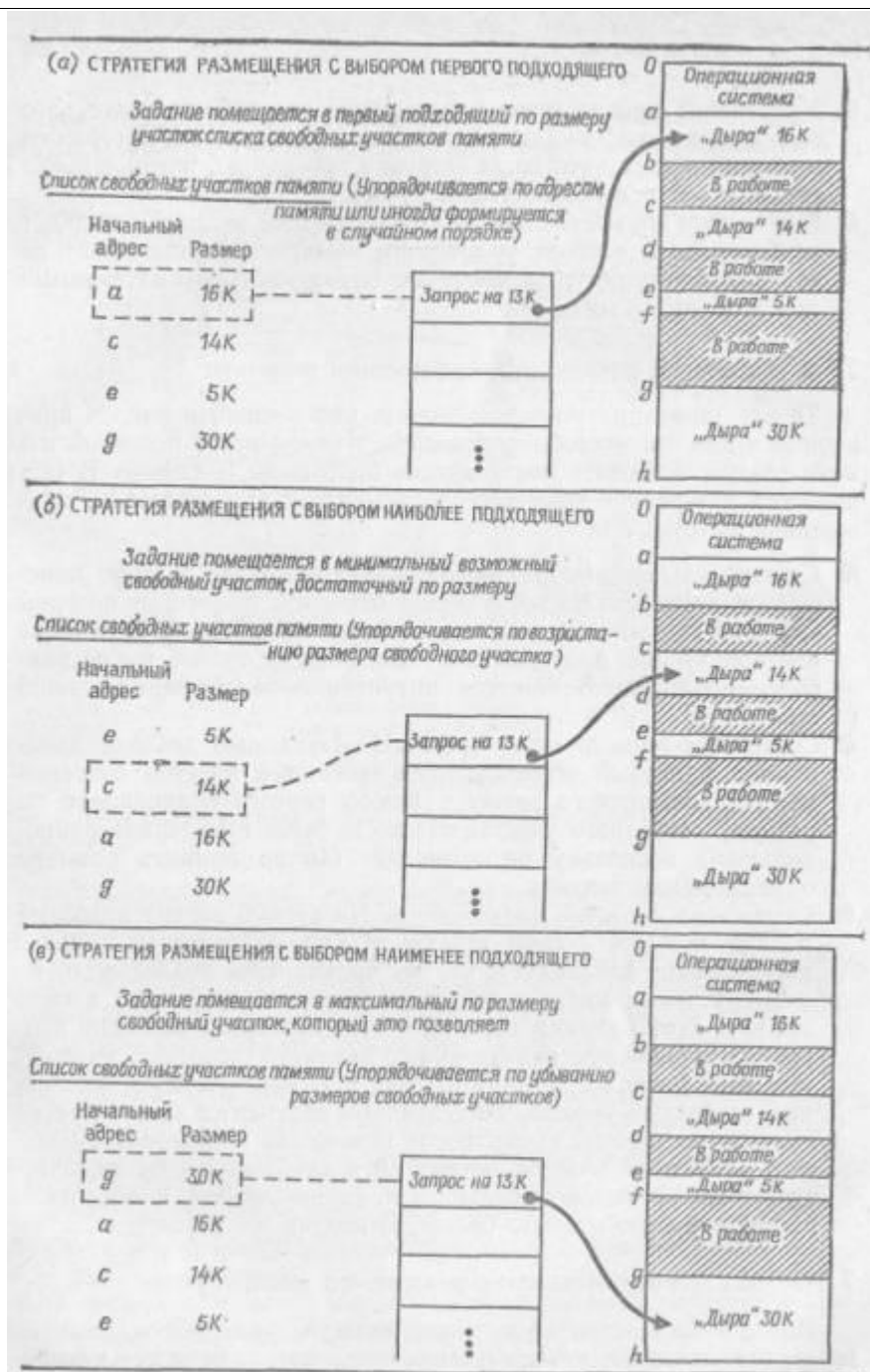


Рис. 7.14 Стратегии размещения информации в памяти по принципу занятия первого подходящего, наиболее подходящего и наименее подходящего по размеру свободного участка.

В некоторых системах со свопингом (рис. 7.15) сразу всю основную память в каждый момент времени занимает одно задание. Это задание выполняется до тех пор, пока оно может продолжаться,

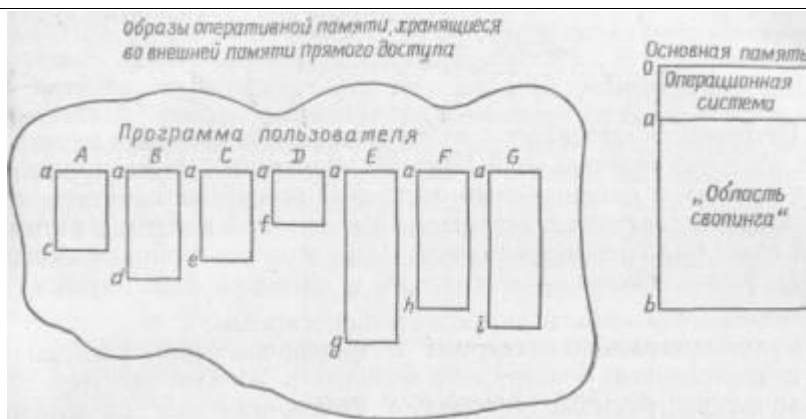


Рис. 7.15 Мультипрограммирование в системе со свопингом, в которой в основной памяти в каждый момент времени размещается только одна программа пользователя. Она работает до тех пор, пока не произойдет одно из следующих событий: а) выдан запрос ВВ; б) появился сигнал таймера; в) программа завершилась. Область свопинга для этой программы копируется во внешнюю память. Образ оперативной памяти для следующей программы загружается в область свопинга, и эта программа выполняется до тех пор, пока она не уступит места следующей программе и т. д. Эта схема широко применялась в первых системах с разделением времени.

а затем освобождает как память, так и ЦП для следующего задания. Таким образом, вся память целиком на короткий период выделяется одному заданию, затем в какой-то момент это задание выводится (*выталкивается*, т. е. осуществляется «откачка») и очередное задание вводится (*вталкивается*, т. е. осуществляется «подкачка»). В обычном случае каждое задание, еще до завершения, будет много раз перекачиваться из внешней памяти в основную и наоборот.

При реализации многих первых систем с разделением времени применялся свопинг. При относительно небольшом количестве абонентов системы со свопингом могли гарантировать приемлемые времена ответа, однако разработчики понимали, что для обслуживания большого числа пользователей потребуются более эффективные методы и средства. На основе систем со свопингом, существовавших в начале 60-х годов, были созданы системы со страничной организацией памяти, которые широко используются в настоящее время. Страничная организация памяти подробно рассматривается в нескольких следующих главах.

В свое время были разработаны и более сложные системы со свопингом, которые позволяли размещать в основной памяти прикладные программы сразу нескольких пользователей. В этих системах программа пользователя выталкивалась из памяти только в случае, когда занимаемое ею место было нужно для размещения другой программы. При наличии достаточно большого объема памяти эти системы позволяли существенно уменьшить время затрачиваемое на свопинг.

Заключение

Исторически сложилось так, что методы организации и управления основной памятью компьютера в существенной степени определяли развитие операционных систем. Организация памяти — это способ представления и использования основной памяти; в настоящей главе было рассмотрено несколько

распространенных способов организации памяти для системы с физической памятью:

- однопрограммные (одноабонентские) системы;
- мультипрограммные системы с фиксированными разделами,
- с трансляцией и загрузкой модулей в абсолютных адресах; 9 мультипрограммные системы с фиксированными разделами,
- с трансляцией и загрузкой перемещаемых модулей; © мультипрограммные системы с переменными разделами;
- системы со свопингом.

Цель стратегий управления памятью заключается в том, чтобы обеспечить наиболее эффективное использование такого дорогостоящего ресурса, каким является основная память, и при этом достигнуть наивысших возможных скоростных характеристик машины. Обсуждались три вида стратегий управления памятью:

- стратегии выборки;
- стратегии размещения;
- стратегии замещения.

Стратегии выборки ставят своей целью определить, когда следует выбирать очередной блок программы или данных для переписи в основную память. Стратегии выборки по запросу (по требованию) предусматривают загрузку в память блоков программ и данных только в тот момент, когда они запрашиваются. В стратегиях выборки с упреждением система пытается предупредить запросы программы пользователя и загружает соответствующие блоки программ и данных в основную память еще до того, как они реально потребуются; таким образом, эти блоки уже будут находиться в основной памяти, так что обращающаяся к ним программа сможет продолжать свое выполнение без задержки.

Стратегии размещения ставят своей целью определить, в какое место основной памяти следует помещать поступающую программу. Рассматривались три стратегии размещения:

- «первый подходящий»;
- «наиболее подходящий»;
- «наименее подходящий».

Выбор первого подходящего свободного участка предусматривает помещение программы в первый найденный свободный участок, который достаточно велик для ее размещения. Выбор наиболее подходящего свободного участка предусматривает помещение программы в «самый тесный» подходящий участок, т. е. в минимальный из имеющихся свободных участков памяти, где программа может уместиться. Выбор наименее подходящего свободного участка предусматривает помещение блока программы или данных в имеющийся свободный участок максимального размера. Выбор первого подходящего участка характеризуется малыми издержками. Выбор наиболее подходящего участка большинству людей интуитивно кажется наиболее выгодным. Выбор наименее подходящего участка имеет то преимущество, что он не оставляет в памяти маленьких дыр.

С определенного времени системы памяти компьютеров становятся иерархическими, распадаясь на несколько уровней. Внешняя, или вторичная, память имеет большую емкость, но стоит относительно недорого. Основная, или первичная, память стоит дороже и имеет меньшую емкость, но более высокие скоростные характеристики, чем вторичная память. Чтобы центральный процессор мог работать с программами и данными, они должны размещаться в основной памяти. Программы и данные, которые в настоящий момент не используются, могут

храниться во внешней памяти, как правило, на магнитных дисках или барабанах. Кэш-память стоит исключительно дорого, однако обеспечивает высокоскоростное выполнение программных команд и обращение к элементам данных.

Системы со связным распределением памяти требуют, чтобы вся программа занимала один блок смежных ячеек памяти. Системы с несвязным распределением допускают, чтобы программы и данные разделялись на ряд блоков меньшего размера, которые могут помещаться в любые свободные участки памяти, даже если эти свободные участки не соседствуют друг с другом. Системы виртуальной памяти, как правило, предусматривают выделение несвязных сегментов памяти и будут рассматриваться в последующих главах.

Первые вычислительные машины предоставлялись в каждый момент времени одному пользователю. Эти однопрограммные машины содержали самые примитивные операционные системы, как правило, системы управления вводом-выводом, существенно упрощающие программирование операций ввода-вывода на машинном уровне. В однопрограммных системах защита памяти осуществлялась при помощи одного граничного регистра, который позволял отделить программу пользователя от операционной системы. Чтобы уменьшить потери машинного времени при переходе с задания на задание и благодаря этому добиться повышения производительности, были разработаны системы однопоточковой пакетной обработки.

Со временем стало ясно, что несколько пользователей могли бы одновременно работать с одной машиной, однако необходимо было обеспечить их защиту друг от друга. В системах мультипрограммирования с фиксированными разделами, с трансляцией и загрузкой модулей в абсолютных адресах программы пользователя подготавливаются таким образом, чтобы выполняться в конкретных разделах. Если раздел, предусмотренный для конкретной программы пользователя, занят, то этому пользователю приходится ждать, даже если имеется другой достаточно большой свободный раздел. В мультипрограммных системах с фиксированными разделами, с трансляцией и загрузкой перемещаемых модулей программу можно загружать в любой свободный раздел, размеры которого это позволяют.

В мультипрограммных системах со связным распределением памяти защита осуществляется при помощи двух граничных регистров, которые входят в состав ЦП. Каждый раз, когда программа обращается к памяти, система проверяет, нет ли здесь попытки выйти за пределы пространства адресов основной памяти, разрешенного для доступа со стороны этой программы. Пользователь, которому нужно войти в операционную систему для выполнения операций ввода-вывода или других функций, делает это при помощи команды вызова супервизора. Эта команда дает программе пользователя возможность пересечь границу операционной системы, чтобы получить определенные услуги. После того как операционная система окажет эти услуги, она возвращает управление программе пользователя.

В мультипрограммных системах с фиксированными разделами значительный объем основной памяти теряется, когда программы пользователя оказываются много меньшими, чем выделяемые для них разделы. В мультипрограммных системах с переменными разделами программа пользователя получает ровно столько памяти, сколько ей требуется (в пределах размера имеющейся основной памяти). Однако в этих системах после завершения выполнения программ пользователя по всей основной памяти остаются разбросанными многочисленные «дыры».

Соседние «дыры» объединяются, чтобы сформировать максимально большие по размеру связные (непрерывные) свободные участки основной памяти. Бывают случаи, когда поступает задание, для которого требуется память, большая по

размеру, чем любой из имеющихся индивидуальных свободных участков, но меньшая суммы всех свободных участков. В этом случае содержимое памяти можно уплотнить, чтобы создать свободный участок, достаточно большой для размещения данной программы. При уплотнении памяти все программы пользователя перемещаются таким образом, чтобы они оказались соседними друг с другом и остался один свободный большой участок памяти.

Как только стратегические средства выборки определяют, что в основную память необходимо ввести новую программу, стратегические средства размещения должны решить, в какое место ее записать. Иногда бывает так, что свободного участка в памяти нет, так что какую-то программу необходимо удалить из основной памяти, чтобы освободить место для записи новой. Подобное решение принимают стратегические средства замещения информации в памяти. Мультипрограммный режим можно реализовать также в системе со свопингом, где в основной памяти в каждый конкретный момент времени находится только одна программа пользователя. Эта программа пользователя выполняется до тех пор, пока это возможно. Затем она «вытаскивается» из основной памяти и вместо нее «втаскивается» следующая программа пользователя, которая также выполняется до тех пор, пока это возможно. Такой метод свопинга эффективен, когда основная память имеет довольно ограниченные размеры. Он использовался во многих первых системах разделения времени с относительно небольшим числом пользователей. Более сложные системы со свопингом позволяют размещать в памяти сразу несколько программ.

Терминология

внешняя, вторичная, память (secondary storage)

выборка (информации) по запросу, по требованию (demand fetch)

заключительное время, время освобождения машины для выполнения следующего задания (job teardown)

защита памяти (storage protection)

иерархия памяти (storage hierarchy)

команда вызова супервизора (supervisor call(SVC)instruction)

кэш-память (cache storage)

мультипрограммирование на физической памяти (real storage multiprogramming)

мультипрограммирование с переменными разделами (variable partition multiprogramming)

мультипрограммирование с фиксированными разделами, с трансляцией и загрузкой модулей в абсолютных адресах (fixed partition multiprogramming with absolute translation and loading)

мультипрограммирование с фиксированными разделами, с трансляцией и загрузкой перемещаемых модулей (fixed partition multiprogramming with relocatable translation and loading)

несвязное распределение памяти (noncontiguous storage allocation)

объединение соседних свободных участков («дыр») памяти, слияние «дыр» (coalescing holes)

оверлей, оверлейный режим, режим работы с наложением, перекрытием (overlay)

однопрограммные, одноабонентские системы (машины) (single user dedicated systems)

организация памяти (storage organization)

основная, первичная, оперативная память (primary storage)

подготовительное время, время подготовки машины к выполнению задания (job setup)

регистры границ, граничных адресов; граничные регистры (boundary registers)

«сбор мусора» (garbage collection)

свободные участки памяти, «дыры» (holes)

свопинг (swapping)

связное распределение памяти (contiguous storage allocation)

сегменты (segments)

система управления вводом-выводом (IOCS) (input/output control system)

системы пакетной обработки с одним потоком заданий, однопотоковые системы пакетной обработки (single-stream batch processing systems)

список свободной памяти (free storage list)

стратегия выборки (fetch strategy)

стратегия замещения (replacement strategy)

стратегия размещения (placement strategy)

стратегия размещения с выбором наиболее подходящего (best-fit storage placement strategy)

стратегия размещения с выбором наименее подходящего (worst-fit storage placement strategy)

стратегия размещения с выбором первого подходящего (first-fit storage placement strategy)

стратегии управления памятью (storage management strategies)

уплотнение памяти (compaction of storage)

управление памятью (storage management)

упреждающая выборка, выборка с упреждением (anticipatory fetch)

«утряска» памяти («burping» the storage)

физическая, реальная, оперативная память (real storage)

фрагментация (fragmentation)

Упражнения

7.1 В системах с иерархической памятью перемещение программ и данных между устройствами памяти различных уровней иерархии сопряжено с определенными накладными расходами. Объясните, почему эти накладные расходы более чем окупаются преимуществами, которыми обладают подобные системы.

7.2 Определенная вычислительная машина может выполнять программы, располагающиеся либо в ее основной памяти, либо в кэш-памяти. Программы могут переписываться из основной памяти в кэш-память и обратно только блоками фиксированного размера. Накладные расходы на каждую подобную перепись являются постоянными. Программы, находящиеся в кэш-памяти, выполняются быстрее, чем в основной памяти.

а) При каких обстоятельствах следует выполнять программы, находящиеся в кэш-памяти?

б) При каких обстоятельствах следует выполнять программы, находящиеся в основной памяти?

в) Предположим, что вам поручили определить, какого размера кэш-память приобрести для вашей вычислительной машины. Вы знаете о том, что кэш-память стоит дорого, но обеспечивает гораздо большую скорость выполнения программ, чем основная память. Какие соображения повлияют на ваше решение?

7.3 Почему выборка информации по запросу в течение столь длительного времени считалась наиболее рациональной? Почему стратегии выборки с упреждением в настоящее время привлекают к себе гораздо большее внимание, чем 10 лет назад?

7.4 Обсудите, каким образом происходит фрагментация памяти для каждой из схем организаций памяти, представленных в настоящей главе.

7.5 При каких обстоятельствах целесообразно использовать оверлейный режим? Когда определенную секцию основной памяти можно использовать как оверлей? Каким образом оверлей влияет на время проектирования программы? Каким образом оверлей влияет на возможности модификации программы?

7.6 Обсудите мотивы и предпосылки для появления мультипрограммирования. Какие характеристики программ и машин делают мультипрограммирование целесообразным? При каких обстоятельствах мультипрограммирование нецелесообразно?

7.7 Предположим, что в вашем распоряжении имеется система с иерархической памятью четырех уровней — она содержит кэш-память, первичную память, вторичную память и третичную память. Предположим, что программы можно

выполнять на любом из этих уровней памяти. Предположим, что каждый уровень имеет идентичную емкость физической памяти и идентичное адресное пространство. Третичная память — самая медленная, а кэш-память — самая быстрая; первичная память в 10 раз медленнее, чем кэш-память, вторичная — в 10 раз медленнее, чем первичная, и третичная — в 10 раз медленнее, чем вторичная. В системе имеется только один ЦП, причем он может выполнять только одну программу в каждый момент времени.

а) Предположим, что программы и данные могут переписываться с любого уровня памяти на любой другой под управлением операционной системы. Время, затрачиваемое на перемещение блоков между двумя конкретными уровнями, определяется скоростью самых низкоуровневых (и самых медленных) устройств, участвующих в обмене. В каком случае операционная система может решить, что программу целесообразно из кэш-памяти переписать сразу во вторичную память, обходя таким образом первичную память? В каких случаях объекты будут переписываться на нижние уровни иерархии? В каких случаях объекты будут переписываться на верхние уровни иерархии?

б) Приведенная выше схема несколько необычна. Как правило, программы и данные перемещаются только между соседними уровнями иерархии. Приведите несколько аргументов против того, чтобы допускать обмены информацией непосредственно между кэш-памятью и любым другим уровнем иерархии, кроме первичной памяти.

7.8 Как системный программист крупного вычислительного комплекса, в котором используется мультипрограммирование с фиксированными разделами, вы получили задание определить, следует ли изменить текущее разбиение памяти системы.

а) Какая информация потребуется вам, чтобы принять обоснованное решение?

б) Если вы легко получите эту информацию, то каким образом вы определите идеальное разбиение памяти на разделы?

в) К каким последствиям может привести изменение разбиения памяти для подобной системы?

7.9 Одна из простых схем реализации перемещения программ в мультипрограммной системе связана с использованием одного регистра перемещения (приписки). Все программы транслируются так, как если бы они размещались в ячейках памяти, начинающихся с нулевой, однако затем каждый адрес, формируемый в процессе выполнения программы, модифицируется путем сложения с регистром перемещения ЦП. Обсудите вопрос использования и управления регистром перемещения при мультипрограммировании с разделами переменных размеров. Каким образом регистр перемещения можно использовать в схеме защиты?

7.10 Стратегии размещения определяют, в какое место основной памяти следует загружать поступающие программы и данные. Предположим, что заданию, ожидающему начала выполнения, требуется память, которая может быть предоставлена немедленно. Следует ли такое задание загрузить и начать его выполнение немедленно?

7.11 Предъявление счетов за ресурсы в мультипрограммных системах может требовать довольно сложных алгоритмов.

а) В однопрограммной системе пользователю обычно приходится платить за всю систему. Предположим, что в настоящий момент на мультипрограммной системе

работает только один пользователь. Должен ли этот пользователь платить за всю систему?

б) Мультипрограммные операционные системы обычно занимают значительные ресурсы машины, поскольку им приходится организовывать работу многих прикладных программ пользователей. Кто должен оплачивать эти ресурсы?

в) Большинство людей сходится во мнении, что плата за использование вычислительной машины должна быть справедливой, однако лишь немногие из нас могут достаточно точно определить, что такое здесь «справедливость». Еще один атрибут схем исчисления арендной платы — но его несколько легче определить — это предсказуемость. Мы хотим быть уверенными в том, что если выполнение некоторого задания обошлось однажды в определенную сумму, то повторное выполнение этого же задания в аналогичных условиях обойдется приблизительно в ту же самую сумму. Предположим, что в мультипрограммной системе плата за выполнение задания будет устанавливаться по календарному времени, т. е. по общему фактическому времени, прошедшему с момента запуска задания до выдачи результатов. Позволит ли подобный подход прогнозировать плату за использование машины? Почему?

7.12 Обсудите достоинства и недостатки системы с несвязным распределением памяти.

7.13 Многие разработчики считают, что операционной системе всегда следует присваивать статус самого полномочного пользователя. Некоторые разработчики полагают, что даже для операционной системы следует устанавливать определенные ограничения, в частности, с точки зрения ее возможности обращаться к различным областям памяти. Обсудите положительные и отрицательные стороны предоставления операционной системе возможностей доступа к полному пространству физических адресов вычислительной машины в любое время.

7.14 Развитие операционных систем, как правило, происходит скорее эволюционно, чем революционно. Опишите основные мотивы разработчиков операционных систем, обусловившие создание системы нового типа из старой, для каждого из следующих переходов:

а) от однопрограммных (одноабонентских) систем к мультипрограммным;

б) от мультипрограммных систем с фиксированными разделами и с трансляцией и загрузкой модулей в абсолютных адресах к аналогичным системам с трансляцией и загрузкой перемещаемых модулей;

в) от мультипрограммных систем с фиксированными разделами к системам с переменными разделами;

г) от систем со связным распределением памяти к аналогичным системам с несвязным распределением;

д) от однопрограммных машин с ручным переключением с задания на задание к однопрограммным машинам с однопотоковыми системами пакетной обработки.

Глава 8

Организация виртуальной памяти

Воображение — в действительности не что иное, как вид памяти, освобожденной от уз времени и пространства.

Сэмюель Тэйлор Кольридж

8.1 Введение

8.2 Эволюция видов организации памяти

8.3 Виртуальная память: основные концепции

8.4 Многоуровневая организация памяти

8.5 Поблочное отображение

8.6 Страничная организация: основные концепции

8.6.1 Преобразование адресов страниц прямым отображением

8.6.2 Преобразование адресов страниц ассоциативным отображением

8.6.3 Преобразование адресов страниц комбинированным ассоциативно-прямым отображением

8.6.4 Коллективное использование программ и данных в системе со страничной организацией

8.7 Сегментная организация

8.7.1 Управление доступом в системах с сегментной организацией

8.7.2 Преобразование адресов сегментов прямым отображением

8.7.3 Коллективное использование программ и данных в системе с сегментной организацией

8.8 Системы с комбинированной странично-сегментной организацией

8.8.1 Динамическое преобразование адресов в системах со странично-сегментной организацией

8.8.2 Коллективное использование программ и данных в системе со странично-сегментной организацией

8.1 Введение

Термин виртуальная память обычно ассоциируется с возможностью адресовать пространство памяти, гораздо большее, чем емкость первичной (реальной, физической) памяти конкретной вычислительной машины. Концепция виртуальной памяти является далеко не новой. Впервые она была реализована в вычислительной машине Atlas, созданной в Манчестерском университете в Англии в 1960 г. Однако широкое распространение системы виртуальной памяти получили относительно недавно.

Существуют два наиболее общепринятых способа реализации виртуальной памяти — страничная и сегментная организации памяти; оба они подробно обсуждаются в настоящей главе. В некоторых системах виртуальной памяти применяется либо тот, либо другой из этих способов, а в некоторых — их комбинация.

Все системы виртуальной памяти характеризуются тем отличительным свойством, что адреса, формируемые выполняемыми программами, не обязательно совпадают с существующими адресами первичной памяти. В действительности виртуальные адреса, как правило, представляют гораздо большее множество адресов, чем имеется в первичной памяти.

8.2 Эволюция видов организации памяти

На рис. 8.1 показано, каким образом эволюционировали виды организации памяти — от систем реальной памяти, выделяемой в почти полное распоряжение одного пользователя, до систем виртуальной памяти, сочетающих методы страничной и сегментной организации. Виды организации реальной памяти рассматривались

Реальная	Реальная		Виртуальная		
Однопользовательские системы	Мультипрограммные системы с реальной памятью		Мультипрограммные системы с виртуальной памятью		
	Мультипрограммирование с фиксированными разделами	Мультипрограммирование с переменными разделами	Чистые страницы	Чистые сегменты	Комбинация страничной и сегментной
	Абсолютные модули	Перемещаемые модули			

Рис. 8.1 Эволюция видов организации памяти.

в предыдущей главе. Поскольку виртуальная память является гораздо более сложной, ее рассмотрение разделено на две главы. В настоящей главе рассматриваются различные виды организации виртуальной памяти, а в следующей главе — методы и стратегии управления ими.

8.3 Виртуальная память: основные концепции

Суть концепции виртуальной памяти заключается в том, что адреса, к которым обращается выполняющийся процесс, отделяются от адресов, реально существующих в первичной памяти.

Те адреса, на которые делает ссылки выполняющийся процесс, называются виртуальными адресами, а те адреса, которые существуют в первичной памяти, называются реальными (или физическими) адресами. Диапазон виртуальных адресов, к которым может обращаться выполняющийся процесс, называется пространством виртуальных адресов V этого процесса. Диапазон реальных адресов, существующих в конкретной вычислительной машине, называется пространством реальных адресов R этого компьютера.

Несмотря на то, что процессы обращаются только к виртуальным адресам, в действительности они должны работать с реальной памятью. Таким образом, во время выполнения процесса виртуальные

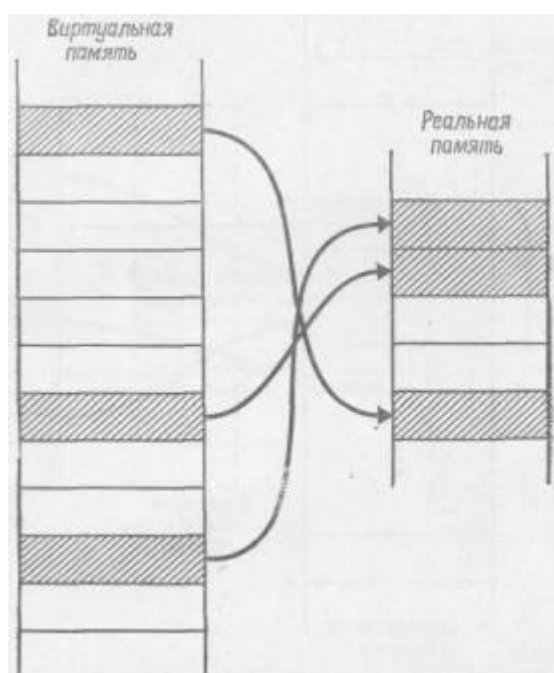


Рис. 8.2 Отображение элементов виртуального адресного пространства на реальное адресное пространство.

адреса необходимо преобразовывать в реальные, причем это нужно делать быстро, ибо в противном случае производительность вычислительной машины будет снижаться до неприемлемых уровней и тем самым практически сведется на нет те преимущества, которые и призвана обеспечить прежде всего концепция виртуальной памяти (рис. 8.2).

Для установления соответствия между виртуальными и реальными адресами разработаны различные способы. Так, механизмы динамического преобразования адресов (DAT) обеспечивают преобразование виртуальных адресов в реальные во время выполнения процесса. Все подобные системы обладают общим свойством: смежные адреса виртуального адресного пространства процесса не обязательно будут смежными в реальной памяти, это свойство называют «искусственной смежностью» (рис. 8.3). Таким образом, пользователь освобождается от необходимости учитывать размещение своих процедур и данных в реальной памяти. Он получает возможность писать программы наиболее естественным образом, прорабатывая только детали

алгоритма и структуры программы и игнорируя конкретные особенности структуры аппаратных

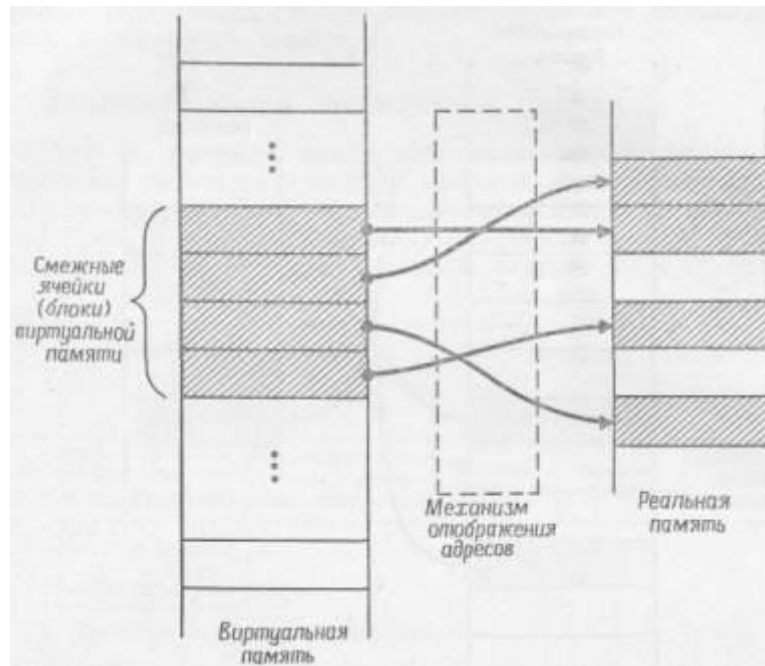


Рис. 8.3 Искусственная смежность. Элементы, которые являются смежными в виртуальной памяти, отображаются не обязательно на смежные элементы реальной памяти.

средств, служащих для выполнения программы. При этом компьютер рассматривается (или может рассматриваться) только как логическое средство, обеспечивающее реализацию необходимых алгоритмов, а не как физическая машина с уникальными характеристиками, часть которых может лишь затруднить процесс проектирования программы.

8.4 Многоуровневая организация памяти

Если мы предусматриваем, что пространство виртуальных адресов пользователя будет больше пространства реальных адресов, и, естественно, если мы ориентируемся на то, что система будет эффективно работать в мультипрограммном режиме с совместным использованием (разделением) ресурсов реальной памяти многими пользователями, то мы должны обеспечить средства хранения программ и данных в большой вспомогательной памяти. Обычно для этого применяется двухуровневая схема построения памяти (рис. 8.4). Первый уровень — это реальная память, в которой находятся выполняемые процессы и в которой должны размещаться данные, чтобы процесс во время работы мог к ним обращаться.

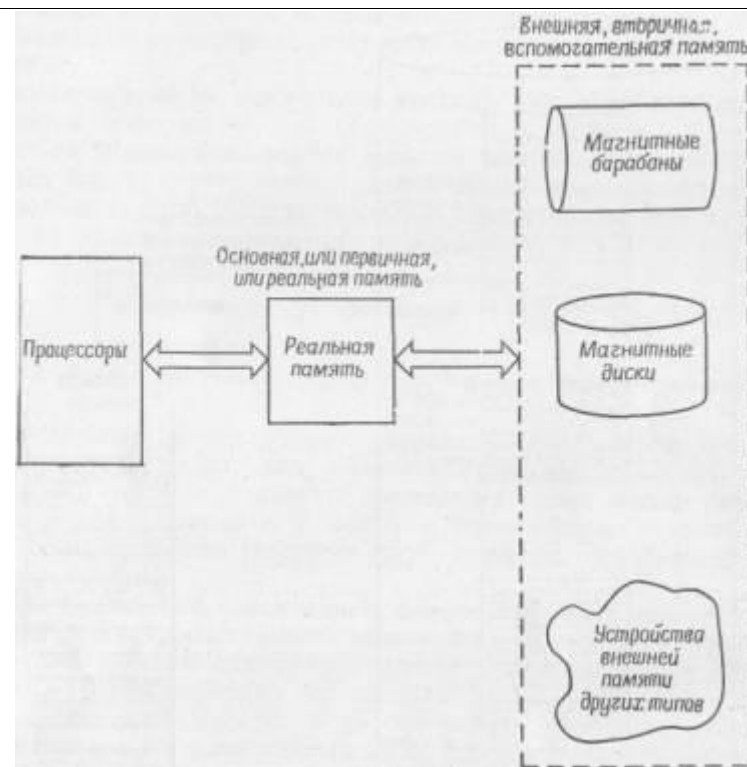


Рис. 8.4 Двухуровневая память.

Второй уровень — это внешняя память большой емкости, например диски или барабаны, способные хранить программы и данные, которые не могут все сразу уместиться в реальной памяти ограниченной емкости. Память этого второго уровня, как правило, называется вторичной, или внешней. Чтобы обеспечить возможность выполнения процесса, его код и данные вводятся в основную память. В настоящей и следующей главах будет подробно описано, каким образом это делается.

Поскольку реальная память разделяется между многими процессами и поскольку каждый процесс может иметь гораздо большее пространство виртуальных адресов, чем реальная память, то в текущий момент времени в реальной памяти имеется возможность держать лишь небольшую часть программных кодов и данных каждого процесса. На рис. 8.5 показана двухуровневая система памяти, в которой реальная память содержит лишь определенные элементы из виртуальных памяти различных пользователей.

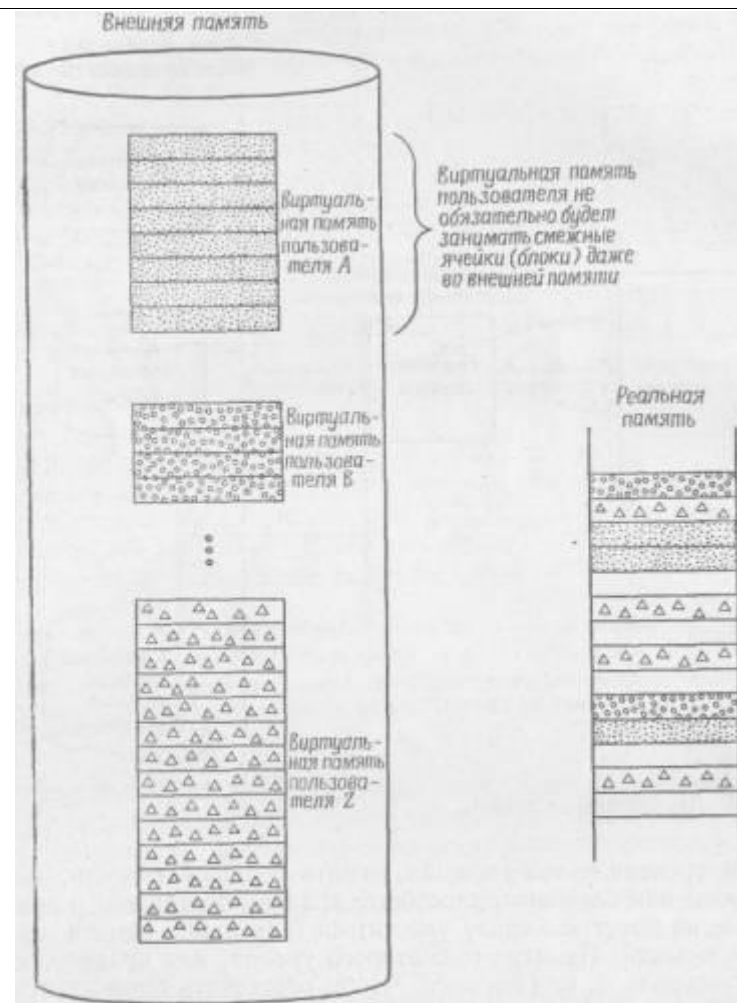


Рис. 8.5 Размещение элементов виртуальной памяти различных пользователей в реальной памяти.

8.5 Поблочное отображение

Механизм динамического преобразования адресов должен вести таблицы, показывающие, какие ячейки виртуальной памяти в текущий момент времени находятся в реальной памяти и где именно они размещаются. Если бы такое отображение осуществлялось пословно или побайтно, то информация об отображении была бы столь велика, что для ее хранения потребовалось бы столько же или даже больше реальной памяти, чем для самих процессов. Поэтому, чтобы реализация виртуальной памяти имела смысл, необходим метод, позволяющий существенно сократить объем информации отображения.

Поскольку мы не можем идти на индивидуальное отображение элементов информации, мы группируем их в блоки, и система следит за тем, в каких местах реальной памяти размещаются различные блоки виртуальной памяти. Чем больше размер блока, тем меньшую долю емкости реальной памяти приходится затрачивать на хранение информации отображения.

Номер блока b	Смещение d	Виртуальный адрес $V=(b,d)$
-----------------	--------------	-----------------------------

Рис. 8.6 Формат виртуального адреса в системе поблочного отображения.

Увеличение размера блоков приводит к уменьшению дополнительных затрат памяти для механизма отображения. Однако крупные блоки требуют большего времени на обмен между внешней и первичной памятью и с большей вероятностью ограничивают количество процессов, которые могут совместно использовать первичную память.

При реализации виртуальной памяти возникает вопрос о том, следует ли все блоки делать одинакового или разных размеров. Если блоки имеют одинаковый размер, они называются страницами, а соответствующая организация виртуальной памяти называется страничной. Если блоки могут быть различных размеров, они называются сегментами, а соответствующая организация виртуальной памяти называется сегментной. В некоторых системах оба этих подхода комбинируются, т. е. сегменты реализуются как объекты переменных размеров, формируемые из страниц фиксированного размера.

Адреса в системе поблочного отображения являются двухкомпонентными («двумерными»). Чтобы обратиться к конкретному элементу данных, программа указывает блок, в котором этот элемент располагается и смещение этого элемента относительно начала блока (рис. 8.6). Виртуальный адрес и указывается при помощи упорядоченной пары (b, d) , где b — номер блока, в котором размещается соответствующий элемент, а d — смещение относительно начального адреса этого блока.

Преобразование адреса виртуальной памяти $V=(b, d)$ в адрес реальной памяти r осуществляется следующим образом (рис. 8.7). Каждый процесс имеет собственную таблицу отображения блоков, которую система ведет в реальной памяти. Реальный адрес a этой таблицы загружается в специальный регистр центрального процессора, называемый *регистром начального адреса таблицы блоков*. Таблица отображения блоков содержит по одной строке для каждого блока процесса, причем эти строки идут в последовательном порядке, сначала блок 0, затем блок 1 и т. д. Номер блока b суммируется с базовым адресом a таблицы, образуя реальный адрес строки

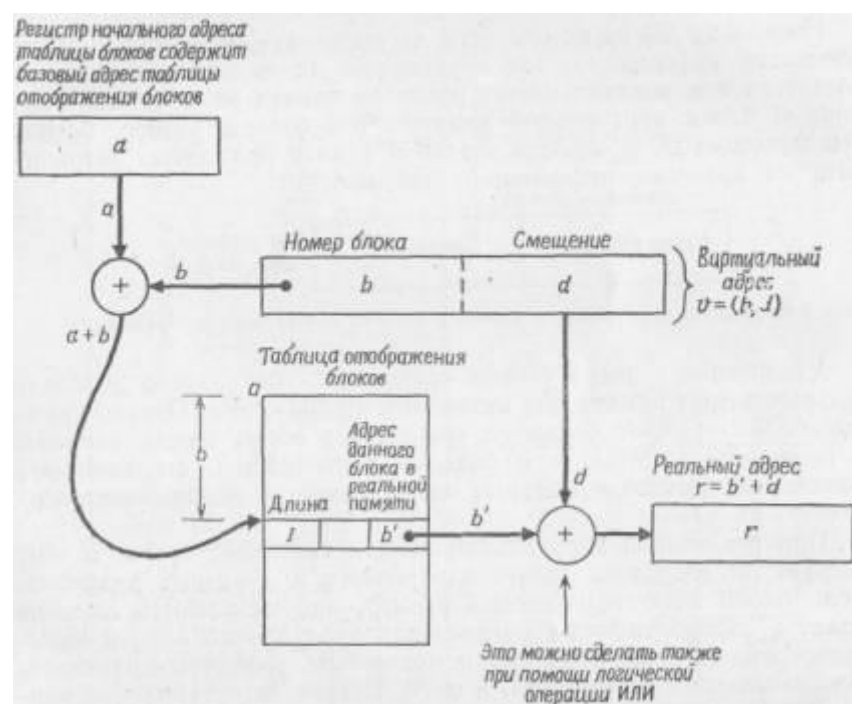


Рис. 8.7 Преобразование виртуального адреса при поблочном отображении.

таблицы для блока b . Эта строка содержит реальный адрес b' блока b в реальной памяти. К этому начальному адресу b' прибавляется смещение d , так что образуется нужный реальный адрес $r=b'+d$.

Все методы поблочного отображения, применяемые в системах с сегментной, страничной и комбинированной странично-сегментной организациями, подобны схеме отображения, которая показана на рис. 8.7.

Важно отметить, что поблочное отображение осуществляется динамически во время выполнения процесса. Если механизм отображения реализован недостаточно эффективно, то накладные расходы могут привести к такому ухудшению характеристик системы, что будут практически сведены на нет преимущества, достигаемые благодаря использованию виртуальной памяти.

8.6 Страничная организация: основные концепции

Памятуя о том, насколько сложнее обращаться с блоками переменных размеров при мультипрограммировании переменными разделами, давайте начнем с рассмотрения поблочного отображения

Номер страницы p	Смещение d	Виртуальный адрес $V=(p,d)$
-----------------------	--------------	--------------------------------

Рис. 8.8 Формат виртуального адреса в системе с чисто страничной организацией.

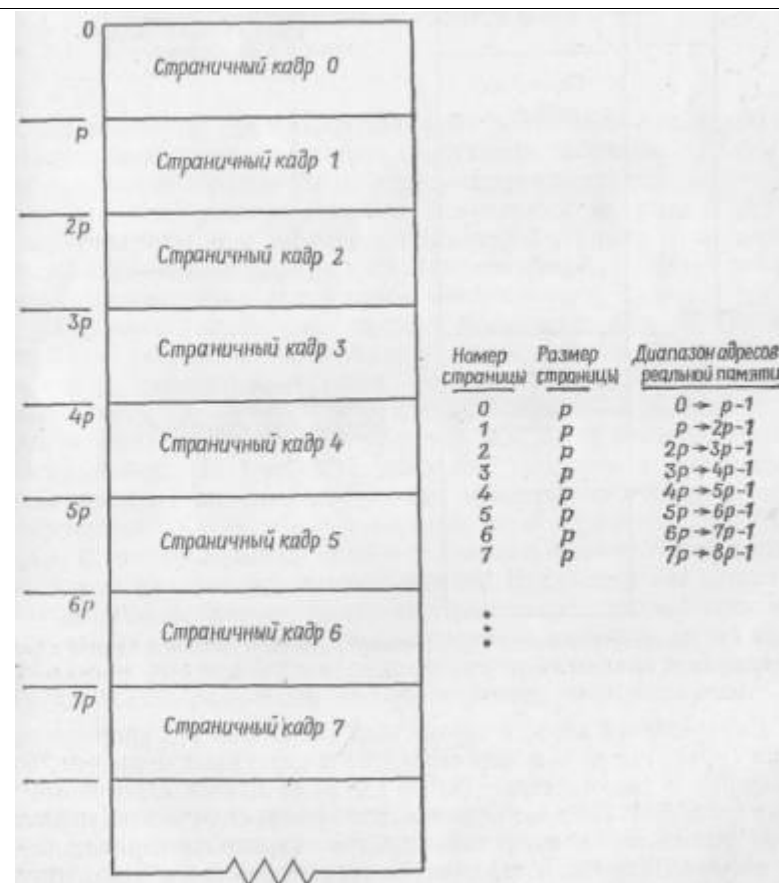


Рис. 8.9 Разделение реальной памяти на страничные кадры.

при фиксированном размере блока, т. е. со страничной организацией памяти. В настоящем разделе мы будем рассматривать чисто страничную, а не странично-сегментную организацию.

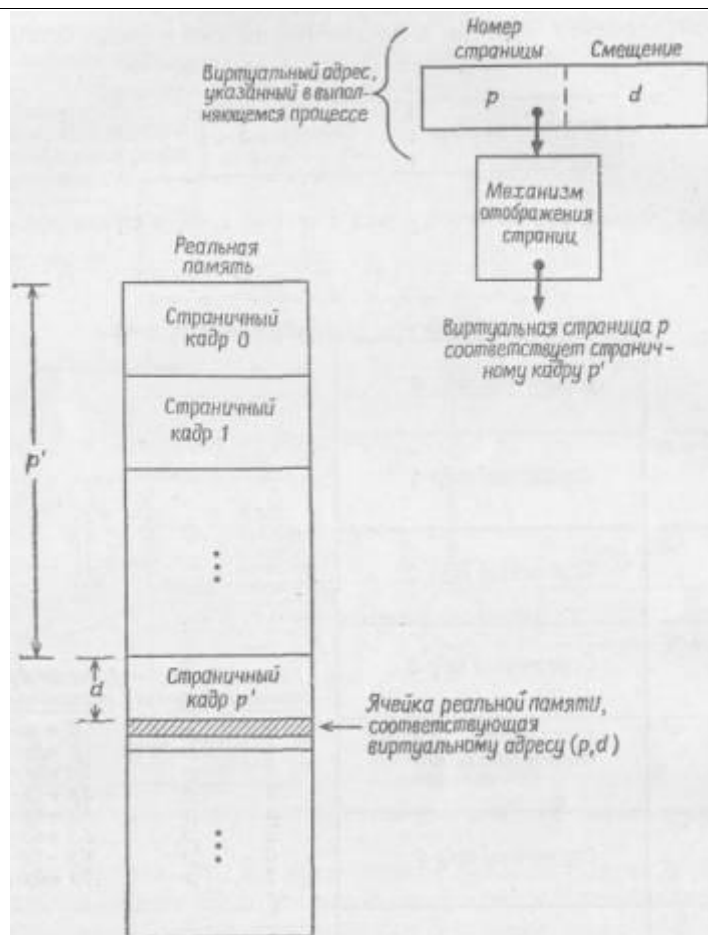


Рис. 8.10 Соответствие между адресами виртуальной и реальной памяти в системе со страничной организацией.

Виртуальный адрес в страничной системе — это упорядоченная пара (p, d) , где p — номер страницы в виртуальной памяти, а d — смещение в рамках страницы p , где размещается адресуемый элемент (рис. 8.8). Процесс может выполняться, если его текущая страница находится в первичной памяти. Страницы переписываются из внешней памяти в первичную и размещаются в ней в блоках, называемых страничными кадрами и имеющих точно такой же размер, как поступающие страницы. Страничные кадры начинаются с адресов первичной памяти, кратных фиксированному размеру страницы (рис. 8.9). Поступающая страница может быть помещена в любой свободный страничный кадр.

Бит-признак присутствия страницы	Адрес внешней памяти (если страницы в реальной памяти нет)	Номер страничного кадра (если страница находится в реальной памяти)
r	s	p'

$r = 0$, если страницы в реальной памяти нет
 $r = 1$, если страница находится в реальной памяти

Рис. 8.11 Строка таблицы страниц.

Динамическое преобразование адресов в системе со страничной организацией осуществляется следующим образом. Выполняющийся процесс обращается по адресу виртуальной памяти $V=(p, d)$. Механизм отображения страниц, показанный на рис. 8.10, ищет номер страницы p в таблице отображения страниц и определяет, что эта страница находится в страничном кадре p' . Адрес реальной памяти формируется затем путем конкатенации p' и d .

Рассмотрим теперь этот процесс более подробно. В частности, поскольку обычно не все страницы процесса находятся в первичной памяти одновременно, таблица отображения страниц должна указывать, есть ли адресуемая страница в первичной памяти, и если есть, то где именно, а если нет, то в каком месте внешней памяти ее можно найти. На рис. 8.11 показана типичная строка таблицы отображения страниц. Бит-признак присутствия страницы r устанавливается в 0, если данной страницы в первичной памяти нет, и в 1, если эта страница есть в первичной памяти. Если страницы нет, то s — ее адрес во внешней памяти. Если страница есть в первичной памяти, то p' — номер ее страничного кадра. Отметим, что p' — это не действительный адрес первичной памяти. Адрес первичной памяти a , с которого начинается страничный кадр p' (если размер страницы равняется p), определяется произведением

$$a=(p)(p')$$

(если страничные кадры имеют номера 0, 1, 2 и т. д.).

В целях экономии емкости первичной памяти в таблицу отображения страниц можно не включать адреса страниц во внешней памяти.

8.6.1 Преобразование адресов страниц прямым отображением

В этом и нескольких следующих разделах мы рассмотрим несколько способов преобразования адресов страниц. Рассмотрим вначале прямое отображение, которое иллюстрирует рис. 8.12.

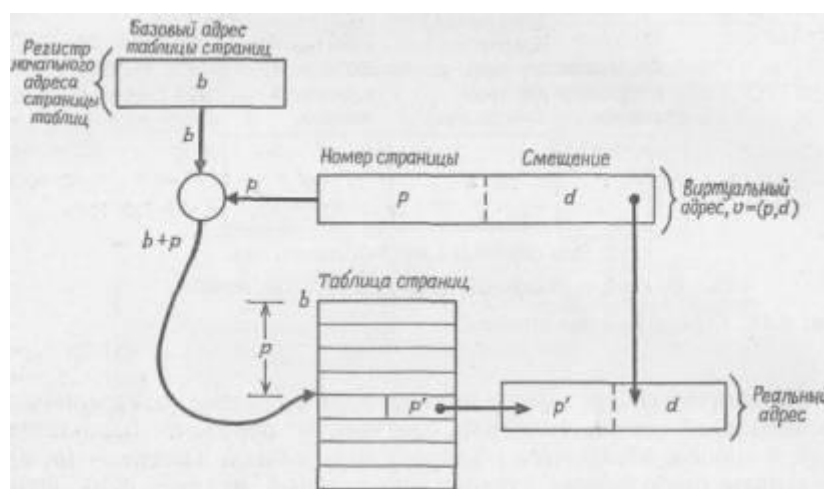


Рис. 8.12 Преобразование адреса страницы путем прямого отображения.

Выполняющийся процесс делает ссылку по виртуальному адресу $V=(p, d)$. Прежде

чем процесс начнет выполняться, операционная система загружает адрес первичной памяти для таблицы отображения страниц в регистр начального адреса этой таблицы. Этот базовый адрес b таблицы прибавляется к номеру страницы p , образуя адрес первичной памяти $b+p$ для строки страницы p в таблице. В этой строке указано, что виртуальной странице p соответствует страничный кадр p' . Затем к значению p' приформировывается (путем конкатенации) смещение d , так что образуется реальный адрес g . Такой подход называется способом прямого отображения потому, что таблица отображения страниц содержит отдельную строку для каждой страницы виртуальной памяти данного процесса. Если процесс имеет в виртуальной памяти n страниц, то таблица при прямом отображении для этого процесса содержит последовательные строки для страницы 0, страницы 1, страницы 2, ..., страницы $n-1$.

Как преобразуемый виртуальный адрес, так и базовый адрес таблицы отображения страниц хранятся в высокоскоростных регистрах управляющего процессора, так что операции с их участием можно выполнять очень быстро внутри командного цикла. Однако таблица прямого отображения страниц, которая может быть довольно большой, обычно находится в первичной памяти. Следовательно, для обращения к этой таблице требуется один полный цикл выборки из первичной памяти. Поскольку время выборки из первичной памяти составляет, как правило, наибольшую долю цикла выполнения команды и поскольку нам требуется еще один цикл обращения к первичной памяти для выборки соответствующей строки таблицы, это означает, что использование метода прямого отображения для преобразования адресов страниц может вызвать снижение скорости вычислительной машины при выполнении программ почти вдвое! Это, естественно, недопустимо. Поэтому необходимы более высокоскоростные способы преобразования адресов. В то же время это не означает, что способ прямого отображения совершенно неприемлем, например некоторые системы с успехом его реализуют, размещая целиком всю таблицу в сверхвысокоскоростной кэш-памяти.

8.6.2 Преобразование адресов страниц ассоциативным отображением

Один из методов ускорения динамического преобразования адресов заключается в том, чтобы всю таблицу преобразования страниц разместить в ассоциативной памяти, длительность цикла которой, быть может, на порядок меньше, чем у первичной памяти. На рис. 8.13 показано, каким образом осуществляется динамическое преобразование адресов с чисто ассоциативным отображением. Работающая программа обращается по виртуальному адресу $V=(p, d)$. Все строки таблицы, хранящиеся в ассоциативной памяти, одновременно сравниваются с адресом страницы p . Ассоциативная память выдает значение p' как адрес страничного кадра, соответствующего странице p , и путем конкатенации p' с значением смещения d формируется реальный адрес g . Отметим, что стрелки, подведенные



Рис. 8.13 Преобразование адреса страницы при чисто ассоциативном отображении.

на рисунке к ассоциативной памяти, фактически затрагивают каждый элемент таблицы. Это означает, что каждый элемент ассоциативной памяти одновременно анализируется на совпадение с адресом p . Именно из-за этого ассоциативная память является столь дорогостоящей.

Здесь опять-таки возникает дилемма: чтобы концепцию виртуальной памяти можно было реализовать на практике, необходимо обеспечить быстрое динамическое преобразование адресов. Однако применение кэш-памяти для реализации чисто прямого отображения или ассоциативной памяти для реализации чисто ассоциативного отображения обходится слишком дорого. Нам нужна некая компромиссная схема, которая обладала бы основными преимуществами методов, предусматривающих использование кэш-памяти или ассоциативной памяти, но при этом обходилась бы гораздо дешевле.

8.6.3 Преобразование адресов страниц комбинированным ассоциативно-прямым отображением

До сих пор в наших рассуждениях мы ориентировались преимущественно на аппаратные средства компьютера, необходимые для эффективной реализации виртуальной памяти. При этом мы рассматривали аппаратные средства скорее логически, чем физически. Нас интересует не конкретное построение устройств, а их функциональная организация и относительные скорости. Именно таким и должен, как правило, видеть аппаратные средства разработчик операционных систем, особенно в тех случаях, когда конструкции аппаратных средств находятся на этапе отработки и могут изменяться.

В течение последних трех десятилетий аппаратные средства вычислительной техники развивались гораздо более высокими темпами, чем программное обеспечение. Сейчас в своем развитии они достигли фактически такого уровня, что обычные программисты не хотят привязываться к конкретной технике, поскольку ожидают, что весьма скоро появятся более совершенные технические средства. Однако разработчики операционных систем, как правило, практически не имеют выбора в этом смысле, поскольку им приходится создавать операционные системы с использованием существующей аппаратуры. При этом они должны также учитывать реальные возможности и экономические характеристики современной техники. В настоящее время кэш-память и ассоциативная память стоят гораздо дороже, чем первичная память произвольного доступа. Это вынуждает нас идти на компромиссное решение при реализации механизма отображения страниц.

Этот механизм предусматривает использование ассоциативной памяти, способной хранить лишь небольшую часть полной таблицы отображения страниц для процесса (рис. 8.14). В строках этой таблицы отображаются только те страницы, к которым были последние ссылки,— это делается с учетом того эвристического соображения, что к странице, к которой осуществлялось обращение в недалеком прошлом, будет, по всей вероятности, новое обращение в ближайшем будущем. Современные системы, в которых используется частичная ассоциативная таблица страниц, обеспечивают скоростные показатели, составляющие 90 и более процентов показателей, возможных при полной ассоциативной таблице.

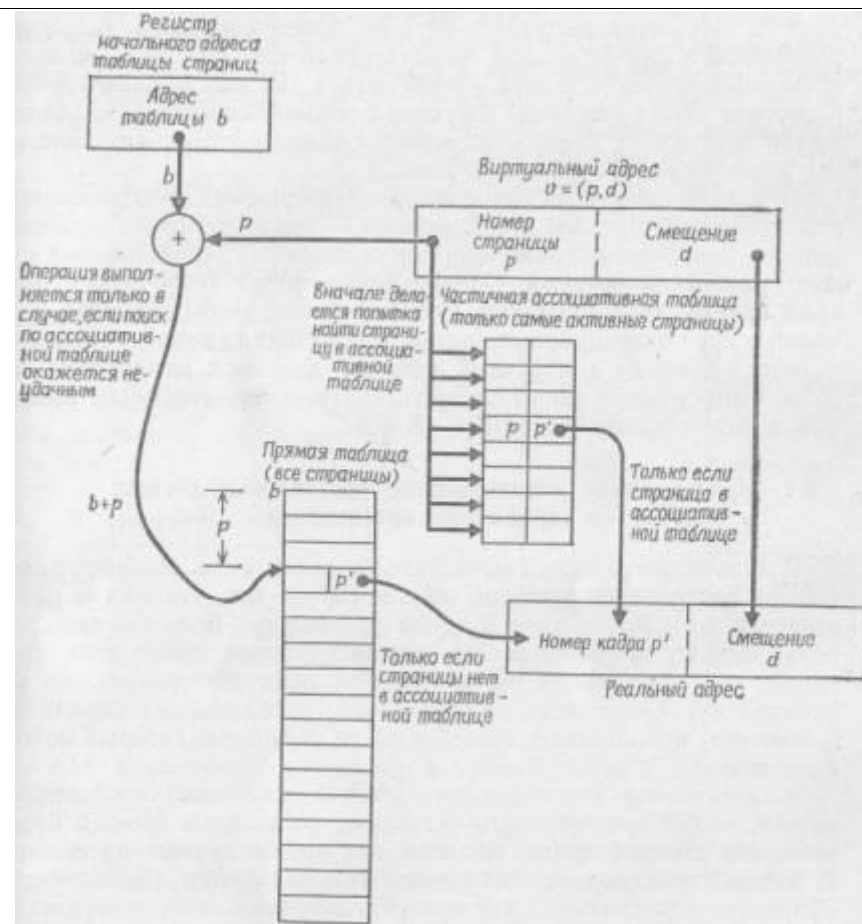


Рис. 8.14 Преобразование адреса страницы при комбинированном ассоциативно-прямом отображении.

Динамическое преобразование адресов в данном случае осуществляется следующим образом. Работая по виртуальному адресу $V = (p, d)$. Механизм преобразования адресов вначале пытается найти страницу p в частичной ассоциативной таблице. Если страница p оказывается здесь, то ассоциативная таблица выдает p' как номер кадра, соответствующего виртуальной странице p , и значение p' сцепляется со смещением d , образуя реальный адрес r , соответствующий виртуальному адресу $V = (p, d)$.

Для того чтобы обеспечивать хорошие скоростные показатели, частичная ассоциативная таблица должна быть небольшой. И действительно, системы, в которых этот подход реализуется с использованием всего лишь восьми или шестнадцати регистров ассоциативной памяти, зачастую достигают 90 или более процентов скорости, возможной в случае полной ассоциативной таблицы, которая в современных системах может потребовать в 100 раз большего числа регистров. Это происходит благодаря особому свойству выполняющихся процессов, которое называется локальностью; это явление мы рассмотрим в следующей главе.

Использование комбинированного механизма преобразования с ассоциативно-прямым отображением — это инженерное решение, которое базируется на экономических факторах, свойственных существующей аппаратуре. Однако современный технический прогресс идет быстрыми темпами. Поэтому важно, чтобы студенты, изучающие курс операционных систем, представляли себе перспективы развития техники аппаратных средств. Хорошим источником подобной информации могут служить книги по архитектурам вычислительных систем, например (Ва80).

8.6.4 Коллективное использование программ и данных в системе со страничной организацией

В мультипрограммных вычислительных системах, особенно в системах с разделением времени, обычно бывает так, что многие пользователи выполняют одни и те же программы. Если бы каждому пользователю предоставлялись индивидуальные копии этих программ, то значительная часть основной памяти затрачивалась бы понапрасну. Очевидное решение этой проблемы — коллективно (совместно) использовать (разделять) те страницы, которые можно разделять.

Коллективное использование должно тщательно координироваться, чтобы предотвратить ситуацию, когда один процесс будет изменять данные, считываемые в это время другим процессом. В большинстве современных вычислительных систем, реализующих коллективное использование памяти, программы обычно состоят из двух отдельных частей, области процедур и области данных. Неизменяемые процедуры называются чистыми, или реентерабельными процедурами. Изменяемые данные, естественно, коллективно использовать нельзя; неизменяемые данные (например, постоянную табличную информацию) — можно. Изменяемые процедуры коллективно использовать нельзя.

Все эти рассуждения говорят о том, что каждую страницу памяти необходимо идентифицировать как либо используемую коллективно, либо нет. После того как страницы каждого процесса будут таким образом разделены на две категории, в системе с чисто страничной организацией можно реализовать коллективное использование, как показано на рис. 8.15. Указывая в строках таблицы

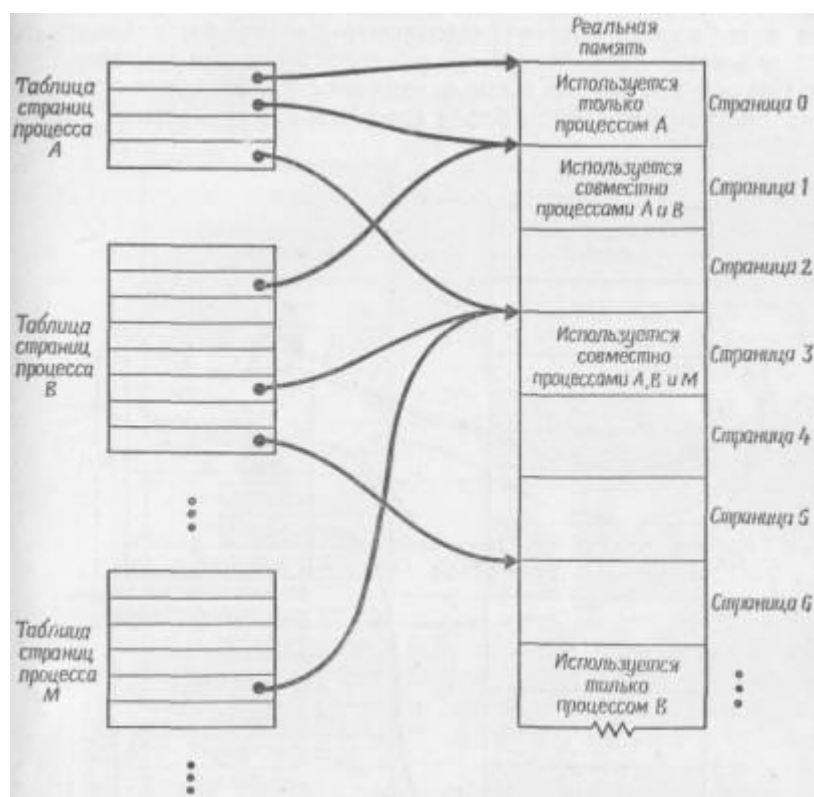


Рис. 8.15 Коллективное использование программ и данных в чисто страничной системе.

страниц различных процессов один и тот же страничный кадр, можно обеспечить совместное использование этого кадра соответствующими процессами. Коллективное использование информации уменьшает объем первичной памяти, необходимой для эффективного выполнения группы процессов, и может обеспечить для данной системы возможность обслуживания большего числа пользователей. В следующей главе будет рассмотрен вопрос о том, каким образом коллективное использование программ и данных отражается на управлении виртуальной памятью.

8.7 Сегментная организация

В главе, посвященной реальной памяти, мы говорили о том, что в системах мультипрограммирования с переменными разделами размещение программ в памяти чаще всего осуществляется в соответствии с выбором «первого подходящего», «наиболее подходящего» или «наименее подходящего» по размеру свободного участка памяти. Однако мы все же были ограничены необходимостью выполнять программы в одном блоке смежных ячеек реальной памяти.

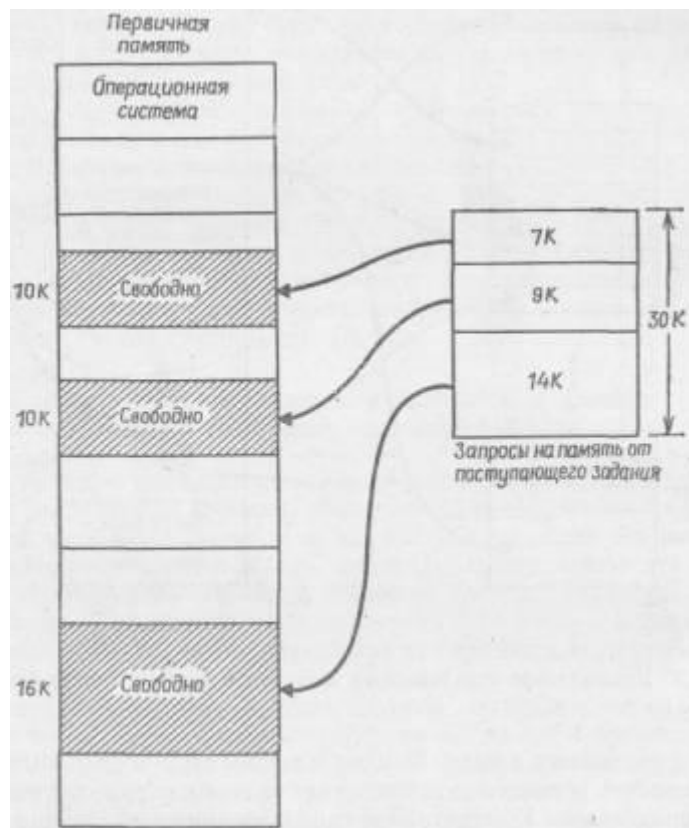


Рис. 8.16 Распределение памяти несмежными блоками.

В системах с сегментной организацией (сегментацией) это ограничение снимается, и программа (и ее данные) может занимать много отдельных блоков первичной памяти (рис. 8.16). Сами блоки не обязательно одинакового размера, все же должны состоять из смежных ячеек, однако отдельные блоки могут размещаться и не рядом друг с другом.

При таком подходе возникает несколько интересных проблем. Например, гораздо более сложной теперь становится проблема защиты программы каждого пользователя от несанкционированного доступа со стороны других пользователей. Парой граничных регистров здесь больше не обойтись. Аналогично становится гораздо труднее ограничивать допустимую область доступа для любой конкретной программы. Один из способов реализации защиты памяти в системах с сегментной организацией — это использование ключей защиты, как показано на рис. 8.17.

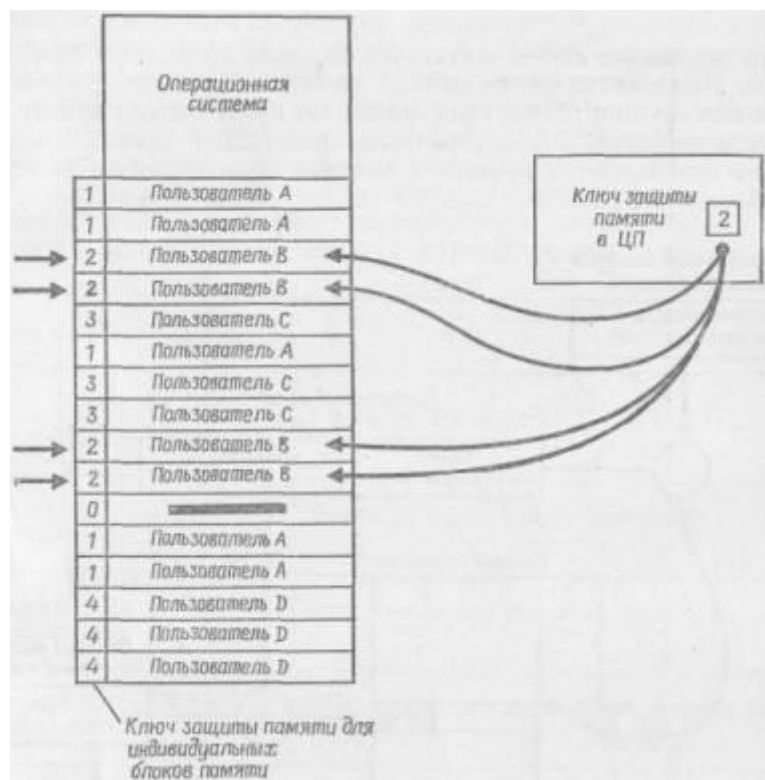


Рис. 8.17 Защита памяти при помощи ключей в мультипрограммных системах с распределением памяти несмежными блоками. Когда в ЦП для ключа защиты памяти устанавливается значение 2, соответствующее пользователю В, программа этого пользователя может обращаться только к тем блокам памяти, которые имеют именно этот ключ защиты (2). Управление ключами защиты осуществляет только операционная система.

Виртуальный адрес в сегментной системе — это упорядоченная пара $V=(s, d)$, где s — номер сегмента виртуальной памяти, а d — смещение в рамках этого сегмента, где находится адресуемый элемент (рис. 8.18). Процесс может выполняться только в случае, если его текущий сегмент (как минимум) размещается в первичной памяти. Сегменты передаются из внешней памяти в первичную целиком. Все ячейки, относящиеся к сегменту, занимают смежные адреса первичной памяти. Поступающий из внешней памяти сегмент может

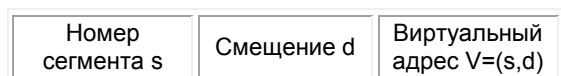


Рис. 8.18 Формат виртуального адреса в чисто сегментной системе.

быть помещен в любой достаточно большой по размеру свободный блок смежных ячеек первичной памяти. Стратегии размещения при сегментации идентичны стратегиям, применяемым при мультипрограммировании с переменными разделами, причем наиболее часто используются принципы «первый подходящий» и «наиболее подходящий».

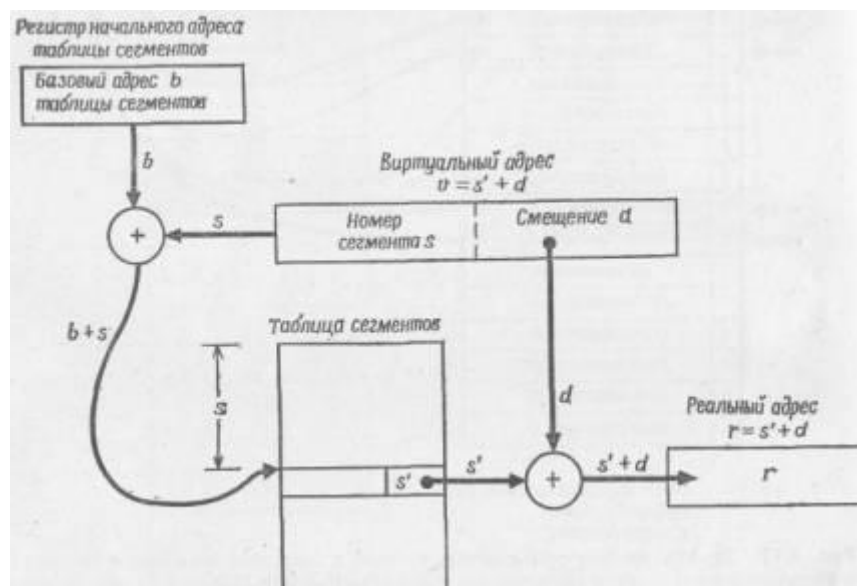


Рис. 8.19 Преобразование виртуального адреса в чисто сегментной системе.

Динамическое преобразование адресов осуществляется следующим образом (рис. 8.19). Выполняющийся процесс выдает адрес виртуальной памяти $V=(s, d)$. Механизм отображения сегментов ищет сегмент s в таблице сегментов и определяет, что этот сегмент находится в реальной памяти, начиная с ячейки s' . Адрес реальной памяти, соответствующий виртуальному адресу $V=(s, d)$, формируется затем путем добавления s' к d . Более подробно эту процедуру мы рассмотрим в нескольких следующих разделах.

8.7.1 Управление доступом в системах с сегментной организацией

Предоставлять каждому процессу неограниченный доступ к любому сегменту в системе нецелесообразно. В действительности одним из достоинств сегментных систем является именно возможность четкого управления доступом. Для этого каждому процессу предоставляются определенные права доступа к каждому сегменту и фактически полностью запрещается доступ ко многим сегментам.

На рис. 8.20 перечислены наиболее распространенные виды управления доступом, применяемые в современных системах. Если процессу разрешено *чтение* сегмента, то процесс может выбрать любой элемент информации, содержащейся в этом сегменте. При желании процесс может сделать полную копию этого сегмента.

Вид доступа	Сокращенное обозначение	Пояснение
-------------	-------------------------	-----------

Read (чтение)	R	Этот блок можно читать
Write (запись)	W	Этот блок можно модифицировать
ExecVte (выполнение)	E	Этот блок можно выполнять (как программу)
Append (дополнение)	A	Этот блок допускает добавление информации в конце

Рис. 8.20 Виды управления доступом.

Если процессу разрешена запись в сегмент, то процесс может изменить любой элемент, содержащийся в сегменте, и поместить в этот сегмент дополнительную информацию. При желании процесс может полностью уничтожить всю информацию сегмента.

Процесс, которому разрешено выполнение сегмента, может работать с этим сегментом как с программой. К сегменту, содержащему данные, доступ для выполнения обычно запрещается.

Процесс, которому разрешено дополнение некоторого сегмента, может записать дополнительную информацию в конце сегмента, но не может изменить существующую информацию.

В системе, предусматривающей все эти четыре вида управления доступом, можно создать 16 различных режимов управления доступом, разрешая или запрещая каждый вид. Некоторые из этих режимов весьма полезны, а некоторые не имеют смысла. Для простоты рассмотрим восемь различных комбинаций видов доступа для чтения, записи и выполнения, как показано на рис. 8.21.

	Чтение	Запись	Выполнение	Пояснение	Применение
Режим 0	Нет	Нет	Нет	Любые виды доступа запрещаются	Для защиты от несанкционированного доступа
Режим 1	Нет	Нет	Да	Только для выполнения	Программа предоставляется пользователям, которые могут ее только выполнять, но не могут ни модифицировать, ни копировать

Режим 2	Нет	Да	Нет	Только для записи	Эти режимы не используются — не имеет смысла предоставлять право записи, но не позволять читать
Режим 3	Нет	Да	Да	Можно писать и выполнять, но нельзя читать	
Режим 4	Да	Нет	Нет	Только для чтения	Для выборки информации
Режим 5	Да	Нет	Да	Для чтения и выполнения	Программу можно копировать и выполнять, но нельзя модифицировать
Режим 6	Да	Да	Нет	Можно читать и писать, но не выполнять	Служит для защиты данных от ошибочной попытки выполнения как программы
Режим 7	Да	Да	Да	Доступ без ограничений	Этот режим доступа предоставляется самым доверенным (привилегированным) пользователям

Рис. 8.21 Комбинирование прав доступа по чтению, записи и выполнению для формирования различных необходимых режимов защиты.

В режиме 0 любые виды доступа к сегменту блокируются. Этот режим необходим для схем защиты от несанкционированного доступа, где к данному сегменту нельзя обращаться со стороны конкретного процесса.

В режиме 1 разрешается доступ к сегменту только для выполнения. Этот режим необходим, когда процессу нужно разрешить использовать программу, содержащуюся в сегменте, но нельзя позволить копировать или изменять ее.

Режимы 2 и 3 практически не используются — не имеет никакого смысла предоставлять процессу право изменять содержимое сегмента, не позволяя в то же время читать тот же сегмент.

Режим 4 разрешает доступ к сегменту только для чтения. Этот режим необходим для информационно-поисковых систем, где процесс должен получать доступ к информации, но не может изменять ее.

Режим 5 разрешает чтение и выполнение. Это необходимо в ситуациях, когда процессу разрешается использовать программу, содержащуюся в сегменте, но нельзя изменять исходную копию. Процесс может, однако, сделать собственную копию этого сегмента, а потом уже ее модифицировать.

Режим 6 разрешает чтение и запись. Это необходимо, когда в сегменте размещаются данные, которые процесс может читать или писать, но которые должны быть защищены от случайной попытки «выполнения» (поскольку этот сегмент не программа).

Режим 7 предоставляет неограниченный доступ к сегменту. Этот режим необходим для того, чтобы позволить процессу полностью распоряжаться своими собственными сегментами и предоставлять ему статус «доверенного лица» для доступа к сегментам других пользователей.

Простой механизм управления доступом, описанный в этом разделе, является основой для сегментной защиты, реализованной во многих реальных системах.

8.7.2 Преобразование адресов сегментов прямым отображением

Как и при страничной организации, в сегментных системах существует много стратегий реализации преобразования адресов сегментов. Это можно делать путем прямого, ассоциативного или комбинированного (ассоциативно-прямого) отображения. Это можно делать при помощи кэш-памяти, достаточно большой для размещения всей таблицы сегментов, или кэш-память можно использовать только для хранения информации о сегментах, обращения к которым производились в самое последнее время. В данном разделе мы рассмотрим определение адресов сегментов путем прямого отображения с ведением полной таблицы сегментов в высокоскоростной кэш-памяти. Формат типичной строки такой таблицы сегментов показан на рис. 8.22.

Рассмотрим вначале случай, когда преобразование адреса происходит нормально, а затем обсудим несколько возможных проблем. Выполняющийся процесс обращается по виртуальному адресу $V=(s, d)$. Номер сегмента s прибавляется к базовому адресу b , находящемуся в регистре начального адреса таблицы сегментов, образуя реальный адрес $b+s$ строки сегмента s в таблице сегментов. Таблица сегментов содержит адрес первичной памяти s' , с которого начинается данный сегмент. Прибавлением смещения d к s' формируется реальный адрес $r=d+s'$, соответствующий виртуальному адресу $V=(s, d)$.

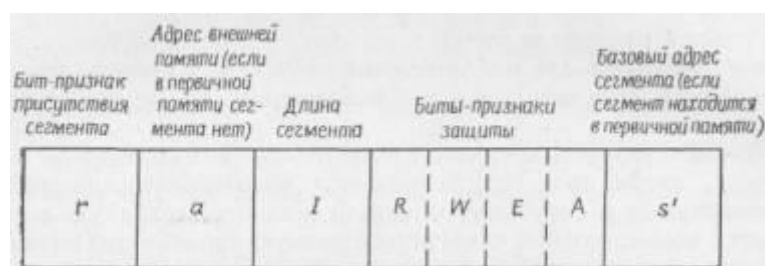


Рис. 8.22 Строка таблицы сегментов; $r=0$, если сегмента нет в первичной памяти, $r=1$, если сегмент в первичной памяти; биты-признаки защиты: (1 — да, 0 — нет), R — доступ для чтения, W — доступ для записи, E — доступ для выполнения, A — доступ для дополнения.

На рис. 8.22 подробно показан формат типичной строки таблицы сегментов. Бит-

признак присутствия r говорит о том, находится или нет данный сегмент в текущий момент времени в первичной памяти. Если сегмент в памяти, то s' — его начальный адрес. Если сегмента в первичной памяти нет, то a — адрес внешней памяти, по которому следует выбрать сегмент, чтобы процесс мог продолжить свое выполнение. Все обращения к сегменту контролируются по размеру — длине сегмента l , чтобы удостовериться в том, что они не приводят к выходу за границы сегмента. Каждое обращение к сегменту контролируется также по битам-признакам защиты, чтобы определить, разрешается ли выполнение соответствующей операции доступа. Таким образом, во время динамического преобразования адресов после нахождения строки в таблице сегментов для конкретного сегмента s прежде всего проверяется признак присутствия r , чтобы установить, находится ли требуемый сегмент в первичной памяти. Если нет, то вырабатывается прерывание по отсутствию сегмента; по этому прерыванию операционная система принимает управление на себя и загружает требуемый сегмент из внешней памяти, где он находится по адресу a . После загрузки сегмента обработка адреса продолжается, смещение d сравнивается с длиной сегмента l . Если d больше l , то вырабатывается прерывание по выходу за пределы сегмента, операционная система принимает управление на себя и прекращает выполнение данного процесса. Если смещение d находится в рамках сегмента, то осуществляется контроль по битам-признакам защиты, чтобы удостовериться в том, что соответствующая операция доступа разрешена. Если да, то, наконец, базовый адрес s' сегмента в первичной памяти суммируется со смещением d и образуется реальный адрес $r = s' + d$, соответствующий виртуальному адресу $V = (s, d)$. Если указанная операция доступа не разрешена, то происходит прерывание по защите сегмента, операционная система принимает управление на себя и прекращает выполнение данного процесса.

8.7.3 Коллективное использование программ и данных в системе с сегментной организацией

Одно из основных преимуществ сегментной организации по сравнению со страничной состоит в том, что эта концепция является скорее логической, чем физической. В своем самом общем виде сегменты не ограничены каким-то определенным размером. Они могут иметь такой размер (естественно, в разумных пределах),



Рис. 8.23 Коллективное использование программ и данных в чисто сегментной системе.

какой нужен. Так, сегмент для размещения некоторого массива будет иметь размер, соответствующий этому массиву. Размеры сегмента, содержащего динамическую структуру данных, могут увеличиваться и уменьшаться в соответствии с расширением и сокращением самой структуры данных. Сегмент, который предназначается для размещения кода процедуры, генерируемого компилятором, будет иметь размер, соответствующий длине этого кода.

Коллективное использование программ и данных в сегментной системе получается гораздо проще, чем в системе с чисто страничной организацией. Если, например, какой-либо массив в чисто страничной системе имеет размер в три с половиной страницы, то вместо

Номер сегмента s	Номер страницы p	Смещение d	Виртуальный адрес $V=(s,d)$
---------------------	---------------------	------------	--------------------------------

Рис. 8.24 Формат виртуального адреса в странично-сегментной системе.

одного указания, что «этот массив используется коллективно», мы должны предусматривать отдельные указания для каждой страницы, которую занимает массив. А работать с неполной страницей может быть весьма затруднительно. Ситуация еще более усугубляется в случае динамической структуры данных. Если динамическая структура данных расширяется с переходом в новую страницу, то признаки совместного использования страниц необходимо перестраивать во время выполнения процессов. А в сегментированной системе, после того как определенные сегменты объявлены как коллективно используемые (разделяемые), их структуры данных могут расширяться и сокращаться произвольно, не меняя логического факта, что они занимают разделяемые сегменты.

На рис. 8.23 показано, каким образом осуществляется коллективное использование в чисто сегментной системе. Два процесса могут совместно использовать некоторый сегмент, если в их таблицах сегментов просто будут строки, указывающие на один и тот же сегмент первичной памяти.

8.8 Системы с комбинированной странично-сегментной организацией

И сегментная, и страничная организации имеют важные достоинства как способы построения виртуальной памяти. Начиная с систем середины 60-х годов, в частности с информационно-вычислительной системы с разделением времени Multics и системы разделения времени TSS корпорации IBM, во многих вычислительных машинах применяется комбинированная странично-сегментная организация памяти (Da68, De71, Do76). Эти системы обладают достоинствами обоих способов реализации виртуальной памяти. Сегменты обычно содержат целое число страниц, причем не обязательно, чтобы все страницы сегмента находились в первичной памяти одновременно, а смежные страницы виртуальной памяти не обязательно должны оказываться смежными в реальной памяти. В системе со странично-сегментной организацией применяется трехкомпонентная (трехмерная) адресация, т. е. адрес

виртуальной памяти V определяется как упорядоченная тройка $V=(s, p, d)$, где s — номер сегмента, p — номер страницы, а d — смещение в рамках страницы, по которому находится нужный элемент (рис. 8.24).

8.8.1 Динамическое преобразование адресов в системах со странично-сегментной организацией

Рассмотрим теперь динамическое преобразование виртуальных адресов в реальные в странично-сегментной системе с применением комбинированного ассоциативно-прямого отображения, как показано на рис. 8.25.

Выполняющийся процесс делает ссылку по виртуальному адресу $V=(s, p, d)$. Самые последние по времени обращения страницы имеют соответствующие строки в ассоциативной таблице. Система производит ассоциативный поиск, пытаясь найти строку с параметрами (s, p) в ассоциативной таблице. Если такая строка здесь обнаруживается, то адрес страничного кадра p' , по которому эта страница размещается в первичной памяти, соединяется со смещением d , образуя реальный адрес r , соответствующий виртуальному адресу V , — и на этом преобразование адреса завершается.

В обычном случае большинство запросов на преобразование адресов удастся удовлетворить подобным ассоциативным поиском. Если же требуемого адреса в ассоциативной памяти нет, то преобразование осуществляется способом полного прямого отображения. Это делается следующим образом: базовый адрес b таблицы сегментов прибавляется к номеру сегмента s , так что образуется адрес $b+s$ строки для сегмента s в таблице сегментов по первичной памяти. В этой строке указывается базовый адрес s' таблицы страниц для сегмента s . Номер страницы p прибавляется к s' , так что образуется адрес $p+s'$ строки в таблице страниц для страницы p сегмента s . Эта таблица позволяет установить, что виртуальной странице p соответствует номер кадра p' . Этот номер кадра соединяется со смещением d , так что образуется реальный адрес r , соответствующий виртуальному адресу $V=(s, p, d)$.

Эта процедура преобразования адресов описана, конечно, в предположении, что каждый необходимый элемент информации находится именно там, где ему положено быть. Однако в процессе преобразования адресов существует много моментов, когда обстоятельства могут складываться не столь благоприятно. Просмотр таблицы сегментов может показать, что сегмента s в первичной памяти нет; при этом возникает прерывание по отсутствию сегмента, операционная система найдет нужный сегмент во внешней памяти, сформирует для него таблицу страниц и загрузит соответствующую страницу в первичную память, быть может, вместо некоторой существующей страницы этого или какого-либо другого процесса. Когда сегмент находится в первичной памяти, обращение к таблице страниц может показать, что нужной страницы в первичной памяти все же нет. При этом произойдет прерывание по отсутствию страницы, операционная система возьмет управление на себя, найдет данную страницу во внешней памяти и загрузит ее в первичную память (быть может, опять-таки с замещением другой страницы). Как и при чисто сегментной системе, адрес виртуальной памяти может выйти за рамки сегмента и произойдет прерывание по выходу за пределы сегмента. Или контроль по битам-признакам защиты может

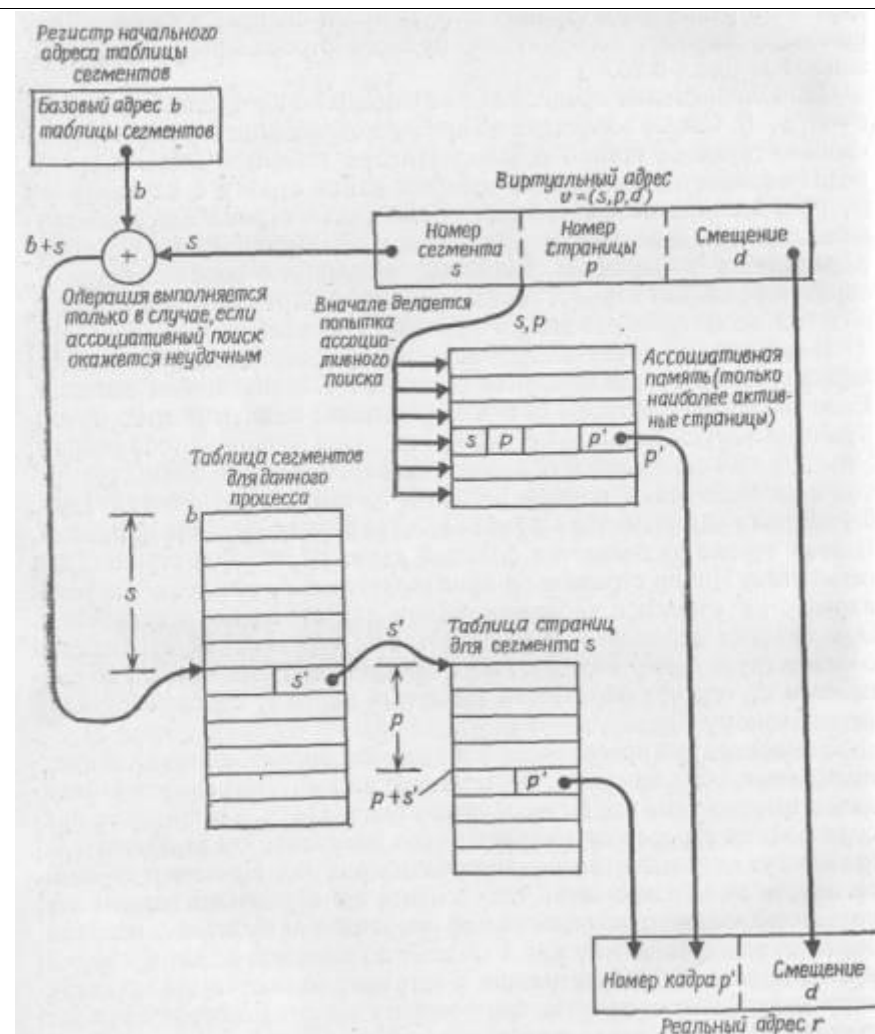


Рис. 8.25 Преобразование виртуального адреса путем комбинированного ассоциативно-прямого отображения в странично-сегментной системе.

выявить, что операция, выполнение которой запрашивается по указанному виртуальному адресу, не разрешена, вследствие чего произойдет прерывание по защите сегмента. Операционная система должна предусматривать обработку всех этих ситуаций.

Ассоциативная память (или аналогично высокоскоростная кэш-память) играет решающую роль в обеспечении эффективной работы механизма динамического преобразования адресов. Если преобразование адресов производить с помощью чисто прямого отображения с ведением в первичной памяти полного набора таблиц соответствия виртуальных и реальных адресов, типичное обращение к виртуальной памяти потребовало бы цикла памяти для доступа к таблице сегментов, второго цикла памяти для доступа к таблице страниц и третьего для доступа к нужному элементу в реальной памяти. Таким образом, каждое обращение к адресуемому элементу занимало бы три цикла памяти, т. е. реальное быстродействие вычислительной системы составило бы лишь приблизительно треть от номинального значения, а две трети затрачивалось бы на преобразование адресов! Интересно, что при всего лишь восьми или шестнадцати ассоциативных регистрах разработчики добиваются для многих систем показателей быстродействия, составляющих 90 и более процентов полных скоростных возможностей их управляющих (центральных) процессоров.

На рис. 8.26 показана подробная структура таблиц, необходимых для

комбинированной странично-сегментной системы. На самом верхнем уровне этой структуры находится таблица процессов, которая содержит по строке для каждого процесса, известного системе. Строка таблицы процессов, соответствующая конкретному процессу, указывает на таблицу сегментов этого процесса. Каждая строка таблицы сегментов процесса указывает на таблицу страниц соответствующего сегмента, а каждая строка таблицы страниц указывает либо на страничный кадр, в котором размещается данная страница, либо на адрес внешней памяти, где можно найти эту страницу. В системе с большим количеством процессов, сегментов и страниц вся эта табличная структура может занимать значительную долю первичной памяти. Здесь следует учитывать, что преобразование адресов производится во время выполнения быстрее, если все таблицы находятся в первичной памяти. Однако чем больше таблиц в памяти, тем меньшее количество процессов система в состоянии поддерживать в текущий период, а это может приводить к снижению пропускной способности. Разработчики операционных систем должны анализировать многие подобные взаимозависимости и принимать тщательно сбалансированные компромиссные решения — только в этом случае можно обеспечить и эффективную загрузку системы, и оперативное предоставление ее услуг пользователям.

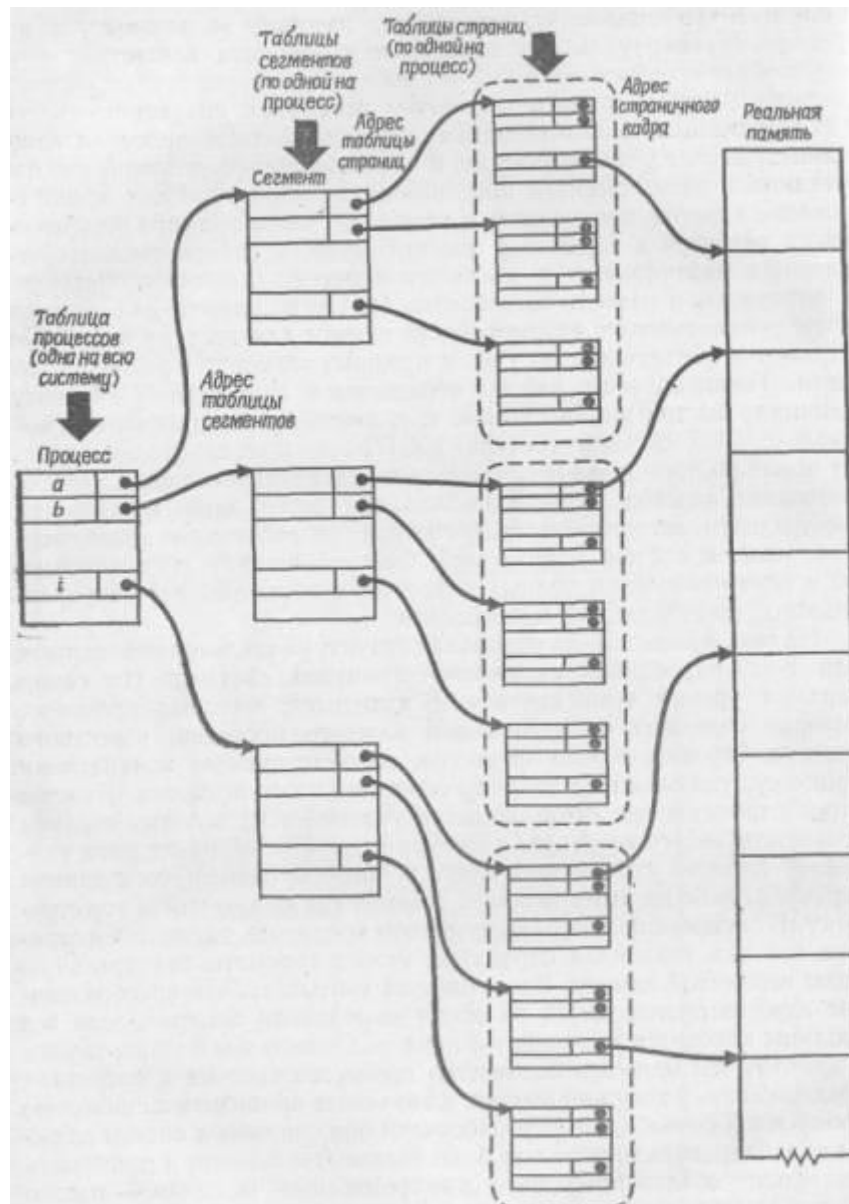


Рис. 8.26 Структура таблиц для странично-сегментной системы.

8.8.2 Коллективное использование программ и данных в системе со странично-сегментной организацией

В странично-сегментной системе преимущества, обеспечиваемые коллективным использованием сегментов, становятся особенно заметными. Коллективное использование здесь реализуется благодаря тому, что в таблицах сегментов для различных процессов содержатся строки, указывающие на одну и ту же таблицу страниц (см. рис. 8.27).

Коллективное использование программ и данных в любой системе, будь то страничная, сегментная или странично-сегментная, требует очень четкого управления со стороны операционной системы. Чтобы убедиться в этом, читатель мог бы рассмотреть случай, когда поступающая страница должна была бы заменить в первичной памяти страницу, коллективно используемую многими процессами.

Заключение

Существуют два наиболее распространенных способа реализации виртуальной памяти — это страничная и сегментная организации. Множество виртуальных адресов, как правило, значительно превышает диапазон адресов первичной памяти.

Суть концепции виртуальной памяти заключается в том, что виртуальные адреса, на которые ссылается выполняющийся процесс, отделяются от адресов, существующих в первичной памяти, т. е. реальных адресов. Диапазон виртуальных адресов, к которым может обращаться выполняющийся процесс, называется виртуальным адресным пространством V этого процесса. Диапазон реальных адресов, существующих в конкретной вычислительной машине, называется реальным адресным пространством R этой машины.

Перевод виртуальных адресов в реальные называется динамическим преобразованием адресов. Динамическое преобразование адресов выполняет сама система, причем это делается прозрачно (невидимо) для пользователя. Искусственная смежность, или непрерывность, означает, что адреса, смежные в виртуальном адресном пространстве V , не обязательно будут смежными в реальном адресном пространстве R .

Только небольшая часть процедур и данных каждого процесса, как правило, размещается в первичной памяти одновременно. Остальная часть хранится на устройствах внешней памяти с быстрым доступом.

Системы виртуальной памяти требуют наличия таблиц отображения виртуальных адресов на реальные адреса. Главная проблема для разработчиков систем виртуальной памяти — это минимизация количества информации отображения, которую необходимо держать в первичной памяти, гарантируя в то же самое время удовлетворительные скоростные характеристики системы. Достижению этой

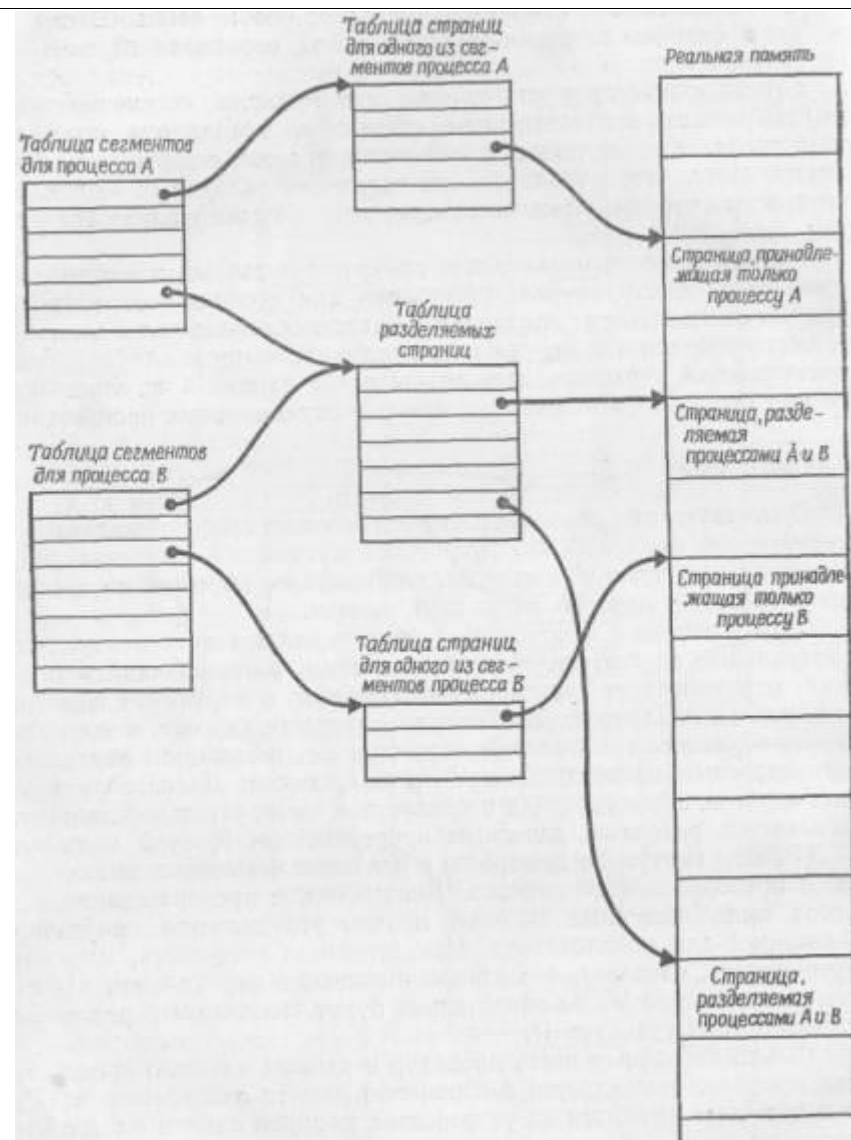


Рис. 8.27 Разделение памяти между двумя процессами в странично-сегментной системе.

цели способствует так называемое поблочное отображение. Блоки фиксированного размера называются страницами; блоки переменного размера называются сегментами. В некоторых системах оба вида блоков комбинируются, причем, как правило, применяются сегменты, длина которых выражается целым числом страниц фиксированного размера. В схемах блочного отображения адреса могут рассматриваться как упорядоченные пары $V=(b, d)$, где b — блок, в котором находится виртуальный адрес U , а d — смещение адреса U относительно начала блока b .

В системе с блочным отображением виртуальные адреса преобразуются в реальные при помощи таблицы отображения блоков. Из скоростных соображений эта таблица зачастую размещается в быстродействующей кэш-памяти или ассоциативной памяти.

Виртуальный адрес при страничной организации — это упорядоченная пара $U=(p, d)$, где p — номер страницы, содержащей U , а d — смещение адреса U относительно начала страницы p . Первичная память разделяется на страничные кадры того же самого размера, что и виртуальные страницы. Поступающая в первичную память страница может быть помещена в любой свободный страничный кадр.

Динамическое преобразование адресов при страничной организации памяти предусматривает отображение номера страницы на страничный кадр p' . Отображение может быть прямым — в этом случае используется полная таблица страниц, размещаемая либо в первичной памяти, либо в высокоскоростной кэш-памяти. Отображение может быть ассоциативным — в этом случае полная таблица страниц размещается в высокоскоростной ассоциативной памяти. Ввиду высокой стоимости ассоциативной памяти и кэш-памяти отображение может быть комбинированным, когда только самые последние по времени обращения страницы содержатся в ассоциативной памяти, а полные таблицы прямого отображения используются, если поиск по ассоциативной памяти дает отрицательный результат. Виртуальная память чаще реализуется с использованием кэш-памяти, чем ассоциативной памяти.

Для коллективного использования (разделения) информации в страничной системе в строках таблиц страниц различных процессов указывается один и тот же страничный кадр. Совместное использование, реализуемое таким способом, связано с неудобствами, поскольку по смыслу процессы коллективно используют такие логические объекты, как процедуры и данные, которые могут зачастую занимать по несколько страниц и расширяться и сокращаться во время выполнения процесса.

В сегментной системе используются блоки переменного размера. Каждый блок, естественно, в разумных пределах, может иметь именно тот размер, который необходим для размещения логического объекта, например процедуры или структуры данных. Не обязательно, чтобы все сегменты процесса одновременно находились в первичной памяти или были в ней смежными. Процесс может выполняться, если в первичной памяти находится (как минимум) его текущий сегмент.

Виртуальный адрес в сегментной системе представляет собой упорядоченную пару $V=(s, d)$, где s — номер сегмента, в котором находится адрес U , а d — смещение этого адреса в рамках сегмента s . Динамическое преобразование адресов может выполняться практически точно так же с использованием прямого, ассоциативного или комбинированного ассоциативно-прямого отображения, как и в системах со страничной организацией. Поскольку сегменты имеют переменную длину, смещение d необходимо контролировать, чтобы быть уверенным в том, что оно не выходит за пределы сегмента.

Защита в сегментных системах оказывается более естественной по сравнению со страничными системами, поскольку защищаются логические, а не физические объекты. Процессам предоставляются различные сочетания прав доступа для чтения, записи, выполнения и дополнения при обращении к различным сегментам. Коллективное использование информации в сегментной системе также реализуется более естественным образом — строки таблиц сегментов различных процессов указывают на совместно используемый сегмент.

Во многих системах реализуется комбинированная странично-сегментная организация. Виртуальный адрес странично-сегментной системы представляет собой упорядоченную тройку $V=(s, p, d)$, где s — сегмент, в котором находится виртуальный адрес U , p — страница в сегменте s , содержащая U , а d — смещение адреса U в странице p . Динамическое преобразование адресов в подобных системах осуществляется довольно сложно, как бы в два этапа, поскольку строки таблиц сегментов указывают на таблицы страниц, а строки таблиц страниц в свою очередь указывают на страничные кадры. В комбинированных странично-сегментных системах почти всегда применяются высокоскоростные ассоциативные устройства или кэш-память, поскольку в противном случае динамическое преобразование адресов может привести к серьезному ухудшению производительности системы.

Коллективное использование в странично-сегментной системе осуществляется благодаря тому, что строки различных таблиц сегментов указывают на одну таблицу страниц коллективно используемого (разделяемого) сегмента.

Терминология

адресное пространство, диапазон адресов (address space)

ассоциативное отображение (associative mapping)

ассоциативная память (associative storage)

блок (block)

бит-признак присутствия (residence bit)

виртуальный адрес (virtual address)

виртуальная память (virtual storage)

двухкомпонентная, двумерная адресация (two-dimensional addressing)

динамическое преобразование адресов (трансляция, перевод) (dynamic address translation, DAT)

доступ для выполнения (execute access)

доступ для дополнения (записи дополнительной информации в конце блока) (append access)

доступ для записи (write access)

доступ для чтения (read access)

защита (protection)

искусственная смежность, непрерывность (artificial contiguity)

коллективное использование (разделение) программного кода и данных (sharing of code and data)

комбинированное ассоциативно-прямое отображение (combined associative/direct mapping)

комбинированная странично-сегментная организация (combined paging/segmentation)

кэш-память (cache storage)

многоуровневая организация памяти (multilevel storage organization)

ожидание страницы (page wait)

отображение адресов (address map)

перемещение страниц (page transport)

поблочное отображение (block mapping)

прерывание по выходу за пределы (переполнению) сегмента (segment overflow fault)

прерывание по защите сегмента (segment protection fault)

прерывание по отсутствию сегмента (missing segment fault)

прерывание по отсутствию страницы (missing page fault)

прерывание по отсутствию элемента (missing item fault)

пространство виртуальных адресов, виртуальное адресное пространство (virtual address space)

пространство реальных адресов, реальное адресное пространство (real address space)

прямое отображение (direct mapping)

коллективное использование (разделение) программного кода и данных (sharing of code and data)

регистр начального (базового) адреса таблицы блоков (block table origin register)

реентерабельная процедура (процедура повторной входимости) (reentrant procedure)

сегмент (segment)

сегментная организация (segmentation)

смещение (displacement)

страница (page)

страничная организация (paging)

странично-сегментная организация (paging/segmentation)

страничный кадр (page frame)

таблица блоков (block map table)

таблица процессов (process table)

таблица сегментов (segment map table)

таблица страниц (page map table)

трехкомпонентная, трехмерная адресация (three-dimensional addressing)

физический (реальный) адрес (real address)

чистая процедура (pure procedure)

чисто сегментная организация (pure segmentation)

чисто страничная организация (pure paging)

Упражнения

8.1 Приведите несколько причин, обуславливающих необходимость отделения виртуального адресного пространства процесса от реального адресного пространства.

8.2 В некоторых системах, эксплуатируемых в настоящее время, виртуальная память по размеру меньше, чем имеющаяся реальная память. Обсудите преимущества и недостатки подобного подхода.

8.3 Одно из главных достоинств виртуальной памяти заключается в том, что благодаря ей пользователям больше не приходится искусственно ограничивать размер своих программ, ориентируясь на недостаточно широкие рамки реальной памяти. Это позволяет сделать стиль программирования более свободной формой выражения алгоритмов и замыслов программиста. Обсудите последствия подобного свободного стиля программирования с точки зрения производительности мультипрограммной системы с виртуальной памятью. Перечислите как положительные, так и отрицательные последствия.

8.4 Расскажите о различных методах, применяемых для преобразования виртуальных адресов в реальные при страничной организации памяти.

8.5 Обсудите относительные достоинства каждого из следующих способов отображения при реализации виртуальной памяти:

- а) прямое отображение,
- б) ассоциативное отображение,
- в) комбинированное ассоциативно-прямое отображение.

8.6 Объясните, каким образом осуществляется преобразование виртуальных адресов в реальные в сегментных системах.

8.7 Объясните, каким образом производится защита памяти в сегментных системах виртуальной памяти.

8.8 Расскажите о различных особенностях аппаратных средств, необходимых для реализации систем виртуальной памяти.

8.9 Расскажите, каким образом проявляется фрагментация в каждом из следующих типов систем виртуальной памяти: ^

- а) в системах с сегментной организацией;
- б) в системах со страничной организацией;
- в) в комбинированных странично-сегментных системах.

8.10 В любой вычислительной системе, независимо от того, является ли она системой реальной или виртуальной памяти, машина редко будет обращаться ко всем командам или данным, записанным в реальной памяти. Назовем это явление кусочной фрагментацией, поскольку оно является следствием того, что элементы памяти обрабатываются, как правило, не индивидуально, а блоками, или «порциями». Поблочная фрагментация вполне может привести к большим непроизводительным затратам емкости первичной памяти, чем все остальные виды фрагментации в совокупности.

а) Почему, в таком случае, кусочной фрагментации не уделяется такого же внимания в технической литературе, как другим видам фрагментации?

б) Каким образом системы виртуальной памяти с динамическим распределением памяти значительно уменьшают степень кусочной фрагментации по сравнению с наблюдаемой в системах реальной памяти?

в) Как могло бы сказаться на кусочной фрагментации уменьшение размеров страниц?

г) Какие факторы, как практические, так и теоретические, не позволяют полностью исключить кусочную фрагментацию?

д) Что может сделать для минимизации кусочной фрагментации каждый из следующих специалистов:

I) программист;

II) разработчик аппаратных средств;

III) разработчик операционной системы.

8.11 Расскажите, каким образом производится преобразование виртуальных адресов в реальные при комбинированной странично-сегментной организации.

8.12 В мультипрограммных системах коллективное использование программного кода и данных позволяет существенно уменьшить объем реальной памяти, необходимой для эффективной работы группы процессов. Кратко расскажите о том, каким образом можно реализовать разделение кода и данных для каждого из следующих типов систем:

а) мультипрограммная система с фиксированными разделами;

б) мультипрограммная система с переменными разделами;

в) система со страничной организацией;

г) сегментная система;

д) система с комбинированной странично-сегментной организацией.

8.13 Почему коллективное использование кода и данных осуществляется гораздо более естественным образом в системах виртуальной памяти, чем в системах реальной памяти?

8.14 Обсудите сходства и различия между страничной и сегментной организацией.

8.15 Укажите общие и отличительные признаки систем с чисто сегментной и комбинированной странично-сегментной организацией.

8.16 Предположим, что вас попросили реализовать сегментную систему на машине, которая имеет аппаратные средства разбиения памяти на страницы, но не имеет аппаратной поддержки сегментов. Вы можете использовать только программные методы. Возможно ли это? Приведите обоснования своего ответа.

8.17 Предположим, что вас попросили реализовать страничную организацию памяти на машине, которая имеет аппаратную поддержку сегментов, но не имеет аппаратуры разбиения памяти на страницы. Вы можете использовать только программные методы. Возможно ли это? Приведите обоснование своего ответа.

8.18 В качестве главного конструктора новой системы виртуальной памяти вы можете выбрать либо страничную организацию, либо сегментную, но не их комбинацию. Какую организацию вы выберете? Почему?

Глава 9

Управление виртуальной памятью

Обычно случается не то, на что мы рассчитываем, а то, чего мы меньше всего ожидаем.

Бенджамин Дизраэли

9.1 Введение

9.2 Стратегии управления виртуальной памятью

9.3 Стратегии выталкивания страниц

9.3.1 Принцип оптимальности

9.3.2 Выталкивание случайной страницы

▣ 9.3.3 Выталкивание первой пришедшей страницы (FIFO)

9.3.3.1 Аномалия FIFO

9.3.4 Выталкивание дольше всего не использовавшейся страницы (LRU)

9.3.5 Выталкивание реже всего используемой страницы (LFU)

9.3.6 Выталкивание не использовавшейся в последнее время страницы (NUR)

9.4 Локальность

9.5 Рабочие множества

9.6 Подкачка страниц по запросу

9.7 Подкачка страниц с упреждением

9.8 Освобождение страниц

9.9 Размер страниц

9.10 Поведение программ при подкачке страниц

9.1 Введение

В предыдущей главе были рассмотрены различные практически реализованные способы организации виртуальной памяти, а именно:

- страничная организация,
- сегментная организация,
- комбинированная странично-сегментная организация.

Мы обсуждали аппаратные и программные механизмы реализации виртуальной памяти. В настоящей главе мы рассмотрим стратегии управления системами виртуальной памяти, а также поведение этих систем при управлении в соответствии с этими стратегиями.

9.2 Стратегии управления виртуальной памятью

В главе, посвященной реальной памяти, рассматривались стратегии управления вталкиванием (выборкой), размещением и выталкиванием (замещением) информации. Здесь мы рассмотрим их заново применительно к системам виртуальной памяти.

Стратегии вталкивания. Их цель - определить, в какой момент следует переписать страницу или сегмент из вторичной памяти в первичную. Вталкивание по запросу (по требованию) предполагает, что система ждет ссылки на страницу или сегмент от выполняющего процесса и только после появления ссылки начинает переписывать данную страницу или сегмент в первичную память. Вталкивание с упреждением (опережением) предполагает, что система пытается заблаговременно определить, к каким страницам или сегментам будет обращаться процесс. Если вероятность обращения высока и в первичной памяти имеется свободное место, то соответствующие страницы или сегменты будут переписываться в основную память еще до того, как к ним будет явно производиться обращение.

Стратегии размещения. Их цель — определить, в какое место первичной памяти помещать поступающую страницу или сегмент. *В системах со страничной организацией решение о размещении принимается достаточно тривиально, поскольку поступающая страница может быть помещена в любой свободный страничный кадр.* Системы с сегментной организацией требуют стратегий размещения, аналогичных тем, которые мы обсуждали применительно к системам мультипрограммирования с переменными разделами.

Стратегии выталкивания. Их цель — решить, какую страницу или сегмент следует удалить из первичной памяти, чтобы освободить место для помещения поступающей страницы или сегмента, если первичная память полностью занята.

9.3 Стратегии выталкивания страниц

В системах со страничной организацией все страничные кадры бывают, как правило, заняты. В этом случае программы управления памятью, входящие в операционную систему, должны решать, какую страницу следует удалить из первичной памяти, чтобы освободить место для поступающей страницы. Мы рассмотрим следующие стратегии выталкивания страниц.

- Принцип оптимальности.
- Выталкивание случайной страницы.

- Первой выталкивается первая пришедшая страница (FIFO).
- Первой выталкивается дольше всего не использовавшаяся страница (LRU).
- Первой выталкивается наименее часто использовавшаяся страница (LFU).
- Первой выталкивается не использовавшаяся в последнее время страница (NUR).
- Рабочее множество.

9.3.1 Принцип оптимальности

Принцип оптимальности (De70) говорит о том, что для обеспечения оптимальных скоростных характеристик и эффективного использования ресурсов следует заменять ту страницу, к которой в дальнейшем не будет новых обращений в течение наиболее длительного времени. Можно, конечно, продемонстрировать, что подобная стратегия действительно оптимальна, однако реализовать ее, естественно, нельзя, поскольку мы не умеем предсказывать будущее.

В связи с этим для обеспечения высоких скоростных характеристик и эффективного использования ресурсов мы попытаемся наиболее близко подойти к принципу оптимальности, применяя различные методы выталкивания страниц, приближающиеся к оптимальному.

9.3.2 Выталкивание случайной страницы

Если нам нужно иметь стратегию выталкивания страниц, которая характеризовалась бы малыми издержками и не являлась бы дискриминационной по отношению к каким-либо конкретным пользователям, то можно пойти по очень простому пути - выбирать случайную страницу. В этом случае все страницы, находящиеся в основной памяти, могут быть выбраны для выталкивания с равной вероятностью, в том числе даже следующая страница, к которой будет производиться обращение (и которую, естественно, удалять из памяти наиболее нецелесообразно). Поскольку подобная стратегия по сути как бы рассчитана на «слепое» везение и похожа на подход «пан или пропал», в реальных системах она применяется редко.

9.3.3 Выталкивание первой пришедшей страницы (FIFO)

При выталкивании страниц по принципу FIFO мы присваиваем каждой странице в момент поступления в основную память временную метку. Когда появляется необходимость удалять из основной памяти какую-нибудь страницу, мы выбираем ту, которая находилась в памяти дольше других. Интуитивный аргумент в пользу подобной стратегии кажется весьма весомым, а именно: у данной страницы уже были возможности «использовать свой шанс», и пора дать подобные возможности и другой странице. К сожалению, стратегия FIFO с достаточно большой вероятностью будет приводить к замещению активно используемых страниц, поскольку тот факт, что страница находится в основной памяти в течение длительного времени, вполне может означать, что она постоянно в работе. Например, для крупных систем разделения времени стандартна ситуация, когда многие пользователи во время ввода и обработки

своих программ совместно используют одну копию текстового редактора. Если в подобной системе выталкивать страницы по принципу FIFO, это может привести к удалению из памяти какой-либо интенсивно используемой страницы редактора. А это будет безусловно нецелесообразно, поскольку ее почти немедленно придется снова переписывать в основную память.

9.3 3.1. Аномалия FIFO

На первый взгляд кажется очевидным, что с увеличением количества страничных кадров, выделяемых процессу, этот процесс будет выполняться с меньшим количеством прерываний по отсутствию нужной страницы в памяти.

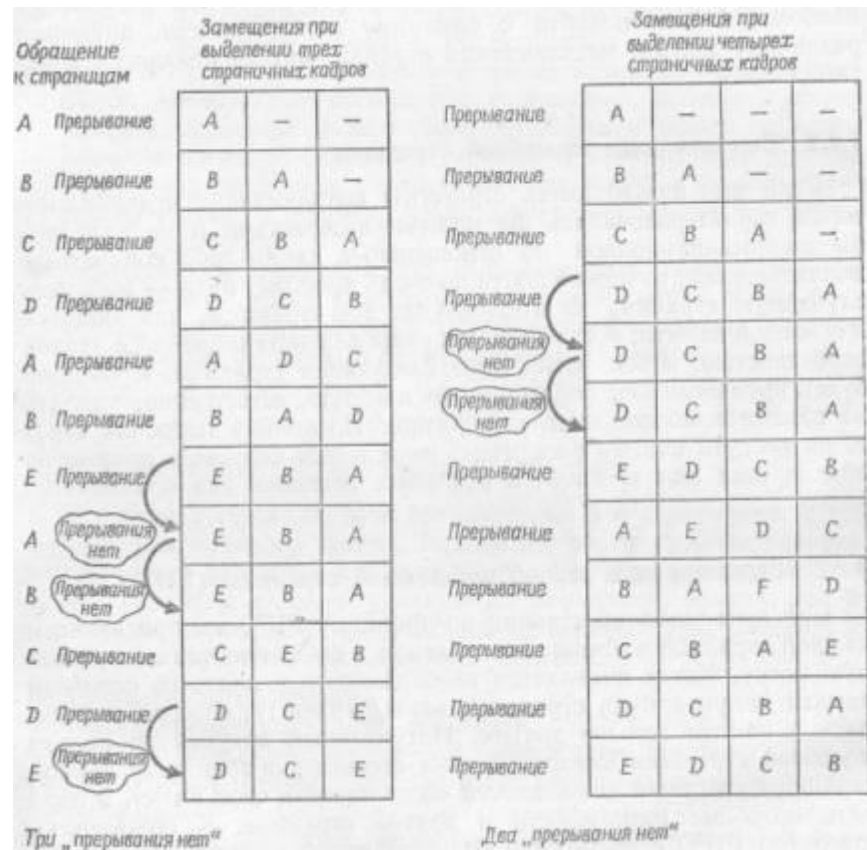


Рис. 9.1 Аномалия FIFO.

Однако, как установили Билейди, Нельсон и Шедлер (Be69b), в стратегии FIFO определенные последовательности обращений к страницам приводят в действительности к увеличению количества прерываний по отсутствию страницы при увеличении количества страничных кадров, выделяемых процессу. Это явление носит название «аномалии FIFO».

Рассмотрим рис. 9.1. Самый левый столбец — это последовательность страниц, к которым обращается процесс. Первая таблица рисунка показывает, как для этой последовательности обращений страницы будут вталкиваться и выталкиваться при реализации стратегии FIFO и выделении данному процессу трех страничных кадров. Вторая таблица показывает, каким образом этот процесс будет работать при тех же самых обстоятельствах, при выделении ему четырех страничных кадров. Слева от каждой таблицы мы отмечаем, будет ли соответствующее новое обращение к странице вызывать прерывание по отсутствию страницы или нет. В действительности оказывается, что при выполнении процесса с четырьмя страничными кадрами количество прерываний по отсутствию страницы на одно больше, чем в случае трех страничных кадров, — факт, явно противоречащий интуиции.

Аномалию FIFO следует, по-видимому, считать скорее курьезом, чем фактором, требующим серьезного к себе отношения. Быть может, истинное значение этого явления для студента, изучающего операционные системы, состоит в том, что оно служит дополнительным указанием на исключительную сложность операционных систем как объектов, где интуитивный подход иногда неприемлем.

9.3.4 Выталкивание дольше всего не использовавшейся страницы (LRU)

Эта стратегия предусматривает, что для выталкивания следует выбирать ту страницу, которая не использовалась дольше других. Здесь мы исходим из эвристического правила, говорящего о том, что недавнее прошлое — хороший ориентир для прогнозирования ближайшего будущего. Стратегия LRU требует, чтобы при каждом обращении к странице ее временная метка обновлялась. Это может быть сопряжено с существенными издержками, и поэтому стратегия LRU, хотя она и кажется весьма привлекательной, в современных системах реализуется редко. Чаще применяются близкие к LRU стратегии, для которых характерны меньшие издержки.

Разработчики операционных систем должны всегда с большой осторожностью применять эвристические правила и рассуждения. Например, при реализации стратегии LRU может быть так, что страница, к которой дольше всего не было обращений, в действительности станет следующей используемой страницей, если программа к этому моменту очередной раз пройдет большой цикл, охватывающий несколько страниц. Таким образом, выталкивая страницу, к которой дольше всего не было обращений, мы можем оказаться вынужденными почти немедленно возвращать ее обратно.

9.3.5 Выталкивание реже всего используемой страницы (LFU)

Одной из близких к LRU стратегий является стратегия, согласно которой выталкивается наименее часто (наименее интенсивно) использовавшаяся страница (LFU). Здесь мы контролируем интенсивность использования каждой страницы. Выталкивается та страница, которая наименее интенсивно используется или обращения к которой наименее часты. Подобный подход опять-таки кажется интуитивно оправданным, однако в то же время велика вероятность того, что удаляемая страница будет выбрана нерационально. Например, наименее интенсивно используемой может оказаться та страница, которую только что переписали в основную память и к которой успели обратиться только один раз, в то время как к другим страницам могли уже обращаться более одного раза. Теперь работающий по принципу LFU механизм вытолкнет эту страницу, а она скорее всего сразу же будет использоваться.

Таким образом, практически любой метод выталкивания страниц, по-видимому, не исключает опасности принятия нерациональных решений. Это действительно так просто потому, что мы не можем достаточно точно прогнозировать будущее. В связи с этим необходима такая стратегия выталкивания страниц, которая обеспечивала бы принятие рациональных решений в большинстве случаев и в то же время не требовала больших накладных расходов.

9.3.6 Выталкивание не использовавшейся в последнее время страницы (NUR)

Один из распространенных алгоритмов, близких к стратегии LRU и характеризующихся малыми издержками, — это алгоритм выталкивания страницы, не

использовавшейся в последнее время (NUR); к страницам, которые в последнее время не использовались, вряд ли будут обращения и в ближайшем будущем, так что их можно заменять на вновь поступающие страницы.

Поскольку желательно заменять ту страницу, которая в период нахождения в основной памяти не изменялась, реализация стратегии NUR предусматривает введение двух аппаратных битов-признаков на страницу. Это

- а) бит-признак обращения = 0, если к странице не было обращений;
= 1, если к странице были обращения;
- б) бит-признак модификации = 0, если страница не изменялась;
= 1, если страница изменялась.

Бит-признак модификации часто называют также «признаком записи» в страницу. Стратегия NUR реализуется следующим образом. Первоначально биты-признаки обращения и модификации для всех страниц устанавливаются в 0, При обращении к какой-либо странице ее бит-признак обращения устанавливается в 1, а в случае изменения содержимого страницы устанавливается в 1 ее бит-признак модификации. Когда нужно выбрать страницу для выталкивания, прежде всего мы пытаемся найти такую страницу, к которой не было обращений (поскольку мы стремимся приблизиться к алгоритму LRU). В противном случае у нас не будет другого выхода, как вытолкнуть страницу, к которой были обращения. Если к странице обращения были, мы проверяем, подверглась ли она изменению или нет. Если нет, мы заменяем ее из тех соображений, что это связано с меньшими затратами, чем в случае замены модифицированной страницы, которую необходимо будет физически переписывать во внешнюю память. В противном случае нам придется заменять модифицированную страницу.

В многоабонентских системах основная память, естественно, работает активно, так что рано или поздно у большинства страниц бит-признак обращения будет установлен в 1, и мы не сможем отличать те страницы, которые вытолкнуть наиболее целесообразно. Один из широко распространенных способов решения этой проблемы заключается в том, что все биты-признаки обращений периодически сбрасываются в 0, с тем чтобы механизм вталкивания оказался в исходном состоянии, а затем снова разрешается установка этих битов-признаков в 1 обычным образом при обращениях. Правда, в этом случае существует опасность того, что могут быть вытолкнуты даже активные страницы, однако только в течение короткого периода после сброса битов-признаков, поскольку почти немедленно биты-признаки обращений для этих страниц будут снова установлены в 1.

Описанный выше алгоритм NUR предусматривает существование четырех групп страниц:

Группа 1	Обращений не было	Модификаций не было
Группа 2	Обращений не было	Модификация была
Группа 3	Обращения были	Модификаций не было
Группа 4	Обращения были	Модификация была

Страницы групп с меньшими номерами следует выталкивать в первую очередь, а с большими — в последнюю. Отметим, что группа 2 обозначает на первый взгляд

нереальную ситуацию, она включает страницы, к которым как бы не было обращений, но они оказались модифицированными. В действительности это просто результат того, что биты-признаки обращения (но не биты-признаки модификации) периодически сбрасываются, т. е. подобная ситуация вполне возможна.

9.4 Локальность

Большинство стратегий управления памятью базируется на концепции локальности, суть которой заключается в том, что распределение запросов процессов на обращение к памяти имеет, как правило, неравномерный характер с высокой степенью локальной концентрации.

Свойство локальности проявляется как во времени, так и в пространстве. Временная локальность — это концентрация во времени. Если, например, в 15 ч наблюдается солнечная погода, то можно с достаточно высокой вероятностью считать (но, естественно, без всяких гарантий), что было солнечно в 14 ч 30 мин и будет солнечно в 15 ч 30 мин. Пространственная локальность означает, что соседние объекты будут, как правило, характеризоваться одинаковыми свойствами. Если опять-таки взять в качестве примера погоду, то в случае, когда в одном городе солнечно, можно с достаточно высокой вероятностью (но без всяких гарантий) считать, что и в близлежащих городах солнечно.

Свойство локальности наблюдается также и в работе операционных систем, в частности при управлении памятью. Свойство это скорее эмпирическое (наблюдаемое), чем теоретически обоснованное. Локальность никак нельзя гарантировать, однако ее вероятность зачастую весьма велика. Например, в системах со страничной организацией мы наблюдаем, что процессы обычно «оказывают предпочтение» определенным подмножествам своих страниц и что эти страницы обычно размещаются по соседству друг с другом в виртуальном адресном пространстве процесса. Это не значит, что процесс не будет иметь ссылок на какую-либо новую страницу — если бы это было так, то процессы вообще не могли бы даже начать выполнение. Это просто означает, что процесс будет, как правило, сосредоточивать свои обращения в течение некоторого временного интервала на некотором конкретном подмножестве своих страниц.

Фактически свойство локальности для вычислительных систем довольно легко объяснить, если учесть, каким образом пишутся программы и организуются данные. В частности,

1. Временная локальность, означающая, что к ячейкам памяти, к которым недавно производилось обращение, с большой вероятностью будет обращение в ближайшем будущем, обуславливается наличием следующих факторов:

- а) программных циклов,
- б) подпрограмм,
- в) стеков,
- г) переменных, используемых в качестве счетчиков и для накопления итоговых сумм.

2. Пространственная локальность, означающая, что обращения к памяти, как правило, концентрируются таким образом, что в случае обращения к некоторой ячейке

памяти с большой вероятностью можно ожидать обращения к близлежащим ячейкам, обусловливается наличием следующих факторов:

- а) организацией данных в виде массивов,
- б) последовательным выполнением кода программы»
- в) тенденцией программистов размещать описания взаимосвязанных переменных поблизости друг от друга.

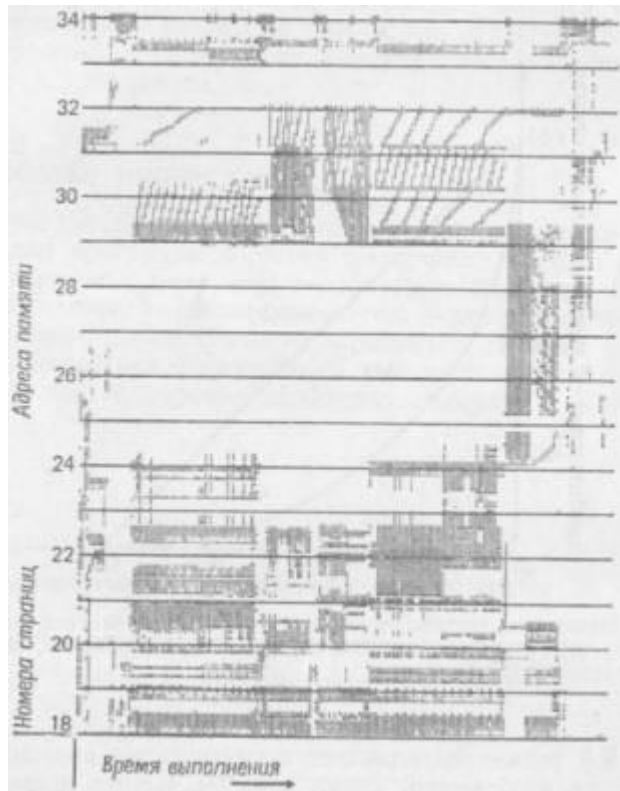


Рис. 9.2 Картина распределения обращений к памяти, свидетельствующая о существовании явления локальности (перепечатывается с разрешения корпорации IBM из IBM Systems Journal. © 1971).

По-видимому, самым важным следствием локализации обращений к памяти является то, что программа может эффективно работать, когда в памяти находится подмножество, включающее наиболее «популярные» ее страницы. На основе изучения свойства локальности Деннинг сформулировал свою *теорию рабочего множества* программ (De68, De68a). Эту теорию мы рассмотрим в следующем разделе.

Было проведено много исследований, иллюстрирующих явление локальности. Так, на рис. 9.2 в графическом виде показано распределение обращений процесса к различным страницам памяти (Ha72). Здесь штрихами показано, к каким областям памяти производились обращения в течение последовательных временных интервалов. Из рисунка ясно видно, что в течение определенных периодов работы процесс обращается преимущественно лишь к некоторому подмножеству своих страниц.

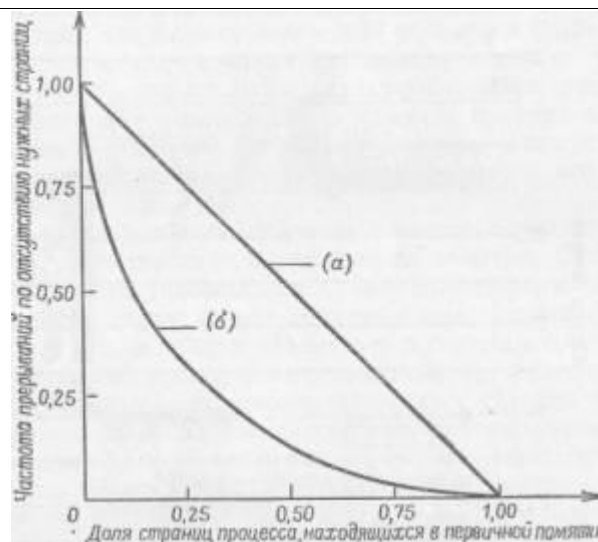


Рис. 9.3 Зависимость частоты прерываний от объема выделяемой первичной памяти: (а) процесс с равновероятными обращениями к страницам, (б) процесс с Локализованными обращениями.

Рис. 9.3 также подтверждает существование явления локальности. Здесь изображено, каким образом частота прерываний по отсутствию нужной страницы для процесса зависит от объема первичной памяти, выделяемой для размещения его страниц. Прямая линия показывает, как выглядела бы эта зависимость в случае, если бы процессы обращались к памяти по случайному закону, с равномерным распределением ссылок на различные свои страницы. Кривая линия показывает, как в действительности ведут себя большинство процессов во время работы. Если количество страничных кадров, выделяемых процессу, уменьшается, то существует некоторый интервал, в котором это не очень заметно сказывается на увеличении частоты прерываний по отсутствию нужной страницы. Однако, начиная с определенного момента при дальнейшем уменьшении количества выделяемых страничных кадров частота прерываний резко возрастает. При этом замечено, что, до тех пор, пока в первичной памяти остается подмножество наиболее интенсивно используемых страниц процесса, частота прерываний меняется мало. Но, как только какие-либо страницы этого подмножества из памяти удаляются, интенсивность подкачки резко возрастает, поскольку процесс постоянно обращается и вновь вызывает эти страницы в первичную память. Все это подтверждает предложенную Деннингом концепцию рабочего множества, которая рассматривается в следующем разделе. (Отметим, что приведенные рассуждения распространяются также по существу и на размещение рабочих множеств в кэш-памяти.)

9.5 Рабочие множества

Деннинг (De68) предложил оценивать интенсивность подкачки страниц для программы согласно концепции, которую он назвал *теорией поведения программы с рабочим множеством*. Если говорить неформально, то рабочее множество — это подмножество страниц, к которым процесс активно обращается. Деннинг утверждал, что для обеспечения эффективного выполнения программы необходимо, чтобы ее рабочее множество находилось в первичной

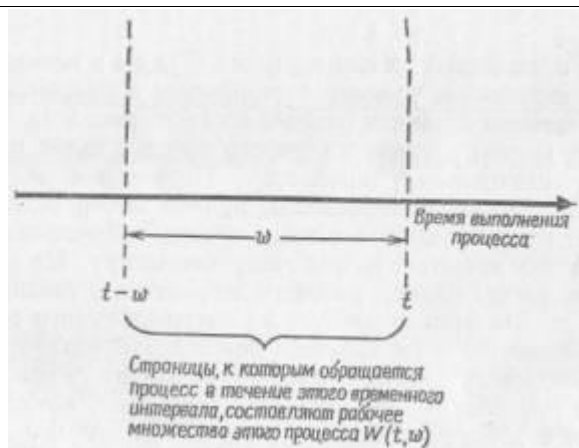


Рис. 9.4 Одно из определений рабочего множества страниц процесса.

памяти. В противном случае может возникнуть режим чрезмерно интенсивной подкачки страниц, так называемое пробуксовывание, или трешинг (De68b), поскольку программа будет многократно подкачивать одни и те же страницы из внешней памяти.

Стратегия управления памятью в соответствии с рабочим множеством стремится к тому, чтобы рабочее множество программы было в первичной памяти. Решение о том, следует ли добавить новый процесс к набору активных процессов (т. е. об увеличении степени мультипрограммирования), должно базироваться на том, имеется ли в основной памяти достаточно свободного места, чтобы разместить рабочее множество страниц этого процесса. Подобное решение, особенно в случае впервые иницизируемых процессов, зачастую принимается эвристически, поскольку система не может заранее знать, каким окажется рабочее множество данного процесса.

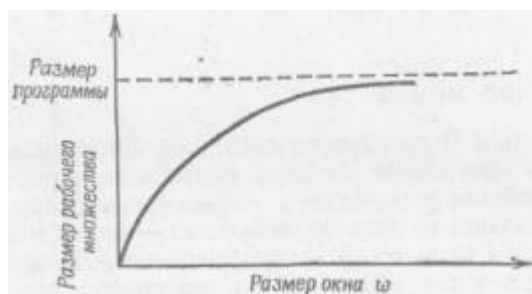


Рис. 9.5 Размер рабочего множества как функция размера окна.

Рабочее множество страниц процесса $W(t, w)$ в момент времени t есть набор страниц, к которым этот процесс обращается в течение интервала времени процесса от $t-w$ до t (см. рис. 9.4). Время процесса — это время, в течение которого процесс имеет в своем распоряжении центральный процессор. Переменная w называется *размером окна рабочего множества*, причем выбор величины этого окна играет решающую роль с точки зрения эффективности стратегии управления памятью по рабочему множеству. На рис. 9.5 показано, как растет размер рабочего множества с увеличением размера окна w . Эта кривая следует из математического определения рабочего множества и не является результатом обработки эмпирически наблюдаемых размеров рабочих множеств. *Реальное рабочее множество процесса — это множество страниц, которые должны находиться в первичной памяти, чтобы процесс мог эффективно выполняться.*

Во время работы процесса его рабочие множества динамически меняются. Иногда к текущему рабочему множеству добавляются или из него удаляются некоторые страницы. Иногда происходят резкие изменения, например, когда процесс переходит к этапу, требующему совершенно нового рабочего множества. Таким образом, любые предположения относительно размера и содержимого начального рабочего множества процесса не обязательно будут справедливыми для последующих рабочих множеств данного процесса. В связи с этим существенно усложняется задача реализации четкой стратегии управления памятью по рабочим множествам.

На рис. 9.6 показано, как мог бы использовать первичную память процесс при управлении памятью по рабочим множествам. Вначале, поскольку процесс запрашивает страницы своего рабочего множества не все сразу, а по одной, он постепенно получает достаточный объем памяти для размещения текущего рабочего множества. Затем на какой-то период времени этот объем памяти стабилизируется, поскольку процесс интенсивно обращается к страницам

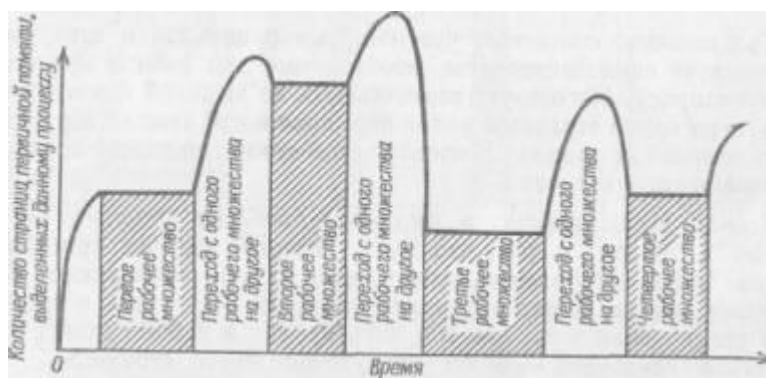


Рис. 9.6 Выделение основной памяти при управлении памятью на основе рабочих множеств.

своего первого рабочего множества. Со временем процесс перейдет на следующее рабочее множество, что показывает кривая линия от первого ко второму рабочему множеству. Вначале эта кривая поднимается выше уровня первого рабочего набора, поскольку для процесса по его запросам быстро производится подкачка страниц нового рабочего множества. При этом система не может сразу определить, то ли процесс просто расширяет свое рабочее множество, то ли он переходит на новое. После того как процесс начнет стабильно обращаться к страницам своего следующего рабочего множества, система видит, что в течение выбранного окна происходит обращение к меньшему количеству страниц, чем ранее, и сокращает объем памяти, выделяемой процессу, до числа страниц этого второго рабочего множества. Каждый раз, когда осуществляется переход с одного рабочего множества на другое, кривая линия поднимается, а затем спадает, показывая, каким образом система адаптируется к подобному переходу.

Рис. 9.6 иллюстрирует одну из серьезных трудностей, связанных с выбором стратегии управления памятью на основе рабочих множеств; проблема состоит в том, что *рабочие множества меняются во времени, причем следующее рабочее множество процесса может существенно отличаться от предыдущего*. Стратегия управления памятью должна обязательно учитывать этот факт, чтобы избежать перегрузок первичной памяти и возникающего вследствие этого трешинга.

9.6 Подкачка страниц по запросу

Традиционно считается, что наиболее рационально загружать в основную память страницы, необходимые для работы процесса, по его запросу. Не следует переписывать из внешней памяти в основную ни одной страницы до тех пор, пока к ней явно не обратится выполняющийся процесс. В пользу такой стратегии можно привести несколько аргументов:

- Теория вычислимости, в частности *проблема останова* (Mi67, He77), говорит о том, что путь, который выберет программа при своем выполнении, точно предсказать невозможно. Поэтому любая попытка заранее загрузить страницы в память в предвидении того, что они потребуются в работе, может оказаться неудачной — будут загружены не те страницы.
- Подкачка страниц по запросу гарантирует, что в основную память будут переписываться только те страницы, которые фактически необходимы для работы процессов.
- Накладные расходы на то, чтобы определить, какие страницы следует передавать в основную память, минимальны. Стратегии вталкивания с упреждением могут потребовать значительных дополнительных затрат процессорного времени.

Подкачка страниц по запросу, как показывает рис. 9.7, имеет свои проблемы. Процесс должен накапливать в памяти, требуемые ему страницы по одной. При появлении ссылки на каждую новую страницу процессу приходится ждать, когда эта страница будет передана в основную память. В зависимости от того, сколько страниц данного процесса уже находятся в основной памяти, эти периоды ожидания будут обходиться все более дорого, поскольку ожидающие процессы будут занимать все больший объем памяти. Рис. 9.7 иллюстрирует понятие произведения «пространство—время», которое часто применяется в операционных системах для оценки использования памяти процессом. Произведение «пространство—время» соответствует площади «под кривой» на этом рисунке. Оно является комплексным показателем, отражающим как объем, так и время использования памяти процессом. Уменьшение произведения «пространство—время» за счет периодов ожидания процессом нужных ему страниц является важнейшей целью всех стратегий управления памятью.

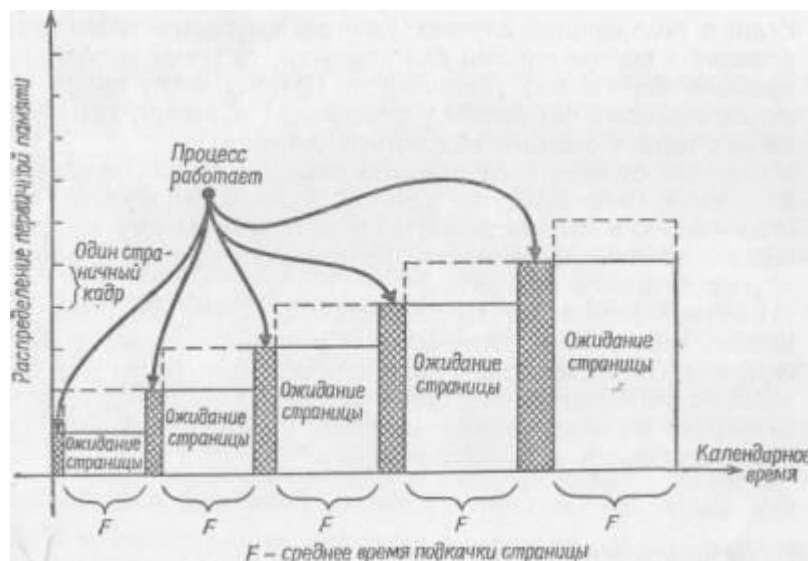


Рис. 9.7 Произведение «пространство-время» при подкачке страниц по запросу.

9.6 Подкачка страниц с упреждением

Главный критерий эффективности управления ресурсами можно сформулировать следующим образом: *интенсивность использования каждого ресурса должна определяться относительной ценностью этого ресурса*. Так, в настоящее время стоимость аппаратуры резко снижается, и поэтому значительно снижается относительная ценность машинного времени по сравнению с временем, затрачиваемым человеком. Сейчас разработчики операционных систем активно ищут пути уменьшения количества времени, в течение которого пользователям приходится ждать получения результатов от компьютера. Одним из весьма перспективных в этом смысле является метод *подкачки страниц с упреждением* (с опережением).

При упреждающей подкачке операционная система пытается заблаговременно предсказать, какие страницы потребуются процессу, а затем, когда в основной памяти появляется свободное место, загружает в нее эти страницы. Пока процесс работает со своими текущими страницами, система запрашивает новые страницы, которые будут уже готовы к использованию, когда процесс к ним обратится. Если решения о выборе страниц для подкачки принимаются правильно, то удастся значительно сократить общее время выполнения данного процесса.

Метод подкачки страниц с упреждением характеризуется следующими преимуществами:

- Если в большинстве случаев удастся принимать правильные решения о выборе страниц для подкачки, то время выполнения процесса значительно уменьшается. Поэтому имеет смысл пытаться создавать механизмы упреждающей подкачки, даже если от них нельзя ожидать абсолютной точности.
- Во многих случаях вполне можно находить правильные решения. Если такое принятие решений удастся реализовать при относительно малых затратах, то выполнение данного процесса можно значительно ускорить, не замедляя при этом работы других активных процессов.
- Поскольку аппаратура вычислительных машин все более дешевеет, последствия неоптимальных решений становятся менее серьезны. Сейчас мы можем позволить себе приобрести дополнительную основную память, обеспечивающую накопление дополнительных страниц, которые механизм упреждающей подкачки будет передавать в основную память.

9.8 Освобождение страниц

При управлении памятью по рабочему множеству программы говорят нам, какие страницы они хотят использовать, явно обращаясь к этим страницам. Страницы, которые больше не нужны, должны каким-то образом выводиться из рабочих множеств. Существует обычно период времени, в течение которого ненужные страницы все еще остаются в основной памяти.

Когда выясняется, что некоторая страница больше не понадобится, пользователь может по собственной инициативе выдать сигнал об удалении страницы из памяти (отказаться от страницы) с освобождением соответствующего страничного кадра. Благодаря этому будет исключен период задержки, неизбежный в случае, если предоставить системе возможность естественным порядком вывести эту страницу из рабочего множества процесса.

Подобное «добровольное освобождение» страниц может исключить

непроизводительное использование памяти и ускорить выполнение программ. Однако большинство пользователей современных вычислительных машин даже не знают, что такое страница, поэтому их вообще нельзя просить принимать решения на системном уровне. Включение команд освобождения страниц в прикладные программы пользователя может значительно замедлить их разработку.

Можно надеяться, что реально эта проблема будет решена при помощи компиляторов и операционных систем, которые будут автоматически обнаруживать ситуации, требующие освобождения страниц, причем, делать это гораздо быстрее, чем позволяют стратегии с использованием рабочих, множеств страниц.

9.9 Размер страниц

В системах со страничной организацией реальная память обычно разбивается на страничные кадры фиксированного размера. Какой величины следует выбирать размер этих страничных кадров? Какой величины следует выбирать размер страницы? Должны ли все страницы в системе иметь одинаковый размер либо следует использовать страницы нескольких различных размеров? Если используются различные размеры страниц, то должны ли большие размеры страниц быть целыми кратными меньших размеров?

Для ответа на эти вопросы требуется тщательный анализ и всестороннее понимание возможностей аппаратных и программных средств и области применения конкретной системы. Здесь не может быть универсальных ответов. Не существует никакой особой необходимости в том, чтобы все вычислительные системы имели одинаковый или единственный размер страниц.

Некоторые соображения, которые необходимо учитывать при выборе того или иного размера страницы, приведены ниже.

- Чем меньше размер страницы, тем большее количество страниц и страничных кадров будет в системе и тем больше будут таблицы страниц. Для систем, в которых таблицы страниц занимают первичную память, это указывает на необходимость увеличенных размеров страниц. Такое неэффективное использование памяти из-за чрезмерно больших таблиц *называется табличной фрагментацией*. Здесь мы отметим, что подобный аргумент в настоящее время не столь актуален, поскольку выпускается недорогая память очень большой емкости.
- При крупных размерах страниц в первичную память при подкачке попадают большие объемы информации, которая, вообще говоря, может и не понадобиться. Это указывает на необходимость уменьшения размеров страниц.
- Поскольку обмены данными с дисковой памятью занимают относительно много времени, мы хотим свести к минимуму число обменов, которые будут производиться для программы во время ее выполнения. Это указывает, по-видимому, на необходимость увеличения размеров страниц.
- Программы, как правило, проявляют свойство локальности обращений, причем размеры подобных локальных участков невелики. Таким образом, уменьшение размера страницы должно способствовать тому, что у программы образуется более компактный набор страниц, т. е. страницы рабочего множества» размещаемые в реальной памяти, будут содержать более интенсивно используемые объекты.
- Поскольку блоки процедур и данных редко представляют целое число страниц, в системах со страничной организацией наблюдается внутренняя фрагментация, иллюстрируемая рис. 9.8. Сегмент длины s с равной вероятностью может иметь свою последнюю страницу почти полную или почти пустую, и, таким образом, в среднем потери из-за внутренней фрагментации

составляют половину страницы (при том ограничительном условии, что страница не может содержать части более чем одного сегмента). Чем меньше размер страницы, тем меньше потери из-за внутренней фрагментации.

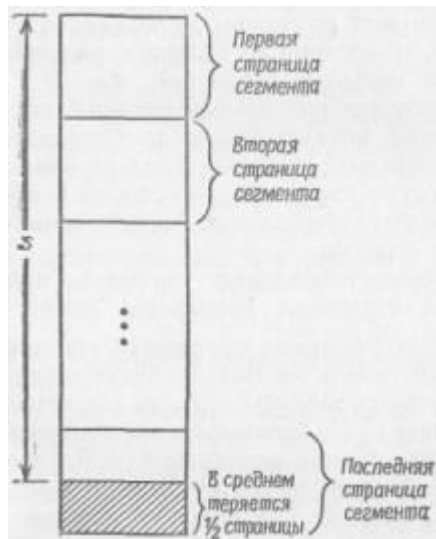


Рис. 9.8 Внутренняя фрагментация в системе со страничной организацией.

Фирма	Модель	Размер страницы	Единицы измерения
Honeywell	Multics	1024	36-битовое слово
IBM	360/67	1024	32-битовое слово
IBM	370/168	1024 или 512	32-битовое слово
DEC	PDP- 10 PDP-20	512	36-битовое слово
DEC	VAX 11/780	128	32-битовое слово

Рис. 9.9 Некоторые стандартные размеры страниц.

Большинство приводящихся в литературе данных (De80), как теоретических, так и эмпирических, указывает на то, что необходимы страницы малых размеров. На рис. 9.9 приведены размеры страниц нескольких популярных компьютеров. Отметим, что для одной из недавно разработанных машин, VAX 11/780, выбран размер страницы 128 слов.

9.10 Поведение программ при подкачке страниц

Страничная организация — весьма эффективная концепция, так что она будет, по-видимому, по-прежнему находить применение в вычислительных системах в течение следующих нескольких десятилетий. Было проведено много исследований,

анализирующих поведение процессов со страничной организацией памяти (Be66, Co68, Ha72, Sp72, O174, Sa75, Ba76, Po77, Sp77, Fr78, De80). В данном разделе мы приведем некоторые качественные результаты этих исследований.

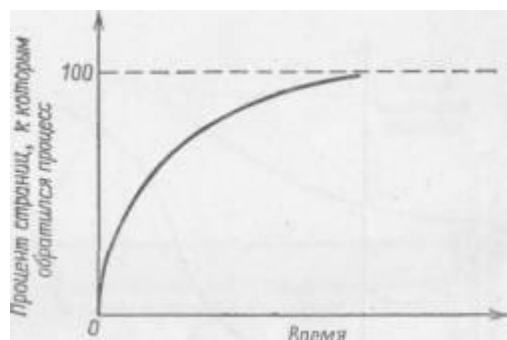


Рис. 9.10

На рис. 9.10 показан график зависимости процента страниц, к которым обращается типичный процесс, от времени, причем с начала выполнения этого процесса. Первоначальный резкий скачок вверх свидетельствует о том, что сразу после начала выполнения процесс обращается к значительной части своих страниц. Со временем наклон кривой уменьшается и она асимптотически приближается к уровню 100%. Безусловно, некоторые процессы используют все 100% своих страниц, однако приведенный качественный график отражает тот факт, что многие процессы могут в течение длительного времени выполняться, не обращаясь ко всем своим страницам. Это часто бывает, например, в случае, когда к определенным подпрограммам обработки ошибок приходится прибегать лишь изредка.

На рис. 9.11 показано, к каким последствиям приводит изменение размера страницы при сохранении постоянного объема первичной памяти. Из графика видно, что для работающего процесса число прерываний по отсутствию нужной страницы увеличивается с увеличением размера страницы. Это происходит потому, что с увеличением размера страницы в первичную память фиксированного размера попадает большее число процедур и данных, к

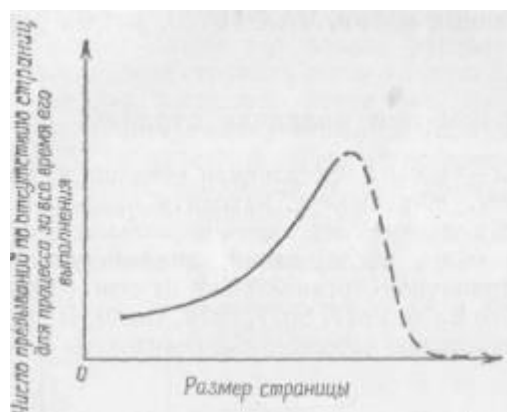


Рис. 9.11

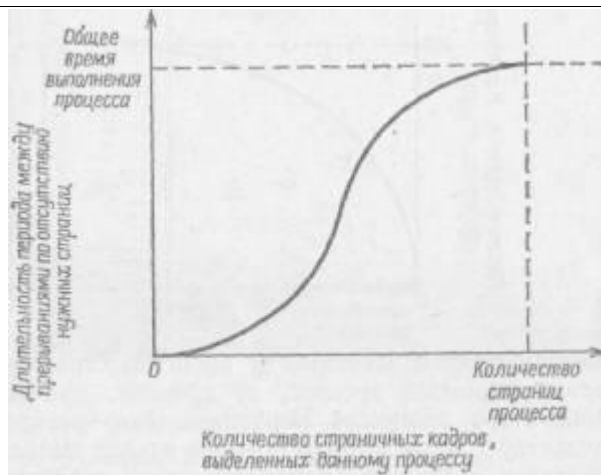


Рис. 9.12

которым не будет обращений. Тем самым уменьшается доля, занимаемая процедурами и данными, к которым будут производиться обращения, в общем ограниченном объеме памяти процессора. А поскольку последовательность обращений процесса к элементам памяти по сути не зависит от объема выделяемой ему первичной памяти, во время выполнения процесса будет возникать большее количество прерываний, вызывающих передачу страниц с требуемыми процедурами и данными в первичную память.

На рис. 9.12 показано, каким образом меняется среднее время между прерываниями по отсутствию нужной страницы в памяти при увеличении числа страничных кадров, выделяемых процессу.

Кривая монотонно растет — чем больше страничных кадров имеет процесс, тем больше интервал между прерываниями по отсутствию нужных страниц. Однако на кривой есть точка перегиба, после которой наклон резко уменьшается. Эта точка соответствует моменту, когда в первичной памяти оказывается все рабочее множество страниц процесса. Первоначально величина интервала между прерываниями растет очень быстро, поскольку растет часть рабочего множества, которая оказывается в первичной памяти. После того как выделенный объем первичной памяти оказывается достаточным для размещения всего рабочего множества, кривая делает перегиб, указывая на то, что выделение дополнительных страничных кадров не приведет к существенному увеличению интервала между прерываниями. Ключевой момент здесь состоит в том, чтобы все рабочее множество попало в первичную память.

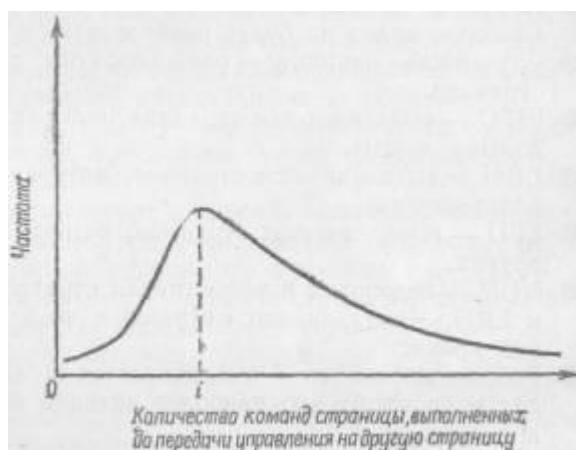


Рис. 9.13

На рис. 9.13 показано, какой процент команд данной страницы выполняется до того, как управление будет передано другой странице. В одном проведенном эксперименте максимум указанной кривой пришелся на точку, соответствующую примерно 200 командам страницы размером 1024 слов. Подобные результаты говорят, по-видимому, о том, что необходимо уменьшать размеры страниц.

В общем качественные оценки и рассуждения, приведенные в этом разделе, свидетельствуют о правомерности концепции рабочего множества и о необходимости уменьшения размеров страниц. По мере развития архитектуры компьютеров эти результаты придется пересматривать.

Заключение

Стратегии управления виртуальной памятью разделяются на три категории. Стратегии вталкивания ставят своей целью определить, в какой момент следует переписывать очередную страницу или сегмент в основную память. В случае вталкиваний по запросу это делается при появлении явной ссылки; при упреждающем вталкивании это делается заблаговременно, когда с достаточно высокой вероятностью можно считать, что к данной странице или сегменту вскоре будет обращение. Стратегии размещения ставят своей целью определить, в какое место первичной памяти помещать поступающую страницу или сегмент. Стратегии выталкивания определяют, какую страницу или сегмент следует заменить, чтобы освободить место для поступающей страницы или сегмента, когда первичная память полностью занята.

В данной главе обсуждались следующие стратегии выталкивания страниц:

- Принцип оптимальности — выталкивается страница, которая наиболее долго не будет использоваться в дальнейшем.
- Случайный принцип — равновероятно выталкивается любая страница.
- FIFO — выталкивается страница, которая находилась в памяти дольше других.
- LRU — выталкивается страница, которая дольше всего не использовалась.
- LFU — выталкивается страница, которая использовалась реже других.
- NUR — недорогая и эффективная стратегия, приближающаяся к LRU, — выталкивается страница, не используемая в последнее время.
- Рабочее множество — выталкивается страница, которая не входит в подмножество наиболее активно используемых страниц процесса.

Аномалия FIFO — это явление, суть которого заключается в том, что иногда увеличение количества страничных кадров, выделяемых процессу, приводит к увеличению частоты прерываний по отсутствию нужных страниц для этого процесса.

Локальность — это свойство выполняющихся процессов, состоящее в том, что процессы, как правило, более активно обращаются к некоторому подмножеству своих страниц в течение некоторого временного интервала выполнения. Временная локальность означает, что процесс, обращающийся к некоторой странице, с большой вероятностью вскоре снова обратится к этой же странице. Пространственная локальность означает, что процесс, обращающийся к некоторой странице, с большой вероятностью вскоре обратится к соседним страницам своего виртуального адресного

пространства. Свойство локальности вполне объяснимо, если учесть, что программы пишутся с использованием циклов, подпрограмм, стеков, счетчиков, переменных для накопления итоговых сумм, массивов, что код программы обычно выполняется последовательно и что у программистов принято размещать описания взаимосвязанных элементов поблизости друг от друга.

Деннинг разработал концепцию рабочих множеств, объясняющую поведение программы на основе понятия локальности. Стратегии управления памятью с использованием рабочих множеств ставят своей целью размещать страницы рабочего множества, т. е. страницы, к которым производились обращения в самое последнее время, в основной памяти, с тем чтобы процесс мог быстро выполняться. Новые процессы можно инициировать только в случае, если в основной памяти имеется свободное место для размещения их рабочих множеств. Деннинг определил рабочее множество $W(t, w)$ в момент времени t как множество страниц, к которым процесс обращается в интервале процессорного времени от $t-w$ до t . Если делается попытка выполнять процессы, не имеющие достаточного места в памяти для размещения рабочих множеств, то часто возникает явление пробуксовки, или трешинга; оно заключается в том, что производится непрерывное выталкивание страниц с последующим немедленным вталкиванием их обратно.

Подкачка страниц по запросу традиционно считается наиболее рациональной стратегией управления памятью, поскольку.

- теория вычислимости говорит о том, что полностью эффективные механизмы упреждающей подкачки реализовать нельзя, так как мы не можем точно предсказывать будущее;
- в основную память вводятся только те страницы, которые фактически необходимы для работы процесса, и
- издержки на выборку страниц минимальны.

В то же время подкачка страниц по запросу приводит к неэффективному использованию первичной памяти, поскольку процессам приходится ждать, пока не поступит каждая нужная страница.

Сторонники подкачки страниц с упреждением, напротив, утверждают, что в большинстве случаев удастся принимать правильные решения по выбору страниц и что процессы будут выполняться гораздо быстрее, когда нужные им страницы будут уже находиться в первичной памяти.

Некоторые системы предусматривают возможность добровольного освобождения страниц, т. е. выполняющийся процесс в явном виде сообщает системе, что какая-то конкретная страница ему больше не понадобится. Такой подход помогает освобождать первичную память от ненужных страниц.

Чтобы определить оптимальный размер страницы для данной системы, необходимо учитывать ряд соображений.

- Малый размер страницы приводит к увеличению таблиц страниц и, как следствие, к табличной фрагментации.
- Большой размер страницы приводит к тому, что в первичную память будут переписываться команды и данные, к которым не будет обращений.
- Ввод-вывод более эффективен при больших размерах страниц.
- Свойство локальности распространяется, как правило, на небольшие участки.
- При малых размерах страниц потери памяти на внутреннюю фрагментацию уменьшаются.

В целом большинство разработчиков придерживается мнения, что все эти факторы, свидетельствуют о необходимости выбора небольших размеров страниц.

Было проведено много экспериментов для исследования поведения программ в вычислительных системах со страничной организацией, причем получены интересные результаты.

- Когда процесс начинает выполнение, он обычно быстро обращается к большому проценту своих страниц.
- Количество прерываний по отсутствию нужной страницы для выполняющегося процесса увеличивается с увеличением размера страницы, если предположить, что объем первичной памяти, выделенной этому процессу, остается постоянным.
- Интервал между прерываниями по отсутствию нужной страницы для работающего процесса растет с увеличением количества выделяемых ему страничных кадров. После того как процессу выделено количество страничных кадров, достаточное для размещения его рабочего множества, скорость увеличения этого периода уменьшается.
- Число команд, выполняемых на странице до передачи управления другой странице, как правило, невелико.

Терминология

аномалия FIFO (FIFO Anomaly)

внутренняя фрагментация (internal fragmentation)

выталкивание страниц по принципу FIFO («первой заменяется первая пришедшая страница») (first-in-first-out (FIFO) page replacement)

выталкивание страниц по принципу LFU («первой заменяется наименее часто используемая страница») (least-frequently-used (LFU) page replacement)

выталкивание страниц по принципу LRU («первой заменяется дольше всего не использовавшаяся страница») (least-recently-used (LRU) page replacement)

выталкивание страниц по принципу NUR («первой заменяется не использовавшаяся в последнее время страница») (not-used-recently (NUR) page replacement)

выталкивание страниц рабочего множества (working set page replacement)

выталкивание страниц по случайному принципу (random page replacement)

добровольное освобождение страницы, отказ от страницы (voluntary page release)

локальность (locality)

локальность временная (temporal locality)

локальность пространственная (spatial locality)

оптимальное выталкивание страниц (optimal page replacement)

подкачка страниц по запросу (по требованию) (demand paging)

подкачка страниц с упреждением (anticipatory paging)

принцип оптимальности (Principle of Optimality)

пробуксовка, трешинг (thrashing)

произведение «пространство—время» (space-time product)

рабочее множество страниц (working set of pages)

размер окна рабочего множества (working set window size)

стратегия вталкивания (fetch strategy)

стратегия выталкивания (page replacement strategy)

стратегия управления памятью на основе рабочих множеств (working set storage management policy)

табличная фрагментация (table fragmentation)

теория поведения программ с рабочим множеством (working set theory of program behavior)

Упражнения

9.1 Обсудите цели каждой из следующих стратегий управления памятью применительно к системам виртуальной памяти со страничной организацией:

- а) стратегия вталкивания,
- б) стратегия размещения,
- в) стратегия выталкивания.

9.2 Объясните, почему управление памятью в чисто сегментных системах весьма похоже на управление памятью в системах мультипрограммирования с переменными разделами.

9.3 Приведите несколько причин, по которым подкачка страниц по запросу в течение многих лет считалась наиболее рациональной стратегией вталкивания страниц.

9.4 В одной конкретной вычислительной системе с комбинированной странично-сегментной виртуальной памятью предусматривается уровень мультипрограммирования, выражаемый показателем 10 (одновременно могут выполняться 10 программ). Страницы команд (реентерабельный код) отделены от страниц данных (допускающих модификацию). Вы исследовали эту систему в работе и сделали следующие наблюдения: (1) большинство процедурных сегментов содержат по много страниц и (2) большинство сегментов данных используют только незначительную часть страницы.

Ваш помощник предложил в качестве одного из способов повышения эффективности использования памяти объединять несколько сегментов данных каждого пользователя в одной индивидуальной странице. Прокомментируйте это предложение с учетом следующих факторов:

- а) использование памяти,

- б) эффективность выполнения программ,
- в) бесконечное откладывание,
- г) защита,
- д) разделение ресурсов.

9.5В настоящее время среди специалистов наблюдается большой интерес к упреждающей подкачке страниц и к упреждающему распределению ресурсов вообще. Какую полезную информацию может предоставить механизму упреждающей подкачки страниц каждый из следующих объектов?

- а) программа пользователя,
- б) языковой транслятор,
- в) операционная система,
- г) журнал с данными о прошлых прогонах программы.

9.6Известно, что в общем случае мы не можем предсказать путь выполнения произвольной программы. А если бы мы могли, то мы оказались бы в состоянии решить проблему останова, которая, как известно, неразрешима. Объясните, каким образом это обстоятельство влияет на эффективность механизмов упреждающего распределения ресурсов.

9.7Предположим, что блок управления памятью при принятии решения о выталкивании страницы должен выбрать одну из двух страниц. Предположим, что одна из этих страниц разделяется несколькими процессами, а другая используется только одним процессом. Должен ли блок управления вытолкнуть эту монопольно используемую страницу? Объясните.

9.8Обсудите каждый из следующих нетрадиционных алгоритмов выталкивания страниц применительно к мультипрограммной системе с виртуальной памятью, обслуживающей пользователей как в пакетном, так и в диалоговом режимах:

- а) «Глобальный LIFO» — выталкивается страница, самой последней поступившая в физическую память.
- б) «Локальный LIFO» — выталкивается самая последняя по времени поступления страница, вызванная процессом, который запросил поступающую новую страницу.
- в) «Утомленная страница» — выталкивается страница, подвергавшаяся наиболее интенсивным обращениям в системе. (Рассмотрите как глобальный, так и локальный варианты этого алгоритма.)
- г) «Истрепанная страница» — выталкивается страница, подвергавшаяся наиболее интенсивным модификациям в системе. (Рассмотрите как глобальный, так и локальный варианты этого алгоритма.)

9.9Некоторые типы процессов хорошо работают при одних стратегиях выталкивания страниц и плохо при других. Обсудите возможность реализации блока управления памятью, который динамически определял бы тип процесса, а затем выбирал бы и использовал соответствующую стратегию выталкивания страниц памяти для этого процесса.

9.10Предположим, что блок управления памятью принимает решение о том, какую страницу следует вытолкнуть, исключительно на основе анализа битов-признаков обращения и модификации для каждого страничного кадра. Приведите несколько примеров некорректных решений, которые мог бы принять такой блок управления.

9.11Перечислите несколько причин, по которым на некоторые страницы должен действовать запрет на выталкивание из первичной памяти.

9.12 Почему в общем случае более целесообразно выталкивать немодифицированную страницу, а не страницу, подвергавшуюся модификации? При каких обстоятельствах может оказаться более целесообразным выталкивать модифицированную страницу?

9.13 Для каждой из приведенных ниже пар стратегий выталкивания укажите последовательность обращений к страницам, при которой обе стратегии выбрали бы для выталкивания (1) одну и ту же страницу и (2) разные страницы:

- а) LRU, NUR,
- б) LRU, LFU,
- в) LRU, FIFO,
- г) NUR, LFU,
- д) LFU, FIFO,
- е) NUR, FIFO.

9.14 Оптимальная стратегия подкачки страниц (OPT) практически неосуществима, поскольку невозможно предсказать будущее. Существуют, однако, обстоятельства, при которых эту стратегию реализовать можно. Что это за обстоятельства?

9.15 Затраты на перепись модифицированной страницы во внешнюю память, с тем чтобы можно было освободить место для новой страницы, настолько велики, что этого по возможности следует избегать. Предположим, что блок управления памятью выбирает для замещения модифицированную страницу. Эту страницу необходимо записать во внешнюю память, чтобы в ее освободившийся страничный кадр можно было поместить новую страницу. Поэтому блок управления планирует произвести выталкивание указанной страницы и заносит ее в список страниц, ожидающих выталкивания. Таким образом, страница будет оставаться в первичной памяти в течение некоторого времени, прежде чем будет вытолкнута во внешнюю память. Предположим теперь, что именно в это время указанную страницу запросит выполняющийся процесс. Как должен отреагировать блок управления памятью на запрос? При ответе не упускайте из виду возможность бесконечного откладывания и необходимость гарантировать приемлемые времена реакции для интерактивных пользователей.

9.16 Стратегию замещения FIFO относительно просто реализовать, причем с малыми издержками. Однако эта стратегия вполне может выбрать для замены и активно используемую страницу. Предложите простую модификацию стратегии FIFO, которая также характеризовалась бы легкостью реализации, малыми издержками и в то же время не позволяла заменять интенсивно используемую страницу.

9.17 Приведите пример аномалии FIFO, отличающийся от указанного в тексте.

9.18 Разработайте экспериментальную программу для демонстрации явления локальности в системе со страничной организацией.

9.19 Программист, который пишет программы, принимая специальные меры для обеспечения высокой степени локальности, может ожидать заметного повышения эффективности работы своих программ. Перечислите несколько принципов, которые может использовать программист для улучшения локальности. В частности, какие средства языков высокого уровня следует применять более широко? Применения каких средств следует избегать?

9.20 Предположим, что вы подключились к каналу ввода-вывода и наблюдаете активное движение страниц. Означает ли это, что имеет место трешинг? Объясните.

9.21 Почему глобальная стратегия выталкивания страниц может оказаться более подверженной трешингу, чем локальная?

9.22 Обсудите взаимосвязи между предоставлением каждому процессу большего

числа страничных кадров, чем ему реально требуется (чтобы предотвратить трешинг), и получающейся фрагментацией первичной памяти.

9.23Предложите несколько эвристических правил, которые мог бы использовать блок управления памятью, чтобы определять, не произошла ли перегрузка первичной памяти.

9.24Рабочее множество страниц процесса может иметь несколько определений. Обсудите достоинства каждой из приведенных ниже схем выбора страниц, составляющих рабочее множество процесса:

а) те страницы, к которым процесс обращался в течение последних w секунд календарного времени;

б) те страницы, к которым процесс обращался в течение последних w секунд виртуального времени (т. е. времени, в течение которого этот процесс фактически имел в своем распоряжении центральный процессор);

в) последние k различных страниц, к которым обращался процесс;

г) те страницы, к которым процесс делал свои последние r обращений за командами или данными;

д) те страницы, к которым процесс обращался в течение последних w секунд виртуального времени с частотой, превышающей f раз в виртуальную секунду.

9.25Приведите пример ситуации, при которой стратегия выталкивания страниц по рабочему множеству будет приводить к замене

а) наиболее удачной страницы,

б) наименее удачной страницы.

9.26Одна из сложностей реализации стратегии управления памятью по рабочим множествам заключается в том, что, когда процесс запрашивает новую страницу, трудно определить, переходит ли этот процесс к новому рабочему множеству или просто расширяет свое текущее рабочее множество. В первом случае блоку управления памятью лучше принять решение о замене одной из страниц этого процесса, а во втором — о добавлении еще одной страницы к уже выделенным страницам процесса. Как может блок управления решить, с какой ситуацией он столкнулся?

9.27Сам факт, что некоторая страница является членом рабочего множества процесса, не обязательно означает, что к ней производятся частые обращения. Аналогично к страницам рабочего множества процесса не обязательно производятся обращения с одной и той же частотой. Предложите модификацию стратегии подкачки по рабочим множествам, которая в случае необходимости замены страницы рабочего множества обеспечивала бы выбор этой страницы согласно одному из прочих алгоритмов, например FIFO, LRU, LFU, NUR и т. д. Сравните эту новую стратегию с чистым принципом рабочих множеств.

9.28Предположим, что все процессы, работающие в мультипрограммной системе, разместили свои рабочие множества в первичной памяти. При изменении картины локальности (распределения обращений) рабочие множества могут возрасти и первичная память может оказаться перегруженной, что приводит к трешингу. Обсудите относительные достоинства указанных ниже стратегий, направленных на предотвращение этого неприятного явления:

а) Никогда не инициировать новый процесс, если реальная память уже загружена на 80% и более.

б) Никогда не инициировать новый процесс, если реальная память уже загружена на 97% и более.

в) При инициировании нового процесса назначать для его рабочего множества максимальный

размер, за пределы которого нельзя будет выходить.

9.29 Критическим фактором с точки зрения обеспечения высоких скоростных характеристик является организация взаимодействия между различными компонентами операционной системы. Обсудите проблемы взаимодействия между блоком управления памятью и модулем инициирования задач в мультипрограммной схеме с виртуальной памятью. В частности, рассмотрите случай, когда блок управления памятью реализует метод управления памятью по рабочим множествам.

9.30 Рассмотрите приведенный ниже эксперимент и объясните наблюдаемые результаты.

Программа выполняется на машине со страничной организацией памяти, причем начинает работу со своей первой процедурной страницы. Во время выполнения программы необходимые страницы подкачиваются по запросу в имеющиеся страничные кадры, число которых значительно больше числа страниц программы. Однако вне компьютера имеется специальное наборное устройство, при помощи которого оператор ЭВМ может задавать максимальное количество страничных кадров, предоставляемых программе для использования.

Первоначально это наборное устройство устанавливается на два кадра и программа выполняется с начала до конца. Затем устройство устанавливается на три кадра и программа снова выполняется с начала до конца. Этот процесс повторяется до тех пор, пока на наборном устройстве не будет установлено полное количество имеющихся страничных кадров реальной памяти, и программа выполняется последний раз. При каждом прогоне фиксируется время выполнения программы. Наблюдаемые результаты:

При изменении числа кадров с двух до трех и затем до четырех времени выполнения программы резко уменьшаются. При переходе от четырех к пяти и затем к шести времени выполнения каждый раз также сокращаются, однако менее резко. После того как на наборном устройстве устанавливается семь кадров и более, времена выполнения программы остаются практически постоянными.

9.31 Разработчик операционной системы предложил так называемую PD-стратегию управления памятью, реализующую следующий алгоритм. Каждому активному процессу выделяются ровно два страничных кадра. В этих кадрах размещается самая последняя по времени обращения процедурная страница, Р-страница, и самая последняя по времени обращения страница данных, D-страница. Когда происходит прерывание по отсутствию страницы и если нужная страница процедурная, она заменяет Р-страницу, а если это страница данных, она заменяет D-страницу.

Разработчик говорит, что главное достоинство такого алгоритма заключается в том, что он до предела упрощает все аспекты управления памятью и благодаря этому характеризуется очень малыми накладными расходами.

а) Каким образом PD-алгоритм сказывается на реализации стратегий (1) вталкивания, (2) размещения и (3) выталкивания при управлении памятью.

б) При каких обстоятельствах PD-алгоритм фактически обеспечивает лучшие результаты, чем управление памятью по рабочим множествам?

в) При каких обстоятельствах PD-алгоритм дает неблагоприятные результаты.

9.32 Опишите, каким образом коллективное использование кода и данных затрагивает перечисленные ниже стратегии и явления:

а) стратегию вталкивания,

б) стратегию размещения,

в) стратегию выталкивания,

г) «локальную локальность» (т. е. локальность в рамках одного процесса),

д) «глобальную локальность» (т. е. локальность для всех процессов),

е) трешинг,

ж) назначение платы за использование ресурсов,

з) фрагментацию

(I) внутреннюю,

(II) внешнюю,

(III) блочную.

9.33 Приведите в сводном виде аргументы за и против (1) страницы малых размеров и (2) страницы больших размеров.

9.34 Система Multics первоначально разрабатывалась с ориентацией на использование страниц двух размеров — 64 слова и 1024 слова (в конце концов от такого подхода разработчики отказались).

а) Какие факторы, по вашему мнению, послужили предпосылками для принятия такого конструкторского решения?

б) Каким образом подобный подход с использованием страниц двух размеров сказался на стратегиях управления памятью?

9.35 Обсудите, каким образом в системах виртуальной памяти могут использоваться следующие аппаратные средства:

а) механизмы динамического преобразования адресов,

б) ассоциативная память,

в) кэш-память,

г) бит-признак обращения к странице,

д) бит-признак модификации страницы,

е) бит-признак «страница в процессе передачи» (означающий, что в настоящее время страница передается в конкретный страничный кадр).

9.36 Обсудите вопрос о том, каким образом стиль программирования влияет на скоростные характеристики системы со страничной организацией. При этом рассмотрите следующие подходы и проблемы:

а) нисходящее программирование,

б) минимальное использование операторов перехода GOTO,

в) модульность,

г) рекурсия,

д) итерация.

ЧАСТЬ 4

Управление процессорами

Глава 10

Планирование заданий и загрузки процессоров

Ничто не может в процессе развития точно следовать первоначальному плану. Рассчитывать на это — все равно что пытаться укачивать взрослого человека в колыбели младенца.

Эдмунд Бёрк

□

Каждая задача имеет по крайней мере одно решение достаточно простое, изящное и... неверное.

Г. Менкен

Не рассказывайте о том, как много и усердно вы работаете. Скажите лучше, что вам удалось сделать.

Джеймс Дж. Линг

В любом деле важнее всего исполнение.

Джозеф Аддисон

10.1 Введение

10.2 Уровни планирования

10.3 Цели планирования

10.4 Критерии планирования

10.5 Планирование с переключением и без переключения

10.6 Интервальный таймер или прерывающие часы-будильник

10.7 Приоритеты

10.7.1 Статические и динамические приоритеты

10.7.2 Покупаемые приоритеты

10.8 Планирование по сроку завершения

10.9 Планирование по принципу FIFO («первый пришедший обслуживается первым»)

10.10 Циклическое планирование (RR)

10.11 Размер кванта времени

10.12 Планирование по принципу SJF («кратчайшее задание — первым»)

10.13 Планирование по принципу SRT («по наименьшему остающемуся времени»)

10.14 Планирование по принципу HRN («по наибольшему относительному времени реакции»)

10.15 Многоуровневые очереди с обратными связями

10.1 Введение

Процессы получают возможность выполнять конкретную работу, когда в их распоряжение выделяются физические процессоры. Распределение процессоров по процессам представляет собой сложную задачу, которую решает операционная система. В этой главе мы рассмотрим проблемы, связанные с определением того, когда следует выделять процессоры и каким именно процессам. Это называется *планированием загрузки процессоров*.

10.2 Уровни планирования

Мы рассмотрим три основных уровня планирования (рис. 10.1).

- *Планирование на верхнем уровне.* Иногда называется *планированием заданий*. Средства этого уровня определяют, каким заданиям будет разрешено активно конкурировать за захват ресурсов системы. Этот вид планирования иногда называют также *планированием допуска*, поскольку на этом уровне определяется, какие задания будут допущены в систему. Вошедшие в систему задания становятся процессами или группами процессов.
- *Планирование на промежуточном уровне.* Средства этого уровня определяют, каким процессам будет разрешено состязаться за захват центрального процессора. Планировщик промежуточного уровня оперативно реагирует на текущие колебания системной нагрузки, кратковременно *приостанавливая* и вновь *активизируя* (или *возбуждая*) процессы, что обеспечивает равномерную работу системы и помогает

достижению определенных глобальных целевых скоростных характеристик. Таким образом, планировщик промежуточного уровня выполняет как бы функции буфера между средствами допуска заданий в систему и средствами предоставления ЦП для выполнения этих заданий.

- *Планирование на нижнем уровне.* Средства этого уровня определяют, какому из готовых к выполнению процессов будет предоставляться освободившийся ЦП, и фактически выделяют ЦП данному процессу (т. е. осуществляют *диспетчерские функции*). Планирование на нижнем уровне производится так называемым *диспетчером*, который работает с большой частотой, много раз в секунду, и поэтому всегда должен располагаться в основной памяти.

В данной главе мы обсудим многие стратегии планирования, применяемые в операционных системах, а также реализующие их *механизмы планирования*. Многие из рассматриваемых стратегий распространяются на планирование как заданий, так и процессов.

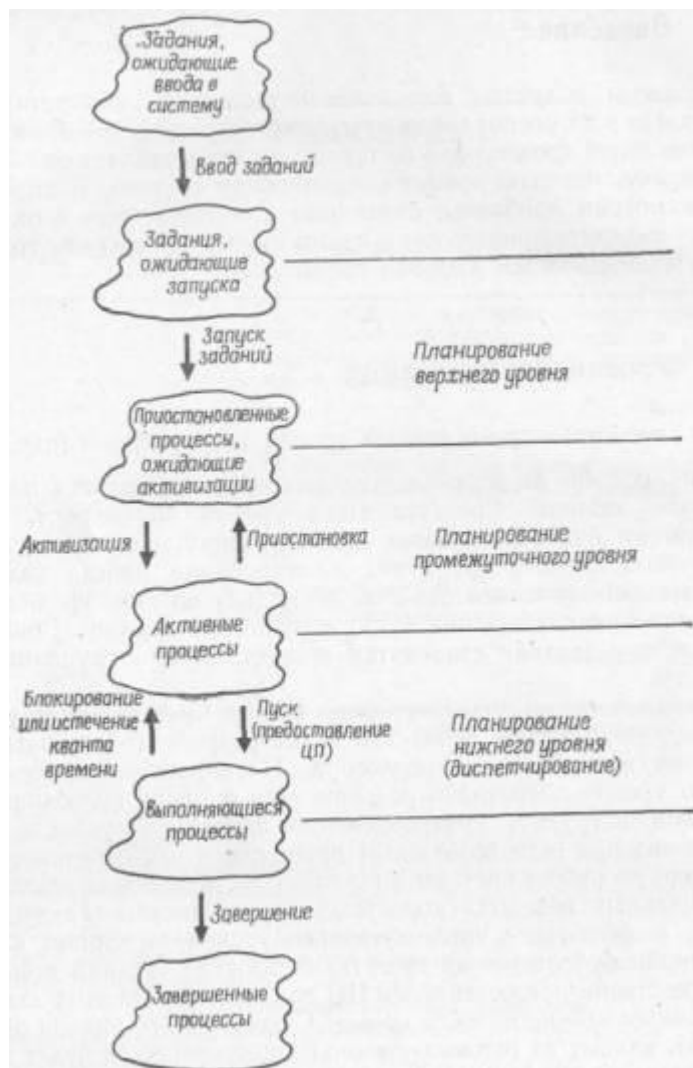


Рис. 10.1 Уровни планирования.

10.3 Цели планирования

Дисциплина планирования должна:

- Быть справедливой. Дисциплина планирования считается справедливой, если ко всем процессам она относится одинаково и ни один процесс не может пострадать от бесконечного откладывания.
- Обеспечивать максимальную пропускную способность системы. Дисциплина планирования должна стремиться к обслуживанию максимально возможного количества процессов в единицу времени.
- Обеспечивать максимальному числу пользователей, работающих в интерактивном режиме, приемлемые времена ответа (т.е. в худшем случае несколько секунд).
- Быть предсказуемой. Данное задание должно выполняться приблизительно за одно и то же время и с приблизительно одинаковой стоимостью независимо от нагрузки на систему.
- Минимизировать накладные расходы. Интересно отметить, что далеко не все специалисты считают подобную цель одной из наиболее важных. Накладные расходы обычно рассматриваются как потеря ресурсов. Однако определённая часть системных ресурсов, относящаяся к категории накладных расходов, может в действительности значительно улучшить общие характеристики системы.
- Сбалансировать использование ресурсов. Механизмы планирования должны стремиться к повышению коэффициента использования системных ресурсов. Предпочтение должно оказываться тем процессам, которые будут занимать недогруженные ресурсы.
- Обеспечивать определенный баланс между временем ответа и коэффициентом использования ресурсов. Наилучший способ гарантировать малые времена ответа - это иметь достаточно свободных ресурсов, чтобы их можно было сразу же использовать, когда они потребуются. Подобная стратегия приводит к снижению общего коэффициента использования ресурсов в системах реального времени минимальное время реакции.
- Исключать бесконечное откладывание. Во многих случаях бесконечное откладывание процессов может приводить к не менее неприятным последствиям, чем тупики. Чтобы, исключив бесконечное откладывание, лучше всего учитывать *старение* процесса - с увеличением периода времени, в течение которого процессу приходится ожидать некоторого ресурса приоритет этого процесса должен расти. В конце концов, приоритет окажется настолько высоким, что процессу будет предоставлен нужный ресурс.
- Учитывать приоритеты. В системах, в которых процессам присваиваются приоритеты, механизм планирования должен оказывать предпочтение процессам с более высокими приоритетами.
- Оказывать предпочтение процессам, занимающим ключевые ресурсы. Может возникнуть ситуация, когда процесс, даже имеющий низкий приоритет, удерживает за собой некоторый ключевой ресурс, который требуется процессам более высокого приоритета. Если этот ресурс не является оперативно перераспределяемым, то механизм планирования должен создавать для процесса лучшие, чем обычно, условия, чтобы этот процесс быстрее освободил удерживаемый им ключевой ресурс.
- Создавать лучшие условия для выполнения процессов, отличающихся более «примерным поведением» (например, требующих менее частой подкачки страниц).
- Характеризоваться постепенностью снижения работоспособности при больших нагрузках. Механизм планирования не должен сразу же терять работоспособность под тяжестью большой системной нагрузки. Он должен либо предотвращать чрезмерную нагрузку, не разрешая создавать новые процессы, когда нагрузка и так достаточно велика, либо в случае

увеличенной нагрузки он должен предоставлять постепенно снижаемый уровень обслуживания для всех процессов.

Многие из перечисленных целей противоречат друг другу, что делает организацию планирования весьма сложной проблемой.

10.4 Критерии планирования

Для того чтобы реализовать перечисленные выше цели планирования, механизм планирования должен учитывать следующие факторы:

- Лимитируется ли процесс вводом-выводом. Когда процесс получает в свое распоряжение ЦП, будет ли он занимать ЦП лишь кратковременно, прежде чем сформировать запрос на ввод-вывод.
- Лимитирует ли процесс ЦП. Когда процесс получает в свое распоряжение ЦП, будет ли он, как правило, использовать ЦП до тех пор, пока не истечет его квант времени?
- Является ли процесс пакетным или диалоговым. Пользователи, работающие в диалоговом режиме, выдают, как правило, довольно тривиальные запросы, на которые следует немедленно реагировать, чтобы гарантировать хорошие времена ответа. Пользователи, программы которых выполняются в пакетном режиме, непосредственно на машине не присутствуют, так что соответствующие процессы могут, как правило, без ущерба переносить значительные задержки.
- Насколько обязательной является быстрая реакция. Процесс, выполняющийся ночью в пакетном режиме, вряд ли потребует немедленной реакции системы. А система управления технологическими процессами реального времени, контролирующая работу нефтеочистительного предприятия, требует исключительно быстрого ответа, быть может, для предотвращения взрыва.
- Приоритетность процессов. Для процессов более высокого приоритета должны создаваться лучшие условия выполнения, чем для процессов более низкого приоритета.
- Насколько часто при выполнении процесса возникают прерывания по отсутствию нужных страниц. Можно полагать, что процессы, генерирующие малое число прерываний по отсутствию нужных страниц, уже собрали свои рабочие множества в основной памяти. А процессы, для которых характерно большое количество прерываний, пока еще не сформировали свои рабочие множества. Обычно считается наиболее целесообразным оказывать предпочтение процессам, которые сформировали свои рабочие множества. Существует и другая точка зрения — предпочтение должно оказываться процессам с высокими частотами прерываний по отсутствию нужных страниц, поскольку они занимают ЦП лишь короткие промежутки времени перед генерацией запроса на ввод-вывод.
- Насколько часто данный процесс приостанавливается из-за переключения на процесс более высокого приоритета. Часто приостанавливаемым процессам следует создавать менее благоприятные условия для продолжения работы. Это объясняется тем, что для запуска подобного процесса операционной системе каждый раз приходится идти на определенные накладные расходы, а короткое время работы процесса перед приостановкой их не оправдывает.
- Сколько времени уже получил данный процесс. Некоторые разработчики считают, что предпочтение должно оказываться процессу, который получил меньше времени. Другие разработчики считают, что процесс, которому уже

выделено много времени, должен быть ближе к завершению и поэтому ему надо оказывать предпочтение, чтобы помочь завершиться и как можно скорее покинуть систему.

- Сколько еще времени требуется данному процессу для завершения. Средние времена ожидания можно уменьшить, если раньше других запускать процессы, требующие меньшего времени для своего завершения. К сожалению, лишь в редких случаях точно известно, сколько времени необходимо каждому процессу для завершения.

10.5 Планирование с переключением и без переключения

Если после предоставления ЦП в распоряжение некоторого процесса отобрать ЦП у этого процесса нельзя, то говорят о дисциплине планирования *без переключения*. Если же ЦП можно отобрать, то говорят о дисциплине планирования *с переключением*.

Планирование с переключением необходимо в системах, в которых процессы высокого приоритета требуют немедленного внимания. Например, в системах реального времени пропажа одного важного сигнала прерывания может привести к катастрофическим последствиям. В интерактивных системах разделения времени планирование с переключением играет важную роль, поскольку позволяет гарантировать приемлемые времена ответа.

Планирование, требующее контекстных переключений, сопряжено с определенными накладными расходами. Чтобы обеспечить его эффективность, в основной памяти должно размещаться много процессов, с тем чтобы очередной процесс был, как правило, готов для выполнения, когда освобождается ЦП. А размещение неработающих программ в основной памяти также сопряжено с накладными расходами.

В системах планирования без переключения коротким заданиям приходится больше ждать из-за выполнения длительных заданий, однако для всех процессов создаются как бы равные условия. Времена ответа здесь более предсказуемы, поскольку поступающие задания высокого приоритета не могут оттеснить уже ожидающие задания.

При проектировании механизма планирования с переключением разработчик должен рассмотреть практически произвольную схему приоритетов (виртуально). Мы можем построить сложную, тщательно обоснованную схему приоритетного переключения, в то время как в действительности сами приоритеты назначаются далеко не обосновано. В операционных системах довольно часто бывает так, что создаются очень сложные механизмы, реализующие в достаточной степени произвольные схемы. Разработчик должен проявлять осторожность и мудрость — он должен тщательно оценивать каждый предлагаемый механизм, прежде чем приступить к его построению. Принцип «главное — это простота» имеет великий смысл, однако, если вам не удастся достигать поставленной цели простыми путями, вы должны по крайней мере стремиться к тому, чтобы создаваемые вами механизмы были эффективными и понятными.

10.6 Интервальный таймер или прерывающие часы-будильник

Говорят, что процесс работает, если в данный момент в его распоряжение

предоставлен ЦП. Если речь идет о процессе операционной системы, то это означает, что в данный момент работает операционная система, причем она может принимать решения, влияющие на функционирование всей машины. Чтобы не допустить монополизации системы пользователями (умышленной, либо случайной), в операционной системе предусмотрены механизмы, позволяющие отбирать ЦП у пользователя.

Операционная система устанавливает *часы* или *интервальный таймер* с целью генерации сигнала прерывания в некоторый конкретный момент времени в будущем (или по истечении некоторого интервала времени в будущем). После прерывания ЦП передается в распоряжение следующего процесса. Этот процесс сохраняет управление центральным процессором до тех пор, пока он добровольно не освободит ЦП, или не произойдет временное прерывание или какое-то другое прерывание не потребует внимания ЦП. Если работает программа пользователя и происходит временное прерывание, это прерывание вызывает включение в работу операционной системы. Операционная система при этом решает, какому процессу следует предоставить далее ЦП.

Временные прерывания помогают гарантировать приемлемые времена ответа для пользователей, работающих в диалоговом режиме, предотвращают «зависание» системы из-за зацикливания какой-то программы пользователя, а также позволяют процессам соответствующим образом реагировать на *события, зависящие от времени*. Процессы, которые должны работать периодически, зависят от временных прерываний.

10.7 Приоритеты

Система может присваивать приоритеты автоматически или они могут назначаться извне. Приоритеты могут быть заслуженными («заработанными») или купленными. Они могут быть либо статическими, либо динамическими. Они могут присваиваться по какому-то рациональному принципу или произвольно назначаться в ситуациях, когда системному механизму необходимо каким-то образом различать процессы, однако он не знает, какой из них в действительности более важен.

10.7.1 Статические и динамические приоритеты

Статические приоритеты не изменяются. Механизмы статической приоритетности легко реализовать и они сопряжены с относительно небольшими издержками. Однако они не реагируют на изменения в окружающей ситуации, изменения, которые могут сделать желательной корректировку приоритетов.

Механизмы *динамических приоритетов* реагируют на изменения в ситуации. Начальное значение приоритета, присвоенное процессу, может действовать в течение лишь короткого периода времени, после чего назначается новое, более подходящее значение. Схемы динамической приоритетности гораздо сложнее в реализации и связаны с большими издержками по сравнению со статическими схемами. Однако предполагается, что эти издержки оправдываются повышением реактивности системы.

10.7.2 Покупаемые приоритеты

Операционная система должна качественно обслуживать большой контингент пользователей на рациональных началах, однако она должна также предоставлять привилегированное обслуживание для тех ситуаций, когда какому-то из пользователей это требуется.

Пользователь, которому необходимо срочно выполнить свое задание, может с готовностью пойти на дополнительную оплату более высокого уровня обслуживания. Такая дополнительная оплата вполне оправдана, поскольку для этого могут потребоваться ресурсы других пользователей, платящих за них деньги. Если бы не взималась повышенная плата за привилегии, то все пользователи запрашивали бы более высокий уровень обслуживания.

10.8 Планирование по сроку завершения

При планировании по сроку завершения принимаются все меры для того, чтобы выполнение определенных заданий заканчивалось к заранее обусловленному времени, или конечному сроку. Результаты этих заданий могут иметь очень важное значение, если они будут получены вовремя, и могут оказаться бесполезными после назначенного срока. Пользователь зачастую с готовностью внесет дополнительную плату, лишь бы система гарантированно завершила выполнение его задания в назначенное время. Планирование по сроку завершения сложно организовать по многим причинам.

- Пользователь должен заранее точно указывать, какие ресурсы потребуются для выполнения его задания. А подобную информацию можно иметь лишь в редких случаях.
- Система должна выполнять задания к сроку без серьезного снижения уровня обслуживания других пользователей.
- Система должна тщательно распределять загрузку своих ресурсов вплоть до указанного срока завершения задания. А это может быть достаточно трудно, поскольку могут поступать новые задания, предъявляющие к системе непредсказуемые требования.
- Если приходится иметь дело одновременно со многими заданиями, для которых задается срок выполнения, то планирование может настолько усложниться, что потребуются специальные методы оптимизации, гарантирующие соблюдение всех сроков.
- Активное управление ресурсами, необходимое при планировании по сроку завершения, может быть связано с существенными накладными расходами. Хотя пользователи согласны идти на достаточно высокую плату за предоставление услуг, обеспечивающих выполнение их заданий в указанные сроки, общий объем затрачиваемых на это системных ресурсов может оказаться настолько большим, что остальные пользователи, работающие с системой, будут страдать от ухудшенного обслуживания. Разработчики операционных систем должны тщательно учитывать возможность подобных конфликтов.

10.9 Планирование по принципу FIFO («первый пришел первый обслуживается»)

По-видимому, наиболее простой дисциплиной планирования является принцип

FIFO («первый пришедший обслуживается первым») (рис. 10.2). Центральный процессор предоставляется процессам в порядке их прихода в очередь готовности. После того как процесс получает ЦП в свое распоряжение, он выполняется

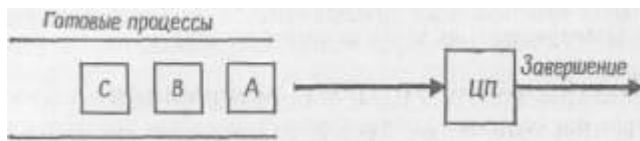


Рис. 10.2 Планирование по принципу FIFO («первый пришедший обслуживается первым»).

до завершения. Принцип FIFO — это дисциплина планирования без переключения. Если рассуждать формально, то этот принцип можно считать справедливым, однако он все-таки в определенном смысле несправедлив тем, что длинные задания заставляют ждать короткие задания, а менее важные задания — более важные. Принцип FIFO характеризуется относительно небольшими колебаниями времен ответа и поэтому большей предсказуемостью, чем большинство других дисциплин обслуживания. Его не рекомендуется применять при планировании заданий пользователей, работающих в интерактивном режиме", поскольку он не может гарантировать хорошие времена ответа.

В современных системах принцип FIFO редко используется самостоятельно в качестве основной дисциплины обслуживания, а чаще комбинируется с другими схемами. Например, во многих схемах планирования предусматривается диспетчирование процессов в соответствии с их приоритетами, однако процессы с одинаковыми уровнями приоритета диспетчируются по принципу FIFO.

10.10 Циклическое планирование (RR)

При *циклическом*, или *круговом*, планировании (round robin, RR) (рис. 10.3) диспетчирование процессов осуществляется по принципу FIFO, однако каждый раз процессу предоставляется ограниченное количество времени ЦП, называемое *временным квантом*. Если процесс не заканчивается до истечения выделенного ему кванта времени ЦП, ЦП у него отбирается и предоставляется следующему ожидающему процессу. Процесс, у которого перехватили ЦП, переходит в конец списка готовых к выполнению процессов.

Дисциплина циклического обслуживания эффективна для работы с разделением времени, когда система должна гарантировать

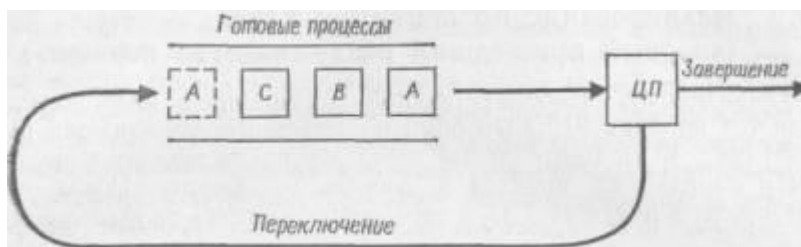


Рис. 10.3 Планирование по циклическому принципу (RR).

приемлемые времена ответа для всех интерактивных пользователей. Накладные расходы на диспетчирование здесь удастся снизить за счет эффективных механизмов контекстного переключения и благодаря предоставлению достаточного объема основной памяти чтобы процессы могли размещаться в ней одновременно.

10.11 Размер кванта времени

Определение размера кванта времени имеет критическое значение для эффективной работы вычислительной системы. Следует ли выбирать большой или малый квант времени? Следует ли делать его фиксированным или переменным? Следует ли задавать одно и то же значение кванта времени для всех пользователей или следует определять его индивидуально для каждого пользователя?

Прежде всего рассмотрим поведение системы в случаях, когда квант времени становится либо очень большим, либо очень маленьким. Если квант времени становится очень большим, то каждому процессу предоставляется практически столько времени, сколько ему требуется для завершения, так что циклическое планирование по сути вырождается в планирование по принципу FIFO. Если квант времени становится очень маленьким, то накладные расходы на контекстные переключения начинают играть доминирующую роль, причем характеристики системы в конце концов настолько ухудшаются, что с какого-то момента основное время затрачивается на переключение процессора, так что лишь незначительная часть времени остается, если вообще остается, на выполнение вычислений для пользователей.

Так в какой же точке между нулем и бесконечностью следует выбирать длину кванта времени q ? Рассмотрим следующий эксперимент. Предположим, что существует некая круговая шкала, имеющая метки от $q=0$ до $q=\text{бесконечность}$. Мы начинаем эксперимент с момента, когда ручка шкалы указывает на нуль. При повороте ручки величина кванта времени для системы меняется. Предположим, что система работает и обслуживает много интерактивных пользователей. Когда ручка шкалы только начинает вращаться, значения кванта времени близки к нулю и накладные расходы на переключения занимают основную часть ресурса ЦП. Перед интерактивными пользователями предстает неповоротливая система, характеризующаяся длительными временами ответа. При дальнейшем повороте ручки размер кванта времени увеличивается и времена ответа системы улучшаются. В какой-то момент мы достигаем точки, когда процент накладных расходов настолько снижается, что ЦП начинает обслуживать пользователей. Однако времена ответа по-прежнему остаются не слишком хорошими.

При дальнейшем повороте ручки времена ответа продолжают улучшаться. В определенный момент система начинает быстро реагировать на запросы пользователей. Однако пока еще не ясно, оптимально ли установленное таким образом значение кванта времени. Ручка поворачивается несколько дальше и времена ответа еще немного улучшаются. Однако затем при дальнейшем повороте ручки времена ответа снова начинают расти. Это происходит потому, что квант времени достигает размера, достаточного для того, чтобы каждый пользователь, получивший в свое распоряжение ЦП, успевал завершить свою программу. При этом дисциплина циклического планирования вырождается в принцип FIFO, при котором более длительные процессы заставляют ждать более короткие, причем среднее время ожидания увеличивается, поскольку эти более длительные процессы выполняются до конца, прежде чем уступить ЦП.

Рассмотрим предположительно оптимальное значение кванта времени

(небольшую долю секунды), при котором обеспечиваются хорошие времена ответа. Чем именно характеризуется подобный квант времени? Он достаточно велик, так что подавляющее большинство интерактивных запросов требует для своего обслуживания меньшего времени, чем длительность кванта. Когда интерактивный процесс начинает выполняться, он, как правило, использует ЦП в течение некоторого времени, после чего генерирует запрос ввода-вывода. Когда запрос ввода-вывода выдан, этот процесс уступает ЦП следующему процессу. Поскольку величина кванта больше, чем это время вычислений до формирования запроса ввода-вывода, процессы пользователей выполняются практически с максимальной скоростью. Каждый раз, когда процесс пользователя получает в свое распоряжение ЦП, он с большой вероятностью доработает до момента выдачи запроса ввода-вывода. Благодаря этому сводятся к минимуму накладные расходы на диспетчирование, обеспечивается максимальное использование ресурсов ввода-вывода и относительно короткие времена ответа.

Так какова же величина подобного оптимального кванта времени? Очевидно, что она меняется от системы к системе, причем меняется в зависимости и от нагрузок. Она меняется также от процесса к процессу, однако наш эксперимент не рассчитан на измерение различий для отдельных процессов.

Если все процессы лимитируются ЦП, то вообще не имеет смысла переключаться с процесса на процесс. Это объясняется тем, что затраты на переключение просто вычитаются из общей производительности системы. Если, однако, в мультипрограммной смеси принимают участие какие-либо интерактивные пользователи, то ЦП необходимо все же периодически переключать, чтобы гарантировать приемлемые времена ответа для этих пользователей.

10.12 Планирование по принципу SJF («кратчайшее задание — первым»)

Принцип SJF («кратчайшее задание — первым») — это дисциплина планирования без переключения, согласно которой следующим для выполнения выбирается ожидающее задание (или процесс) с минимальным оценочным рабочим временем, остающимся до завершения. Принцип SJF обеспечивает уменьшение среднего времени ожидания по сравнению с дисциплиной FIFO. Однако времена ожидания при этом колеблются в более широких пределах (т. е. менее предсказуемы), чем в случае FIFO, особенно при больших заданиях.

Дисциплина SJF оказывает предпочтение коротким заданиям (или процессам) за счет более длинных. Многие разработчики считают, что чем короче задания, тем лучшие условия для его выполнения следует создавать. Далеко не все специалисты придерживаются подобного мнения, особенно применительно к случаям, когда необходимо учитывать приоритетность заданий.

Механизм SJF выбирает для обслуживания задания таким образом, чтобы очередное задание завершалось и покидало систему как можно быстрее. Благодаря этому обеспечивается уменьшение количества ожидающих заданий, а также количества заданий, стоящих в очереди за длинными заданиями. В результате дисциплина SJF позволяет свести к минимуму среднее время ожидания для заданий, проходящих через систему.

Очевидная проблема, связанная с реализацией принципа SJF, состоит в том, что он требует точно знать, сколько времени будет выполняться задание или процесс, а такой информации обычно не бывает. Самое большее, что может сделать механизм SJF, — это полагаться на оценочные значения времен выполнения,

указываемые пользователями. В условиях производственного счета, когда одни и те же задачи выполняются регулярно, довольно точные и обоснованные оценки, по-видимому, возможны. Однако в исследовательской работе пользователи редко знают, как долго будут выполняться их программы.

Ориентация механизма планирования на оценки пользователей приводит к любопытным последствиям. Если пользователи знают, что система оказывает предпочтение заданиям с малыми оценочными временами выполнения, они могут задавать малые оценочные времена. Можно, однако, спроектировать планировщик заданий таким образом, чтобы избавить их от подобного соблазна. Пользователя можно заранее предупредить о том, что в случае, если его задание будет выполняться дольше оценочного времени, то оно будет просто выводиться из системы, причем пользователю придется платить за всю работу. Второе возможное решение заключается в том, чтобы выполнять задание в течение указанного оценочного времени плюс небольшой дополнительный процент, а затем «консервировать» задание (De68), т. е. запоминать его в текущем виде, с тем чтобы впоследствии можно было вновь продолжить его выполнение. При этом пользователю, естественно, придется оплачивать затраты на консервацию и последующую расконсервацию (т. е. перезапуск), и, кроме того, он пострадает от задержки в завершении его задания. Существует еще одно возможное решение — выполнять задание в течение оценочного времени по обычному тарифу, а затем за дополнительное время предъявлять счет по повышенному тарифу, значительно превышающему обычный. При таких условиях пользователь, указывающий нереально низкие оценочные времена, чтобы получить улучшенное обслуживание, будет в итоге платить значительно больше, чем обычно.

Принцип SJF подобно FIFO — это дисциплина обслуживания без переключения, поэтому ее не рекомендуется применять в системах разделения времени, где необходимо гарантировать приемлемые времена ответа.

10.13 Планирование по принципу SRT («по наименьшему остающемуся времени»)

Принцип SRT — это аналог принципа SJF, но с переключением, применимый в системах с разделением времени. По принципу SRT всегда выполняется процесс, имеющий минимальное оценочное время до завершения, причем с учетом новых поступающих процессов. По принципу SJF задание, которое запущено в работу, выполняется до завершения. По принципу SRT выполняющийся процесс может быть прерван при поступлении нового процесса, имеющего более короткое оценочное время работы. Чтобы механизм SRT был эффективным, опять-таки нужны достаточно точные оценки будущего, причем разработчик системы должен позаботиться о мерах против неправильного использования прикладными программистами особенностей стратегий планирования.

Дисциплина SRT характеризуется более высокими накладными расходами, чем SJF. Механизм SRT должен следить за текущим временем обслуживания выполняющегося задания и обрабатывать возникающие прерывания. Поступающие в систему небольшие процессы будут выполняться почти немедленно. Однако более длительные задания будут иметь даже большее среднее время ожидания и больший разброс времени ожидания, чем в случае SJF.

Реализация принципа SRT требует, чтобы регистрировались истекшие времена обслуживания, а это приводит к увеличению накладных расходов. Теоретически принцип SRT обеспечивает минимальные времена ожидания. Однако из-за издержек на переключения может оказаться так, что в определенных ситуациях в

действительности лучшие показатели будет иметь SJF.

Предположим, что текущее задание уже почти завершается, а в этот момент поступает новое задание с очень малым оценочным временем обслуживания. Следует ли переключаться? Механизм планирования, реализующий чистый принцип SRT, произвел бы переключение, однако целесообразно ли это? Проблему можно решить, если предусмотреть некоторое пороговое значение остающегося времени выполнения, с тем чтобы в случае, если текущему заданию требуется для своего завершения меньше указанного количества времени, система гарантировала бы его завершение без прерываний.

Предположим, что поступает задание, оценочное время которого лишь немногим меньше времени, необходимого для завершения текущего задания, причем в основном процессорного времени. В этом случае механизм, реализующий чистый принцип SRT, также пошел бы на переключение. Если, однако, затраты на переключение превышают разность времен обслуживания обоих заданий, то прерывание текущего задания фактически приведет к снижению производительности системы. Цель всех этих рассуждений состоит в том, чтобы показать, что *разработчики операционных систем должны тщательно взвешивать накладные расходы механизмов управления ресурсами в сравнении с предположительными выгодами.*

10.14 Планирование по принципу HRN («по наибольшему относительному времени реакции»)

Бринк Хансен (Br71) разработал стратегию планирования HRN (highest-response-ratio-next, «по наибольшему относительному времени ответа»), которая компенсирует некоторые из слабостей, присущих дисциплине SJF, в частности чрезмерное предубеждение против длинных заданий и чрезмерную благосклонность по отношению к коротким новым заданиям. HRN — это дисциплина планирования без переключений, согласно которой приоритет каждого задания является не только функцией времени обслуживания этого задания, но также времени, затраченного заданием на ожидание обслуживания. После того как задание получает в свое распоряжение ЦП, оно выполняется до завершения. Динамические приоритеты при дисциплине HRN вычисляются по формуле

$$\text{приоритет} = \frac{\text{время ожидания} + \text{время обслуживания}}{\text{время обслуживания}}$$

Поскольку время обслуживания находится в знаменателе, предпочтение будет оказываться более коротким заданиям. Однако, поскольку в числителе имеется время ожидания, более длинные задания, которые уже довольно долго ждут, будут также получать определенное предпочтение. Отметим, что сумма

$$\text{время ожидания} + \text{время обслуживания}$$

есть время ответа системы для данного задания, если бы это задание инициировалось немедленно.

10.15 Многоуровневые очереди с обратными связями

Когда процесс получает в свое распоряжение ЦП, особенно если он до сих пор не имел возможности каким-то образом проявить себя, планировщик не имеет ни малейшего представления о том, какое именно количество времени ЦП потребуется данному процессу. Процессы, лимитируемые вводом-выводом, могут использовать ЦП лишь кратковременно, перед тем как сформировать запрос ввода-вывода. Процессы, лимитируемые ЦП, могут занимать его часами, если нет переключений.

Механизм планирования должен:

- оказывать предпочтение коротким заданиям;
- оказывать предпочтение заданиям, лимитируемым вводом-выводом, чтобы обеспечить хороший коэффициент использования устройств ввода-вывода;
- как можно быстрее определять характер задания и соответствующим образом планировать выполнение этого задания.

Многоуровневые очереди с обратными связями (рис. 10.4) представляют структуру, которая обеспечивает достижение этих целей. Новый процесс входит в сеть очередей с конца верхней очереди. Он перемещается по этой очереди, реализующей принцип FIFO, пока не получит в свое распоряжение ЦП. Если задание завершается или освобождает ЦП, чтобы подождать завершения операции ввода-вывода или наступления некоторого события, то оно, это задание, выходит из сети очередей. Если выделенный квант времени истекает до того, как процесс добровольно освободит ЦП, этот процесс помещается в конец следующей очереди более низкого уровня. Этот процесс в следующий раз получит в свое распоряжение ЦП, когда он достигнет начала данной очереди, если при этом не будет ожидающих в первой очереди. Если данный процесс будет продолжать использовать полный квант времени, предоставляемый на каждом уровне, он продолжит переход в конец очереди следующего ниже лежащего уровня. Обычно в системе предусматривается некоторая очередь самого нижнего уровня, которая реализует принцип циклического обслуживания и в которой данный процесс циркулирует до тех пор, пока не завершится.

Во многих структурах многоуровневых очередей с обратными связями квант времени, предоставляемый данному процессу при Переходе в каждую очередь более низкого уровня, увеличивается.

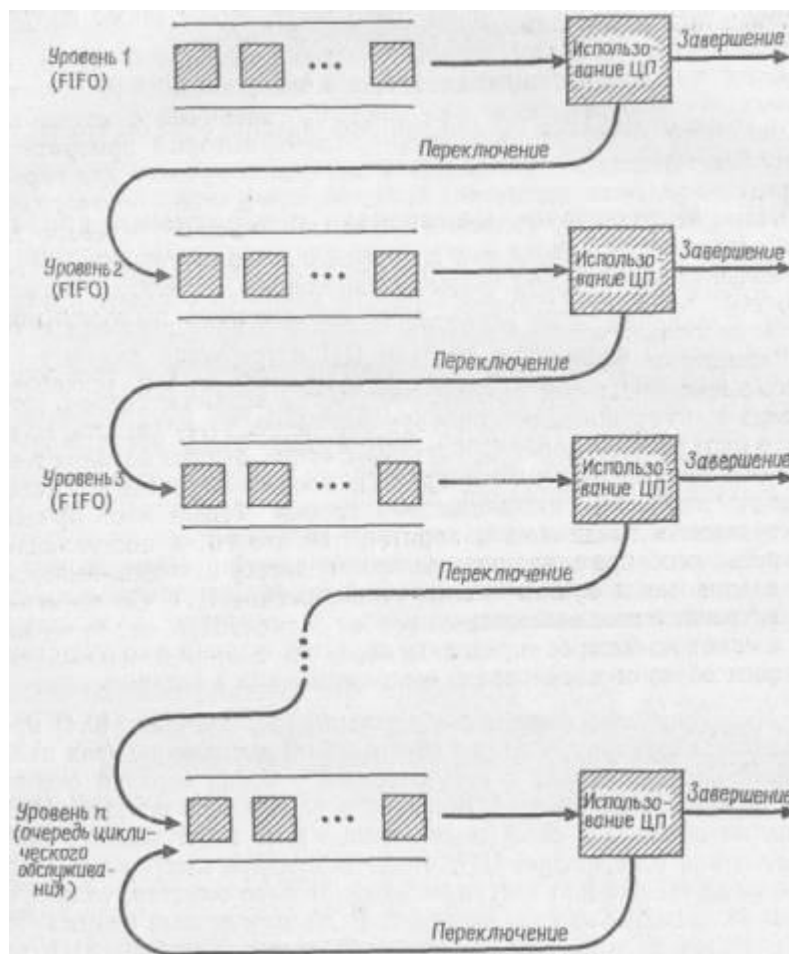


Рис. 10.4 Многоуровневые очереди с обратными связями.

Таким образом, чем дольше находится процесс в сети очередей, тем больший квант времени выделяется ему с каждым разом, когда он получает в свое распоряжение ЦП. Однако ему не удастся получить ЦП слишком часто, поскольку процессы, находящиеся в очередях более высоких уровней, имеют и более высокий приоритет. Процесс, находящийся в данной очереди, может начать выполнение только в случае, если нет ожидающих во всех очередях более высоких уровней. Выполняющийся процесс прерывается, если поступает новый процесс в очередь более высокого уровня.

Теперь, после того как мы описали работу сети очередей, рассмотрим, каким образом подобный механизм реагирует на различные типы процессов. Этот механизм должен оказывать предпочтение процессам, лимитируемым вводом-выводом, чтобы обеспечить хорошую загрузку устройств ввода-вывода и малое время реакции на запросы интерактивных пользователей. И действительно, так и будет, поскольку процесс, связанный с вводом-выводом, будет поступать в очередь с очень высоким приоритетом и ему будет быстро предоставляться ЦП. Квант времени для первой очереди выбирается достаточно большим, с тем, чтобы подавляющее большинство заданий, связанных с вводом-выводом, успевало выдать запрос ввода-вывода еще до истечения первого кванта. Когда подобный процесс выдает запрос ввода-вывода, он выходит из сети очередей, причем, получив истинно приоритетное обслуживание, что и требовалось.

Рассмотрим теперь задание, лимитируемое ЦП и требующее много времени ЦП.

Это задание поступает в верхнюю очередь сети, очередь с очень высоким приоритетом. Первый раз задание получает в свое распоряжение ЦП весьма быстро, однако после истечения выделенного ему кванта времени процесс переходит в конец очереди следующего ниже лежащего уровня. Теперь этот процесс будет иметь более низкий приоритет, так что вновь поступающие процессы, особенно связанные по преимуществу с вводом-выводом, будут первыми получать в свое распоряжение ЦП. Со временем данный вычислительный процесс все же получит ЦП, ему будет выделен квант времени большей величины, чем в очереди наивысшего приоритета, но он снова не успеет завершиться по истечении этого полного кванта. Затем он будет помещен в конец очереди следующего ниже лежащего уровня. Таким образом данный процесс будет продолжать переходить в очереди с более низкими приоритетами, будет дольше ждать в промежутках между выделяемыми ему квантами времени и каждый раз полностью использовать свой квант, когда будет получать в свое распоряжение ЦП (если при этом не будет прерываться из-за поступления процесса с более высоким приоритетом). В конце концов, этот вычислительный процесс окажется в очереди самого низкого уровня с циклическим обслуживанием, где и будет циркулировать, пока не завершится.

Многоуровневые очереди с обратными связями — это идеальный механизм, позволяющий разделять процессы на категории в соответствии с их потребностями во времени ЦП. В системе с разделением времени каждый раз, когда процесс выходит из сети очередей, он может быть помечен признаком очереди самого низкого уровня, в которой он побывал. Когда этот процесс впоследствии вновь войдет в сеть очередей, он будет направлен прямо в ту очередь, в которой он последний раз завершал свою работу. Здесь планировщик действует на основе следующего эвристического правила: поведение процесса в недавнем прошлом может служить хорошим ориентиром для определения его поведения в ближайшем будущем. Поэтому процесс, лимитируемый ЦП, при своем возвращении в сеть очередей не будет помещаться в очереди более высоких уровней, где он только мешал бы обслуживать короткие процессы высокого приоритета или процессы, лимитируемые вводом-выводом.

Если процессы всегда помещать в сеть очередей на самый низкий уровень, который они занимали прошлый раз, то система не сможет реагировать на изменения характера процесса, например на то, что процесс, бывший по преимуществу вычислительным, становится преимущественно «обменным». Эту проблему можно решить, если в метке, которой сопровождается процесс, указывать длительность его выполнения при последнем пребывании в сети очередей. Когда такой процесс вновь поступит в сеть очередей, его можно будет поместить в нужную очередь. Если процесс войдет в новую фазу, в которой он вместо лимитируемого ЦП станет обменным, первоначально он будет несколько страдать от «неповоротливости» системы, пока система не определит, что характер процесса изменился. Однако описываемый механизм планирования довольно быстро реагирует на подобные изменения. Еще один путь обеспечения быстрой реакции системы на изменения в поведении процесса — это позволить процессу перемещаться на один уровень вверх в сети очередей каждый раз, когда он добровольно освободит ЦП еще до истечения выделяемого ему кванта времени.

Многоуровневые очереди с обратными связями — показательный пример адаптивного механизма, механизма, который может реагировать на изменение поведения контролируемой им системы. Адаптивные механизмы, как правило, требуют больших накладных расходов, чем неадаптивные, однако присущая им чувствительность к изменениям режима работы делает систему более реактивной и компенсирует накладные расходы.

Одним из распространенных вариантов механизма многоуровневых очередей с обратными связями является вариант, в котором процесс несколько раз циркулирует в каждой очереди, прежде чем перейти в очередь следующего

нижележащего уровня. Как правило, количество подобных циклов в каждой очереди увеличивается по мере перехода процесса на нижележащие уровни.

Заключение

Планировщики процессоров определяют, когда следует предоставлять процессоры и каким процессам. Планирование на верхнем уровне, или планирование заданий, определяет, какие задания будут допускаться в систему. После допуска в систему эти задания становятся процессами или группами процессов. Планирование на нижнем уровне, или диспетчирование, определяет, какому из готовых к выполнению процессов будет предоставлен в распоряжение центральный процессор в следующий раз. Планирование на промежуточном уровне определяет, каким процессам будет разрешено состязаться за захват ЦП и какие процессы будут кратковременно приостанавливаться в связи с текущими изменениями в нагрузке на систему.

Дисциплина планирования должна быть справедливой, обеспечивать максимальную пропускную способность системы, приемлемые времена ответа для максимального количества пользователей, работающих в интерактивном режиме, предсказуемость, минимальные накладные расходы, сбалансированное использование ресурсов, сбалансированность времени ответа и коэффициента использования ресурсов, должна исключать бесконечное откладывание процессов, учитывать приоритеты, оказывать предпочтение тем процессам, которые занимают ключевые ресурсы, предусматривать улучшенное обслуживание для процессов, отличающихся «примерным поведением», и «плавно деградировать» при увеличении нагрузок. Многие из этих целей противоречат друг другу, что делает планирование весьма сложной проблемой.

Для достижения указанных целей механизм планирования должен учитывать, лимитируется ли процесс вводом-выводом или вычислениями, является ли процесс пакетным или диалоговым, обязательно ли малое время ответа, приоритет каждого процесса, частоту генерации каждым процессом прерываний по отсутствию нужных страниц, частоту переключений с низкоприоритетных процессов на поступающие процессы более высокого приоритета, приоритеты процессов, ожидающих освобождения уже занятых ресурсов, длительность периода времени, в течение которого ожидает каждый процесс, суммарное время работы каждого процесса и оценочное время, необходимое каждому процессу для завершения.

Планирование считается планированием без переключения, если центральный процессор нельзя отнять у занимающего его процесса; в противном случае планирование считается планированием с переключением. Планирование с переключением играет важную роль в мультипрограммных системах, в которых некоторые процессы должны обслуживаться очень быстро, причем особенно важное значение они имеют в системах реального времени и системах с разделением времени.

Полезной компонентой системы планирования с переключением является интервальный таймер, или прерывающие часы. По истечении заданного временного интервала таймер генерирует сигнал прерывания, по которому управление центральным процессором переходит от текущего процесса к операционной системе. Операционная система может после этого запустить в работу следующий процесс.

Статические приоритеты остаются одинаковыми на всем протяжении выполнения процесса; динамические приоритеты меняются в ответ на изменения ситуации в системе. Некоторые системы предоставляют пользователю возможность

покупать для своих программ более высокие приоритеты.

Планирование по сроку завершения организуется таким образом, чтобы определенные процессы завершались в определенные сроки. Планирование по сроку завершения — это сложная проблема, особенно в условиях, когда в промежутке между моментом начала выполнения процесса и запланированным сроком его завершения в систему могут поступать дополнительные задания.

Принцип FIFO («первый пришедший обслуживается первым») — это дисциплина планирования без переключения, при которой процессам предоставляется ЦП в соответствии со временем их поступления в список готовых к выполнению. Это распространенная дисциплина планирования заданий в системах пакетной обработки, однако, она не позволяет гарантировать приемлемые времена ответа для интерактивных пользователей.

Циклическое планирование (RR) — это по сути вариант FIFO с переключением ЦП. Диспетчирование процессов осуществляется здесь по принципу FIFO, однако ЦП предоставляется им только на ограниченное время, называемое квантом времени. Если квант времени некоторого процесса истечет до того, как этот процесс добровольно освободит ЦП, у него будет отобран ЦП, а затем выделен другому готовому для выполнения процессу. Циклическое планирование применяется обычно для того, чтобы гарантировать приемлемые времена ответа для интерактивных пользователей.

Определение оптимального размера кванта времени представляет собой сложную проблему. Размер кванта имеет первостепенное значение с точки зрения обеспечения рационального использования ресурсов системы и получения приемлемых времен ответа. Очень большой размер кванта приводит к тому, что любая дисциплина планирования с переключением начинает приближаться по своим характеристикам к аналогу без переключения. При очень малом размере кванта время ЦП непроизводительно расходуется на многочисленные контекстные переключения при переходе с процесса на процесс. Как правило, размер кванта стараются выбирать настолько большим, чтобы подавляющее большинство тривиальных запросов можно было полностью обслужить в рамках одного кванта. Например, в системе, лимитируемой вводом-выводом, размер кванта выбирается таким, чтобы большинство процессов успевало сформировать и выдать свой следующий запрос ввода-вывода в пределах одного кванта времени.

Планирование по принципу SJF («кратчайшее задание — первым») — это планирование без переключений, применяемое преимущественно для планирования пакетных заданий. Оно обеспечивает минимальное среднее время ожидания для заданий, однако для длинных заданий время ожидания может оказаться большим.

Дисциплина SRT («по наименьшему остающемуся времени») — это аналог принципа SJF с переключением. При планировании по принципу SRT выполнение текущего процесса может быть прервано при поступлении нового процесса с более коротким оценочным временем выполнения. Механизму SRT свойственны более высокие накладные расходы, чем SJF, однако он обеспечивает лучшее обслуживание поступающих коротких заданий. Он позволяет еще больше снизить средние времена ожиданий для всех заданий, однако длинные задания могут испытывать даже большие задержки, чем в случае SJF.

Предложенный Бринком Хансеном принцип HRN («по наибольшему относительному времени ответа») — это планирование без переключения, при котором в определенной степени корректируются некоторые недостатки SJF, в частности исключается чрезмерное игнорирование длинных заданий и чрезмерная

благосклонность к коротким новым заданиям.

Одним из наиболее совершенных механизмов планирования, применяемых в настоящее время, является сеть многоуровневых очередей с обратными связями. Это схема планирования с переключением процессов, которая особенно эффективна для систем, где выполняются смеси разнородных заданий. Новые процессы поступают в сеть очередей с очень высоким начальным приоритетом и быстро обслуживаются, если они являются либо интерактивными, либо лимитируемыми вводом-выводом. Процессы, лимитируемые ЦП, полностью используют выделенный им квант времени, а затем переходят в конец очереди следующего, более низкого приоритетного уровня. Чем дольше данный процесс занимает ЦП, тем ниже становится его приоритет, пока процесс не опускается в очередь самого низкого приоритета, которая реализует принцип циклического обслуживания и в которой он циркулирует до тех пор, пока не завершится. Как правило, размер кванта времени, предоставляемого процессу, увеличивается по мере перехода процесса в каждую следующую очередь.

Сеть многоуровневых очередей с обратными связями — это пример адаптивного механизма планирования, который реагирует на изменение поведения контролируемой им системы.

Терминология

адаптивный механизм (adaptive mechanism)

время ответа (response time)

динамические приоритеты (dynamic priorities)

диспетчер (dispatcher)

диспетчирование (dispatching)

дисциплины планирования (scheduling disciplines)

интервальный таймер (interval timer)

квант (времени) (quantum)

«консервация» (откладывание) процесса (shelve a process)

критерии планирования (scheduling criteria)

определение величины кванта времени (quantum determination)

переключение (preemption)

планирование (scheduling)

планирование без переключений (nonpreemptive scheduling)

планирование допуска (задания в систему) (admission scheduling)

планирование на верхнем уровне (high-level scheduling)

планирование на нижнем уровне (диспетчирование) (low-level scheduling (dispatching))

планирование на промежуточном уровне (intermediate-level scheduling)

планирование по предельному сроку, по сроку завершения (deadline scheduling)

планирование с переключением (preemptive scheduling)

планировщик (scheduler)

предсказуемость, прогнозируемость (predictability)

событие, связанное со временем (time-dependent event)

старение (aging)

статические приоритеты (static priorities)

уровни планирования (scheduling levels)

цели планирования (scheduling objectives)

часы-будильник, прерывающие часы (interrupting clock)

FIFO first-in-first-out («первый пришедший обслуживается первым»)

HRN highest-response-ratio-next («по наибольшему относительному времени реакции»)

SJF shortest-job first («кратчайшее задание — первым»)

SRT shortest-remaining-time («по наименьшему остающемуся времени»)

Упражнения

10.1 Укажите различия между планировщиками следующих трех уровней;

- а) планировщик заданий;
- б) планировщик промежуточного уровня;
- в) диспетчер.

10.2 Планировщик какого уровня должен принимать решение по каждому из следующих вопросов:

- а) Какому из готовых к выполнению процессов следует предоставить ЦП, когда он освободится?
- б) Какое из ряда ожидающих пакетных заданий, ранее введенных в дисковую память,

должно быть инициировано следующим?

в) Какие процессы следует временно приостановить, чтобы снять короткий пик нагрузки на ЦП?

г) Какой из кратковременно приостановленных процессов (о котором известно, что он лимитируется вводом-выводом) следует активизировать, чтобы сбалансировать мультипрограммную смесь?

10.3 Укажите различия между принципами планирования и механизмами планирования.

10.4 Дисциплина планирования должна, как правило, реализовать следующие цели:

а) быть справедливой;

б) обеспечивать максимальную пропускную способность системы;

в) обеспечивать приемлемые времена ответа для максимального числа интерактивных пользователей;

г) обеспечивать предсказуемость;

д) обеспечивать минимальные накладные расходы;

е) обеспечивать сбалансированное использование ресурсов;

ж) обеспечивать сбалансированность между временем ответа и коэффициентом использования ресурсов;

з) исключать бесконечное откладывание; и) учитывать приоритеты;

к) оказывать предпочтение процессам, занимающим ключевые ресурсы; л) устанавливать низкий приоритет обслуживания процессов с высокими накладными расходами;

м) постепенно снижать уровень работоспособности при увеличении нагрузок. Какая из указанных целей имеет наиболее непосредственное отношение к каждому из следующих случаев?

(I) Если некий пользователь уже давно ждет, ему следует оказать предпочтение.

(II) Пользователь, который выполняет задание по составлению платежной ведомости для компании, имеющей 1000 служащих, рассчитывает, что на выполнение этого задания будет затрачиваться приблизительно одинаковое количество времени каждую неделю.

(III) В систему следует в первую очередь допускать такие задания, чтобы создавалась смесь, обеспечивающая занятость для большинства устройств.

(IV) Система должна оказывать предпочтение наиболее важным заданиям. (V) Важный процесс, поступивший в систему, не может выполняться,

поскольку необходимые ему ресурсы в данный момент занимает процесс меньшей важности.

(VI) В течение пиковых периодов система не должна терять работоспособность из-за чрезмерных накладных расходов, необходимых для манипуляций большим количеством процессов.

10.5 Ниже перечисляются обычные критерии планирования:

- а) лимитируется ли процесс ввода-вывода;
 - б) лимитируется ли процесс ЦП;
 - в) является ли данный процесс пакетным или интерактивным;
 - г) обязателен ли быстрый ответ;
 - д) приоритет процесса;
 - е) частота прерываний по отсутствию нужных страниц;
 - ж) частота переключений процесса из-за появления процессов более высокого приоритета;
 - з) приоритеты процессов, ожидающих освобождения ресурсов, которые заняты другими процессами;
 - и) суммарное время ожидания;
 - к) суммарное время выполнения;
 - л) оценочное время ЦП, необходимое для завершения процесса. Укажите, какой из перечисленных выше критериев планирования имеет наиболее непосредственное отношение к каждому из следующих случаев:
 - (I) В системе реального времени, управляющей космическим полетом, компьютер должен немедленно реагировать на сигналы, принимаемые с космического корабля.
 - (II) Время от времени ЦП предоставляется в распоряжение процесса, однако в его выполнении наблюдается лишь номинальный прогресс.
 - (III) Как часто данный процесс добровольно освобождает ЦП для выполнения операции ввода-вывод до того, как выделенный ему квант времени истечет?
 - (IV) Присутствует ли пользователь на машине и ждет ли он ответа при работе в диалоговом режиме или пользователь отсутствует?
 - (V) Одна из целей планирования — обеспечить минимизацию средних времен ожиданий.
- 10.6** Известно, что при выполнении некоторого процесса формируется большое число прерываний по отсутствию нужных страниц. Приведите аргументы за и против того, чтобы этому процессу присвоить высокий приоритет с точки зрения предоставления ЦП.
- 10.7** Укажите, какие из приведенных ниже утверждений истинны, а какие ложны. Объясните, почему вы так считаете.
- а) Планирование является планированием с переключением, если у процесса нельзя принудительно отобрать выделенный ему ЦП.
 - б) В системах реального времени применяется, как правило, планирование с переключением ЦП.
 - в) В системах разделения времени применяется, как правило, планирование без переключения ЦП.
 - г) Времена ответа являются более предсказуемыми в системах с переключением, чем без переключения.
 - д) Один из недостатков схем с переключением заключается в том, что система будет точно

соблюдать приоритеты, однако сами приоритеты могут устанавливаться далеко не разумно.

10.8 Почему нельзя разрешать пользователям устанавливать самим часы-будильник?

10.9 Придайте приведенным ниже предложениям законченный вид, используя выражения «статические приоритеты» и «динамические приоритеты».

а) _____ легче реализовать, чем _____ .

б) _____ требуют меньших затрат машинного времени, чем _____ .

в) _____ более чувствительны к изменениям в условиях выполнения процесса, чем _____ .

г) _____ требуют более тщательного обдумывания при выборе начальных значений приоритетов, чем _____ .

10.10 Покупка приоритетов не обязательно означает, что пользователю придется платить дороже, чем при обычном обслуживании. И действительно, некоторые системы предоставляют пользователям возможность покупать очень низкий приоритет (т. е. «экономичную очередь») для заданий, которые не требуется выполнять очень быстро. При каких обстоятельствах может пользователь, которому необходимо быстрое обслуживание, фактически запросить низкий уровень приоритета по более дешевому тарифу?

10.11 Укажите несколько причин, обуславливающих исключительную сложность организации планирования по предельному сроку.

10.12 Приведите пример, показывающий, почему принцип FIFO является неподходящей дисциплиной планирования ЦП для интерактивных пользователей.

10.13 На примере из предыдущего пункта покажите, почему циклическое планирование является более приемлемым для обслуживания интерактивных пользователей.

10.14 Определение размера кванта времени — сложная и важная задача. Предположим, что среднее время контекстного переключения при переходе с процесса

на процесс есть s , а среднее количество времени, которое использует процесс» лимитируемый вводом-выводом, прежде чем сформирует запрос ввода-вывода, есть $t(t > s)$. Обсудите, к каким последствиям приведет выбор каждого из следующих значений кванта времени:

а) $q = \text{бесконечность}$,

б) q несколько больше нуля,

в) $q = s$,

г) $s < q < t$,

д) $q = t$,

е) $q > t$.

10.15 Обсудите последствия, к которым приведет реализация каждого из следующих методов назначения q :

1. q выбирается фиксированным и идентичным для всех пользователей; 2. q выбирается фиксированным и уникальным для каждого пользователя

3. q выбирается переменным и идентичным для всех пользователей;

4. q выбирается переменным и уникальным для каждого пользователя.

а) Расположите перечисленные выше принципы в порядке возрастания накладных расходов времени ЦП от минимума к максимуму.

б) Расположите перечисленные выше принципы в порядке возрастания чувствительности к изменениям характеристик индивидуальных процессов и нагрузки системы.

в) Установите связь между своими ответами для заданий (а) и (б).

10.16 Укажите, почему неправильно каждое из следующих утверждений:

а) Планирование по принципу SRT всегда характеризуется меньшим средним временем ответа, чем SJF.

б) Принцип SJF справедлив.

в) Чем короче задание, тем лучшие условия обслуживания должны ему предоставляться.

г) Поскольку принцип SJF отдает предпочтение коротким заданиям, его целесообразно применять в системах разделения времени.

10.17 Укажите некоторые недостатки принципа SRT. Каким образом могли бы вы его модифицировать, чтобы обеспечить более высокие скоростные характеристики?

10.18 Ответьте на каждый из следующих вопросов, относящихся к предложенной Бринком Хансеном стратегии HRN:

а) Каким образом стратегия HRN предотвращает бесконечное откладывание?

б) Каким образом стратегия HRN уменьшает предпочтение, оказываемое другими стратегиями коротким новым заданиям?

в) Предположим, что два задания находятся в режиме ожидания приблизительно одинаковое время. Являются ли их приоритеты приблизительно одинаковыми? Объясните свой ответ.

10.19 Покажите, каким образом многоуровневые очереди с обратными связями обеспечивают достижение каждой из следующих целей планирования:

а) Оказывать предпочтение коротким заданиям;

б) Оказывать предпочтение заданиям, лимитируемым вводом-выводом, чтобы обеспечить хорошую загрузку устройств ввода-вывода;

в) Определять характер задания как можно быстрее и соответствующим образом планировать выполнение этого задания.

10.20 Одно из эвристических правил, часто используемых планировщиками ЦП, заключается в том, что прошлое поведение процесса может служить хорошим ориентиром для прогнозирования его будущего поведения. Приведите пример ситуации, когда планировщик ЦП, следующий этому эвристическому правилу, выберет плохое решение.

Глава 11

Мультипроцессорные системы

Иметь в жизни одного друга — это очень много, двух — это даже слишком, а трех — вряд ли возможно. Дружба требует определенного параллелизма жизни, общности мыслей и интересов, некоего соперничества в достижении целей.

Генри Брукс Адама

Тот, кто стремится к власти над другими, должен прежде всего научиться управлять собой.

Филип Массинджер

□

Никто не может служить двум господам.

Евангелие от Матфея, 6.24.

11.1 Введение

11.2 Надежность

11.3 Использование параллелизма

11.4 Максимальное распараллеливание

11.5 Цели мультипроцессорных систем

11.6 Автоматическое распараллеливание

11.6.1 Расщепление цикла

11.6.2 Редукция высоты дерева

11.7 Правило «никогда не ждать»

11.8 Организация мультипроцессорной аппаратуры

11.8.1 Общая шина

11.8.2 Матрица координатной коммутации

11.8.3 Многопортовая память

11.9 Системы со слабо и сильно связанными процессорами

11.10 Организация «главный — подчиненный»

11.11 Мультипроцессорные операционные системы

11.12 Организация мультипроцессорных операционных систем

11.12.1 Организация «главный — подчиненный»

11.12.2 Организация с отдельными мониторами

11.12.3 Симметричная организация

11.13 Производительность мультипроцессорных систем

11.14 Экономическая эффективность мультипроцессорных систем

11.15 Восстановление после ошибок

11.16 Симметричная мультипроцессорная система TOPS-10

11.17 Системы $S.mmp$ и Sm^*

11.18 Перспективы мультипроцессорных систем

11.1 Введение

Одна из важных тенденций развития современной вычислительной техники состоит в расширенном внедрении *мультипроцессорных систем (архитектур)*, т. е. построении вычислительных комплексов, содержащих по несколько процессоров (рис. 11.1). Мультипроцессорные архитектуры реализуются в вычислительных системах уже в течение нескольких десятилетий, однако сейчас интерес к ним снова обостряется в связи с появлением недорогих микропроцессоров.

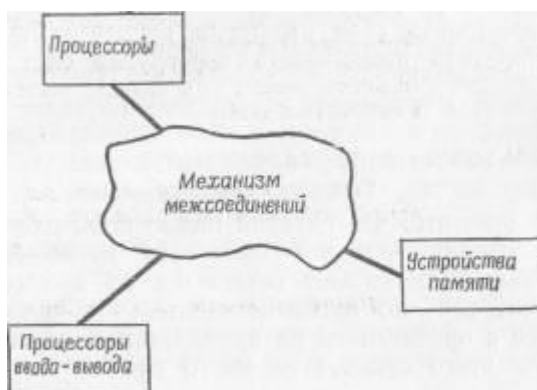


Рис. 11.1 Упрощенная схема мультипроцессорной организации.

Микропроцессоры позволили получать процессорное время по номинальной цене, а их малые габариты делают вполне обоснованной постановку вопроса о комплексировании многих подобных устройств в единой системе. В настоящее время вполне можно представить себе систему, построенную с использованием нескольких сотен микропроцессоров. А дальнейшее развитие техники обеспечит, по-видимому, возможность создания систем, содержащих тысячи и даже миллионы процессоров.

Представление о процессоре как о наиболее ценном ресурсе данной вычислительной системы, обычное для первых двух десятилетий развития современной вычислительной техники, сейчас уже практически устарело. В настоящее время с появлением мультипроцессорных архитектур гораздо большее значение приобретают проблемы надежности, параллелизма в вычислениях, оптимальных схем коммутации и состязаний между процессорами, пытающимися получить доступ к одним и тем же ресурсам.

11.2 Надежность

Одно из основных достоинств мультипроцессорных систем состоит в том, что в случае выхода из строя одного процессора оставшиеся процессоры могут продолжать работать. Однако само собой это не происходит. Для этого требуется тщательная проработка конструкции системы. Вышедший из строя процессор должен каким-то образом проинформировать другие процессоры о том, что они должны принять на себя его нагрузку. Работоспособные процессоры должны быть в состоянии обнаружить процессор, который вышел из строя. Операционная система должна заметить, что какой-то конкретный процессор отказал и его необходимо исключить из списка распределяемых ресурсов. Кроме того, поскольку общий объем ресурсов теперь уменьшился, операционная система должна несколько перестроить свои стратегии распределения ресурсов, чтобы предотвратить возможность перегрузки системы.

11.3 Использование параллелизма

Интересно отметить, что главной целью *большинства мультипроцессорных систем является повышение производительности*. Программирование продолжает оставаться по преимуществу последовательным, так что лишь незначительное количество программ пишется с ориентацией на параллельное, совмещенное выполнение. Для этого существует много причин.

- Люди обычно мыслят последовательными категориями. Человеческий разум по сути просто не приспособлен для представления параллельных категорий.
- Ни в одном естественном человеческом языке нет средств, удобных для выражения параллелизма. Существует несколько языков программирования для компьютеров — таких, как Ада, CSP/k, параллельный Паскаль и Модула. Однако, ни один из них пока еще не получил широкого распространения.
- Поскольку сами мультипроцессорные системы в общем-то практически еще не использовались в разработке параллелизма, имеется лишь очень небольшой опыт решения связанных с этим проблем.
- Аппаратные средства компьютеров ориентированы на последовательную работу.
- Отлаживать параллельные программы исключительно сложно.

- Гораздо более сложно доказывать корректность параллельных программ, чем последовательных.

Сейчас появились аппаратные средства для параллельного выполнения операций, однако пройдет еще много времени, прежде чем будут осмыслены и решены основные проблемы параллельного программирования и будет написано большое количество параллельных программ. Все эти трудности позволяют понять, почему современные мультимикропроцессорные комплексы не часто используют потенциальные преимущества параллелизма. Реальный прогресс в этой области начнется, по-видимому, после того, как появятся компиляторы и операционные системы, которые будут обнаруживать и осуществлять распараллеливание автоматически.

11.4 Максимальное распараллеливание

Одним из ключевых факторов, способствующих использованию потенциальных преимуществ параллелизма в будущих системах, станет, безусловно, наличие достаточного числа процессоров, с тем чтобы все операции, которые можно выполнять параллельно, удавалось распределить по отдельным процессорам. Вместо того чтобы говорить о мультимикропроцессорных системах в сегодняшнем смысле, т. е. о системах, содержащих два (или, быть может, немногим более) процессора, мы теперь обычно говорим о *максимальном распараллеливании*.

Максимальное распараллеливание предоставляет нам возможность привести данную программу к завершению в самое короткое время. Здесь интересен следующий чисто теоретический вопрос: «За какое минимальное время можно выполнить программу, реализующую конкретный алгоритм, при наличии максимального распараллеливания?»

11.5 Цели мультимикропроцессорных систем

Благодаря использованию большого количества центральных процессоров разработчики вычислительных систем могут реализовать множество целей. Среди них следует отметить достижение очень высокой надежности и коэффициента готовности, а также повышенной вычислительной мощности.

Многопроцессорные системы обладают повышенной надежностью, поскольку в случае выхода из строя одного процессора остальные могут продолжать функционировать. Система будет продолжать работать, хотя и при несколько уменьшенных функциональных возможностях. Повышение вычислительной мощности получается за счет объединения мощностей нескольких процессоров.

Снижение стоимости аппаратных средств привело к тому, что разработчики стали объединять большое количество микропроцессоров, образуя мультимикропроцессорные системы (Ch79, Si78a, Si78b, Jo77). Таким образом можно достичь высокого уровня вычислительной мощности системы без применения дорогостоящих сверхбыстродействующих процессоров.

Мультимикропроцессорная архитектура может явиться средством значительного увеличения мощности вычислительных установок без существенного увеличения стоимости. Иногда для повышения вычислительной мощности гораздо

целесообразнее перейти на мультипроцессорную систему, чем приобретать дополнительно функционально законченные машины. Установка многих машин может быть связана с серьезными проблемами производственных площадей, может потребовать дополнительного персонала для их обслуживания и т. д.

Важным качеством мультипроцессорных комплексов является их гибкость. Модульная архитектура мультипроцессорных комплексов позволяет легко увеличивать вычислительные мощности путем подключения при необходимости дополнительных процессоров.

11.6 Автоматическое распараллеливание

Мультипроцессорные комплексы позволяют воспользоваться преимуществами параллелизма. Большинство программ пишутся для последовательного выполнения. Вычислительные системы получают больше пользы от параллельной обработки благодаря мультипрограммному выполнению нескольких процессов, чем используя параллелизм в рамках одного процесса. Обнаружение параллелизма (распараллеливание), выполняемое программистами, языковыми трансляторами, аппаратными средствами или операционными системами — это сложная проблема, вызывающая сегодня особый интерес. Независимо от того, каким образом в конце концов будет обнаруживаться параллелизм, мультипроцессорные системы позволяют с успехом использовать его, одновременно выполняя параллельные ветви вычислений.

Параллелизм в программах может быть либо *явным*, либо *неявным* (скрытым). Явный параллелизм программист указывает в своей программе при помощи специальной конструкции, обозначающей параллельное вычисление, например COBEGIN/COEND, следующим образом:

COBEGIN;

оператор-1;

оператор-2;

...

оператор-n;

COEND;

В мультипроцессорной системе, рассчитанной на реализацию параллелизма, каждый из программных операторов может выполнять отдельный процессор, с тем, чтобы все вычисления завершались быстрее, чем при чисто последовательной работе.

Явное указание параллелизма налагает определенную ответственность на программиста. Это достаточно трудоемкая процедура, причем в ряде случаев программист может ошибочно указать, что определенные операции можно выполнять параллельно, в то время как в действительности этого делать нельзя. Программист может упустить многие ситуации, допускающие распараллеливание. Весьма вероятно, что программист обнаружит параллелизм и закодирует его явным образом в наиболее очевидных ситуациях. Однако многие случаи параллелизма вручную в алгоритмах обнаружить трудно, поэтому они будут просто пропускаться.

Одно из довольно неприятных последствий явного указания параллелизма заключается в том, что программы могут оказаться более сложными для модификации. При внесении изменений в программу, имеющую большое число явных параллельных конструкций, легко можно допустить ошибки, причем, вообще говоря, довольно тонкие.

Реально, на что можно надеяться при решении подобной проблемы,— это на автоматическое обнаружение неявного параллелизма, т. е. параллелизма, присущего алгоритму, но не указанного явно программистом. В компиляторы, операционные системы и аппаратные средства компьютеров необходимо включать специальные механизмы распараллеливания. Это с гораздо большей вероятностью, чем явное указание параллелизма, обеспечит создание быстро выполняющихся и корректных программ.

Два распространенных способа, реализуемых в компиляторах для использования неявного параллелизма программ,— это *расщепление цикла* и *редукция высоты дерева*.

11.6.1 Расщепление цикла

Программный цикл предполагает многократное выполнение некоторой последовательности операторов, называемой телом цикла, до тех пор, пока не станет истинным соответствующее условие завершения. В определенных случаях в тело цикла входят операторы, которые могут допускать параллельное выполнение. Рассмотрим следующий цикл:

FORI = 1 TO 4 DO

A(I)=B(I)+C(I);

Этот цикл суммирует соответствующие элементы массивов В и С и помещает суммы этих элементов в массив А. По данному программному циклу последовательный процессор выполняет поочередно следующие операции:

A(1)=B(1)+C(1);

A(2)=B(2)+C(2);

A(3)=B(3)+C(3);

A(4)=B(4)+C(4);

В мультипроцессорной системе эти операторы можно выполнять параллельно при помощи четырех процессоров, благодаря чему существенно уменьшится время выполнения всего цикла. Компилятор, осуществляющий автоматическое распараллеливание, может преобразовать приведенный выше цикл в следующую форму:

COBEGIN;

A(1)=B(1)+C(1);

A(2)=B(2)+C(2);

$A(3)=B(3)+C(3);$

$A(4)=B(4)+C(4);$

COEND;

Тем самым он указывает вычислительной системе на возможность параллельного выполнения операторов. Этот способ называется расщеплением, или разбиением программного цикла на параллельные цепочки, и его относительно легко реализовать. Естественно, что во многих программных циклах число повторений гораздо больше четырех, как в данном примере, поэтому метод разбиения цикла обычно дает более высокую степень параллелизма, чем количество имеющихся процессоров. Операционная система и аппаратные средства компьютера должны затем решить, какое подмножество параллельных ветвей будет выполняться одновременно.

11.6.2 Редукция высоты дерева

Применяя ассоциативные, коммутативные и дистрибутивные свойства арифметики, компиляторы могут обнаруживать неявный параллелизм в алгебраических выражениях и генерировать объектный код для мультипроцессорных систем с указанием операций, которые можно выполнять одновременно. Компиляторы должны действовать в соответствии со старшинством операций. Не все компиляторы используют одинаковые правила старшинства, однако во многих случаях применяются примерно следующие правила:

1. Прежде всего выполняются операции, указанные во вложенных скобках, и, в частности, те операции, которые имеют более глубокий уровень вложенности.
2. Затем выполняются операции в скобках.
3. Затем выполняются операции возведения в степень.
4. Затем выполняются операции умножения и деления.
5. Затем выполняются операции сложения и вычитания.
6. Если операции, которые можно выполнить следующими, имеют одинаковое старшинство, то они выполняются слева направо.

Применение таких правил старшинства операций дает компилятору возможность устанавливать однозначный последовательный порядок действий для алгебраического выражения, а затем переходить к генерации машинного кода, реализующего вычисление этого выражения.

Зачастую однозначность и последовательность порядка действий при вычислениях не обязательны. Например, поскольку операции сложения и умножения являются коммутативными, компилятор, генерирующий код для перемножения p и q , может выдать либо $p*q$, либо $q*p$; результаты вычислений будут, естественно, одинаковыми. Аналогично, сложение p и q можно выполнить либо как $p+q$, либо как $q+p$. Используя свойство коммутативности, а также ассоциативности и дистрибутивности, компилятор может довольно гибко перестраивать выражения, делая их более удобными для параллельных вычислений.

На рис. 11.2 показано, как обычный компилятор оттранслировал бы алгебраическое выражение и как оно могло бы быть обработано распараллеливающим компилятором. Левая диаграмма на этом рисунке —

структура дерева, показывающего, каким образом обычный компилятор может генерировать код программы для выполнения приведенных операций. Цифры 1, 2 и 3 показывают порядок,

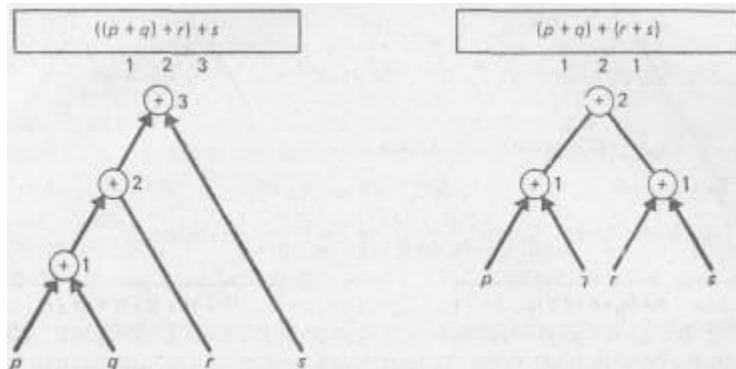


Рис. 11.2 Уменьшение высоты дерева за счет ассоциативности.

в котором должны выполняться эти операции. Используя ассоциативность сложения, можно преобразовать выражение

$$((p+q)+r)+s$$

в выражение вида

$$(p+q)+(r+s),$$

которое более приспособлено для параллельных вычислений. Структуре первого выражения соответствует трехуровневое дерево, в то время как структуре второго — лишь двухуровневое, и поэтому его можно выполнить гораздо быстрее на мультипроцессорной системе.

Цель многих систем новых поколений будет заключаться в том, чтобы производить вычисления в минимально возможное время, независимо от того, сколько процессоров придется использовать для выполнения этой работы. Поэтому новые вычислительные системы будут содержать специальные механизмы, которые будут преобразовывать последовательные алгоритмы в параллельные, разлагая их на шаги, которые можно было бы выполнять одновременно на многих процессорах системы.

На рис. 11.3 показано, каким образом благодаря коммутативности сложения можно преобразовать последовательную цепь операций, изображаемую трехуровневым деревом, в параллельный вариант, требующий всего лишь двух уровней. На рис. 11.4 показано, каким образом дистрибутивность умножения и сложения может быть использована для уменьшения числа уровней дерева с четырех до трех.

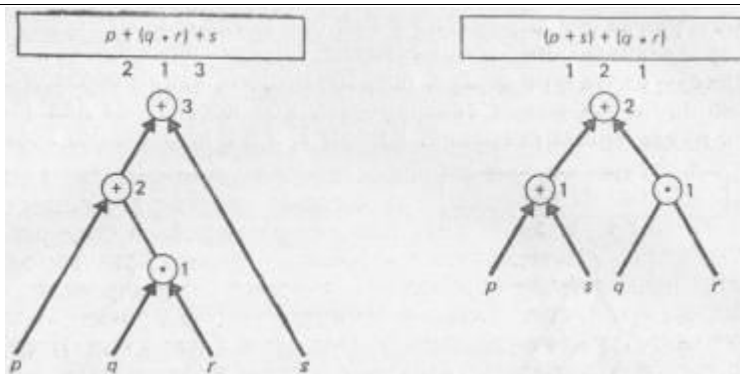


Рис. 11.3 Уменьшение высоты дерева за счет коммутативности.

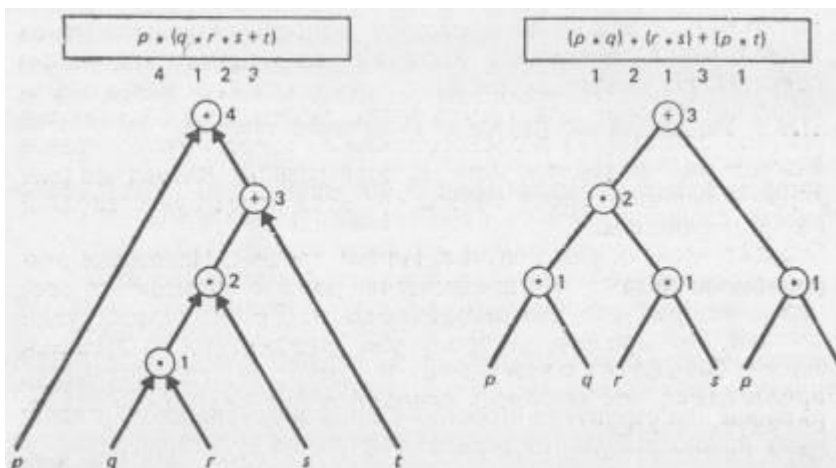


Рис. 11.4 Уменьшение высоты дерева за счет дистрибутивности.

Применение указанных методов компиляции с целью оптимизации программ для выполнения на мультипроцессорных системах не обходится бесплатно. Дело в том, что за уменьшение количества времени выполнения, затрачиваемого на данное вычисление, приходится платить увеличенными затратами времени и ресурсов в период компиляции. Такую взаимосвязь необходимо учитывать, тщательно оценивая необходимые затраты и возможные выгоды в каждом индивидуальном случае. Например, для производственного счета целесообразно добиваться минимального времени выполнения программ. Однако в условиях разработки, когда программа, возможно, будет выполняться всего один или два раза до внесения очередных изменений и повторной компиляции, затраты на оптимизацию могут значительно превысить получаемые выгоды.

11.7 Правило «никогда не ждать»

В некоторых случаях целесообразно, чтобы процессоры выполняли вычисления, результаты которых могут так и не потребоваться, если в действительности существует вероятность того, что эти результаты все же будут использованы и ускорят вычисления.

Правило «никогда не ждать» говорит о том, что лучше поручить некоторому процессору работу, которая то ли будет, то ли не будет использована в

дальнейшем, чем оставить этот процессор бездействующим. Идея состоит в том, что если все же результаты понадобятся, то вычисления можно будет выполнить гораздо быстрее. Рассмотрим следующий пример:

$A = B * C;$

IF $A = 9$ THEN $D = 10;$

$E = D * F$

В зависимости от результата, получаемого при выполнении первого оператора, второй оператор может либо изменить значение D , либо оставить его прежним. Если значение D не меняется, то третий оператор можно было бы выполнять параллельно с первым. Если D изменяется, то при выполнении третьего оператора необходимо использовать новое значение D . Стратегия «никогда не ждать» заключается в том, чтобы всегда выполнять третий оператор параллельно с первым. Если значение D изменится, то третий оператор придется выполнить заново. Если, однако, значение D не изменится, то результат выполнения третьего оператора будет уже готов, и все вычисление можно будет завершить быстрее.

11.8 Организация мультипроцессорной аппаратуры

Одной из ключевых проблем при проектировании мультипроцессорных систем является выбор способов подключения многих центральных процессоров и процессоров ввода-вывода к устройствам памяти. Для этого разработано несколько схем.

В следующих разделах мы будем рассматривать только действительно мультипроцессорные комплексы, которые можно охарактеризовать следующим образом (Еп77):

- Мультипроцессорный вычислительный комплекс содержит два или более сравнимых по производительности процессоров.
- Все процессоры имеют коллективный доступ к общей памяти.
- Все процессоры имеют коллективный доступ к каналам, контроллерам и устройствам ввода-вывода.
- Весь комплекс работает под управлением одной операционной системы, обеспечивающей взаимодействие между процессорами и их программами на уровнях заданий, задач, шагов, наборов данных и элементов данных.

Имеются три наиболее распространенных вида организации мультипроцессорных комплексов:

- с общей шиной,
- с матрицей координатной коммутации,
- с многопортовой памятью.

11.8.1 Общая шина

Организация с общей шиной, показанная на рис. 11.5, использует единый канал (тракт) связи между всеми функциональными устройствами комплекса, а

именно процессорами, устройствами памяти и процессорами ввода-вывода. В приведенной простой схеме общая шина, по сути играет роль пассивного устройства, т. е. операции обмена информацией между функциональными устройствами осуществляются под управлением шинных интерфейсов самих устройств.

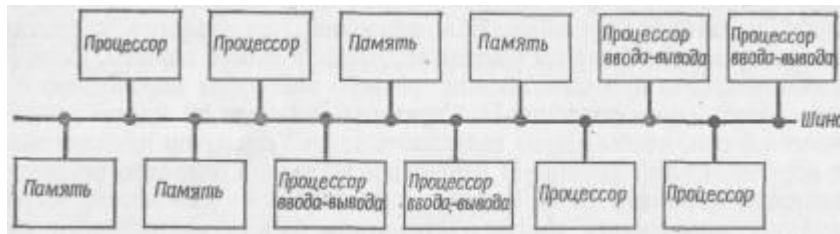


Рис. 11.5 Мультипроцессорная организация с общей шиной.

Центральный процессор или процессор ввода-вывода, желающий произвести передачу данных, должен вначале проверить готовность шины и готовность устройства-потребителя, проинформировать устройство-потребителя о том, что ему нужно будет сделать с получаемыми данными, а затем инициировать фактическую передачу данных. Принимающее устройство должно быть в состоянии определить, что сообщения, передаваемые по шине, адресованы именно ему, причем оно должно действовать в соответствии с сигналами управления, принимаемыми от передающего устройства, и подтверждать получение этих сигналов.

Архитектура с общей шиной позволяет легко вводить новые устройства, подключая их непосредственно к шине. Чтобы устройства комплекса могли обмениваться информацией, каждое из них должно знать, какие еще устройства подключены к шине, однако это делается, как правило, при помощи программных средств.

Главные недостатки систем с единственным каналом связи между устройствами сводятся к следующему:

- Весь комплекс выходит из строя, если возникает неисправность шины (т. е. неисправность шины — это *катастрофический отказ*).
- Общая скорость передачи данных в системе ограничивается общей пропускной способностью шины.
- Состязание за захват шины в загруженной системе может вызвать серьезную деградацию производительности.

Организация с общей шиной экономична, проста и гибка, однако перечисленные ограничения приводят к тому, что она применяется преимущественно при построении лишь небольших мультипроцессорных комплексов.

11.8.2 Матрица координатной коммутации

Увеличение числа шин системы с общей шиной до числа устройств памяти создает мультипроцессорную организацию с так называемой *матрицей*

координатной коммутации, в которой имеется самостоятельный тракт для обмена с каждым устройством памяти (рис. 11.6). В этой схеме обращения к двум различным устройствам памяти могут происходить одновременно — они не мешают друг

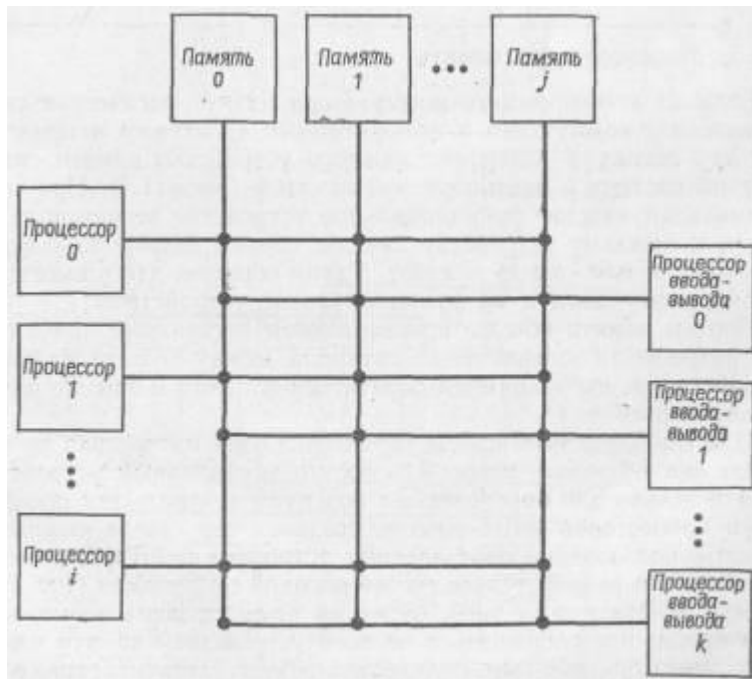


Рис. 11.6 Мультипроцессорная организация с матрицей координатной, или перекрестной коммутации.

другу. И действительно, в системе с матрицей координатной коммутации все устройства памяти могут работать одновременно. Координатные коммутаторы впервые нашли применение в автоматических телефонных станциях много десятилетий назад.

Аппаратные средства, необходимые для построения координатного коммутатора, могут оказаться довольно сложными. Например, такой коммутатор должен уметь разрешать конфликтные ситуации при обращениях к одному и тому же устройству памяти. Однако сложность коммутатора в действительности дает возможность значительно упростить интерфейсы функциональных устройств.

Наличие многих каналов связи между устройствами позволяет достичь очень высоких скоростей передачи. Добавление функциональных устройств в систему с координатной коммутацией приводит, как правило, к более заметному повышению производительности, чем при других схемах построения комплексов.

Координатный коммутатор для повышения надежности может содержать избыточную логику для контроля. Наличие многих трактов связи упрощает разбиение системы на части. Благодаря этому неработоспособное устройство можно легко исключать из системы, в то время как она будет продолжать функционировать.

11.8.3 Многопортовая память

Если из координатного коммутатора изъять логические схемы управления, коммутации и приоритетного арбитража и поместить всю эту логику в интерфейс каждого устройства памяти, то мы получим систему с многопортовой памятью (рис. 11.7). При такой организации каждое функциональное устройство может получить доступ к каждому устройству памяти, однако только по *конкретному порту*, или *каналу памяти*. Таким образом, здесь имеется по одному порту памяти на функциональное устройство.

Портам памяти обычно присваиваются постоянные приоритеты для разрешения конфликтных ситуаций между функциональными устройствами, пытающимися обратиться к одной и той же памяти одновременно.

Для многопортовой схемы характерна одна интересная возможность: она позволяет разрешать доступ к различным устройствам памяти только для определенных подгрупп центральных процессоров и процессоров ввода-вывода, создавая тем самым «закрытый» режим использования определенных устройств памяти. Это особенно важно для разработчиков систем высокой секретности (рис. 11.8). Однако независимо от того, будет ли предоставлена возможность всем процессорам обращаться ко всем устройствам памяти или доступ будет ограниченным, количество кабеля, затрачиваемого в многопортовых системах на проведение всех необходимых соединений, зачастую оказывается гораздо большим, чем при других схемах.

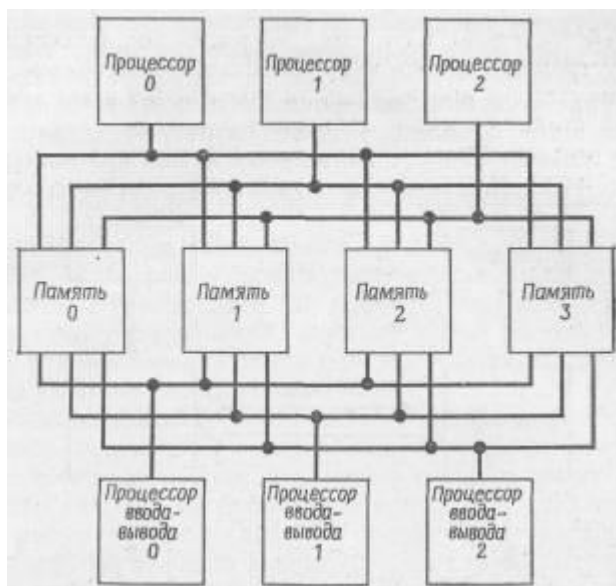


Рис. 11.7 Мультипроцессорная организация с многопортовой памятью.

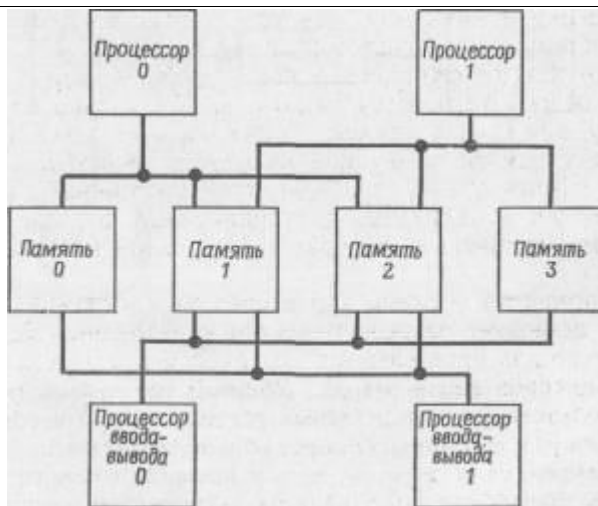


Рис. 11.8 Мультипроцессорная организация с многопортовой памятью и с выделением «частных» устройств памяти.

11.9 Системы со слабо и сильно связанными процессорами

Мультипроцессорные системы со *слабо связанными процессорами* (многомашинные комплексы) предусматривают соединение двух или более независимых вычислительных машин при помощи канала связи (рис. 11.9). При этом каждая машина имеет свою собственную



Рис. 11.9 Мультипроцессорная система со слабо связанными процессорами (многомашинный вычислительный комплекс).

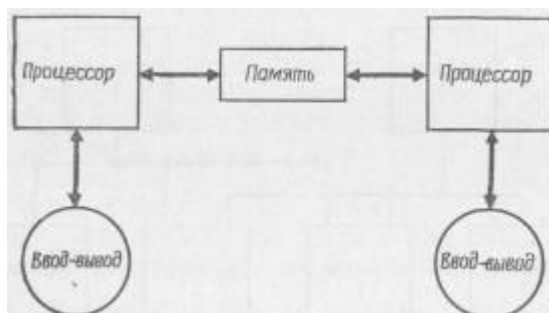


Рис. 11.10 Мультипроцессорная система с сильно связанными процессорами.

операционную систему и память. Машины могут функционировать независимо друг от друга и взаимодействовать при необходимости. Эти самостоятельные машины могут обращаться к файлам друг друга при помощи канала связи, а в некоторых случаях они могут передавать задачи для выполнения на менее загруженных процессорах, что позволяет в определенной степени балансировать нагрузку.

Мультипроцессорные системы с *сильно связанными процессорами* (многопроцессорные вычислительные комплексы) (рис. 11.10) имеют единую память, коллективно используемую различными процессорами, и единую операционную систему, которая управляет работой всех процессоров и других аппаратных средств системы.

11.10 Организация «главный — подчиненный»

В мультипроцессорной системе с организацией *«главный — подчиненный»* один из процессоров выполняет функции главного, а другие являются подчиненными. В качестве главного используется процессор общего назначения, который может выполнять как вычисления, так и операции ввода-вывода. Подчиненные процессоры выполняют только вычисления.

Подчиненные процессоры могут эффективно выполнять вычисления, задания же, в которых преобладают операции ввода-вывода, вызывают частое обращение к услугам, которые может предоставлять только главный процессор, что приводит к снижению эффективности работы системы. Оценивая подобную организацию с точки зрения надежности, можно отметить, что в случае выхода из строя одного из подчиненных процессоров система потеряет часть своих вычислительных возможностей, но все же будет продолжать функционировать. Если же выйдет из строя главный процессор, система не сможет выполнять операции ввода-вывода. В некоторых системах эта проблема решается путем переключения устройств ввода-вывода на один из подчиненных процессоров, с тем чтобы при рестарте системы этот процессор выполнял функции главного.

За последние годы скорости работы процессоров удалось повысить гораздо более существенно, чем скорости выполнения операций ввода-вывода. Это привело к тому, что задания, которые ранее считались преимущественно вычислительными, теперь стали требовать относительно больших затрат времени на ввод-вывод, поскольку вычисления, выполняемые между последовательными операциями ввода-вывода, начали производиться намного быстрее. А это в свою очередь привело к тому, что задания, которые ранее могли эффективно выполняться на одной модели мультипроцессорной системы с организацией *«главный — подчиненный»*, стало теперь менее целесообразно выполнять на новых моделях систем подобной организации с усовершенствованной аппаратурой. Таким образом, основной недостаток мультипроцессорных систем с организацией *«главный — подчиненный»* заключается в асимметрии аппаратных средств. В такой системе процессоры не эквивалентны, поскольку только главный процессор может выполнять как вычисления, так и операции ввода-вывода. Поэтому более удачной можно считать симметричную мультипроцессорную систему, в которой все процессоры функционально эквивалентны и могут выполнять и вычисления, и операции ввода-вывода.

11.11 Мультипроцессорные операционные системы

Различие между операционными системами мультипрограммных и мультипроцессорных комплексов не так велико, как могло бы показаться на первый взгляд. Оба типа систем предусматривают следующие функциональные возможности (Еп77):

- распределение и управление ресурсами;
- защиту наборов данных и системных таблиц;
- предотвращение системных тупиков (дедлоков);
- аварийное завершение заданий;
- балансирование нагрузки по вводу-выводу;
- балансирование процессорной нагрузки;
- реконфигурацию.

Механизмы для реализации трех последних из перечисленных функций характеризуются значительными отличиями для мультипроцессорных систем. Параллелизм в аппаратуре и в программах имеет первостепенное значение для мультипроцессорных комплексов, так что автоматическое распараллеливание является ключевым фактором мультипроцессорных операционных систем.

Увеличение количества процессоров, а также усложнение связей с памятью и процессорами ввода-вывода значительно повышают стоимость аппаратуры комплекса. Поэтому операционная система должна эффективно управлять дополнительными аппаратными средствами, с тем чтобы получаемые выгоды превосходили увеличенные исходные затраты. Безусловно, нельзя также игнорировать дополнительные затраты на программное обеспечение — построение мультипроцессорного вычислительного комплекса требует не только дополнительной аппаратуры, но и более сложной операционной системы.

11.12 Организация мультипроцессорных операционных систем

Одно из основных различий между операционными системами мультипроцессорных и однопроцессорных вычислительных комплексов состоит в том, каким образом организуется и строится операционная система с учетом взаимодействия со многими процессорами. Существуют три основных варианта организации операционных систем для мультипроцессорных комплексов:

- «главный—подчиненный»;
- свой монитор в каждом процессоре;
- симметричная организация (процессоры идентичны).

11.12.1 Организация «главный — подчиненный»

Организацию «главный — подчиненный» реализовать легче всего, причем часто ее можно создать просто путем расширения существующей мультипрограммной системы. Однако, как мы покажем ниже, такая организация не обеспечивает оптимального использования аппаратуры комплекса.

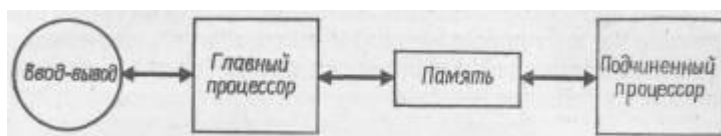


Рис. 11.11 Мультипроцессорная система с организацией «главный — подчиненный».

При организации «главный — подчиненный» (рис. 11.11) операционная система выполняется только на одном конкретном процессоре, главном процессоре. На подчиненном процессоре могут выполняться только программы пользователей. Когда процесс на подчиненном процессоре требует внимания операционной системы, он генерирует сигнал прерывания и ждет, чтобы главный процессор обработал это прерывание. Если подчиненных процессоров много и они активно генерируют сигналы прерывания, то у главного процессора могут создаваться большие очереди. Интересно отметить, что здесь операционная система не обязательно должна быть реентерабельной, поскольку она работает только на одном процессоре и только для одного пользователя в каждый конкретный момент времени.

Решение проблемы взаимного исключения существенно упрощается при обращении к системным таблицам, поскольку операционная система работает только на одном процессоре. Организация «главный — подчиненный» характеризуется меньшей надежностью по сравнению с другими видами организации, поскольку выход главного процессора из строя вызывает катастрофический отказ всей системы.

Если главный процессор не будет достаточно эффективно обслуживать запросы подчиненного, то подчиненный процессор не сможет работать эффективно. Вариант организации «главный — подчиненный» можно считать вполне приемлемым для работы в условиях с четко определенными нагрузками, поскольку здесь имеется возможность добиться оптимального планирования загрузки главного процессора. Этот вариант вполне пригоден также для асимметричных систем, в которых подчиненные процессоры обладают гораздо меньшей вычислительной мощностью, чем главный.

Организация «главный — подчиненный» предполагает, что операционная система всегда работает только на одном из процессоров. Этот процессор может иметь специальную конструкцию, обеспечивающую эффективную работу операционной системы, либо он может быть таким же, как другие процессоры.

Подчиненный процессор, который освобождается в момент, когда главный процессор занят, должен будет ждать, пока главный процессор не предоставит ему дополнительную работу. Если подчиненные процессоры выполняют большое количество коротких задач, это может привести к чрезмерной дополнительной нагрузке для главного процессора. Если главный процессор не сможет быстро реагировать на поступающие запросы, то значительная часть вычислительных мощностей подчиненных процессоров будет оставаться неиспользуемой.

11.12.2 Организация с раздельными мониторами

При организации с *раздельными мониторами (исполнительными программами)* каждый процессор содержит собственную операционную систему, которая соответствующим образом реагирует на прерывания от программ пользователей, работающих на этом процессоре. Поскольку некоторые таблицы содержат

глобальную информацию для всей системы (например, список процессоров, известных системе), доступ к этим таблицам должен осуществляться под строгим контролем с применением методов взаимоисключения.

Организация с отдельными мониторами является в принципе более надежной, чем организация «главный — подчиненный». Отказ какого-то одного процессора здесь вряд ли станет катастрофическим отказом системы, однако рестарт системы с отказавшим процессором может оказаться достаточно сложным.

Каждый процессор управляет своими собственными ресурсами, например файлами и устройствами ввода-вывода. Реконфигурация оборудования ввода-вывода системы может потребовать подключения устройств ввода-вывода к другим процессорам с другими операционными системами. Такая процедура может быть достаточно сложной и потребовать значительных ручных усилий.

При организации с отдельными мониторами каждый процессор имеет свою собственную операционную систему и работает подобно однопроцессорной машине со своими собственными ресурсами. Процесс, запланированный для выполнения на каком-то конкретном процессоре, выполняется на нем до завершения.

Функции операционных систем реализуются на каждом процессоре, обслуживающем свои собственные запросы и запросы идущих на нем процессов. Здесь наблюдается лишь минимальное количество конфликтных ситуаций при работе с таблицами операционных систем, поскольку эти таблицы распределены между индивидуальными операционными системами и используются в основном только ими.

Каждый процессор имеет свой собственный «частный» набор устройств ввода-вывода³ которыми он управляет независимо от других процессоров. Операции ввода-вывода по запросам процесса выполняет исключительно тот процессор, который выделен данному процессу. Возврат из прерываний ввода-вывода осуществляется непосредственно на те процессоры, которые их инициируют. Поскольку устройства ввода-вывода подключаются прямо к индивидуальным процессорам, реконфигурация системы требует ручного вмешательства.

При организации с отдельными мониторами не предусматривается никакого взаимодействия процессоров при выполнении индивидуального процесса. Не исключается возможность, что некоторые из процессоров будут оставаться свободными, в то время как один процессор выполняет длинный процесс.

11.12.3 Симметричная организация

Симметричная организация мультипроцессорного вычислительного комплекса является наиболее сложной для реализации и в то же время наиболее эффективной. Здесь все процессоры идентичны. Операционная система управляет пулом идентичных процессоров, каждый из которых может управлять работой любого устройства ввода-вывода или обращаться к любому устройству памяти.

Поскольку программы операционной системы могут выполняться на многих процессорах одновременно, реентерабельный код и взаимоисключение являются обязательными. Благодаря симметричности системы имеется возможность более точно сбалансировать рабочую нагрузку, чем при других видах организации.

При симметричной организации особенно важное значение приобретают аппаратные и программные средства для разрешения конфликтных ситуаций.

Конфликты между процессорами, пытающимися получить доступ к одной и той же памяти в одно и то же время, разрешаются; как правило, аппаратными средствами. Конфликты при доступе к системным таблицам разрешаются обычно программными средствами.

Симметричные мультипроцессорные комплексы являются в общем случае наиболее надежными — отказ одного процессора приводит к тому, что операционная система исключает этот процессор из пула имеющихся процессоров и уведомляет об этом оператора. Комплекс может продолжать работать с несколько пониженным уровнем функциональных и скоростных возможностей (плавная деградация), пока вышедший из строя процессор не будет отремонтирован.

В симметричной системе процесс может в разные периоды времени выполняться на любом из эквивалентных процессоров. Все процессоры могут кооперироваться при выполнении конкретного процесса.

Операционная система *перемещается* от одного процессора комплекса к другому. Процессор, ответственный в данный момент за ведение системных таблиц и реализацию системных функций, называется мониторным процессором. В каждый конкретный момент времени мониторным может быть только один процессор — благодаря этому предотвращаются конфликты при работе с глобальной системной информацией.

Симметричная организация позволяет лучше использовать ресурсы. Легче сбалансировать нагрузку системы в целом, поскольку большинство видов работ можно передавать любому имеющемуся свободному процессору.

Проблема состязаний может оказаться очень острой, особенно в связи с тем, что в режиме супервизора здесь могут работать несколько процессоров одновременно. Во избежание чрезмерного количества конфликтов необходимо тщательно продумывать конструкцию системных таблиц. Одним из способов, помогающих свести к минимуму число конфликтов, является разбиение системных данных на ряд самостоятельных и независимых объектов, которые можно было бы закрывать индивидуально.

11.13 Производительность мультипроцессорных систем

Даже в совершенно симметричных мультипроцессорных комплексах введение нового процессора не приведет к тому, что общая производительность возрастет на номинальную вычислительную мощность этого нового процессора. Это объясняется многими причинами, в том числе

- дополнительными накладными расходами на работу операционной системы;
- обострением конкуренции за системные ресурсы;
- задержками в аппаратуре коммутации и маршрутизации между увеличившимся количеством компонент.

Одна из исследовательских работ, проведенных с целью оценки производительности мультипроцессорных комплексов, показала, что производительность двухпроцессорного комплекса увеличивается в 1.8 раза, а трехпроцессорного — всего лишь в 2.1 раза по сравнению с однопроцессорным. Подобные результаты помогают понять, почему большинство мультипроцессорных

комплексов, созданных до настоящего времени, содержат относительно небольшое количество процессоров (как правило, от двух до четырех). Это не означает, что нельзя на комплексе с большим количеством процессоров достичь лучших результатов. Это просто говорит о том, что существующие архитектуры придется изменять, чтобы промышленность могла осуществить свои надежды по внедрению мультипроцессорных комплексов в широких масштабах. В некоторых недавно созданных мультипроцессорных комплексах удалось достичь гораздо лучших показателей производительности. Например, четырехпроцессорный вычислительный комплекс 1100/84 фирмы Sperry Univac приблизительно в 3.6 раза превосходит по производительности однопроцессорную систему 1100/81.

11.14 Экономическая эффективность мультипроцессорных систем

Выбирая мультипроцессорное направление развития вычислительных мощностей вместо многомашинного, нужно тщательно анализировать специфические требования различных организаций и вычислительных центров. Ниже перечисляются некоторые преимущества мультипроцессорной архитектуры:

- Дополнительные процессоры в мультипроцессорный вычислительный комплекс, как правило, можно вводить без привлечения дополнительного обслуживающего персонала.
- Если основная выгода, которую необходимо получить,— это увеличение процессорных вычислительных мощностей, т. е. если имеющегося объема памяти и средств ввода-вывода достаточно, то введение дополнительных процессоров экономически более оправданно, чем приобретение самостоятельных вычислительных машин.
- Симметричные мультипроцессорные вычислительные комплексы модульной конструкции позволяют быстро и экономично подключать новые процессоры.
- Для установки дополнительного процессора требуется гораздо меньшая площадь, чем для целой вычислительной машины.
- Вычислительную мощность системы можно наращивать плавно, меньшими квантами.
- Теория массового обслуживания и теория очередей (см. гл. 15 «Аналитическое моделирование») говорят о том, что при направлении единого потока входящих запросов в систему с многими обслуживающими единицами мы обеспечиваем лучшую пропускную способность, чем в случае, когда входящий поток разделяется на много потоков и направляется на отдельные обслуживающие единицы (если обслуживающие единицы в обоих типах систем имеют одинаковые интенсивности обслуживания).
- Мультипроцессорные системы экономически более эффективны в широком диапазоне рабочих нагрузок.

Однако мультипроцессорным архитектурам свойственно и много недостатков:

- Мультипроцессорные операционные системы являются, как правило, более сложными, чем однопроцессорные.
- Мультипроцессорные аппаратные средства более сложны, чем однопроцессорные.
- После того как в вычислительном центре будет в конце концов установлено максимально возможное количество процессоров мультипроцессорного комплекса, дальнейшее расширение вычислительных мощностей может обойтись довольно дорого, если будет приобретаться второй

мультипроцессорный комплекс.

11.15 Восстановление после ошибок

Одним из самых важных достоинств мультипроцессорных операционных систем является их способность противостоять аппаратным ошибкам, возникающим в отдельных процессорах, и обеспечивать сохранение работоспособности комплекса. Эти возможности создаются преимущественно благодаря тщательно продуманным программным средствам. Для восстановления работоспособности системы при ошибках применяются различные приемы. В частности:

- Критические данные для системы и для различных процессов должны иметь несколько копий. Эти копии должны размещаться в отдельных модулях памяти, с тем чтобы отказы индивидуальных компонент не приводили к полному разрушению этих данных.
- Операционная система должна проектироваться таким образом, чтобы она могла эффективно управлять максимальной конфигурацией аппаратных средств, а также и сокращенными конфигурациями в случае отказов.
- В систему следует включать средства обнаружения и исправления аппаратных ошибок, с тем чтобы можно было осуществлять достаточно жесткий контроль, но не мешая эффективной работе системы.
- Незагруженные процессорные ресурсы следует пытаться использовать для обнаружения потенциальных ошибок еще до того, как они возникнут.
- Операционная система должна обеспечивать передачу процесса, выполнявшегося на вышедшем из строя процессоре, на работоспособный процессор.

11.16 Симметричная мультипроцессорная система TOPS-10

Разработчикам фирмы Digital Equipment Corp. (DEC) при создании системы TOPS-10 удалось преодолеть ограничения, свойственные предыдущим вариантам с организацией «главный — подчиненный», благодаря тому, что они наделили каждый процессор полными функциональными возможностями (Wi80). Полученная организация получила название симметричный мультипроцессор SMP (рис. 11.12). Здесь устройства ввода-вывода могут подключаться к каждому из процессоров. Все вызовы монитора, в том числе запросы на ввод-вывод, могут выполняться на любом процессоре. Тот факт, что вызовы монитора могут обслуживаться любым процессором, означает, что процессор, выполняющий программу, которая выдает запрос ввода-вывода для устройства, подключенного к другому процессору, может после выдачи запроса продолжать выполнение своего задания. А запрос ввода-вывода ставится в очередь для инициирования соответствующим процессором.

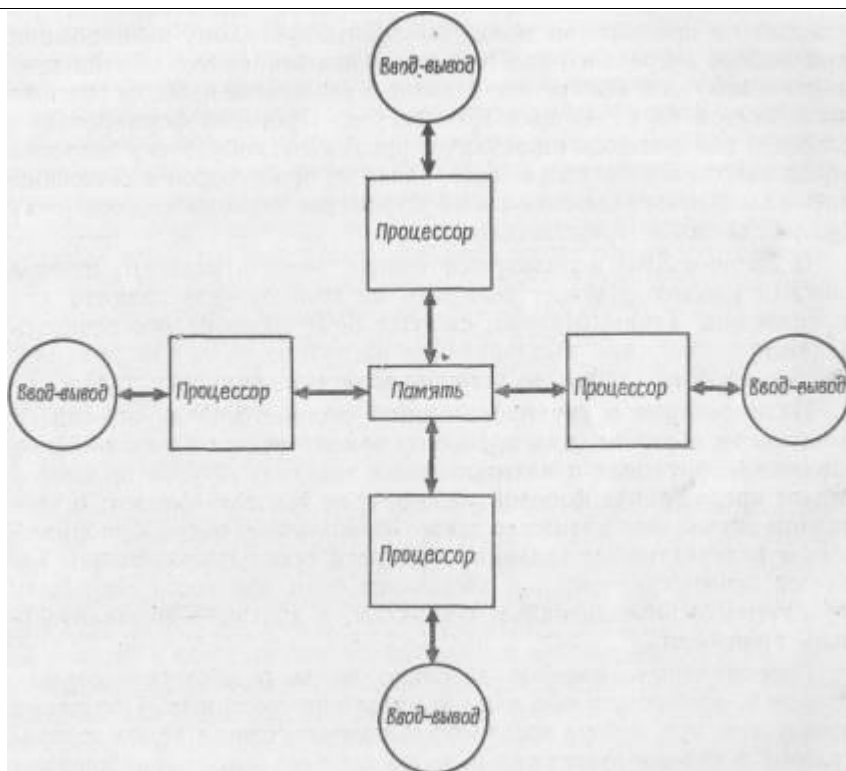


Рис. 11.12. Симметричная мультипроцессорная система TOPS-IO SMP.

В системе SMP процессор, выполняющий конкретное задание, называется *процессором-исполнителем*. Процессор, к которому подключаются различные устройства, используемые при выполнении этого задания, называется *процессором-владельцем* ресурсов. Запросы на ввод-вывод задания, выполняемого на самом процессоре-исполнителе, попадают в очередь ввода-вывода этого процессора-владельца. Запросы на ввод-вывод для других процессоров процессор-исполнитель направляет процессорам-владельцам в их очереди ввода-вывода. Выдав запрос на ввод-вывод, процессор-исполнитель покидает монитор и возобновляет выполнение своего задания — а процессор-владелец обрабатывает запрос на ввод-вывод. Одно из преимуществ такого подхода по сравнению с организацией «главный — подчиненный» заключается в том, что значительно уменьшаются накладные расходы на контекстные переключения. При организации «главный — подчиненный» появление запросов ввода-вывода в подчиненном процессоре вызывает контекстное переключение для главного процессора, а это может обходиться дорого.

В системе SMP планирование построено таким образом, что каждый из процессоров может выполнять программу-планировщик при выборе очередного задания для выполнения. Это обычно приводит к тому, что конкретное задание в различные периоды времени выполняется на различных процессорах. Протокол формирования очередей ввода-вывода гарантирует правильную обработку запросов ввода-вывода независимо от того, какой из процессоров в настоящее время выполняет задание и какие устройства ввода-вывода содержат файлы данного пользователя.

В системе SMP используется единая очередь заданий, причем каждый процессор может выбирать из этой очереди задание для выполнения. Таким образом, система SMP способна обеспечивать балансирование, или выравнивание нагрузки, чего, как правило, не делается в других мультипроцессорных

архитектурах.

Планирование в двухпроцессорной системе SMP производится следующим образом: один процессор ориентируется на выполнение высокоприоритетных и интерактивных заданий. Другой процессор отдает предпочтение фоновой работе, если таковая имеется; в противном случае этот процессор также обрабатывает высокоприоритетные и интерактивные задания. Подобная схема планирования является асимметричной; она заставляет один процессор оказывать предпочтение интерактивным процессам, а другой — вычислительным процессам.

Переключения, которые довольно часты в однопроцессорных системах, происходят еще чаще в мультипроцессорных. Поскольку процессоры при выборе процессов выполняют одни и те же подпрограммы и анализируют одни и те же данные, между процессорами может возникнуть сильная конкуренция. Чтобы свести ее к минимуму в системе SMP, синхронизаторы всех процессоров работают со смещением, с тем чтобы временные прерывания происходили в различные моменты времени.

Когда по временному прерыванию процессор входит в диспетчер, он просматривает также глобальную очередь запросов на ввод-вывод, чтобы определить, какие операции ввода-вывода он должен выполнить для процессов на других процессорах. После завершения необходимых операций ввода-вывода ожидающие процессы получают возможность дальнейшего выполнения. Эти процессы могут затем выполняться на любом свободном процессоре.

Одна интересная проблема, возникающая в этой мультипроцессорной конфигурации, заключается в том, что один процессор может изменить данные процесса, хранящиеся в кэш-памяти, а другой процессор может попытаться выполнить этот процесс, используя устаревшие данные в основной памяти. Чтобы предотвратить такую ситуацию, процессор должен обеспечить замену старых данных основной памяти обновленными данными кэш-памяти, прежде чем будут сделаны какие-либо другие попытки использовать эти данные в следующий раз.

В системе SMP имеют место частые взаимодействия между процессорами, особенно в связи с тем, что все процессоры коллективно используют одну копию операционной системы. Для реализации межпроцессорных взаимодействий разработчики системы SMP остановились на *программном звонке в дверь*. Процессорам не разрешается прерывать работу друг друга при помощи механизма аппаратных прерываний. Вместо этого каждый процессор, когда ему больше нечего делать, сам «слушает», нет ли «звонка в дверь».

Чтобы повысить надежность и коэффициент готовности системы SMP, в ней предусмотрен контроль за работоспособностью основной памяти, в которой размещается операционная система. Когда операционная система определяет, что память работает с ошибками, она просто перемещает сама себя в правильно функционирующую область памяти.

11.17 Системы C.mmp и Cm*

Специалисты Университета Карнеги—Меллона разработали несколько интересных мультипроцессорных систем, в том числе C.mmp и Cm*. Система C.mmp (Si78a, Si78b, Wu80) — это 16-процессорный вычислительный комплекс, объединяющий миникомпьютеры PDP-11/40. Эти процессоры коллективно используют 16 модулей памяти, к которым они подключаются при помощи матрицы координатной коммутации. Благодаря симметричности такой схемы перекрестных соединений обращения от любого процессора к любому модулю памяти

осуществляются при одинаковых затратах. В системе могут одновременно происходить 16 обращений к памяти при условии, если каждое из них связано со своим портом памяти. Поскольку сложность матрицы координатной коммутации увеличивается пропорционально квадрату числа процессоров и устройств памяти, архитектуру системы C.mmp практически нельзя будет применять при создании очень крупных мультипроцессорных комплексов.

Система Cm* (Ar75, Ch79, Fv78, Ha82, Jo77, Si78a, Si78b, Sw77a, Sw77b) содержит 50 микропроцессоров LSI-11. Система построена из пар элементов «процессор-память», называемых *компьютерными модулями (Cm)*. Компьютерные модули Cm объединяются в *кластеры*. Кластеры соединяются друг с другом при помощи *межкластерных шин*. Затраты на обращение к памяти здесь зависят от того, производится ли обращение к локальной памяти модуля, к памяти другого модуля того же самого кластера или к памяти модуля другого кластера. Затраты для этих трех различных типов обращений к памяти имеют соотношение 1 : 3 : 9. В связи с этим возможность достижения оптимальной производительности системы при решении конкретных прикладных задач определяется тем, удастся ли обеспечить, чтобы большинство обращений относилось к локальной памяти соответствующих модулей.

Обе указанные мультипроцессорные системы применяются для решения большого числа задач, характеризующихся высокой степенью внутреннего параллелизма. Во многих случаях удается достигнуть почти линейного увеличения производительности (по сравнению с однопроцессорными машинами). Линейное увеличение производительности означает, что при наличии n процессоров данное задание будет выполняться в n раз быстрее. Такой эффект наблюдается, например, при решении некоторых математических прикладных задач, в частности задач целочисленного программирования и дифференциальных уравнений в частных производных. Результаты этих работ показывают, что мультипроцессорные системы с большим числом процессоров могут значительно уменьшить время выполнения программ, реализующих алгоритмы с высокой степенью параллелизма.

11.18 Перспективы мультипроцессорных систем

Практически все говорит за то, что в будущем применение мультипроцессорных систем и комплексов значительно расширится. Существует много причин для столь оптимистического прогноза.

- Надежность вычислительных машин и комплексов становится все более важным фактором.
- Благодаря достижениям микроэлектроники стоимость процессоров быстро снижается.
- Предполагается значительно более широкое использование языков, которые позволят программистам явно указывать параллелизм. Мультипроцессорные системы могут выполнять программы с высокой степенью параллелизма гораздо быстрее, чем однопроцессорные. В будущем безусловно появятся прикладные задачи, для решения которых исключительно важную роль будет играть истинная одновременность работы процессоров мультипроцессорной системы.
- Интенсивно развиваются методы автоматического обнаружения параллелизма.
- Однопроцессорные вычислительные машины сейчас приближаются к пределам своих возможностей. Электромагнитные сигналы нельзя передавать быстрее скорости света, а некоторые однопроцессорные машины уже работают на частотах, при которых электромагнитные сигналы

за время выполнения одной операции компьютера могут переместиться менее чем на 30 см. Ограничения по размерам и длинам связей со временем вынудят разработчиков для дальнейшего повышения производительности идти по пути применения мультипроцессорных архитектур.

Различные исследования тенденций развития архитектур вычислительных машин указывают на необходимость создания систем, в которых при построении иерархии процессоров сочетаются как симметричные, так и несимметричные связи. Такие системы стали называть *полипроцессорами*.

Независимо от того, какая архитектура будет доминировать в будущих мультипроцессорных вычислительных комплексах, очевидно, что она будет существенным образом влиять на операционные системы будущего.

Заключение

Мультипроцессорная система содержит несколько процессоров. Поскольку процессорное время сейчас становится все более дешевым, главные проблемы, которые приходится решать разработчикам мультипроцессорных систем,— это надежность, параллелизм в вычислениях, оптимальные схемы соединений и разрешение конфликтов между процессорами.

В настоящее время лишь немногие программы пишутся в расчете на параллельные вычисления. Можно надеяться, что аппаратура, компиляторы и операционные системы со временем будут автоматически обнаруживать параллелизм в алгоритмах.

При максимальном распараллеливании система содержит достаточное количество процессоров, с тем чтобы все части программы, которые в принципе могут выполняться совмещенно, удавалось распределять по отдельным процессорам.

Существуют два распространенных способа использования неявного параллелизма — это расщепление цикла и редукция высоты дерева. Расщепление цикла предполагает, что повторяющиеся операции, которые могут выполняться независимо друг от друга, разделяются на цепочки для параллельного выполнения. Редукция высоты дерева опирается на коммутативность, ассоциативность и дистрибутивность арифметических операций для преобразования алгебраических выражений для проведения параллельных вычислений.

Правило «никогда не ждать» говорит о том, что лучше поручить некоторому процессору вычисление, результаты которого могут вообще не понадобиться, чем позволить этому процессору оставаться без нагрузки.

Существуют три наиболее распространенных вида организации мультипроцессорных систем: с разделяемой во времени, или общей шиной, с матрицей координатной, или перекрестной, коммутации и с многопортовой памятью. Разделяемая во времени шина — это один канал связи между всеми функциональными устройствами. Это простейшая из трех перечисленных схем. Она обеспечивает простоту подключения новых функциональных устройств. Ее главные недостатки — что вся система выходит из строя в случае отказа шины, общая скорость обмена информацией в системе ограничивается скоростью

передачи данных по шине и конфликтные ситуации при обращении к шине в загруженной системе могут привести к серьезному снижению производительности (деградации).

В системе с матрицей координатной коммутации предусматривается отдельный тракт связи с каждым устройством памяти. Благодаря этому может производиться обмен информацией со всеми устройствами памяти одновременно. Главный недостаток подобной организации — сложность координатного коммутатора.

В схеме с многопортовой памятью каждое функциональное устройство может получать доступ к каждому устройству памяти, однако только по конкретному порту памяти. Многопортовая организация позволяет ограничивать доступ к различным устройствам памяти, разрешая его только для определенных подгрупп центральных процессоров и процессоров ввода-вывода. Это полезно для создания систем высокой секретности. Основной недостаток такой организации — большое количество кабельных связей.

Мультипроцессорные системы со слабо связанными процессорами, или многомашинные вычислительные комплексы, предусматривают соединение двух или более независимых вычислительных машин при помощи канала передачи данных. В мультипроцессорных системах с сильно связанными процессорами применяется единая память, коллективно используемая всеми процессорами.

В мультипроцессорной системе, реализующей принцип «главный — подчиненный», один процессор выполняет функции главного, а остальные являются подчиненными. Главный процессор может выполнять как вычисления, так и операции ввода-вывода. Подчиненные процессоры, или сопроцессоры, выполняют только вычисления. Такие подчиненные процессоры лучше всего подходят для выполнения вычислительных заданий. Подчиненный процессор, который выполняет задание, связанное с большим объемом операций ввода-вывода, генерирует частые прерывания для главного процессора, который должен выполнять запросы на ввод-вывод для своего подчиненного. Подобная асимметричность аппаратных средств — наибольшая проблема мультипроцессорных систем с организацией «главный — подчиненный».

В мультипроцессорных комплексах применяются три основных способа организации операционных систем: «главный — подчиненный», с отдельными мониторами и симметричная организация (при которой все процессоры идентичны). Организацию «главный — подчиненный» реализовать легче всего. Только главный процессор может выполнять программы операционной системы. Программы операционной системы не обязательно должны быть реентерабельными, а взаимное исключение при обращении к системным таблицам существенно упрощается. Отказ главного процессора является катастрофическим отказом системы.

При организации с отдельными мониторами каждый процессор имеет свою собственную операционную систему. Для управления доступом к глобальной системной информации применяются методы взаимного исключения. Отказ одного процессора вряд ли будет катастрофическим для системы. Каждый процессор управляет своими собственными ресурсами. Каждый процесс выполняется до завершения на том процессоре, которому он передан для выполнения. Загрузка ресурсов может оказаться неэффективной — не исключено, что некоторые процессоры будут работать вхолостую, в то время как один процессор будет выполнять длинный процесс.

Симметричная мультипроцессорная архитектура является наиболее сложной для реализации и наиболее эффективной. Операционная система управляет пулом идентичных процессоров, один из которых может использоваться для управления

любым устройством ввода-вывода обращаться к любому устройству памяти. Реентерабельный код и методы взаимоисключения здесь обязательны. Благодаря симметричности архитектуры возможно хорошее балансирование рабочей нагрузки. Отказ одного процессора приводит к тому, что операционная система исключает этот процессор из пула процессоров, однако вычислительный комплекс продолжает функционировать. Некоторый процесс может в различные периоды времени выполняться на различных процессорах. Операционная система перемещается с процессора на процессор. Соревнование может стать серьезной проблемой, поскольку несколько процессоров могут одновременно работать в режиме супервизора.

Кратко рассмотрены три реальные мультипроцессорные системы: TOPS-10 SMP фирмы DEC и системы C.mmp и Cm*, разработанные в Университете Карнеги—Меллона. В системе SMP применяется интересный асимметричный алгоритм планирования. Чтобы свести к минимуму вероятность конфликтных ситуаций в системе SMP, синхронизаторы всех процессоров работают с небольшим смещением, так что временные прерывания происходят в различные моменты времени. Взаимодействие между процессорами в системе SMP осуществляется программно по принципу «звонка в дверь». Система C.mmp — это крупный мультипроцессорный комплекс, построенный путем объединения 16 миникомпьютеров PDP-11/40 при помощи матрицы координатной коммутации. Система Cm* — очень большой мультипроцессорный комплекс, созданный путем соединения 50 микропроцессоров LSI-11 фирмы DEC, которые входят в отдельные кластеры. В системе Cm* затраты на обращение

к памяти получаются различными в зависимости от того, производятся ли обращения к локальной памяти каждого модуля, к памяти другого модуля того же самого кластера или к памяти модуля другого кластера. Оба комплекса обеспечивают почти линейное увеличение производительности (по сравнению с однопроцессорными машинами) при решении задач с высокой степенью собственного параллелизма.

Мультипроцессорные архитектуры сейчас начинают играть все более важную роль и безусловно окажут серьезное влияние на развитие операционных систем.

Терминология

автоматическое распараллеливание (automatic detection of parallelism)

асимметричная мультипроцессорная система (архитектура) (asymmetric multiprocessing)

балансирование, выравнивание нагрузки процессоров (processor load balancing)

взаимодействие, связь, обмен информацией между процессорами (interprocessor communication)

восстановление (после ошибок) (recovery)

«главный — подчиненный» (master/slave)

главный процессор (master processor)

готовность, работоспособность (availability)

избыточность (redundancy)

катастрофический отказ (catastrophic failure)

кластер (cluster (Cm*))

компьютерные модули (пары «процессор—память») (computer modules)

конфликтная ситуация, состязание при обращении к памяти (storage contention)

максимальное распараллеливание (massive parallelism)

матрица координатной (перекрестной) коммутации (crossbar-switch matrix)

межкластерная шина (intercluster bus (Cm*))

многопортовая память (multiport storage)

мониторный процессор (executive processor)

мультипроцессорные комплексы с сильно связанными процессорами; мультипроцессорные вычислительные комплексы, архитектуры (tightly coupled multiprocessing)

мультипроцессорные комплексы со слабо связанными процессорами,* многомашинные вычислительные комплексы, архитектуры (loosely coupled multiprocessing)

мультипроцессорная архитектура, мультипроцессорный режим (multiprocessing)

мультипроцессорная система, мультипроцессорный (многопроцессорный) вычислительный комплекс, мультипроцессор (multiprocessor)

надежность (reliability)

неявный (скрытый) параллелизм (implicit parallelism)

операторы COBEGIN/COEND

перемещающаяся операционная система, исполнительная программа (floating executive)

подчиненный процессор (slave processor)

поли процессорная система, полипроцессор (polyprocessor)

порт, или канал памяти (storage port)

постепенная (плавная) деградация, снижение производительности системы; снижение функциональных и скоростных возможностей системы (graceful degradation)

правило «никогда не ждать» (Never Wait Rule)

приоритетная арбитражная логика (priority arbitration logic)

программный «звонок в дверь» (software doorbell)

процессор-владелец (ресурсов) (owning processor (SMP))

процессор-исполнитель (executing processor (SMP))

раздельные мониторы, исполнительные программы, или операционные системы (separate executives)

разделяемая во времени, или общая, шина (timeshared or common bus)

расщепление программного цикла (loop distribution)

редукция высоты дерева (tree height reduction)

реконфигурация (reconfiguration)

симметричная мультипроцессорная система (архитектура) (symmetric multiprocessing)

симметричная мультипроцессорная система TOPS-10 SMP

система Cm*

система C.mmp

система массового обслуживания со многими обслуживаемыми единицами (multiple server queuing system)

система массового обслуживания с одной обслуживаемой единицей (single server queuing system)

соревнование, состязание, конкуренция (contention)

шина (bus)

явный параллелизм (explicit parallelism)

Упражнения

11.1 Что такое расщепление цикла? При каких обстоятельствах целесообразно применять этот способ? Какие можно привести аргументы за и против расщепления цикла?

11.2 Что такое «редукция высоты дерева»? При каких обстоятельствах оправдано применение этого способа?

11.3 Что такое правило «никогда не ждать»? При каких обстоятельствах оправдано применение этого правила?

11.4 Опишите мультипроцессорную организацию с разделяемой во времени, или общей шиной. Каковы ее преимущества и недостатки?

11.5 Что такое катастрофический отказ в мультипроцессорной системе? Что могут сделать

разработчики для того, чтобы свести вероятность катастрофического отказа к минимуму?

11.6 Опишите мультипроцессорную организацию с матрицей координатной, или перекрестной, коммутации. Каковы ее преимущества и недостатки?

11.7 Каким образом следует произвести перестройку механизмов распределения ресурсов симметричной мультипроцессорной системы, чтобы отреагировать на потерю вычислительной мощности в случае выхода из строя одного процессора?

11.8 Какова главная цель создания большинства мультипроцессорных систем? Перечислите несколько преимуществ мультипроцессорных архитектур по сравнению с однопроцессорными. Перечислите несколько недостатков.

11.9 Что такое максимальное распараллеливание? Какие виды проблем можно решить с помощью такого подхода?

11.10 Что такое явный параллелизм? В чем его недостатки?

11.11 Что такое неявный параллелизм? Обсудите, почему так важно разрабатывать механизмы, которые будут автоматически обнаруживать неявный параллелизм.

11.12 Опишите мультипроцессорную организацию с многопортовой памятью. В чем ее преимущества и недостатки?

11.13 Что такое мультипроцессорная система со слабо связанными процессорами? Что такое мультипроцессорная система с сильно связанными процессорами?

11.14 Обсудите мультипроцессорную организацию типа «главный — подчиненный». Сравните ее с симметричной мультипроцессорной архитектурой.

11.15 Укажите сходства и различия следующих видов организации мультипроцессорных операционных систем: «главный — подчиненный», с отдельными мониторами, симметричная.

11.16 Какой аспект архитектуры системы S.mmp делает ее непригодной для построения очень крупных мультипроцессорных комплексов? Почему архитектура системы Sm* более подходит для этой цели?

11.17 Тщательно проанализируйте программу, которую вы написали, и укажите, какие операции можно было бы выполнять параллельно.

а) Сколько времени вы на это затратили?

б) Насколько вы уверены в том, что нигде не ошиблись?

в) Предположим, что вам потребовалось модифицировать программу. Как вы думаете, какую программу легче было бы модифицировать, последовательную или параллельную? Объясните.

11.18 Как разработчику вычислительных систем, имеющему опыт создания и аппаратных, и программных средств, вам поручили задачу проектирования сверхвысоконадежной мультипроцессорной системы. Расскажите, какие аппаратные и программные методы и средства вы могли бы выбрать для этого.

11.19 Задание, которое обычно выполнялось на однопроцессорном микрокомпьютере LSI-11, необходимо перевести на мультипроцессорный вычислительный комплекс Sm*, содержащий n взаимосвязанных микрокомпьютеров LSI-11. Приведите по меньшей мере два весомых соображения, на основании которых можно было бы судить о том, удастся ли достигнуть почти линейного повышения производительности.

11.20 Рассмотрите следующий цикл на Фортране:

DO 1, I=1, 20

IF (X.EQ.0) Y(1) = 3.5

1IF (X.NE.0) Y(1) = 5

а) Перекодируйте этот фрагмент программы на язык ассемблера.

б) Перекодируйте этот фрагмент программы таким образом, чтобы он выполнялся гораздо более эффективно на однопроцессорной машине.

в) Предположим, что имеется четырехпроцессорный симметричный вычислительный комплекс. Напишите как можно более эффективную параллельную программу, которая делала бы точно то же самое, что и указанный цикл на Фортране.

г) Предположим, что имеется симметричный мультипроцессорный комплекс с максимальным распараллеливанием. Напишите как можно более эффективную параллельную программу, которая делала бы то же самое, что и указанный цикл на Фортране.

д) Сравните скорость выполнения цикла на Фортране со скоростями выполнения программ, написанных вами по заданиям (б), (в) и (г). При этом предположите, что все процессоры обладают идентичными характеристиками.

11.21 Используя методы редукции высоты дерева, преобразуйте следующее выражение к виду, более пригодному для параллельных вычислений. В каждом случае вычертите деревья вычислений для исходного выражения и для нового выражения.

а) $(p+(q+(r+s)))$;

б) $((a+b*c*d)*e)$;

в) $(m+(n*p*q*r)+a+b+c)$;

г) $(a*(b+c+d*(e+f)))$.