В. Г. Олифер, Н. А. Олифер

Сетевые операционные системы

2-е издание

ПИТЕР

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2009

ББК 32.973.2-018.2я7
УДК 004.451(075)
О-54

Рецензенты:

Тювин Ю. Д., профессор, заведующий кафедрой «Вычислительная техника» факультета «Вычислительные машины и системы» Московского государственного института радиотехники, электроники и автоматики (технического университета МИРЭА)

Кузнецов С. Д., доктор технических наук, ИСП РАН

Олифер В., Олифер Н.

О-54 Сетевые операционные системы: Учебник для вузов. 2-е изд. — СПб.: Питер, 2009. — 669 с.: ил.

ISBN 978-5-91180-528-9

Эта книга — не о конкретной системе и даже не о конкретном типе операционных систем. Она рассматривает фундаментальные концепции и принципы построения, справедливые для большинства известных на сегодня операционных систем. В первую очередь это издание рекомендуется студентам и аспирантам различных специальностей направления «Информатика и вычислительная техника» как учебное пособие по курсам «Операционные системы» и «Организация вычислительных процессов». Кроме того, оно может быть полезно специалистам: программистам, сетевым администраторам. И наконец, книга может заинтересовать всех, кто имеет дело с компьютерами и хочет больше узнать о том, как устроены современные операционные системы.

Допущено Министерством образования Российской Федерации в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника».

ББК 32.973.2-018.2я7
УДК 004.451(075)

ISBN 978-5-91180-528-9

Краткое содержание

Содержание

Благодарности

Прежде всего, мы хотим поблагодарить наших читателей за их многочисленные пожелания, вопросы и замечания. Мы особенно признательны профессору кафедры «Вычислительные машины и сети» Санкт-Петербургского государственного университета аэрокосмического приборостроения Гордееву А. В., а также доценту кафедры информатизации структур государственной службы Российской академии государственной службы при Президенте РФ Бородько В. П. за их доброжелательные советы и конструктивные предложения, касающиеся второго издания этой книги.

Мы благодарим сотрудников издательства «Питер», которые участвовали в работе над книгой. Огромное спасибо нашему неизменному литературному редактору Алексею Жданову, тщательность и «дотошность» которого при работе с текстом часто выходили за рамки традиционного литературного редактирования, что в значительной степени помогло улучшить учебник.

Виктор и Наталья Олифер

Предисловие авторов ко второму изданию

Во втором издании сохранилась основная направленность книги, а именно — изучение **принципов организации и базовых механизмов современных операционных систем**, в отличие от многих других книг, посвященных конкретным приемам работы с какой-либо определенной ОС. При этом в книге рассматриваются принципы и механизмы, нашедшие свое воплощение как в применяемых сегодня популярных ОС, таких как Unix/Linux, Microsoft Windows и Mac OS, так и в ОС 70–90-х годов, таких как IBM 370, Novell NetWare, но затем по каким-либо причинам не реализованные в более поздних ОС.

Такой подход оказывается очень полезным при изучении любой области, в том числе и области операционных систем. Нельзя отказываться от изучения некоторого метода организации вычислительных процессов только потому, что он не нашел применения в последней версии популярной ОС. «Новое — это хорошо забытое старое», и очень часто приходится возвращаться к технологиям, которые, казалось бы, совсем ушли в прошлое. Ярким примером тому может служить возвращение мультипрограммирования в ОС персональных компьютеров. Старое возвращается и получает новое развитие — именно это и есть развитие по спирали, которое, как мы считаем, должно лежать в основе любого процесса обучения. Специалист должен включать в свой багаж все наработанные к данному моменту фундаментальные концепции, с тем чтобы распознать в новой модной технологии давно известную, хотя и замаскированную новыми терминами идею, и, пользуясь этим фактом, быстро ухватить суть «новинки».

Основные отличия второго издания от первого заключаются в следующем:

- Во второе издание книги включена новая глава, в которой в компактной форме дано **введение в организацию транспортной подсистемы компьютерных сетей**, в том числе описаны транспортные средства ОС. Собственно, транспортные средства ОС отдельного компьютера являются неотъемлемой частью коммуникационных средств компьютерной сети, которые, помимо конечных узлов сети — компьютеров, — включают также промежуточные узлы, такие как маршрутизаторы и коммутаторы. Читатели знакомятся с основным

методом передачи компьютерных данных — коммутацией пакетов, а также принципами работы использующих этот метод сетевых устройств — коммутаторов и маршрутизаторов. Описание работы сети дается на основе классической семиуровневой модели OSI, которая стандартизует многоуровневый подход к организации транспортных протоколов. Очевидно, что из всех возможных стеков протоколов для рассмотрения в этой книге был выбран **стек TCP/IP**, являющийся необходимым функциональным элементом любой современной сетевой ОС и составляющий основу Интернета. Читатели узнают о классах IP-адресов и использовании масок, о системе доменных имен (DNS) и протоколе динамического распределения адресов (DHCP), о создании таблиц маршрутизации в автоматическом и ручном режимах. В качестве упражнения в книге описывается процедура конфигурирования параметров стека TCP/IP универсальной ОС, а в качестве развернутого примера технологии канального уровня в книгу включено описание технологии Ethernet, доминирующей при построении локальных сетей. Эти дополнения сделали учебник самодостаточным, в отличие от первого издания, в котором читатель для изучения вопросов организации транспортной системы отсылался к книге авторов «Компьютерные сети».

- Другим важным дополнением книги является описание особенностей **операционных систем маршрутизаторов на примере Cisco IOS**. Маршрутизаторы и коммутаторы компьютерной сети работают под управлением собственного программного обеспечения, которое во многих случаях имеет достаточно сложную организацию, что дает право называть его специализированными сетевыми ОС. В разделе, посвященном специализированной ОС Cisco IOS, рассматривается модульная структура этой ОС, управление процессами и обработка прерываний, организация памяти и работа с буферами пакетов, поддержка программной маршрутизации и ускоренная коммутация пакетов. В заключение описываются средства IOS, направленные на поддержку качества обслуживания трафика.
- Во втором издании существенно расширен раздел, посвященный **справочным сетевым службам**. При описании конкретных реализаций акцент сделан на справочную службу **Active Directory** компании Microsoft: рассмотрены принципы иерархической организации этой справочной службы на основе доменов и организационных единиц, а также методы оптимизации репликации с учетом определенных в системе сайтов.
- Обновлено значительное число **примеров**, в которых использованы новые версии ОС и системного программного обеспечения.

Кроме того, авторы стремились сделать учебник более понятным, для чего значительная часть текста претерпела существенную литературную правку, в некоторых случаях изменялась и структура материала. Мы надеемся, что текст стал более «читабельным» также благодаря тому, что были широко использованы разнообразные средства форматирования и выделения текста, призванные подчеркивать новые термины и понятия, а также выделять главные тезисы.

Для кого эта книга

В первую очередь, эта книга адресуется студентам и аспирантам различных специальностей направления «Информатика и вычислительная техника» как учебное пособие по курсам «Операционные системы» и «Организация вычислительных процессов». В основу книги положен многолетний опыт одного из авторов по преподаванию указанных курсов на кафедре вычислительной техники в МИРЭА.

Книга может быть также полезна специалистам: программистам, которые хотят более эффективно использовать системные средства, сетевым администраторам, которым нужны знания принципов работы ОС для оптимизации операционной сетевой среды, специалистам по коммуникационному оборудованию, создающим транспортную инфраструктуру сети. Авторы убедились в этом, проводя в течение ряда лет курсы в московском Центре информационных технологий для специалистов-практиков, повышавших свою квалификацию в области операционных систем.

И, наконец, книга может заинтересовать всех, кто имеет дело с компьютерами и хочет больше узнать о том, как устроены современные операционные системы.

Структура книги

Первые три главы вводят читателя в проблематику операционных систем.

Глава 1, посвященная **истории и эволюции операционных систем**, во втором издании учебника **значительно расширена**. При написании этой главы авторы столкнулись с непростой проблемой. С одной стороны, как рассказать читателям о развитии идей в области операционных систем, если читатели еще даже не начали изучение предмета и не готовы глубоко понять суть этих идей. С другой стороны, прежде чем начать изучение конкретных концепций, методов и технологий, авторам хотелось донести до читателей динамику и революционность избранной ими области знания, рассказать о ее первопроходцах, а это невозможно сделать без упоминания созданных за эти несколько десятилетий методов, технологий и систем. Выбранное решение оказалось компромиссным. При описании достижений, ставших «историческими вехами», авторы позволили себе использовать термины и названия, о большинстве из которых читатели, только приступающие к изучению операционных систем, имеют весьма смутное представление. Однако во многих случаях эти термины и понятия сопровождаются некоторыми минимально необходимыми краткими пояснениями. Кроме того, авторы надеются, что читатели могут еще раз вернуться к этой главе после того, как все те концепции, упоминаниями о которых так насыщена эта вводная глава, станут им понятными.

В *главах 2 и 3* обсуждается назначение и архитектура современных операционных систем. Здесь исследуются функциональная и структурная организация ОС, а также основные подсистемы и компоненты, используемые для управления

как локальными, так и разделяемыми сетевыми ресурсами. При этом рассматриваются классическая многослойная организация ОС с монолитным ядром и микроядерная архитектура.

Следующие пять глав (с *четвертой* по *восьмую*) посвящены концепциям и механизмам управления локальными ресурсами компьютера: процессором, памятью и внешними устройствами. Изучаются понятия процесса и потока, различные дисциплины планирования и диспетчеризации, применяемые в системах пакетной обработки, разделения времени и реального времени. Достаточно подробно рассматриваются разные схемы реализации механизма прерываний и его роль в организации вычислительного процесса.

Методы управления памятью даны в исторической ретроспективе — от самых простых схем с фиксированными разделами до современной сегментно-страничной организации памяти. Детально исследуется концепции виртуальной памяти и кэширования данных.

При рассмотрении файловой системы и внешних устройств применяется современных подход, согласно которому файловая система является неотъемлемой частью подсистемы ввода-вывода, состоящей из драйверов различного уровня, объединенных общим менеджером. В качестве примеров используются наиболее распространенные файловые системы, такие как ufs, NTFS и FAT. Изучаются такие важные функции файловых систем, как устойчивость к сбоям и отказам, а также контроль доступа к хранимым данным.

Управление локальными ресурсами тесно связано с аппаратными средствами организации вычислительного процесса. *Глава 6* полностью посвящена подробному описанию такого рода средств, встроенных в процессоры семейства Pentium. На примере этого процессора показано взаимодействие программных и аппаратных средств компьютера при планировании процессов и потоков, распределении памяти, защите данных на разных уровнях.

Глава 9 является компактным введением в сетевые технологии, о ее содержании уже рассказывалось при описании отличий второй редакции учебника.

В *главе 10* обсуждаются концепция распределенных вычислений, типы многозвенных приложений и средства их реализации — системы передачи сообщений и удаленного вызова процедур. Теоретические схемы иллюстрируются конкретными механизмами, такими как сокеты Беркли и механизмы Sun RPC и DCE RPC.

Концепция распределенных вычислений широко используется при изучении сетевых служб, которым посвящена *глава 11*. Подробно описана файловая служба, являющаяся одним из основных компонентов любой сетевой ОС. Рассматриваются различные протоколы взаимодействия клиентской и серверной частей файловой службы, такие как NFS, SMB, FTP и NCP. Далее изучаются назначение и архитектура службы каталогов как центрального элемента современной ОС на примере Microsoft Active Directory. Отдельный раздел посвящен проблеме взаимодействия сетевых служб в гетерогенной среде с использованием методов трансляции, мультиплексирования и инкапсуляции протоколов.

Завершает книгу *глава 12*, посвященная сетевой безопасности. Она включает описание основных угроз, возникающих при работе в сети, и различных подходов, используемых для защиты данных. Рассматриваются базовые технологии сетевой безопасности: шифрование, аутентификация, авторизация, цифровая подпись, цифровые сертификаты, аудит, защищенный канал. Совместная работа различных технологий безопасности иллюстрируется на примере интегрированной системы защиты данных Kerberos.

В конце каждой главы даются выводы, а также приводятся задачи и упражнения. Книга снабжена индексным указателем и списком рекомендуемой литературы.

Мы с благодарностью примем ваши отзывы по адресам Natalia@Olifer.co.uk и Victor@Olifer.co.uk.

Виктор Олифер, к.т.н., CCIP
Наталья Олифер, к.т.н., доцент

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

Глава 1

Эволюция операционных систем

История развития любой отрасли науки или техники позволяет не только удовлетворить естественное любопытство, но и глубже понять сущность основных достижений этой отрасли, осознать существующие тенденции и правильно оценить перспективность тех или иных направлений развития. За почти полувековой период своего существования операционные системы (ОС) прошли сложный путь, насыщенный многими важными событиями. Огромное влияние на развитие операционных систем оказали успехи в совершенствовании элементной базы и вычислительной аппаратуры, поэтому многие этапы развития ОС тесно связаны с появлением новых типов аппаратных платформ, таких как миникомпьютеры или персональные компьютеры. Серьезную эволюцию операционные системы претерпели в связи с новой ролью компьютеров в локальных и глобальных сетях. Важнейшим фактором развития ОС стал Интернет. По мере того как эта Сеть приобретает черты универсального средства массовых коммуникаций, ОС становятся все более простыми и удобными в использовании, включают развитые средства поддержки мультимедийной информации, снабжаются надежными механизмами защиты.

Первые операционные системы

Идея компьютера была предложена английским математиком Чарльзом Бэбиджем (Charles Babage) в середине девятнадцатого века. Его механическая «аналитическая машина» так и не смогла по-настоящему заработать, потому что технологии того времени не удовлетворяли требованиям, необходимым для изготовления нужных деталей точной механики. Конечно, никакой речи об операционной системе для этого «компьютера» не шло.

Настоящее рождение цифровых вычислительных машин произошло вскоре после окончания второй мировой войны. В середине 40-х в США и Европе (Англия) примерно одновременно были созданы первые ламповые вычисли-

тельные устройства. Чуть позже, в 1951 году, была создана первая электронная вычислительная машина в СССР.

В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Не было никакого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм, которые программист мог использовать для того, чтобы не писать каждый раз коды, вычисляющие значение какой-либо математической функции или управляющие стандартным устройством ввода-вывода. Операционные системы все еще не появились, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления, который представлял собой примитивное устройство ввода-вывода, состоящее из кнопок, переключателей и индикаторов.

С середины 50-х годов начался следующий период в развитии вычислительной техники, связанный с появлением новой технической базы — полупроводниковых элементов. Выросло быстродействие процессоров, увеличились объемы оперативной и внешней памяти. Компьютеры стали более надежными, теперь они могли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно практически важных задач.

Наряду с совершенствованием аппаратуры заметный прогресс наблюдался в области автоматизации программирования и организации вычислительных работ. В эти годы появились первые алгоритмические языки, и, таким образом, к библиотекам математических и служебных подпрограмм добавился новый тип системного программного обеспечения — трансляторы.

Выполнение каждой программы стало включать большое количество вспомогательных работ: загрузка нужного транслятора (АЛГОЛ, ФОРТРАН, КОБОЛ, и т. п.), запуск транслятора и получение результирующей программы в машинных кодах, связывание программы с библиотечными подпрограммами, загрузка программы в оперативную память, запуск программы, вывод результатов на периферийное устройство. Для организации эффективного совместного использования трансляторов, библиотечных программ и загрузчиков в штат многих вычислительных центров были введены должности операторов, профессионально выполнявших работу по организации вычислительного процесса для всех пользователей этого центра.

Но как бы быстро и надежно ни работали операторы, они никак не могли состязаться в производительности с работой устройств компьютера. Большую часть времени процессор простаивал в ожидании, пока оператор запустит очередную задачу. А поскольку процессор представлял собой весьма дорогое устройство, то низкая эффективность его использования означала низкую эффективность использования компьютера в целом. Для решения этой проблемы были разработаны первые системы пакетной обработки, которые автоматизировали

всю последовательность действий оператора по организации вычислительного процесса.

Ранние **системы пакетной обработки** явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными не для обработки данных, а для управления вычислительным процессом.

В ходе реализации систем пакетной обработки был разработан формализованный **язык управления заданиями**, с помощью которого программист сообщал системе и оператору, какие действия и в какой последовательности он хочет выполнить на вычислительной машине. Типовой набор директив обычно включал признак начала отдельной работы, вызов транслятора, вызов загрузчика, признаки начала и конца исходных данных.

Оператор составлял *пакет заданий*, которые в дальнейшем без его участия последовательно запускались на выполнение управляющей программой — *монитором*. Кроме того, монитор был способен самостоятельно обрабатывать наиболее часто встречающиеся при работе пользовательских программ аварийные ситуации, такие как отсутствие исходных данных, переполнение регистров, деление на ноль, обращение к несуществующей области памяти и т. д. Пакет обычно представлял собой набор перфокарт, но для ускорения работы он мог переноситься на более удобный и емкий носитель, например на магнитную ленту или магнитный диск. Сама программа-монитор в первых реализациях также хранилась на перфокартах или перфоленте, а в более поздних — на магнитной ленте и магнитных дисках.

В числе первых программ этого типа можно назвать монитор IBSYS, управляющий выполнением заданий на компьютере IBM 7090 (1960 год), и разработанный в Советском Союзе монитор для ЭВМ БЭСМ-4 (1962 год).

Ранние системы пакетной обработки значительно сократили затраты времени на вспомогательные действия по организации вычислительного процесса, а значит, был сделан еще один шаг в направлении повышения эффективности использования компьютеров. Однако при этом программисты-пользователи лишились непосредственного доступа к компьютеру, что снижало эффективность их работы — внесение любого исправления требовало значительно больше времени, чем при интерактивной работе за пультом машины.

Еще одним важным событием стала разработка механизма **виртуальной памяти**. При этом способе организации вычислительного процесса программе, выполняемой под управлением ОС, предоставляется виртуальное адресное пространство оперативной памяти очень большого размера, который превосходит размер реальной физической оперативной памяти компьютера. Адреса виртуального адресного пространства динамически отображаются операционной системой на все виды памяти, которая имеется у компьютера, то есть помимо оперативной памяти на дисковую память, память на магнитной ленте и т. п. Это позволяет значительно смягчить ограничения на объем программ. Первая ОС

с поддержкой виртуальной памяти была разработана в 1961 году компанией Burroughs для ее компьютера B5000. Позже эта функция стала для ОС стандартной.

Мультипрограммные операционные системы для мэйнфреймов

Следующий важный период развития операционных систем относится к 1965–1975 годам.

В это время в технической базе вычислительных машин произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам, что открыло путь к появлению следующего поколения компьютеров. Большие функциональные возможности интегральных схем сделали возможным реализацию на практике сложных компьютерных архитектур, таких, например, как IBM System/360.

В этот период были реализованы практически все основные механизмы, присущие современным ОС: мультипрограммирование, мультипроцессирование, поддержка многотерминального многопользовательского режима, виртуальная память, файловые системы, разграничение доступа и сетевая работа. В эти годы начинается расцвет системного программирования. Из направления прикладной математики, представляющего интерес лишь для узкого круга специалистов, системное программирование превращается в отрасль индустрии, оказывающую непосредственное влияние на практическую деятельность миллионов людей.

Революционным событием данного этапа явилась промышленная реализация мультипрограммирования. (Заметим, что в виде концепции и экспериментальных систем этот способ организации вычислений существовал уже около десяти лет.) В условиях резко возросших возможностей компьютера по обработке и хранению данных выполнение только одной программы в каждый момент времени оказалось крайне неэффективным. Решением стало **мультипрограммирование** — способ организации вычислительного процесса, при котором в памяти компьютера находилось одновременно несколько программ, попеременно выполняющихся на одном процессоре. Эти усовершенствования значительно повысили эффективность вычислительной системы: компьютер теперь мог использоваться почти постоянно, а не простаивать большую часть времени, как это было раньше.

Мультипрограммирование было реализовано в двух вариантах — в системах пакетной обработки и разделения времени.

Мультипрограммные системы пакетной обработки, так же как и их однопрограммные предшественники, имели своей целью обеспечение максимальной загрузки аппаратуры компьютера, однако решали эту задачу более эффективно. В мультипрограммном пакетном режиме процессор не простаивал, пока одна программа выполняла операцию ввода-вывода (как это происходило при последовательном выполнении программ в системах ранней пакетной обработки),

а переключался на другую готовую к выполнению программу. В результате достигалась сбалансированная загрузка всех устройств компьютера, а следовательно, увеличивалось число задач, решаемых в единицу времени.

Однако в мультипрограммных системах пакетной обработки пользователь по-прежнему был лишен возможности интерактивно взаимодействовать со своими программами. Для того чтобы хотя бы частично вернуть пользователям ощущение непосредственного взаимодействия с компьютером, был разработан другой вариант мультипрограммных систем — **системы разделение времени**. Этот вариант рассчитан на **многотерминальные системы**, когда каждый пользователь работает за своим терминалом.

В числе первых операционных систем разделения времени, разработанных в середине 60-х годов, были TSS/360 (компания IBM), CTSS и MULTICS (Массачусетсский технологический институт совместно с Bell Labs и компанией General Electric). В СССР первая развитая система разделения времени АИСТ-0 была создана в конце 60-х годов. Эта ОС предназначалась для работы на многомашинном комплексе, организованном из отечественных ЭВМ М-220 и Минск-22. В операционной системе АИСТ-0 были реализованы ставшие классическими сейчас принципы построения ОС, в частности иерархическая архитектура с четко выделенным ядром, а также новаторские для того времени идеи многопроцессорной обработки и разделения времени. Руководителем проекта АИСТ был академик Андрей Петрович Ершов, выдающийся программист и математик, лидер советского программирования.

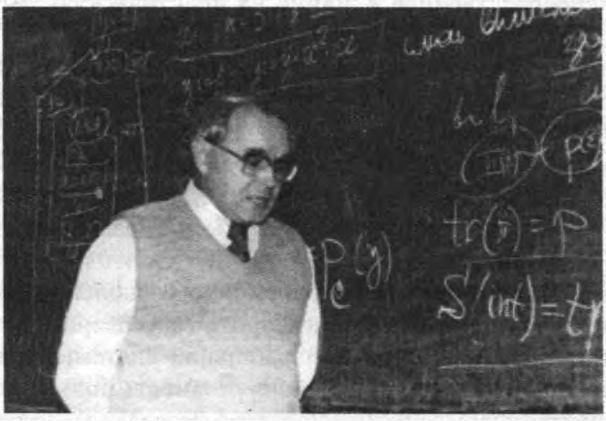


Рис. 1.1. Академик А. П. Ершов читает лекцию о смешанных вычислениях в Институте математики СО АН СССР

Вариант мультипрограммирования, применяемый в системах разделения времени, был нацелен на создание для каждого отдельного пользователя иллюзии единоличного владения вычислительной машиной за счет периодического выделения каждой программе своей доли процессорного времени. В системах разделения времени эффективность использования оборудования ниже, чем

в системах пакетной обработки, что является платой за удобства работы пользователя.

Многотерминальный режим использовался не только в системах разделения времени, но и в системах пакетной обработки. При этом и оператор, и все пользователи получали возможность формировать свои задания и управлять их выполнением со своего терминала. Такие операционные системы получили название **систем удаленного ввода заданий**. Терминальные комплексы могли располагаться на большом расстоянии от процессорных стоек, соединяясь с ними с помощью различных глобальных связей — модемных соединений телефонных сетей или выделенных каналов. Для поддержания удаленной работы терминалов в операционных системах появились специальные программные модули, реализующие различные (в то время, как правило, нестандартные) протоколы связи.

Вычислительные системы с удаленными терминалами, сохраняя централизованный характер обработки данных, в какой-то степени являлись прообразом современных сетей, а соответствующее системное программное обеспечение — прообразом **сетевых операционных систем**.

К этому времени можно констатировать существенное изменение в распределении функций между аппаратными и программными средствами компьютера. Операционные системы становились неотъемлемыми элементами компьютеров, играя роль «продолжения» аппаратуры. В первых вычислительных машинах программист, напрямую взаимодействуя с аппаратурой, мог выполнить загрузку программных кодов, используя пультовые переключатели и лампочки индикаторов, а затем вручную запустить программу на выполнение, нажав кнопку «пуск». В компьютерах 60-х годов большую часть действий по организации вычислительного процесса взяла на себя операционная система. (В большинстве современных компьютеров не предусмотрено даже теоретической возможности выполнения какой-либо вычислительной работы без участия операционной системы. После включения питания автоматически происходит поиск, загрузка и запуск операционной системы, а в случае ее отсутствия компьютер просто останавливается.)

Реализация мультипрограммирования потребовала внесения очень важных изменений в аппаратуру компьютера, непосредственно направленных на поддержку нового способа организации вычислительного процесса. При разделении ресурсов компьютера между программами необходимо обеспечить быстрое переключение процессора с одной программы на другую, а также надежно защитить коды и данные одной программы от непреднамеренной или преднамеренной порчи другой программой. В процессорах появился привилегированный и пользовательский режимы работы, специальные регистры для быстрого переключения с одной программы на другую, средства защиты областей памяти, а также развитая система прерываний.

В привилегированном режиме, предназначенном для работы программных модулей операционной системы, процессор мог выполнять все команды, в том числе и те из них, которые позволяли осуществлять распределение и защиту

ресурсов компьютера. Программам, работающим в пользовательском режиме, некоторые команды процессора были недоступны. Таким образом, только ОС могла управлять аппаратными средствами и исполнять роль монитора и арбитра для пользовательских программ, которые выполнялись в непривилегированном, или пользовательском, режиме. Система прерываний позволяла синхронизировать работу различных устройств компьютера, работающих параллельно и асинхронно, таких как каналы ввода-вывода, диски, принтеры и т. п. *Аппаратная поддержка операционных систем* стала с тех пор неотъемлемым свойством практически любых компьютерных систем, включая персональные компьютеры.

Многотерминальность выявила еще одну важную функцию, которую необходимо было делегировать ОС, — *защиту данных*. Так как в таких системах с компьютером одновременно общаются несколько пользователей, к тому же часто географически удаленных от компьютера, возникла необходимость контроля доступа пользователей к компьютеру (а точнее, к ОС, предоставляющей ресурсы компьютера пользователю). Для решения этой задачи были созданы системы **аутентификации**, которые проверяли легальность пользователей, как правило, на основе паролей. Кроме того, даже легальный пользователь, успешно прошедший аутентификацию, должен быть ограничен в доступе только к тем ресурсам, которые ему по тем или иным соображениям были выделены. Для реализации контролируемого доступа многотерминальные системы стали оснащать системами **авторизации**. В дальнейшем появление локальных и глобальных сетей только обострило задачу обеспечения безопасности компьютеров и обрабатываемых и хранимых ими данных.

Еще одной важной тенденцией этого периода является создание *семейств программно-совместимых машин* и *операционных систем* для них. Примерами семейств программно-совместимых машин, построенных на интегральных микросхемах, являются серии машин IBM System/360 и System/370 (аналоги этих семейств советского производства — машины серии ЕС), PDP-11 (советские аналоги — СМ-3, СМ-4, СМ-1420). Вскоре идея программно-совместимых машин стала общепризнанной.

Программная совместимость требовала и совместимости операционных систем. Однако такая совместимость подразумевает возможность работы на больших и на малых вычислительных системах, с большим и малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением удовлетворить всем этим противоречивым требованиям, оказались чрезвычайно сложными. Они состояли из многих миллионов ассемблерных строк, написанных тысячами программистов, содержали тысячи ошибок, вызывающих нескончаемый поток исправлений. Так, объем кода системы OS/360, разработанной для серии компьютеров IBM System/360, составил 8 Мбайт, что для того времени было абсолютным рекордом.

Однако несмотря на необозримые размеры и множество проблем, OS/360 и другие ей подобные операционные системы этого поколения действительно

удовлетворяли большинству требований потребителей. За это десятилетие был сделан огромный шаг вперед и заложен прочный фундамент для создания современных операционных систем.

Первые сетевые операционные системы

В начале 70-х годов появились первые **сетевые операционные системы**, которые в отличие от многотерминальных ОС позволяли не только рассредоточить пользователей, но и организовать распределенное хранение и обработку данных между несколькими компьютерами, связанными электрическими связями. Любая сетевая операционная система, с одной стороны, выполняет все функции локальной операционной системы, а с другой стороны, обладает некоторыми дополнительными средствами, позволяющими ей взаимодействовать по сети с операционными системами других компьютеров. Программные модули, реализующие сетевые функции, появлялись в операционных системах постепенно, по мере развития сетевых технологий, аппаратной базы компьютеров и возникновения новых задач, требующих сетевой обработки.

Хотя теоретические работы по созданию концепций сетевого взаимодействия велись почти с самого появления вычислительных машин, значимые практические результаты по объединению компьютеров в сети были получены в конце 60-х, когда с помощью глобальных связей и техники коммутации пакетов удалось реализовать взаимодействие машин класса мэйнфреймов и суперкомпьютеров. Эти дорогостоящие компьютеры часто хранили уникальные данные и программы, доступ к которым необходимо было обеспечить широкому кругу пользователей, находившихся в различных городах на значительном расстоянии от вычислительных центров.

В 1969 году министерство обороны США инициировало работы по объединению суперкомпьютеров оборонных и научно-исследовательских центров в единую сеть. Эта сеть получила название ARPANET и явилась отправной точкой для создания всемирной компьютерной сети — *Интернета*. На рис. 1.2 показан один из самых первых набросков структуры этой сети, состоящей всего из 4 узлов.

Компьютеры сети ARPANET работали под управлением различных ОС, которые были дополнены модулями, реализующими коммуникационные протоколы, общие для всех компьютеров сети.

> Операционные системы, установленные на компьютерах сети ARPANET, в совокупности с дополнительными модулями, обеспечивающими взаимодействие узлов сети, явились **первыми сетевыми ОС**, имеющими практическое значение.

В 1974 году компания IBM объявила о создании собственной сетевой архитектуры для своих мэйнфреймов, получившей название SNA (System Network Architecture). Эта многоуровневая архитектура, во многом подобная стандартной модели OSI, появившейся несколько позже, обеспечивала взаимодействие

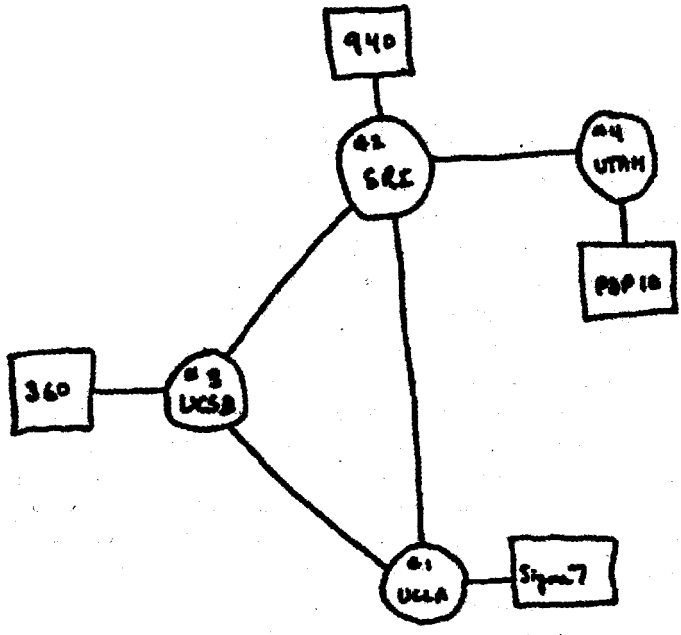


Рис. 1.2. Большое начинается с малого — один из ранних эскизов структуры ARPANET

по глобальным связям типа «терминал-терминал», «терминал-компьютер» и «компьютер-компьютер». Нижние уровни архитектуры были реализованы специализированными аппаратными средствами, наиболее важным из которых являлся процессор телеобработки. Функции верхних уровней SNA выполнялись программными модулями. Один из них составлял основу программного обеспечения процессора телеобработки. Другие модули работали на центральном процессоре в составе стандартной операционной системы IBM для мэйнфреймов.

В это же время в Европе велись активные работы по созданию и стандартизации сетей X.25. Эти сети с коммутацией пакетов не были привязаны к какой-либо конкретной операционной системе. После получения статуса международного стандарта в 1974 году протоколы X.25 стали поддерживаться многими операционными системами. С 1980 компания IBM включила поддержку протоколов X.25 в архитектуру SNA и в свои операционные системы.

Операционные системы миникомпьютеров и первые локальные сети

К середине 70-х годов наряду с мэйнфреймами широкое распространение получили **миникомпьютеры**, такие как PDP-11, Nova, HP. В миникомпьютерах впервые использовались преимущества больших интегральных схем, позволив-

шие реализовать достаточно мощные функции при сравнительно невысокой стоимости компьютера.

Архитектура миникомпьютеров была значительно упрощена по сравнению с мэйнфреймами, что нашло отражение и в их операционных системах. Многие функции мультипрограммных многопользовательских ОС мэйнфреймов были усечены, учитывая ограниченность ресурсов миникомпьютеров. Операционные системы миникомпьютеров часто стали делать специализированными, например, только для управления в реальном времени (ОС RT-11 для миникомпьютеров PDP-11) или только для поддержания режима разделения времени (RSX-11M для тех же компьютеров). Эти операционные системы не всегда были многопользовательскими, что во многих случаях оправдывалось невысокой стоимостью компьютеров.

Важной вехой в истории миникомпьютеров и вообще в истории операционных систем явилось создание *ОС Unix*. Первоначально эта ОС предназначалась для поддержания режима разделения времени в миникомпьютере PDP-7. С середины 70-х годов началось массовое использование ОС Unix. К тому времени программный код для Unix был на 90 % написан на языке высокого уровня С. Широкое распространение эффективных С-компиляторов сделало Unix уникальной для того времени ОС, обладающей возможностью сравнительно легкого переноса на различные типы компьютеров: суперкомпьютеры, мэйнфреймы, миникомпьютеры, серверы и рабочие станции на базе RISC-процессоров, персональные компьютеры. Поскольку эта ОС поставлялась вместе с исходными кодами, то она стала первой открытой ОС, которую могли совершенствовать простые пользователи-энтузиасты.

> Гибкость, элегантность, мощные функциональные возможности и открытость позволили операционной системе Unix занять прочные позиции во всех классах компьютеров.

Доступность миникомпьютеров и, вследствие этого, их распространенность на предприятиях, послужила мощным стимулом для создания **локальных сетей**. Предприятие могло себе позволить иметь несколько миникомпьютеров, находящихся в одном здании или даже в одной комнате. Естественно возникала потребность в обмене информацией между ними и в совместном использовании дорогого периферийного оборудования.

Первые локальные сети строились с помощью нестандартного коммуникационного оборудования, в простейшем случае — путем прямого соединения последовательных портов компьютеров. Программное обеспечение также было нестандартным и реализовывалось в виде пользовательских приложений. Первое сетевое приложение для ОС Unix — программа UUCP (Unix-to-Unix Copy Program) появилась в 1976 году и начала распространяться с версией 7 AT&T Unix с 1978 года. Эта программа позволяла копировать файлы с одного компьютера на другой в пределах локальной сети через различные аппаратные интерфейсы — RS-232, токовую петлю и т. п., а, кроме того, могла работать через глобальные связи, например, модемные.

Развитие операционных систем в 80-е годы

К наиболее важным событиям 80-х годов можно отнести создание стека TCP/IP, становление Интернета, разработку новых версий ОС Unix, стандартизацию технологий локальных сетей, появление персональных компьютеров и операционных систем для них.

Рабочий вариант **стека протоколов TCP/IP** был создан в конце 70-х годов. Этот стек представлял собой набор общих протоколов для разнородной вычислительной среды и предназначался для связи экспериментальной сети ARPANET с другими «сателлитными» сетями. В 1983 году стек протоколов TCP/IP был принят министерством обороны США в качестве военного стандарта. Переход компьютеров сети ARPANET на стек TCP/IP ускорила его реализация для операционной системы BSD Unix. С этого времени началось совместное существование Unix и протоколов TCP/IP и практически все многочисленные версии Unix стали сетевыми.

Внедрение протоколов TCP/IP в ARPANET придало этой сети все основные черты, которые отличают современный Интернет. В 1983 году сеть ARPANET была разделена на две части: MILNET, поддерживающую военные ведомства США, и новую ARPANET. Для обозначения составной сети ARPANET и MILNET стало использоваться название **Internet**, которое в русском языке превратилось в **Интернет**. Интернет стал отличным полигоном для испытаний многих сетевых операционных систем, позволившим проверить в реальных условиях возможности их взаимодействия, степень масштабируемости, способность работы при экстремальной нагрузке, создаваемой сотнями и тысячами пользователей. Стек протоколов TCP/IP также ждала завидная судьба. Независимость от производителей, гибкость и эффективность, доказанные успешной работой в Интернете, а также открытость и доступность стандартов сделали протоколы TCP/IP не только главным транспортным механизмом Интернета, но и основным стеком большинства сетевых операционных систем.

Все десятилетие было отмечено появлением новых, все более совершенных версий ОС Unix. Среди них были и фирменные версии Unix: SunOS, HP-UX, Irix, AIX, QNX и многие другие, в которых производители компьютеров адаптировали код ядра и системных утилит для своей аппаратуры. Разнообразие версий породило проблему их совместимости, которую периодически пытались решить различные организации. В результате были приняты стандарты POSIX и XPG, определяющие интерфейсы ОС для приложений, а специальное подразделение компании AT&T выпустило несколько версий Unix System III и Unix System V, призванных консолидировать разработчиков на уровне кода ядра.

Начало 80-х годов связано с еще одним знаменательным для истории операционных систем событием — появлением **персональных компьютеров**. С точки зрения архитектуры персональные компьютеры ничем не отличались от класса миникомпьютеров типа PDP-11, но их стоимость была существенно ниже. Если миникомпьютер позволял иметь собственную вычислительную машину отделу предприятия или университету, то персональный компьютер дал такую воз-

можность отдельному человеку. Компьютеры стали широко использоваться неспециалистами, что потребовало разработки «дружественного» программного обеспечения, и предоставление этих «дружественных» функций стало прямой обязанностью операционных систем.

Персональные компьютеры послужили также мощным катализатором для бурного роста локальных сетей, создав для этого отличную материальную основу в виде десятков и сотен компьютеров, принадлежащих одному предприятию и расположенных в пределах одного здания. В результате поддержка сетевых функций стала для ОС персональных компьютеров необходимым условием.

Однако и дружественный интерфейс, и сетевые функции появились у операционных систем персональных компьютеров не сразу. Первая версия наиболее популярной операционной системы раннего этапа развития персональных компьютеров — MS-DOS компании Microsoft (1981 год) — была лишена этих возможностей. Это была однопрограммная однопользовательская ОС с интерфейсом командной строки, способная стартовать с дискеты. Основными задачами для нее были управление файлами, расположенными на гибких и жестких дисках в Unix-подобной иерархической файловой системе, а также поочередный запуск программ. MS-DOS не была защищена от программ пользователя, так как процессор Intel 8088 не поддерживал привилегированного режима. Разработчики первых персональных компьютеров считали, что при индивидуальном использовании компьютера и ограниченных возможностях аппаратуры нет смысла в поддержке мультипрограммирования, поэтому в процессоре не были предусмотрены привилегированный режим и другие механизмы поддержки мультипрограммных систем. Однако довольно скоро ПК стали возвращать себе многое из утраченной было функциональности и добавлять новые свойства.

Ориентация на массового, а значит, непрофессионального пользователя резко повысила интерес к *интуитивно понятному пользовательскому интерфейсу*, который ранее не относился к приоритетным свойствам ОС.

Важным событием для этого направления стало представление компанией Apple в начале 1984 года новой операционной системы Mac OS для своего персонального компьютера Macintosh. Эта ОС резко отличалась ото всех существовавших тогда ОС тем, что для взаимодействия с пользователем она использовала не обычный для того времени интерфейс командной строки, а значительно более удобный графический пользовательский интерфейс, включающий столь знакомые всем нам теперь окна, меню, значки (ярлыки) файлов и программ (рис. 1.3). Это революционное изменение позволило пользователю управлять компьютером с помощью простого и удобного нового устройства — мыши, а не набирать команды на клавиатуре[1].

[1] Историческая справедливость требует заметить, что первые успешные попытки реализации графического интерфейса состоялись значительно раньше. Так изобретение мыши — устройства столь необходимого для удобного взаимодействия с компьютером — относится к ранним 60-м, а полноценный графический интерфейс был представлен корпорацией Xerox еще в 1973 году.

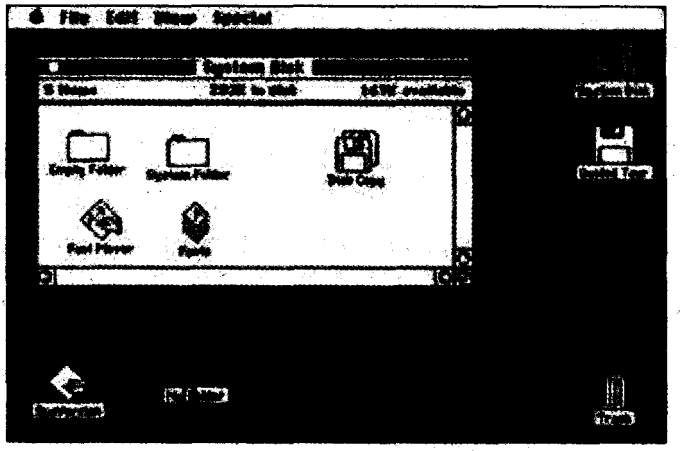


Рис. 1.3. Вид рабочего экрана первой версии Mac OS

Компания Microsoft в своих ОС Windows также сделала ставку на использование оконного графического интерфейса. Первая версия операционной системы Windows компании Microsoft появилась в 1985 году и осталась практически незамеченной, но начиная с версии 3.0, это семейство ОС оказало сильнейшее влияние на развитие индустрии персональных компьютеров. Наиболее серьезные изменения в последней версии Microsoft Windows Vista, ставшей доступной для корпоративных пользователей в конце 2006 года (на момент написания этих строк индивидуальные пользователи все еще ждали своей версии этой системы), также произошли в области пользовательского интерфейса.

Что же касается возвращения утраченных свойств, то, как уже было сказано ранее, после появления персональных компьютеров первое время казалось, что концепция многозадачности, когда на одном компьютере одновременно выполняется несколько задач, исчерпала себя. Действительно, если каждому пользователю теперь выделяется целиком компьютер, то зачем нагружать операционную систему громоздкими функциями по разделению ресурсов между несколькими программами, запускаемыми разными пользователями? Однако очень скоро наступило осознание того, что даже при автономном использовании ПК (то есть без подключения его к сети) по-прежнему актуальна многозадачность, только теперь ресурсы компьютера должны разделяться не между программами разных пользователей, а между несколькими программами одного и того же пользователя. Более того, в ОС персональных компьютеров оказались востребованными и функции многопользовательской защиты для тех случаев, когда ПК поочередно используется несколькими людьми.

В соответствии с этой стратегией в 1987 году в результате совместных усилий Microsoft и IBM появилась первая *мультипрограммная* операционная система *для персональных компьютеров* с процессором Intel 80286, в полной мере использующая возможности защищенного режима — OS/2. Эта ОС с ее развитыми функциями многозадачности и файловой системой, снабженной встроенными

средствами многопользовательской защиты, оказалась хорошей платформой для построения локальных сетей персональных компьютеров. В семействе ОС Windows многозадачность тоже была введена почти с первых версий. Причем долгое время существовали две реализации многозадачности. Первый вариант был характерен для тех версий Windows, которые фактически представляли собой надстройку (оболочку) над MS-DOS (то есть версии 1.0, 2.0, 3.0, 3.1, Windows 95/98/ME). Эта многозадачность была невытесняющей, то есть ОС не могла прервать выполняемую задачу, пока эта задача по своей инициативе не передавала управление операционной системе. Это ограничение снижало эффективность ОС. По-другому была решена проблема многозадачности в автономной линии Windows, начатой Windows NT и продолженной Windows 2000, Windows XP Professional, Windows Server 2003 и Vista (рис. 1.4). В этих ОС, для которых в этой книге используется обобщенное наименование «ОС семейства Windows NT», поддерживается эффективная вытесняющая многозадачность, полноценный многопользовательский режим, а также многопоточный режим, когда каждая задача может быть распараллелена на несколько независимо выполняющихся подзадач.

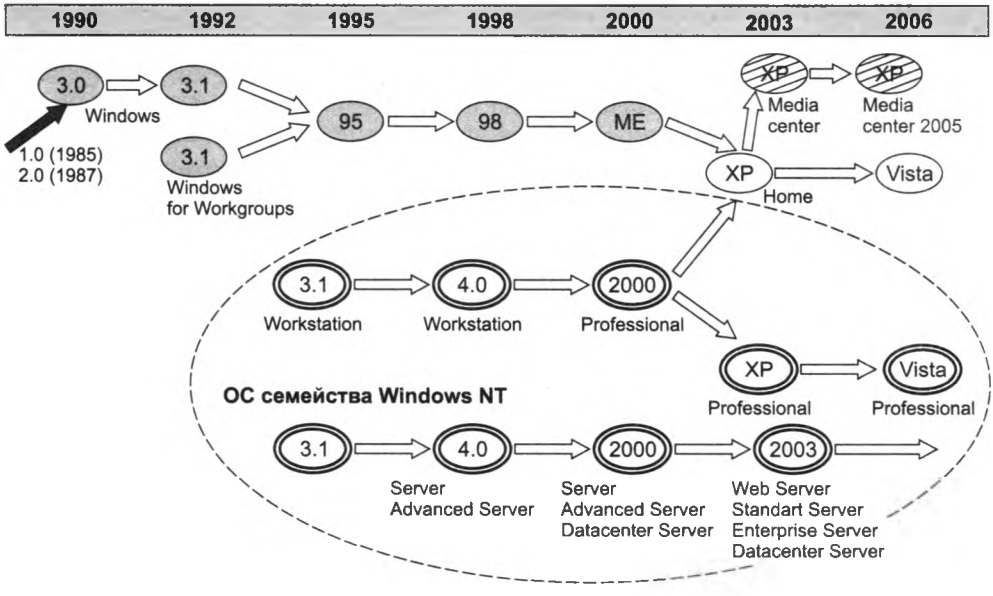


Рис. 1.4. Операционные системы Windows компании Microsoft. Хронологическая схема

Поддержка многопользовательского и многозадачного режимов стала в современных ОС для ПК стандартом, этими свойствами обладают и две другие популярные ОС, работающие на ПК, — Mac OS X и Linux/Fedora.

Еще одним мощным фактором возвращения в ОС ПК многозадачности и многопользовательского режима стали локальные сети. Действительно, если компьютер подключен к локальной сети, то к его ресурсам (файлам, непосред-

ственно подключенному принтеру и т. д.) может обращаться не только тот пользователь, который работает за клавиатурой, но и все остальные пользователи сети — если, конечно, эти ресурсы сделаны разделяемыми. Не способные выполнить эту функцию первые ОС персональных компьютеров, такие, например, как MS-DOS, дополнялись внешними программами, называемыми сетевыми оболочками. В простейшем случае они просто позволяли нескольким пользователям работать с одним файлом за счет дополнительных средств блокировки доступа. В MS-DOS такая блокировка была введена, начиная с версии 3.0 (это не означает, что MS-DOS 3.0 обладала встроенными сетевыми функциями, блокировка файлов была только необходимым условием для работы сетевых оболочек сторонних производителей или самой Microsoft).

Наибольшее распространение получили сетевые оболочки LAN Manager компании Microsoft и LAN Server компании IBM, разработанные этими компаниями на основе одного базового кода. Эти оболочки уступали по производительности специализированной ОС NetWare и потребляли больше аппаратных ресурсов, но имели важные достоинства универсальной ОС — они позволяли, во-первых, выполнять на сервере любые программы, разработанные для OS/2, MS-DOS и Windows, а во-вторых, использовать компьютер, на котором они были запущены, в качестве рабочей станции. Не очень удачная рыночная судьба OS/2 не позволила системам LAN Manager и LAN Server захватить заметную долю рынка, но принципы работы этих сетевых систем во многом нашли свое воплощение в более удачливой операционной системе 90-х годов: Microsoft Windows NT, содержащей встроенные сетевые компоненты, некоторые из которых имеют приставку LM — от LAN Manager.

Иной путь выбрала компания Novell. Она изначально сделала ставку на разработку операционной системы со встроенными сетевыми функциями и добилась на этом пути выдающихся успехов. Ее сетевые операционные системы NetWare на долгое время стали эталоном производительности, надежности и защищенности для локальных сетей. С самой первой версии ОС NetWare (1983 г.) распространялась как операционная система для центрального сервера локальной сети, которая за счет специализации на выполнении функций файл-сервера обеспечивала максимально возможную для данного класса компьютеров скорость удаленного доступа к файлам и повышенную безопасность данных.

В 80-е годы были приняты основные *стандарты на коммуникационные технологии для локальных сетей*: в 1980 году — Ethernet, в 1985 — Token Ring, в конце 80-х — FDDI. Это позволило обеспечить совместимость сетевых операционных систем на нижних уровнях, а также стандартизовать интерфейс ОС с драйверами сетевых адаптеров.

Для персональных компьютеров применялись не только специально разработанные для них операционные системы, подобные MS-DOS, NetWare и OS/2, но и адаптировались уже существующие ОС. Появление процессоров Intel 80286 и особенно 80386 с поддержкой мультипрограммирования позволило перенести на платформу персональных компьютеров ОС Unix. Наиболее извест-

ной системой этого типа в 1980-е годы была версия Unix компании Santa Cruz Operation (SCO Unix).

Появление высокоскоростных и надежных локальных сетей дало мощный импульс развитию распределенных вычислительных систем. Наибольшую популярность завоевали **системы клиент-сервер**, в которых сервер чаще всего выполнял функции базы данных, а клиент предоставлял пользователю удобный графический интерфейс и выполнял обработку данных, которые были найдены сервером и переданы клиенту по сети.

Развитие операционных систем в 90-е годы

В 90-е годы практически все операционные системы, занимающие заметное место на рынке, стали *сетевыми*. Сетевые функции сегодня встраиваются в ядро ОС, являясь ее неотъемлемой частью. Операционные системы получили средства для работы со всеми основными технологиями локальных (Ethernet, Fast Ethernet, Gigabit Ethernet, Token Ring, FDDI, ATM) и глобальных (X.25, frame relay, ISDN, ATM) сетей, а также средства для создания составных сетей (IP, IPX, AppleTalk, RIP, OSPF, NLSP). В операционных системах используются инструменты мультиплексирования нескольких стеков протоколов, за счет чего компьютеры могут поддерживать одновременную сетевую работу с разнородными клиентами и серверами. Появились специализированные ОС, предназначенные исключительно для решения коммуникационных задач. Например, сетевая операционная система IOS компании Cisco Systems, работающая в маршрутизаторах, организует в мультипрограммном режиме выполнение набора программ, каждая из которых реализует один из коммуникационных протоколов.

Во второй половине 90-х годов все производители операционных систем резко усилили поддержку средств работы с *Интернетом*. Если до этого времени стек протоколов TCP/IP, на котором построен Интернет, поддерживался в основном семейством Unix, то теперь этот стек стал проникать во все популярные ОС. Помимо самого стека TCP/IP, в комплект поставки начали включать утилиты, реализующие такие популярные сервисы Интернета, как telnet, FTP, e-mail. Влияние Интернета проявилось и в том, что компьютер превратился из чисто вычислительного устройства в средство коммуникаций с развитыми вычислительными возможностями. Появление службы World Wide Web (WWW) в 1991 году придало мощный импульс развитию популярности Интернета, так как позволило пользователям удобно, эффективно и единообразно искать и просматривать текстовую и графическую информацию, расположенную на многочисленных серверах, работающих под управлением различных ОС. Интернет приобрел после этого черты гигантской справочной системы и библиотеки, где можно найти все и чаще всего — бесплатно.

Интернет принес не только совершенно новые фантастические возможности как всемирная энциклопедия и средство мультимедийного общения, но и совершенно новые угрозы компьютерной безопасности. Среди миллионов пользователей Интернета есть разные люди, в том числе и большое количество

хакеров, которые корыстно или бескорыстно пытаются взломать средства защиты ОС различными остроумными способами, включая коллективные атаки.

Проблема безопасности в век Интернета вышла на качественно новый уровень, и с этого момента *средства безопасности* приобрели самое приоритетное значение для разработчиков и пользователей ОС.

Особое внимание в течение всех 90-х годов уделялось *корпоративным сетевым операционным системам*. Их дальнейшее развитие представляет одну из наиболее важных задач и в обозримом будущем. Корпоративная операционная система отличается способностью хорошо и устойчиво работать в крупных сетях, которые характерны для больших предприятий, имеющих отделения в десятках городов и, возможно, в разных странах. Таким сетям органически присуща высокая степень неоднородности программных и аппаратных средств, поэтому корпоративная ОС должна беспроблемно взаимодействовать с операционными системами разных типов и работать на различных аппаратных платформах. Лидерами этого периода в классе корпоративных ОС были Novell NetWare 4.x и 5.0, Microsoft Windows NT 4.0, а также Unix-системы различных производителей аппаратных платформ. Для корпоративной ОС очень важно наличие средств централизованного администрирования и управления, позволяющих в единой базе данных хранить учетные записи о десятках тысяч пользователей, компьютеров, коммуникационных устройств и модулей программного обеспечения, имеющихся в корпоративной сети. В современных операционных системах средства централизованного администрирования обычно базируются на единой **справочной службе**. Первой успешной реализацией справочной службы корпоративного масштаба была система StreetTalk компании Banyan. Наибольшее признание в это время получила справочная служба NDS компании Novell, выпущенная впервые в 1993 году для первой корпоративной версии NetWare 4.0. Роль централизованной справочной службы настолько велика, что именно по качеству справочной службы оценивают пригодность операционной системы для работы в корпоративном масштабе.

После некоторого «забвения» снова вышли на первый план мощные *корпоративные системы на базе мэйнфреймов*.

В конце 80-х годов в связи с популярностью локальных сетей и распределенных вычислений по схеме клиент-сервер интерес к мэйнфреймам упал. Многие стали рассматривать их как тупиковую ветку из-за того, что появилась возможность объединить вычислительные мощности десятка ПК и получить в результате более производительную и менее дорогую систему, чем мэйнфрейм.

Однако вскоре выяснилось, что это не всегда и не совсем так. Простое суммирование вычислительных мощностей отдельных компьютеров не всегда отражает их возможности по выполнению программ. Так, это может быть справедливо для достаточно простых задач поиска информации в базе данных, когда клиентские части никак логически между собой не связаны, поскольку

обслуживают различных пользователей. В то же время сложные вычислительные алгоритмы, обладающие тесными логическими связями между отдельными ветками вычислительного процесса, распараллеливаются совсем не так эффективно, в результате суммарная мощность распределенной вычислительной системы оказывается намного меньше суммы мощностей входящих в нее отдельных компьютеров. К тому же накладные расходы на координирование работы отдельных компьютеров в сети при решении комплекса тесно взаимосвязанных задач оказываются часто очень высокими, делая такую организацию неэффективной или даже практически нереализуемой. Это, а также то, что в 90-е годы мэйнфреймы резко подешевели из-за прогресса в технологиях для их элементной базы, возродило интерес к мэйнфреймам.

Возобновилось и приостановившееся было развитие ОС мэйнфреймов. Ярким представителем этого класса является появившаяся в 1995 году OS/390 компании IBM для мэйнфреймов System/370 и System/390. Эта ОС по существу является развитием популярной в середине 70-х годов системы пакетной обработки ОС MVS, которая, в свою очередь, ведет историю от одной из первых мультипрограммных систем OS/360 (1964 г.). Помимо выполнения базовых функций операционной системы, OS/390 включает средства сетевого взаимодействия с пользователями и устройствами в разнородной вычислительной среде на базе протоколов SNA и TCP/IP. OS/390 поддерживает стандарты системных вызовов Unix, что является отражением общей тенденции в мире мэйнфреймов.

Современный этап развития операционных систем персональных компьютеров

Следует признать, что процесс революционных изменений в области архитектурных и функциональных решений ОС затормозился, и сегодня мы видим в основном плавную эволюцию тех свойств, механизмов и функций ОС, которые появились в 60-е и 90-е годы. Иллюстрацией этого тезиса, в частности, является новая версия семейства ОС Windows Vista, на разработку которой корпорация Microsoft потратила 5 лет (больше, чем на разработку любой другой версии Windows). Несмотря на большой объем затраченных на ее создание усилий, Vista не показала принципиально новых архитектурных решений и функциональных возможностей, чем вызвала разочарование некоторых специалистов.

И все же в мире ОС персональных компьютеров имеется одна четко выраженная долговременная тенденция развития — *повышение удобства работы человека с компьютером*. Эффективность работы человека становится основным фактором, определяющим эффективность вычислительной системы в целом. Идеальной целью является приближение ПК по уровню надежности эксплуатации и удобства в обращении к бытовым приборам.

Надежность

Компьютер должен работать так же надежно, как телевизор, телефон, электрический чайник, наконец. Телевизор не позволяет себе выводить на экран грозные сообщения «Произошла серьезная ошибка. Некоторые данные могут быть потеряны. Требуется перезагрузка», — а после этого утешительно вопрошать: «Желаете ли вы послать отчет об ошибке в компанию Sony?»

Надежность ОС можно обеспечить различными способами, один из наиболее очевидных — добиться высокой степени отлаженности кода ядра. Существует также возможность повысить надежность за счет архитектурных решений. В частности, **микроядерная архитектура** снова стала обсуждаться как стратегически важное направление. Идея состоит в том, чтобы сократить объем кода, работающего в привилегированном режиме ядра; это позволит снизить вероятность ошибок, которые могут привести к краху системы. Все остальные функции ОС (файловая система, драйверы) реализуются в виде приложений. Если ошибка возникает в одном из приложений, то аварийно завершается только это приложение, и система продолжает функционировать. Однако этот подход приводит к снижению производительности, поэтому микроядерные ОС по большей части оставались уделом учебных ОС и исследовательских проектов. Возможно, возросшие требования к надежности ОС и постоянный рост быстродействия процессоров и объемов оперативной памяти изменят эту ситуацию и заставят разработчиков коммерческих ОС перейти преимущественно к архитектуре на базе микроядра.

В своей последней ОС Windows Vista компания Microsoft предприняла усилия по повышению надежности Windows, в частности за счет встраивания механизма транзакций в файловую систему, которая хранит несколько предыдущих (теневых) версий диска, всех его каталогов и файлов, так что возможен гибкий «откат» после краха всей файловой системы или ее части. Можно, конечно, считать эту новую функцию и архитектурным новшеством, но скорее это все-таки перенос в операционную систему некоторых функций приложений, в данном случае — баз данных.

Простота обслуживания

Пользователь компьютера желает получать от него услуги, не расплачиваясь за это услугами по его обслуживанию (администрированию). Усилия человека не должны тратиться на настройку параметров вычислительного процесса, как это происходило в ОС предыдущих поколений. Например, в системах пакетной обработки для мэйнфреймов каждый пользователь должен был с помощью языка управления заданиями задать большое количество параметров, относящихся к организации вычислительных процессов в компьютере. В частности, в OS/360 язык управления заданиями (Job Command Language, JCL) предусматривал возможность определения пользователем более 40 параметров, среди которых были приоритет задания, требования к основной памяти, предельное время выполнения задания, перечень используемых устройств ввода-вывода и режимы их работы.

Современная операционная система берет решение задачи выбора параметров операционной среды на себя, используя для этой цели различные адаптивные алгоритмы. Например, тайм-ауты в коммуникационных протоколах часто определяются в зависимости от условий работы сети. Распределение оперативной памяти между процессами осуществляется автоматически с помощью механизмов виртуальной памяти в зависимости от активности этих процессов и информации о частоте использования ими той или иной страницы. Мгновенные приоритеты процессов определяются динамически в зависимости от предыстории, включающей, например, время нахождения процесса в очереди, процент использования выделенного кванта времени, интенсивность ввода-вывода и т. п.

Даже в процессе установки большинство ОС предлагают режим выбора параметров по умолчанию, который гарантирует пусть не оптимальное, но всегда приемлемое качество работы систем. Такие ОС, как Windows, Linux и Mac OS X, прошли уже большой путь в этом направлении. Однако настройка параметров ОС по умолчанию раздражает некоторых профессиональных пользователей, лишая их возможности проделывать эти операции вручную и так, как они это хотят. Здесь уместна аналогия с автоматической коробкой передач — водителям-новичкам она нравится, так как освобождает от необходимости постоянно следить за тем, на какой передаче они едут. В то же время некоторые опытные водители предпочитают иметь более полный контроль над автомобилем, который дает ручная коробка передач. Поэтому желательно, чтобы ОС ПК учитывала интересы обеих категорий пользователей.

Для массовых непрофессиональных пользователей такая ОС должна быть незаметной: она должна решать все задачи по установке новых программ и оборудования без малейшего вовлечения пользователя в этот процесс. Этого, к сожалению, пока не происходит, несмотря на прогресс стандарта Plug&Play и развитие интеллектуальных инсталляторов программного обеспечения. Все обычно идет хорошо до первой непредвиденной ситуации, и тут на пользователя обрушивается шквал вопросов «растерянной» ОС, требующей от него знания специальных терминов или структуры системных каталогов. Еще хуже обстоят дела, когда ОС вдруг перестает нормально стартовать — здесь рядовому пользователю никаких инструкций, как «оживить» систему, не дается, кроме рекомендации вызвать системного администратора, который в комплект стандартной поставки, к сожалению, не входит.

В то же время для профессионалов ОС должна предоставлять все возможности по своей тонкой настройке, позволяя легко обходить режим работы с массовым пользователем и изменять параметры, которые заданы по умолчанию или выбраны ОС автоматически.

Пользовательский интерфейс

На эффективность работы пользователя огромное влияние оказывает то, насколько удобные средства предоставляет ему ОС для взаимодействия с компьютером. Появление графического интерфейса, управляемого мышью, было прорывом в этой области, так как избавило массовых пользователей от необходимости

запоминать текстовые команды, состоящие из не всегда осмысленных слов и еще более непонятных символов-ключей. Тем не менее некоторые приверженцы Unix по-прежнему предпочитают использовать во многих случаях командную строку, а не графический интерфейс, — и такой способ общения в некоторых случаях действительно намного быстрее приводит к желаемому результату, которого иногда с помощью оконного интерфейса просто невозможно добиться. Поэтому хорошая ОС должна поддерживать оба способа работы. Эта тенденция была отражена при разработке Microsoft Vista, которая по замыслу разработчиков должна была включать Windows PowerShell, мощный командный язык, устраняющий многолетнее отставание Windows от семейства Unix, с самого рождения имевшего развитый командный язык. Компания Microsoft не успела доработать PowerShell до необходимого уровня к моменту выпуска Windows Vista, но он будет поставляться как отдельный продукт (или же войдет в дистрибутив Vista несколько позже), так что этот пробел Windows будет устранен.

Что же касается оконного графического интерфейса, то прогресс в последние годы в этой области был не такой заметный, как этого хотелось бы пользователям. Примеры улучшений, сделанных в Windows Vista, хорошо это показывают: окна стали перемещаться по экрану быстрее и без искажений, их можно показывать на рабочем столе в виде трехмерного стека, но это, пожалуй, и все. Так что работы в этом направлении много, в частности в области распознавания речи, использования сенсорного экрана монитора, а может быть и открытия каких-нибудь принципиально новых средств общения пользователя с компьютером, каким в свое время стал оконный графический интерфейс.

Средства информационной самоорганизации

Кто из нас не терял фотографии, письма или статьи в «дебрях» своего компьютера? Работа по информационной «уборке и чистке» компьютера может занимать не меньше времени, чем реальная генеральная уборка жилища. Очевидно, что интеллектуальная мощь компьютера должна прийти на помощь человеку, помочь ему сохранять порядок в информационном хозяйстве. Современные диски, хранящие сотни гигабайтов информации, сделали проблему организации и поиска файлов очень острой. Несмотря на то что системы управления базами данных существуют много лет, они пока практически не используются для упорядочивания всей массы разнородной информации, хранящейся в файлах различных форматов в пользовательских каталогах персональных компьютеров. Конечно, очень хорошо было бы, если бы можно было спросить у ОС своего ПК: «А где эта статья о собаках, что я нашел пару недель назад на сайте какого-то университета? Я еще тогда несколько фото выкачал с этого сайта, неплохо бы их тоже найти» — и получить через несколько секунд и статью, и фото, которые случайно попали в разные каталоги и под странными именами, не дающими никакого намека на их содержание.

Для того чтобы начать движение в этом направлении, компания Microsoft планировала включить в версию Vista файловую систему Windows Future Storage (WinFS) со встроенной поддержкой SQL. Однако этим амбициозным

планам не суждено было сбыться в положенный срок, так что и эту новую функцию пришлось удалить из этой новой версии Windows, выделив ее в отдельный продукт. Тем не менее Microsoft реализовала кое-что из этой области в Windows Vista: включила в файловую систему метаданные, на основе которых работает новая версия быстрого поиска файлов по мета-атрибутам, причем файлы ищутся везде, в том числе и среди архивов электронных писем (проблема, правда, остается с присвоением значений метаданным файла — если вы не сделали этого на этапе создания или сохранения файла, то и найти файл по этим признакам будет невозможно). В Microsoft Vista появилась еще одна удобная функция — представление файлов в виде виртуальных папок. Для того чтобы сгруппировать файлы по какому-то признаку независимо от папок их физического хранения, необходимо просто произвести поиск файлов по какому-то критерию и объявить результаты поиска виртуальным каталогом. Далее с этим каталогом можно работать так же, как с обычным.

Защита данных

На современном этапе развития операционных систем на передний план вышли средства обеспечения безопасности. Это связано с возросшей ценностью информации, обрабатываемой компьютерами, а также с повышенным уровнем угроз, существующих при передаче данных через Интернет. Разнообразные атаки, вирусы, троянские кони и невероятно увеличившееся количество спама серьезно мешают эффективной работе на компьютере, подключенном к Интернету. Интернет был создан как открытая среда для общения исследователей, и с приходом службы Web его открытость послужила причиной его огромной популярности для массового пользователя. Но сегодня эта открытость одновременно является и источником многочисленных помех в работе, так как каждый хакер в Интернете может атаковать ваш компьютер, рассылая вирусы или просто бомбардируя вас ненужными письмами. В результате значительная часть ресурсов ОС направлена сегодня на защиту компьютера от злоумышленников. Все больше средств распознавания и блокировки подозрительных действий включается в ОС и приложения, что, с одной стороны, защищает пользователя, а с другой — мешает его нормальной работе, так как даже для выполнения часто повторяющихся и рутинных операций, например для сохранения вложений в письма, приходится преодолевать блокировки ОС, что для рядового пользователя иногда весьма непросто.

Виртуальные распределенные вычислительные системы суперкомпьютеров

Для ОС суперкомпьютеров и серверов одной из перспективных задач является поддержка работы компьютера в составе виртуальной распределенной вычислительной системы, узлы которой общаются через сеть, желательно — через

Интернет. В сущности, это возврат к первоначальному назначению Интернета, который создавался для связи нескольких суперкомпьютеров и организации доступа к их вычислительным ресурсам многочисленных сотрудников исследовательских центров. Впоследствии Интернет стал основой для других сетевых сервисов: сначала сервисов файловых архивов и электронной почты, а потом и такого массового сервиса, как Web, дающего возможность использовать интернет-серверы как универсальные справочные системы, хранящие информацию практически любого вида в свободном формате, удобном для просмотра пользователями с помощью единственной программы — интернет-браузера. Эти сервисы используют в основном способности компьютеров по хранению данных в виде обычного набора файлов, их передаче по сети и представлению в удобном формате. При этом вычислительная мощность удаленных серверов (которые могут представлять собой и суперкомпьютеры) остается недоступной для пользователей Интернета — вы не можете запустить свою программу на удаленном сервере, так как этот сервер обычно не поддерживает такую услугу.

Долгое время совместное использование вычислительной мощности компьютеров через Интернет не было первоочередной задачей для разработчиков ОС и приложений, так как массовый пользователь не сталкивается с задачами, требующими для своего выполнения вычислительной мощности суперкомпьютеров. Однако сегодня очередь дошла до применения Интернета в интересах и других классов пользователей, которым нужны сверхвысокие вычислительные мощности, таких как ученые-исследователи, инженеры-конструкторы, медики, метеорологи и другие категории профессионалов. Для решения такого рода задач традиционно использовались мэйнфреймы и суперкомпьютеры. При этом для особо требовательных приложений суперкомпьютеры объединялись в **кластер** — группу тесно связанных компьютеров, которая скоординировано выполняла параллельные ветви единой задачи. Но хотя классические кластеры и были распределенными системами, их узлы были сосредоточены в пределах сравнительно небольшой области покрытия локальной сети и принадлежали они, как правило, одной организации.

Сегодняшние возможности, предоставляемые Интернетом по связи компьютеров в масштабах всего земного шара, а также новые потребности пользователей привели к созданию кластеров компьютеров, рассредоточенных по различным странам и лабораториям, но способных работать как традиционные кластеры по решению сложных вычислительных задач.

Возник новый термин — **GRID Computing** (или просто GRID), который произошел от другого давно существующего термина Power Grid, относящегося к энергетической сети, объединяющей электростанции и потребителей электроэнергии. GRID Computing — это метафора, цель которой показать, что использование вычислительной мощности удаленных мощных компьютеров становится таким же простым, как потребление электроэнергии мощных электростанций в каждом жилом доме.

Разработка и поддержание сервисов разделения вычислительной мощности компьютеров через Интернет является весьма сложной задачей. Конечной

целью является создание сервиса, который бы позволял создавать виртуальные организации, получающие по запросу доступ к вычислительным мощностям и накопителям компьютеров, рассредоточенным по различным суперкомпьютерным центрам и лабораториям. В таком случае, например, любая небольшая лаборатория, которой потребовалось обрабатывать в течение месяца большой массив экспериментальных данных, может получить в свое распоряжение дорогой суперкомпьютер, направив заявку в соответствующую службу, в результате чего необходимый виртуальный компьютер становится доступным сотрудникам лаборатории через Интернет.

Сегодня GRID — это концепция, подкрепленная набором стандартов. Существует форум Open Grid, который помогает разработке таких стандартов и реализации на их базе программных средств и соответствующих сервисов. Первые практические GRID-сервисы были созданы в CERN, том же самом европейском центре ядерных исследований, в котором появился на свет сервис WWW. Разработка GRID-сервисов широко поддерживается Европейским Союзом, в результате чего было создано несколько пан-европейских GRID-сервисов, например, в рамках проекта Enabling Grids for E-sciencE (EGEE). На момент написания этой книги (начало 2007 года) GRID-сервис EGEE состоял из более 20 000 процессоров и обладал суммарной емкостью дисковой памяти в 5 000 000 гигабайтов. Еще одной практической реализацией идеи GRID является сервис PlanetLab, который также предоставляет виртуальную вычислительную мощность ученым-исследователям.

На пути использования виртуальных GRID-кластеров стоит проблема распараллеливания сложных вычислительных задач. Сегодня существует много задач, которые не распараллеливаются естественным образом, так что ученым еще предстоит потрудиться для того, чтобы сделать их пригодными для GRID-сервисов. Но существует и большое количество распределенных баз данных, для которых GRID является естественной средой выполнения. Например, множество медицинских данных создается в каждом крупном медицинском центре при наблюдении над больными, и возможность сопоставить их с помощью виртуальной GRID-машины является очень ценной для медиков. Такие же проблемы у метеорологов, собирающих данные от многочисленных датчиков и спутников; ученые астрономы также стоят перед задачей обработки данных, получаемых от телескопов, расположенных в разных странах. Так что потенциальных пользователей у виртуальных GRID-систем много, осталось сделать эти сервисы доступными.

Для организации GRID-вычислений нужна их программная поддержка. До недавнего времени эта поддержка реализовывалась в дополнительном относительно ОС слое программного обеспечения, но очевидно, что было бы хорошо внедрить в ОС некоторые дополнительные функции, позволяющие разделять процессор через Интернет. В середине 2006 года стартовал проект XtreemOS, спонсируемый Европейским Союзом, который ставит своей задачей создание специализированной Grid-ОС на основе кода Linux. Компания Microsoft в конце 2006 года при презентации Windows Vista также объявила о серьезности

своих намерений в области создания операционных систем для вычислительных кластеров.

На этом мы заканчиваем обзор важнейших событий, которые произошли в мире операционных систем за последние полвека. Краткую хронологию этих событий вы найдете в табл. 1.1. Но эволюция ОС на этом не заканчивается. Возможно, именно сейчас зарождается нечто, чему мы пока не способны дать правильную оценку, как не могли, например, оценить перспективы Интернета при его зарождении в конце 60-х, то есть нечто, что через несколько лет будет признано революционным событием, повлиявшим на всю дальнейшую историю операционных систем.

Таблица 1.1. Эволюция ОС. Хронология событий

2006	ОС Windows Vista Старт второй очереди проекта виртуальной распределенной вычислительной системы (GRID)
2000	Windows 2000
1995	ОС для мэйнфреймов OS/390
1993	Windows NT 3.1; NetWare 4.0
1991	Первая ОС семейства LINUX
1987	OS/2 – первая мультипрограммная ОС для персонального компьютера
1985	Первая ОС семейства Windows
1984	Первая ОС семейства Mac
1983	Первая ОС семейства NetWare
1981	Операционная система MS-DOS для персонального компьютера
1980	Принят стандарт технологии Ethernet локальных сетей
1979	Рабочий вариант стека TCP/IP
1976	Первое сетевое приложение UUCP для Unix
1974	Сетевые технологии SNA и X.25
1973	Операционные системы RSX-11 и RT-11 для миникомпьютеров
1970	Глобальная вычислительная сеть ARPANET
1969	Первая ОС семейства Unix
1968	Многомашинная система разделения времени АИСТ
1965	Первые мультипрограммные ОС: MULTICS, OS/360
1964	Первое семейство программно совместимых компьютеров IBM System/360
1962	Программный монитор для БЭСМ-6
1961	Первая реализация виртуальной памяти компанией Burroughs для ее компьютера B5000
1960	Первые программные мониторы – прообразы операционных систем
1951	Первая советская электронная вычислительная машина М-1
1950	
Вторая половина 40-х	Первые ламповые компьютеры в Западной Европе и США

Выводы

История ОС насчитывает примерно полвека. Она во многом определяется развитием элементной базы и вычислительной аппаратуры.

Первые цифровые вычислительные машины, появившиеся в начале 40-годов, работали без операционных систем, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления.

Прообразом современных операционных систем явились мониторные системы середины 50-х, которые автоматизировали действия оператора по выполнению пакета заданий.

В 1965–1975 годах переход к интегральным микросхемам открыл путь к появлению следующего поколения компьютеров, ярким представителем которых является IBM/360. В этот период были реализованы практически все основные концепции, присущие современным ОС: мультипрограммирование, мультипроцессирование, многотерминальный режим, виртуальная память, файловые системы, разграничение доступа и сетевая работа.

Реализация мультипрограммирования потребовала внесения очень важных изменений в аппаратуру компьютера. В процессорах появились привилегированный и пользовательский режимы работы, специальные регистры для быстрого переключения с одной задачи на другую, средства защиты областей памяти, а также развитая система прерываний.

В конце 60-х были начаты работы по созданию глобальной сети ARPANET, явившейся отправной точкой для Интернета, — глобальной общедоступной сети, которая стала для многих сетевых ОС испытательным полигоном, позволившим проверить в реальных условиях возможности их взаимодействия, степень масштабируемости, способность работы при экстремальной нагрузке.

К середине 70-х годов широкое распространение получили миникомпьютеры. Архитектура миникомпьютеров была значительно упрощена по сравнению с мэйнфреймами, что нашло отражение и в их ОС. Экономичность и доступность миникомпьютеров послужила мощным стимулом для создания локальных сетей. Предприятие, которое теперь могло позволить себе иметь несколько миникомпьютеров, нуждалось в организации совместного использования данных и дорогого периферийного оборудования. Первые локальные сети строились с помощью нестандартного коммуникационного оборудования и нестандартного программного обеспечения.

С середины 70-х годов началось массовое использование Unix, уникальной для того времени ОС, которая сравнительно легко переносилась на различные типы компьютеров. Хотя ОС Unix была первоначально разработана для миникомпьютеров, ее гибкость, элегантность, мощные функциональные возможности и открытость позволили ей занять прочные позиции во всех классах компьютеров.

В конце 70-х годов был создан рабочий вариант стека протоколов TCP/IP. В 1983 году стек протоколов TCP/IP был стандартизован. Независимость от производителей, гибкость и эффективность, доказанные успешной работой

в Интернете, сделали протоколы TCP/IP не только главным транспортным механизмом Интернета, но и основным стеком большинства сетевых ОС.

Начало 80-х годов ознаменовалось знаменательным для истории операционных систем событием — появлением персональных компьютеров, которые стали мощным катализатором бурного роста локальных сетей, создав для этого отличную материальную основу в виде десятков и сотен компьютеров, расположенных в пределах одного здания. В результате поддержка сетевых функций стала для ОС персональных компьютеров необходимым условием.

В 80-е годы были приняты основные стандарты на коммуникационные технологии для локальных сетей: в 1980 году — Ethernet, в 1985 — Token Ring, в конце 80-х — FDDI. Это позволило обеспечить совместимость сетевых ОС на нижних уровнях, а также стандартизовать интерфейс ОС с драйверами сетевых адаптеров.

К началу 90-х практически все ОС стали сетевыми, способными поддерживать работу с разнородными клиентами и серверами. Появились специализированные сетевые ОС, предназначенные исключительно для решения коммуникационных задач, например система IOS компании Cisco Systems, работающая на маршрутизаторах.

Особое внимание в течение всего последнего десятилетия уделялось корпоративным сетевым ОС, для которых характерны высокая степень масштабируемости, поддержка сетевой работы, развитые средства обеспечения безопасности, способность работать в гетерогенной среде, наличие средств централизованного администрирования и управления.

Развитие ОС для персональных компьютеров ставит задачи повышения их надежности, удобства эксплуатации, эффективности поиска и представления информации.

Операционные системы суперкомпьютеров будут наделяться функциями поддержки виртуальных кластеров, способных разделять вычислительную мощность компьютера через Интернет.

Задачи и упражнения

1. Какие события в развитии технической базы вычислительных машин стали вехами в истории операционных систем?
2. В чем состояло принципиальное отличие первых мониторов пакетной обработки от уже существовавших к этому времени системных обрабатывающих программ — трансляторов, загрузчиков, компоновщиков, библиотек процедур?
3. Может ли компьютер работать без операционной системы?
4. Как эволюционировало отношение к концепции мультипрограммирования на протяжении всей истории ОС?
5. Какое влияние на развитие ОС оказал Интернет?
6. Чем объясняется особое место ОС Unix в истории операционных систем?
7. Опишите историю сетевых ОС.
8. В чем состоят современные тенденции развития ОС?

Глава 2

Назначение и функции операционной системы

Сегодня существует большое количество разных типов операционных систем, отличающихся областями применения, аппаратными платформами и методами реализации. Естественно, это обуславливает и значительные функциональные различия этих ОС. Даже у конкретной операционной системы набор выполняемых функций зачастую определить не так просто — та функция, которая сегодня выполняется внешней по отношению к ОС, завтра может стать ее неотъемлемой частью, и наоборот. Поэтому при изучении операционных систем очень важно из всего многообразия выделить те функции, которые присущи всем операционным системам как классу продуктов.

Операционные системы для автономного компьютера

Операционная система компьютера — это комплекс взаимосвязанных программ, который действует как интерфейс между приложениями и пользователями, с одной стороны, и аппаратурой компьютера, с другой стороны.

В соответствии с этим определением ОС выполняет две группы функций:

- предоставление пользователю или программисту вместо реальной аппаратуры компьютера расширенной *виртуальной машины*, с которой удобней работать и которую легче программировать;
- *управление ресурсами компьютера* с целью повышения эффективности его использования.

ОС как виртуальная машина

Для того чтобы успешно решать свои задачи, современный пользователь или даже прикладной программист может обойтись без досконального знания аппа-

ратного устройства компьютера. Ему не обязательно быть в курсе того, как функционируют различные электронные блоки и электромеханические узлы компьютера. Рядовому пользователю достаточно уметь работать с графическим интерфейсом, находить на рабочем столе или в папках нужные файлы и запускать приложения на выполнение.

Программист может писать программы, не зная систему команд процессора, так как операционная система предоставляет ему в распоряжение мощные высокоуровневые языковые средства. Так, например, при работе с диском программисту, пишущему приложение для работы под управлением ОС, достаточно представлять его в виде некоторого набора файлов, каждый из которых имеет имя. Последовательность действий при работе с файлом заключается в его открытии, выполнении одной или нескольких операций чтения или записи и последующего закрытия файла. Такие частности, как используемая при записи частотная модуляция или текущее состояние двигателя механизма перемещения магнитных головок чтения/записи, не должны волновать программиста. Именно операционная система скрывает от программиста большую часть особенностей аппаратуры и предоставляет возможность простой и удобной работы с требуемыми файлами.

Если бы программист работал непосредственно с аппаратурой компьютера, без участия ОС, то для организации чтения блока данных с диска программисту пришлось бы использовать более десятка команд с указанием множества параметров: номера блока на диске, номера сектора на дорожке и т. п. А после завершения операции обмена с диском он должен был бы предусмотреть в своей программе анализ результата выполненной операции. Учитывая, что контроллер диска способен распознавать более двадцати различных вариантов завершения операции, можно считать программирование обмена с диском на уровне аппаратуры не самой тривиальной задачей. Не менее обременительной выглядит и работа пользователя, если бы ему для чтения файла с терминала потребовалось задавать числовые адреса дорожек и секторов.

Операционная система избавляет программистов не только от необходимости напрямую работать с аппаратурой дискового накопителя, предоставляя им простой файловый интерфейс, но и берет на себя все другие рутинные операции, связанные с управлением разнообразными аппаратными устройствами компьютера: физической памятью, таймерами, принтерами и т. д.

> В результате работы операционной системы реальная машина, способная выполнять только небольшой набор элементарных действий, определяемых ее системой команд, превращается в **виртуальную машину**, выполняющую широкий набор значительно более мощных функций.

В этой книге мы часто будем использовать слово «виртуальный» для обозначения объектов, с точки зрения пользователя или пользовательской программы обладающих свойствами, которыми они в действительности не обладают. Так, виртуальная машина подобно реальному компьютеру управляется командами, но это уже команды другого, более высокого уровня: удалить файл с определенным именем, запустить на выполнение некоторую прикладную программу,

повысить приоритет задачи, вывести текст из файла на печать. Таким образом, назначение ОС состоит в предоставлении пользователю/программисту некоторой усовершенствованной (часто говорят — расширенной) виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальный компьютер или реальную сеть.

ОС как система управления ресурсами

Операционная система не только предоставляет пользователям и программистам удобный интерфейс к аппаратным средствам компьютера, но и является механизмом, распределяющим ресурсы компьютера.

К числу основных ресурсов современных вычислительных систем могут быть отнесены такие ресурсы, как процессоры, основная память, таймеры, наборы данных, диски, накопители на магнитных лентах, принтеры, сетевые устройства и некоторые другие.

Ресурсы распределяются между процессами.

Процесс (задача) — это единица вычислительной работы, создаваемая операционной системой в момент запуска программы на выполнение.

Иногда процесс кратко определяют просто как программу в стадии выполнения. Однако не следует путать понятия «программа» и «процесс». Программа является *статическим* объектом, представляющим собой файл с кодами и данными, которые могут быть записаны на разных типах носителей — на листке бумаги, перфокартах, магнитном диске или в оперативной памяти. Процесс — это *динамический* объект, который возникает в операционной системе после того, как пользователь или сама операционная система решают «запустить программу на выполнение», то есть создать новую единицу вычислительной работы. Например, ОС может создать процесс в ответ на команду пользователя `run prg1.exe`, где `prg1.exe` — это имя файла, в котором хранится код программы.

В общем случае нет однозначного соответствия между процессами и программами. Один и тот же программный файл может породить несколько параллельно выполняемых процессов, а процесс может в ходе своего выполнения сменить программный файл и начать выполнять другую программу.

ПРИМЕЧАНИЕ Во многих современных ОС для обозначения минимальной единицы работы ОС используют термин «нить», или «поток», при этом изменяется суть термина «процесс». Подробнее об этом рассказывается в главе 4. В остальных главах мы будем придерживаться упрощенного толкования, в соответствии с которым для обозначения выполняемой программы будет использоваться только термин «процесс».

Основным назначением операционной системы является управление ресурсами вычислительной системы с целью наиболее эффективного их использования. Например, мультипрограммная операционная система организует одновременное выполнение сразу нескольких процессов на одном компьютере, поочередно переключая процессор с одного процесса на другой, исключая

простои процессора, вызываемые обращениями процессов к вводу-выводу. Операционная система также отслеживает и разрешает конфликты, возникающие при обращении нескольких процессов к одному и тому же устройству ввода-вывода или к одним и тем же данным.

Критерий эффективности, в соответствии с которым ОС организует управление ресурсами компьютера, может быть различным. Например, в одних системах важен такой критерий, как пропускная способность вычислительной системы, в других — время ее реакции. Очевидно, что ОС, построенные в соответствии с разными критериями эффективности, будут по-разному организовывать вычислительный процесс.

Управление ресурсами включает решение следующих общих, не зависящих от типа ресурса задач:

- планирование ресурса — то есть определение, какому процессу, когда и в каком количестве (если ресурс может выделяться частями) следует выделить данный ресурс;
- удовлетворение запросов на ресурсы;
- отслеживание состояния и учет использования ресурса — то есть поддержание оперативной информации о том, занят или свободен ресурс и какая доля ресурса уже распределена;
- разрешение конфликтов между процессами.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, особенности которых, в конечном счете, и определяют облик ОС в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Например, применяемый алгоритм управления процессором в значительной степени определяет, может ли ОС использоваться как система разделения времени, система пакетной обработки или система реального времени.

Задача организации эффективного совместного использования ресурсов несколькими процессами является весьма сложной, и сложность эта порождается в основном случайным характером возникновения запросов на потребление ресурсов. В мультипрограммной системе образуются очереди заявок от одновременно выполняемых программ к разделяемым ресурсам компьютера: процессору, странице памяти, к принтеру, к диску. Операционная система организует обслуживание этих очередей по разным алгоритмам: в порядке поступления, на основе приоритетов, путем кругового обслуживания и т. д. Анализ и определение оптимальных дисциплин обслуживания заявок является предметом специальной области прикладной математики — теории массового обслуживания. Эта теория иногда используется для оценки эффективности тех или иных алгоритмов управления очередями в операционных системах. Очень часто в ОС реализуются и эмпирические алгоритмы обслуживания очередей, прошедшие проверку практикой.

Таким образом, управление ресурсами составляет важную часть функций любой операционной системы, в особенности мультипрограммной. В отличие

от функций расширенной машины, большинство функций управления ресурсами выполняется операционной системой автоматически и прикладному программисту недоступно.

Функциональные компоненты операционной системы автономного компьютера

Функции операционной системы автономного компьютера обычно группируются либо в соответствии с типами локальных ресурсов, которыми управляет ОС, либо в соответствии со специфическими задачами, применимыми ко всем ресурсам. Иногда такие группы функций называют подсистемами. Наиболее важными подсистемами управления ресурсами являются подсистемы управления процессами, памятью, файлами и внешними устройствами, а подсистемами, общими для всех ресурсов, являются подсистемы пользовательского интерфейса, защиты данных и администрирования.

Управление процессами

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема *управления процессами*.

Для каждого вновь создаваемого процесса ОС генерирует системные информационные структуры, которые содержат данные о потребностях процесса в ресурсах вычислительной системы, а также о фактически выделенных ему ресурсах. Таким образом, процесс можно также определить как некоторую заявку на потребление системных ресурсов.

Чтобы процесс мог быть выполнен, операционная система должна назначить ему область оперативной памяти, в которой будут размещены коды и данные процесса, а также предоставить ему необходимое процессорное время. Кроме того, процессу может понадобиться доступ к таким ресурсам, как файлы и устройства ввода-вывода.

В информационные структуры процесса часто включаются вспомогательные данные, характеризующие историю пребывания процесса в системе (например, какую долю времени процесс потратил на операции ввода-вывода, а какую — на вычисления), его текущее состояние (активное или заблокированное), степень привилегированности процесса (значение приоритета). Данные такого рода могут учитываться операционной системой при принятии решения о предоставлении ресурсов процессу.

В мультипрограммной операционной системе одновременно может существовать несколько процессов. Часть процессов порождается по инициативе пользователей и их приложений, такие процессы обычно называют **пользовательскими**. Другие процессы, называемые **системными**, инициализируются самой операционной системой для выполнения своих функций.

Поскольку процессы часто одновременно претендуют на одни и те же ресурсы, то в обязанности ОС входит поддержание очередей заявок процессов на ресурсы, например очереди к процессору, к принтеру, к последовательному порту.

> Важной задачей операционной системы является *защита* ресурсов, выделенных данному процессу, от остальных процессов.

Одним из наиболее тщательно защищаемых ресурсов процесса являются области оперативной памяти, в которой хранятся коды и данные процесса. Совокупность всех областей оперативной памяти, выделенных операционной системой процессу, называется его **адресным пространством**. Говорят, что каждый процесс работает в своем адресном пространстве, имея в виду защиту адресных пространств, реализуемую ОС. Защищаются и другие типы ресурсов, такие как файлы, внешние устройства и т. д. Операционная система может не только защищать ресурсы, выделенные одному процессу, но и организовывать их совместное использование, например разрешать доступ к некоторой области памяти нескольким процессам.

На протяжении периода существования процесса его выполнение может быть многократно прервано и продолжено. Для того чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды идентифицируется состоянием регистров и программного счетчика, режимом работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и т. д. Эта информация называется **контекстом процесса**. Говорят, что при смене процесса происходит переключение контекстов.

Операционная система берет на себя также функции синхронизации процессов, позволяющие процессу приостанавливать свое выполнение до наступления какого-либо события в системе, например завершения операции ввода-вывода, осуществляемой по его запросу операционной системой.

Для реализации сложных программных комплексов полезно бывает организовать их работу в виде нескольких параллельных процессов, которые периодически взаимодействуют друг с другом и обмениваются некоторыми данными. Так как операционная система защищает ресурсы процессов и не позволяет одному процессу писать в память или читать из памяти другого процесса, то для оперативного взаимодействия процессов ОС должна предоставлять особые средства, которые называют средствами межпроцессного взаимодействия.

Таким образом, подсистема управления процессами ОС выполняет следующие действия:

- распределяет процессорное время между несколькими одновременно выполняемыми в системе процессами;
- занимается созданием и уничтожением процессов;
- обеспечивает процессы необходимыми системными ресурсами;

- поддерживает синхронизацию процессов;
- реализует взаимодействие между процессами.

Управление памятью

Память является для процесса не менее важным ресурсом, чем процессор, так как процесс может выполняться процессором только в том случае, если его коды и данные (не обязательно все) находятся в оперативной памяти.

Управление памятью включает распределение имеющейся физической памяти между всеми существующими в системе в данный момент процессами, загрузку кодов и данных процессов в отведенные им области памяти, настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области, а также защиту областей памяти каждого процесса.

Существует большое разнообразие алгоритмов распределения памяти. Они могут отличаться, например, количеством выделяемых процессу областей памяти (в одних случаях память выделяется процессу в виде одной непрерывной области, а в других — в виде нескольких несмежных областей), степенью свободы границы областей (она может быть жестко зафиксирована на все время существования процесса или же динамически перемещаться при выделении процессу дополнительных объемов памяти). В некоторых системах распределение памяти выполняется страницами фиксированного размера, в других — сегментами переменной длины.

Одним из наиболее популярных механизмов управления памятью в современных операционных системах является механизм так называемой **виртуальной памяти**.

> *Механизм виртуальной памяти* позволяет программисту писать программу так, как будто в его распоряжении имеется однородная оперативная память большого объема, часто существенно превышающего объем имеющейся физической памяти.

В действительности все данные, используемые программой, хранятся на диске и при необходимости частями (сегментами или страницами) отображаются на физическую память. При перемещении кодов и данных между оперативной памятью и диском подсистема виртуальной памяти выполняет трансляцию виртуальных адресов, полученных в результате компиляции и компоновки программы, в физические адреса ячеек оперативной памяти. Очень важно, что все операции по перемещению кодов и данных между оперативной памятью и дисками, а также трансляция адресов выполняются ОС прозрачно для программиста.

Защита памяти — это избирательная способность предохранять выполняемую задачу от записи или чтения памяти, назначенной другой задаче. Правильно написанные программы не пытаются обращаться к памяти, назначенной другим. Однако реальные программы часто содержат ошибки, в результате которых такие попытки иногда предпринимаются. Средства защиты памяти,

реализованные в операционной системе, должны пресекать несанкционированный доступ процессов к чужим областям памяти.

Таким образом, функциями ОС по управлению памятью являются:

- отслеживание свободной и занятой памяти;
- выделение памяти процессам и освобождение памяти при завершении процессов;
- защита памяти;
- вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти недостаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место;
- настройка адресов программы на конкретную область физической памяти.

Управление файлами и внешними устройствами

Способность ОС к «экранированию» сложностей реальной аппаратуры очень ярко проявляется в одной из основных подсистем ОС — *файловой системе*. Операционная система виртуализирует отдельный набор данных, хранящихся на внешнем накопителе, в виде файла — простой неструктурированной последовательности байтов, имеющей символьное имя. Для удобства работы с данными файлы группируются в каталоги, которые, в свою очередь, образуют группы — каталоги более высокого уровня. Пользователь может с помощью ОС выполнять над файлами и каталогами такие действия, как поиск по имени, удаление, вывод содержимого на внешнее устройство (например, на дисплей), изменение и сохранение содержимого.

Чтобы представить большое количество наборов данных, разбросанных случайным образом по цилиндрам и поверхностям дисков различных типов, в виде хорошо всем знакомой и удобной иерархической структуры файлов и каталогов, операционная система должна решить множество задач. Файловая система ОС выполняет преобразование символьных имен файлов, с которыми работает пользователь или прикладной программист, в физические адреса данных на диске, организует совместный доступ к файлам, защищает их от несанкционированного доступа.

При выполнении своих функций файловая система тесно взаимодействует с подсистемой управления внешними устройствами, которая по запросам файловой системы осуществляет передачу данных между дисками и оперативной памятью.

Подсистема управления внешними устройствами, называемая также подсистемой ввода-вывода, исполняет роль интерфейса ко всем устройствам, подключенным к компьютеру. Спектр этих устройств очень обширен. Номенклатура выпускаемых накопителей на жестких, гибких и оптических дисках, принтеров, сканеров, мониторов, плоттеров, модемов, сетевых адаптеров и более специальных устройств ввода-вывода, таких как, например, аналого-цифровые преобразователи, может насчитывать сотни моделей. Эти модели могут существенно отличаться набором и последовательностью команд, с помощью которых осу-

ществляется обмен информацией с процессором и памятью компьютера, скоростью работы, кодировкой передаваемых данных, возможностью совместного использования и множеством других деталей.

Программа, управляющая конкретной моделью внешнего устройства и учитывающая все его особенности, обычно называется **драйвером** этого устройства (от английского drive — управлять, вести). Драйвер может управлять единственной моделью устройства, например принтером Deskjet 3420 компании Hewlett-Packard, или же группой устройств определенного типа, например любыми Hayes-совместимыми модемами. Для пользователя очень важно, чтобы операционная система включала как можно больше разнообразных драйверов, так как это гарантирует возможность подключения к компьютеру большого числа внешних устройств различных производителей. От наличия подходящих драйверов во многом зависит успех операционной системы на рынке (например, отсутствие многих необходимых драйверов внешних устройств было одной из причин сравнительно низкой популярности OS/2).

Созданием драйверов устройств занимаются как разработчики конкретной ОС, так и специалисты компаний, выпускающих внешние устройства. Операционная система должна поддерживать хорошо определенный интерфейс между драйверами и остальной частью ОС, чтобы разработчики из компаний-производителей устройств ввода-вывода могли поставлять вместе со своими устройствами драйверы для данной операционной системы.

Прикладные программисты могут пользоваться интерфейсом драйверов при разработке своих программ, но это не очень удобно — такой интерфейс обычно представляет собой низкоуровневые операции, обремененные большим количеством деталей.

Поддержание высокоуровневого унифицированного интерфейса прикладного программирования к разнородным устройствам ввода-вывода является одной из наиболее важных задач ОС. Со времени появления ОС Unix такой унифицированный интерфейс в большинстве операционных систем строится на основе концепции файлового доступа. Эта концепция заключается в том, что обмен с любым внешним устройством выглядит как обмен с файлом, имеющим имя и представляющим собой неструктурированную последовательность байтов. В качестве файла может выступать как реальный файл на диске, так и алфавитно-цифровой терминал, печатающее устройство или сетевой адаптер. Здесь мы опять имеем дело со свойством операционной системы подменять реальную аппаратуру удобными для пользователя и программиста абстракциями.

Защита данных и администрирование

Безопасность данных вычислительной системы обеспечивается средствами отказоустойчивости ОС, направленными на защиту от сбоев и отказов аппаратуры и ошибок программного обеспечения, а в многопользовательских ОС также средствами защиты от несанкционированного доступа. В последнем случае ОС защищает данные от ошибочного или злонамеренного поведения пользователей системы.

Первым рубежом обороны при защите данных от несанкционированного доступа является *процедура логического входа*. Операционная система должна убедиться, что в систему пытается войти пользователь, вход которого разрешен администратором. Функции защиты ОС вообще очень тесно связаны с функциями *администрирования*, так как именно администратор определяет права пользователей при их обращении к разным ресурсам системы — файлам, каталогам, принтерам, сканерам и т. п. Кроме того, администратор ограничивает возможности пользователей в выполнении тех или иных системных действий. Например, пользователю может быть запрещено выполнять процедуру завершения работы ОС, устанавливать системное время, завершать чужие процессы, создавать учетные записи пользователей, изменять права доступа к некоторым каталогам и файлам. Администратор может также урезать возможности пользовательского интерфейса, убрав, например, некоторые пункты из меню операционной системы, выводимого на дисплей пользователя.

Важным средством защиты данных являются функции *аудита* ОС, заключающиеся в фиксации всех событий, от которых зависит безопасность системы. Например, попытки удачного и неудачного логического входа в систему, операции доступа к некоторым каталогам и файлам, использование принтеров и т. п. Список событий, которые необходимо отслеживать, определяет администратор ОС.

Поддержка отказоустойчивости реализуется операционной системой, как правило, на основе *резервирования*. Чаще всего в функции ОС входит поддержание нескольких копий данных на разных дисках или разных дисковых накопителях. Резервируются также принтеры и другие устройства ввода-вывода. При отказе одного из избыточных устройств операционная система должна быстро и прозрачным для пользователя образом произвести реконфигурирование и продолжить работу с резервным устройством. Особым случаем обеспечения отказоустойчивости является использование нескольких процессоров, то есть мультипроцессирование, когда система продолжает работу при отказе одного из процессоров, хотя и с меньшей производительностью. (Необходимо отметить, что многие ОС использует мультипроцессорную конфигурацию компьютера только для ускорения работы, и при отказе одного из процессоров прекращают работу.)

Поддержка отказоустойчивости также входит в обязанности системного администратора. В состав ОС обычно входят утилиты, позволяющие администратору выполнять регулярные операции резервного копирования для быстрого восстановления важных данных.

Интерфейс прикладного программирования

Прикладные программисты используют в своих приложениях обращения к ОС, когда для выполнения тех или иных действий им требуется особый, привилегированный статус, которым обладает только операционная система[1]. Например, в большинстве современных ОС все действия, связанные с управлением аппа-

[1] Точнее — только некоторые модули ОС.

ратными средствами компьютера, может выполнять только ОС. Поэтому для того, чтобы, скажем, установить таймер или получить дополнительную память, приложению нужно обратиться к ОС. Помимо функций, связанных с аппаратными ресурсами, ОС предоставляет прикладному программисту набор функций ОС, которые упрощают написание приложений. Функции такого типа реализуют универсальные действия, часто требующиеся в различных приложениях, такие, например, как обработка текстовых строк. Эти функции в принципе могут быть выполнены и самим приложением (что и делается в тех случаях, когда качество системной процедуры не устраивает прикладного программиста), однако часто более предпочтительным является использование уже готовых, отлаженных процедур, включенных в состав операционной системы.

Возможности операционной системы доступны прикладному программисту в виде набора функций, называющегося **интерфейсом прикладного программирования** (Application Programming Interface, **API**).

От конечного пользователя эти функции скрыты за оболочкой алфавитно-цифрового или графического пользовательского интерфейса.

Для разработчиков приложений все особенности конкретной операционной системы представлены особенностями ее API. Поэтому операционные системы с различной внутренней организацией, но с одинаковым набором API-функций кажутся им одной и той же ОС, что упрощает стандартизацию операционных систем и обеспечивает переносимость приложений между внутренне различными ОС, соответствующими определенному стандарту на API. Например, следование общим стандартам API Unix, одним из которых является стандарт Posix, позволяет говорить о некоторой обобщенной операционной системе Unix, хотя многочисленные версии этой ОС от разных производителей иногда существенно различаются внутренней организацией.

Приложения выполняют обращения к API-функциям с помощью **системных вызовов**. Способ, которым приложение получает услуги операционной системы, очень похож на вызов подпрограмм. Информация, нужная ОС и состоящая обычно из идентификатора команды и данных, помещается в определенное место памяти, в регистры и/или стек. Затем управление передается операционной системе, которая выполняет требуемую функцию и возвращает результаты через память, регистры или стеки. Если операция проведена неуспешно, то результат включает индикацию ошибки.

Реализация системных вызовов зависит от структурной организации ОС, которая, в свою очередь, тесно связана с особенностями аппаратной платформы. Кроме того, она зависит от языка программирования. При использовании ассемблера программист устанавливает значения регистров и/или областей памяти, а затем выполняет специальную инструкцию вызова сервиса или программного прерывания для обращения к некоторой функции ОС. При применении языков высокого уровня функции ОС вызываются тем же способом, что и написанные пользователем подпрограммы, требуя задания определенных аргументов в определенном порядке.

Пользовательский интерфейс

Операционная система должна предоставлять удобный интерфейс не только для прикладных программ, но и для человека, работающего за терминалом. Этот человек может быть конечным пользователем, администратором ОС или программистом.

В ранних операционных системах пакетного режима функции пользовательского интерфейса были сведены к минимуму и не требовали наличия терминала. Команды языка управления заданиями набивались на перфокарты, а результаты выводились на печатающее устройство.

Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалами двух типов: алфавитно-цифровыми и графическими.

При работе за *алфавитно-цифровым терминалом* пользователь имеет в своем распоряжении систему команд, возможности которой отражают функциональные возможности данной ОС. Обычно командный язык ОС позволяет запускать и останавливать приложения, выполнять различные операции с файлами и каталогами, получать информацию о состоянии ОС (количество работающих процессов, объем свободного пространства на дисках и т. п.), администрировать систему. Команды могут вводиться не только в интерактивном режиме с терминала, но и считываться из так называемого **командного файла**, содержащего некоторую последовательность команд.

Программный модуль ОС, ответственный за чтение отдельных команд или же последовательностей команд из командного файла, иногда называют **командным интерпретатором.**

Ввод команды может быть упрощен, если операционная система поддерживает *графический пользовательский интерфейс*. В этом случае пользователь для выполнения нужного действия с помощью мыши выбирает на экране нужный пункт меню или графический символ.

Как показал опыт использования ОС, оба типа интерфейса нужны пользователям. Массовый непрофессиональный пользователь в основном применяет графический оконный интерфейс, в то время как профессионалы задействуют оба в зависимости от решаемых задач: более мощный по функциональным возможностям интерфейс командной строки, когда нужно достичь достаточно специфического и тонкого эффекта, и графический интерфейс для рутинных операций.

Сетевые операционные системы

Компьютерная сеть — это набор компьютеров, связанных коммуникационной системой и снабженных соответствующим программным обеспечением, позволяющим пользователям сети получать доступ к ресурсам этого набора компьютеров. Сеть могут образовывать компьютеры разных типов, в частности небольшие микропроцессоры, рабочие станции, миникомпьютеры, персональные компьютеры или суперкомпьютеры. **Коммуникационная система** может вклю-

чать кабели, повторители, коммутаторы, маршрутизаторы и другие устройства, обеспечивающие передачу сообщений между любой парой компьютеров сети.

Сетевая операционная система позволяет пользователю работать со своим компьютером как с автономным и добавляет к этому возможность доступа к информационным и аппаратным ресурсам других компьютеров сети.

При организации сетевой работы операционная система играет роль интерфейса, экранирующего от пользователя все детали низкоуровневых программно-аппаратных средств сети. Например, вместо числовых адресов компьютеров сети, таких как MAC-адрес и IP-адрес, операционная система компьютерной сети позволяет оперировать удобными для запоминания символьными именами. В результате в представлении пользователя сеть с ее множеством сложных и запутанных реальных деталей превращается в достаточно понятный набор разделяемых ресурсов.

Сетевая ОС предоставляет пользователю некую виртуальную вычислительную систему, работать с которой гораздо проще, чем с реальной сетевой аппаратурой. В то же время эта виртуальная система не полностью скрывает распределенную природу своего реального прототипа. При обращении к ресурсам компьютеров сети пользователи сетевой ОС всегда помнят, что они имеют дело с сетевыми ресурсами, и для доступа к ним нужно выполнить некоторые особые операции, например отобразить удаленный разделяемый каталог на вымышленную локальную букву дисковода или поставить перед именем каталога еще и имя компьютера, на котором тот расположен. Пользователи сетевой ОС обычно должны быть в курсе того, где хранятся их файлы, и использовать явные команды передачи файлов для перемещения фалов с одной машины на другую.

Работая в среде сетевой ОС, пользователь, хотя и может запустить задание на любой машине компьютерной сети, всегда знает, на какой машине выполняется его задание. По умолчанию пользовательское задание выполняется на той машине, на которой пользователь сделал логический вход. Если же он хочет выполнить задание на другой машине, то ему нужно либо выполнить удаленный логический вход на эту машину, используя команду типа `remote login`, либо ввести специальную команду удаленного выполнения, в которой он должен указать информацию, идентифицирующую удаленный компьютер.

Магистральным направлением развития сетевых операционных систем является достижение как можно более высокой степени *прозрачности* сетевых ресурсов. В идеальном случае сетевая ОС должна представить пользователю сетевые ресурсы не в виде сети, а в виде ресурсов единой централизованной виртуальной машины. Для такой операционной системы используют специальное название — **распределенная ОС**, или **истинно распределенная ОС**.

Распределенная ОС существует как единая операционная система в масштабах вычислительной системы. Каждый компьютер сети, работающей под управлением распределенной ОС, выполняет часть функций этой глобальной ОС. Распределенная ОС, динамически и автоматически распределяя работы по

различным машинам системы для обработки, заставляет набор сетевых машин действовать как виртуальный унипроцессор. Пользователь распределенной ОС, вообще говоря, не имеет сведений о том, на какой машине выполняется его работа.

В настоящее время практически все операционные системы относятся к классу сетевых, однако они еще очень далеки от идеала истинной распределенности. Современная компьютерная сеть работает под управлением совокупности сетевых ОС, установленных на каждом из компьютеров. Эти ОС могут быть одинаковыми или разными. Например, на всех компьютерах сети может работать одна и та же ОС Unix. Более реалистичным вариантом является сеть, в которой работают разные ОС, например, часть компьютеров работает под управлением Unix FreeBSD, часть — под управлением NetWare 6.5, а остальные — под управлением Windows Vista. Все эти операционные системы функционируют независимо друг от друга в том смысле, что каждая из них принимает независимые решения о создании и завершении собственных процессов и управлении локальными ресурсами. Но в любом случае операционные системы компьютеров, работающих в сети, должны включать взаимно согласованный набор коммуникационных протоколов для организации взаимодействия процессов, выполняющихся на разных компьютерах сети, и разделения ресурсов этих компьютеров между пользователями сети.

Функциональные компоненты сетевой ОС

На рис. 2.1 показаны основные функциональные компоненты сетевой ОС:

- *средства управления локальными ресурсами* компьютера реализуют все функции ОС автономного компьютера (распределение оперативной памяти между процессами, планирование и диспетчеризацию процессов, управление процессорами в мультипроцессорных машинах, управление внешней памятью, интерфейс с пользователем и т. д.);
- *сетевые средства*, в свою очередь, можно разделить на три компонента:
  - средства предоставления локальных ресурсов и услуг в общее пользование — *серверная часть ОС*;
  - средства запроса доступа к удаленным ресурсам и услугам — *клиентская часть ОС*;
  - *транспортные средства ОС* совместно с коммуникационной системой обеспечивают передачу сообщений между компьютерами сети.

Упрощенно работа сетевой ОС происходит следующим образом. Предположим, что пользователь компьютера *A* решил разместить свой файл на диске другого компьютера сети — компьютера *B*. Для этого он набирает на клавиатуре соответствующую команду. Программный модуль ОС, отвечающий за интерфейс с пользователем, принимает эту команду и передает ее клиентской части ОС компьютера *A*.

Клиентская часть ОС не может получить непосредственный доступ к ресурсам другого компьютера — в данном случае к дискам и файлам компьютера *B*. Она может только «попросить» об этом серверную часть ОС, работающую на

Рис. 2.1. Функциональные компоненты сетевой ОС

том компьютере, которому принадлежат эти ресурсы. Эти «просьбы» выражаются в виде **сообщений**, передаваемых по сети. Сообщения могут содержать не только команды на выполнение некоторых действий, но и собственно данные, например содержимое некоторого файла.

Управляют передачей сообщений между клиентской и серверной частями по коммуникационной системе сети транспортные средства ОС. Эти средства выполняют такие функции, как формирование сообщений, разбиение сообщения на части (пакеты, кадры), преобразование имен компьютеров в числовые адреса, организация надежной доставки сообщений, определение маршрута в сложной сети и т. д.

Правила взаимодействия компьютеров при передаче сообщений по сети фиксируются в **коммуникационных протоколах**, таких как Ethernet, Token Ring, IP, IPX и пр. Чтобы два компьютера смогли обмениваться сообщениями по сети, транспортные средства их ОС должны поддерживать некоторый общий набор коммуникационных протоколов. Коммуникационные протоколы переносят сообщения клиентских и серверных частей ОС по сети, не вникая в их содержание.

На стороне компьютера *B*, на диске которого пользователь хочет разместить свой файл, должна работать серверная часть ОС, постоянно ожидающая прихода из сети запросов на удаленный доступ к ресурсам этого компьютера. Серверная часть, приняв запрос из сети, обращается к локальному диску и записывает в один из его каталогов указанный файл. Конечно, для выполнения этих действий требуется не одно, а целая серия сообщений, переносящих между компьютерами команды ОС и части передаваемого файла.

Очень удобной и полезной способностью клиентской части ОС является способность отличить запрос к удаленному файлу от запроса к локальному файлу. Если клиентская часть ОС умеет это делать, то приложения не должны заботиться о том, с локальным или удаленным файлом они работают — клиентская программа сама распознает и перенаправляет (redirect) запрос к удаленной машине. Отсюда и название, часто используемое для клиентской части сетевой ОС, — **редиректор**. Иногда функции распознавания выделяются в отдель-

ный программный модуль, в этом случае редиректором называют не всю клиентскую часть, а только этот модуль.

Клиентские части сетевых ОС выполняют также преобразование форматов запросов к ресурсам. Они принимают запросы от приложений на доступ к сетевым ресурсам в локальной форме, то есть в форме, принятой в локальной части ОС. В сеть же запрос передается клиентской частью в другой форме, соответствующей требованиям серверной части ОС, работающей на компьютере, где расположен требуемый ресурс. Клиентская часть также осуществляет прием ответов от серверной части и преобразование их в локальный формат, так что для приложения выполнение локальных и удаленных запросов неразличимо.

Сетевые службы и сетевые сервисы

Совокупность серверной и клиентской частей ОС, предоставляющих доступ к конкретному типу ресурса компьютера через сеть, называется **сетевой службой**. В приведенном ранее примере клиентская и серверная части ОС, которые совместно обеспечивают доступ через сеть к файловой системе компьютера, образуют файловую службу. Говорят, что сетевая служба предоставляет пользователям сети некоторый набор **услуг**. Например, большинство файловых служб может по запросу пользователя или приложения выполнять следующие действия (предоставлять услуги): создавать, удалять, копировать, переименовывать файлы и др. Услуги сетевых служб иногда называют **сетевым сервисом** (от английского service).

ПРИМЕЧАНИЕ Термин «service» в технической литературе переводится и как «сервис», и как «услуга», и как «служба». Хотя указанные термины иногда используются как синонимы, следует иметь в виду, что в некоторых случаях различие в значениях этих терминов носит принципиальный характер. Далее в тексте под службой мы будем понимать сетевой компонент, который реализует некоторый набор услуг, а под сервисом — описание того набора услуг, который предоставляется данной службой.

Каждая служба связана с определенным типом сетевых ресурсов и/или определенным способом доступа к этим ресурсам. Например, служба печати обеспечивает доступ пользователей сети к разделяемым принтерам сети и предоставляет сервис печати, а почтовая служба предоставляет доступ к информационному ресурсу сети — электронным письмам. Способом доступа к ресурсам отличается, например, служба удаленного доступа — она предоставляет пользователям компьютерной сети доступ ко всем ее ресурсам через коммутируемые телефонные каналы. Для получения удаленного доступа к конкретному ресурсу, например к принтеру, служба удаленного доступа взаимодействует со службой печати. Среди сетевых служб можно выделить такие, которые ориентированы не на простого пользователя, а на администратора. Такие службы предназначены для организации работы сети. Например, централизованная справочная служба, или служба каталогов, предназначена для ведения базы данных о пользовате-

лях сети, обо всех ее программных и аппаратных компонентах[1]. Другими примерами сетевых служб, предоставляющих сервис администратору, являются служба мониторинга сети, позволяющая захватывать и анализировать сетевой трафик, служба безопасности, в функции которой может, в частности, входить выполнение процедуры логического входа с проверкой пароля, служба резервного копирования и архивирования.

От того, насколько богатый набор сетевых служб и услуг предлагает операционная система конечным пользователям, приложениям и администраторам сети, зависит ее позиция в общем ряду сетевых ОС.

Сетевые службы по своей природе являются клиент-серверными системами. Поскольку при реализации любого сетевого сервиса естественно возникает источник запросов (клиент) и исполнитель запросов (сервер), то и любая сетевая служба содержит в своем составе две несимметричные части — клиентскую и серверную (рис. 2.2). Сетевая служба может быть представлена в операционной системе либо обоими (клиентской и серверной) частями, либо только одной из них.

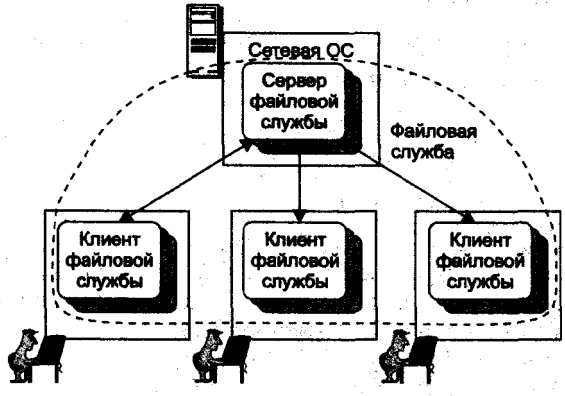


Рис. 2.2. Клиент-серверная природа сетевых служб

Обычно говорят, что сервер предоставляет свои ресурсы клиенту, а клиент ими пользуется. Необходимо отметить, что при предоставлении сетевой службой некоторой услуги используются ресурсы не только сервера, но и клиента. Клиент может затрачивать значительную часть своих ресурсов (дискового пространства, процессорного времени и т. п.) на поддержание работы сетевой службы. Например, при реализации почтовой службы на диске клиента может храниться локальная копия базы данных, содержащей его обширную переписку. В этом случае клиент выполняет большую работу при формировании сообщений в различных форматах, в том числе и сложном мультимедийном, поддерживает ведение адресной книги и выполняет еще много различных вспомогательных работ.

[1] Например, служба каталогов Active Directory компании Microsoft.

Принципиальной же разницей между клиентом и сервером является то, что инициатором выполнения работы сетевой службой всегда выступает клиент, а сервер всегда находится в режиме пассивного ожидания запросов. Например, почтовый сервер осуществляет доставку почты на компьютер пользователя только при поступлении запроса от почтового клиента.

Обычно взаимодействие между клиентской и серверной частями стандартизуется, так что один тип сервера может быть рассчитан на работу с клиентами разного типа, реализованными различными способами и, возможно, разными производителями. Единственное условие для этого — клиенты и сервер должны поддерживать общий стандартный протокол взаимодействия.

Встроенные сетевые службы и сетевые оболочки

На практике сложилось несколько подходов к построению сетевых операционных систем, различающихся глубиной внедрения сетевых служб в операционную систему (рис. 2.3):

- сетевые службы объединены в виде некоторого набора — *оболочки*;
- сетевые службы производятся и поставляются в виде *отдельного продукта*;
- сетевые службы глубоко *встроены* в ОС.

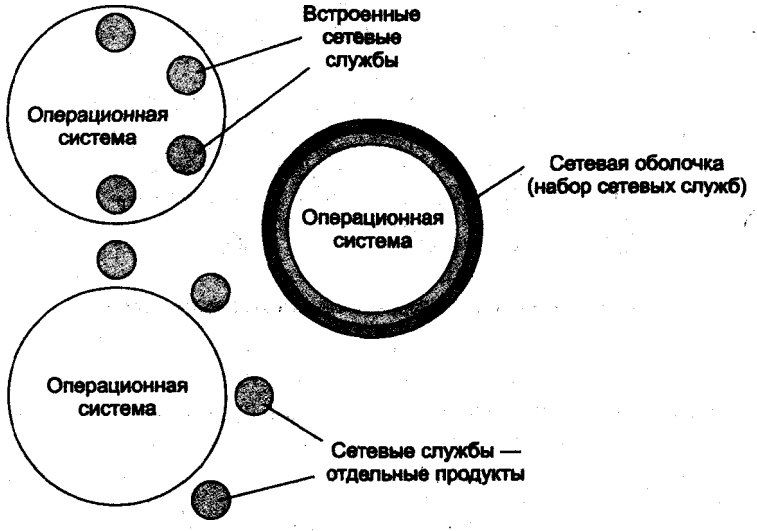


Рис. 2.3. Варианты построения сетевых ОС

Первый подход был характерен для первых сетевых ОС, которые создавались как совокупность уже существующей локальной ОС и надстроенной над ней сетевой оболочки. Сетевая оболочка представляет собой набор согласованных между собой сетевых служб, как правило, оформленных как самостоятельный программный продукт. Сетевые оболочки могут подразделяться на клиентские и серверные. Оболочка, которая преимущественно содержит клиентские части сетевых служб, называется клиентской. Серверная оболочка, как мини-

мум, содержит серверные компоненты двух основных сетевых служб — файловой службы и службы печати[1].

Однако в дальнейшем разработчики сетевых ОС посчитали более эффективным подход, при котором сетевая ОС с самого начала работы над ней задумывается и проектируется специально для сети. Сетевые функции у этих ОС глубоко встраиваются в основные модули системы, что обеспечивает логическую стройность ОС, а также потенциально более высокую производительность. При таком подходе отсутствует избыточность. Если все сетевые службы хорошо интегрированы и рассматриваются как неотъемлемые части ОС, то все внутренние механизмы такой операционной системы могут быть оптимизированы для выполнения сетевых функций.

Одноранговые и серверные сетевые операционные системы

В зависимости от того, как распределены функции между компьютерами сети, они могут выступать в трех разных ролях:

- компьютер, занимающийся исключительно обслуживанием запросов других компьютеров, играет роль **выделенного сервера** сети;
- компьютер, обращающийся с запросами к ресурсам другой машины, исполняет роль **клиентского узла**;
- компьютер, совмещающий функции клиента и сервера, является **одноранговым узлом**.

Очевидно, что сеть не может состоять только из клиентских или только из серверных узлов. Сеть, оправдывающая свое назначение и обеспечивающая взаимодействие компьютеров, может быть построена по одной из трех следующих схем:

- сеть на основе одноранговых узлов — **одноранговая сеть**;
- сеть на основе клиентов и выделенных серверов — **сеть с выделенными серверами**;
- сеть, включающая узлы всех типов, — **гибридная сеть**.

Каждая из этих схем обладает своими достоинствами и недостатками, определяющими их области применения.

ОС в одноранговых сетях

В одноранговых сетях все компьютеры равны в возможностях доступа к ресурсам друг друга (рис. 2.4). Каждый пользователь может по своему желанию объ-

[1] В качестве примера серверной сетевой оболочки можно указать популярный в начале 90-х годов продукт компании Microsoft LAN Server, который существовал в различных вариантах, ориентированных на операционные системы VAX VMS, VM, OS/400, AIX, OS/2. Не менее популярен в то время был типичный набор программного обеспечения рабочей станции в сети NetWare, представляющий собой систему MS-DOS с установленной поверх нее клиентской оболочкой, состоящей из клиентских частей файловой службы и службы печати ОС NetWare.

явить какой-либо ресурс своего компьютера разделяемым, после чего другие пользователи могут его задействовать. В одноранговых сетях на всех компьютерах устанавливается такая операционная система, которая предоставляет всем компьютерам в сети *потенциально* равные возможности. Сетевые операционные системы такого типа называются **одноранговыми ОС**. Очевидно, что одноранговые ОС должны включать как серверные, так и клиентские компоненты сетевых служб (на рисунке они обозначены буквами соответственно *С* и *К*). Примерами одноранговых ОС могут служить Windows 2000 Professional, Windows XP/Vista.

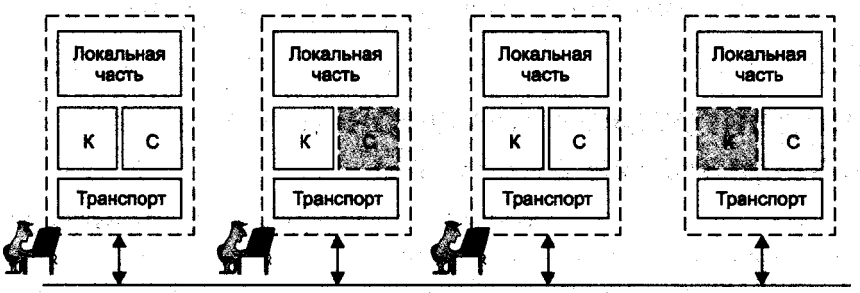


Рис. 2.4. Одноранговая сеть

При потенциальном равноправии всех компьютеров в одноранговой сети часто возникает функциональная несимметричность. Обычно в сети имеются пользователи, которые не желают предоставлять свои ресурсы для совместной работы. В таком случае серверные возможности их операционных систем не активизируются, и компьютеры исполняют роль «чистых» клиентов (на рисунке неиспользуемые компоненты ОС изображены затемненными).

В то же время администратор может закрепить за некоторыми компьютерами сети только функции по обслуживанию запросов остальных компьютеров, превратив их, таким образом, в «чистые» серверы, за которыми не работают пользователи. В такой конфигурации одноранговые сети становятся похожими на сети с выделенными серверами, но это только внешняя схожесть — между этими двумя типами сетей остается существенное внутреннее различие. Изначально в одноранговых сетях специализация ОС не зависит от того, какую функциональную роль исполняет компьютер — клиента или сервера. Изменение роли компьютера в одноранговой сети достигается за счет того, что функции серверной или клиентской части просто не используются.

Одноранговые сети проще в организации и эксплуатации, по этой схеме работают небольшие сети, в которых количество компьютеров не превышает 10–20. В этом случае нет необходимости в применении централизованных средств администрирования — нескольким пользователям нетрудно договориться между собой о перечне разделяемых ресурсов и паролях доступа к ним.

Однако в больших сетях средства централизованного администрирования, хранения и обработки данных, а особенно защиты данных становятся необходимыми, и такие средства легче реализовать в сетях с выделенными серверами.

ОС в сетях с выделенными серверами

В сетях с выделенными серверами используются специальные варианты сетевых ОС, которые оптимизированы для роли серверов и называются **серверными ОС** (рис. 2.5). Пользовательские компьютеры в этих сетях работают под управлением **клиентских ОС**.

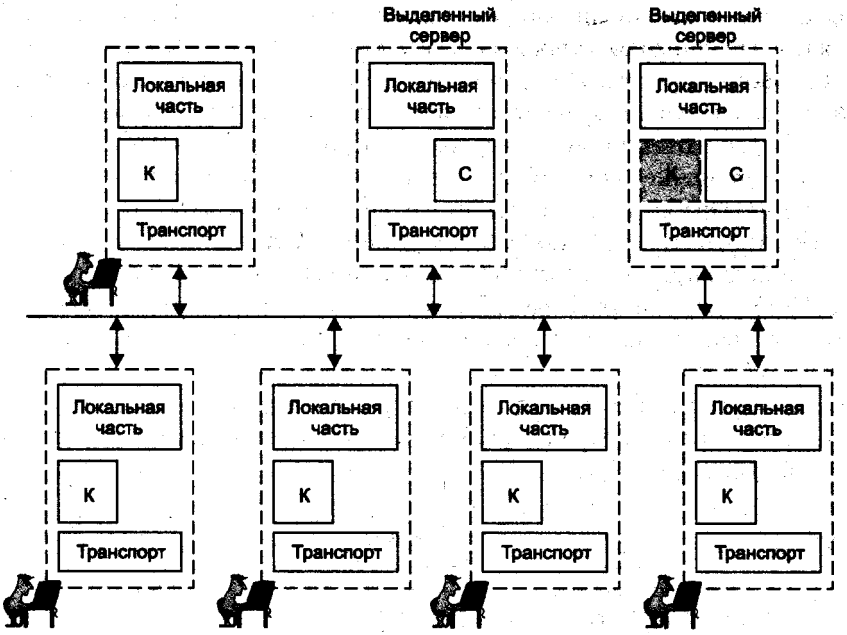


Рис. 2.5. Сеть с выделенными серверами

Специализация операционной системы для работы в качестве сервера является естественным способом повышения эффективности серверных операций. А необходимость такого повышения часто ощущается весьма остро, особенно в крупной сети. При существовании в сети сотен или даже тысяч пользователей интенсивность запросов к общим ресурсам может быть очень большой, и сервер должен справляться с этим потоком запросов без больших задержек. Очевидным решением этой проблемы является использование в качестве сервера компьютера с мощной аппаратной платформой и операционной системой, оптимизированной для серверных функций.

Чем меньше функций выполняет ОС, тем более эффективно можно их реализовать, поэтому для оптимизации серверных операций разработчики ОС вынуждены ущемлять некоторые другие ее функции, причем иногда вплоть до полного их отбрасывания.

Ярким примером такого подхода являются популярные в 90-х годах серверные ОС NetWare 3.x и 4.x компании Novell. Разработчики этих ОС ставили перед собой цель оптимизировать выполнение файлового сервиса и сервиса печати.

Для этого они полностью исключили из системы многие элементы, важные для универсальной ОС, в частности графический интерфейс пользователя, поддержку универсальных приложений, защиту приложений мультипрограммного режима друг от друга, механизм виртуальной памяти. Все это позволило добиться уникальной скорости файлового доступа и вывело семейство NetWare в лидеры серверных ОС того времени.

Однако слишком узкая специализация некоторых серверных ОС является одновременно их слабой стороной. Так, отсутствие в ОС NetWare универсального интерфейса программирования и средств защиты приложений затрудняло ее применение в качестве среды для выполнения приложений, приводило к необходимости включения в сеть других серверных ОС в тех случаях, когда требовалось выполнение функций, отличных от файлового сервиса и сервиса печати. Поэтому разработчики компании Novell в следующих версиях NetWare 5.x и 6.x отказались от функциональной ограниченности и, последовав примеру создателей большинства серверных операционных систем (Windows Server NT/2000/2003, Solaris, Linux, FreeBSD), включили в состав своих серверных ОС все компоненты, позволяющие использовать их в качестве универсального сервера.

В больших сетях наряду с отношениями клиент-сервер сохраняется необходимость и в одноранговых связях, поэтому такие сети чаще всего строятся по гибридной схеме (рис. 2.6).

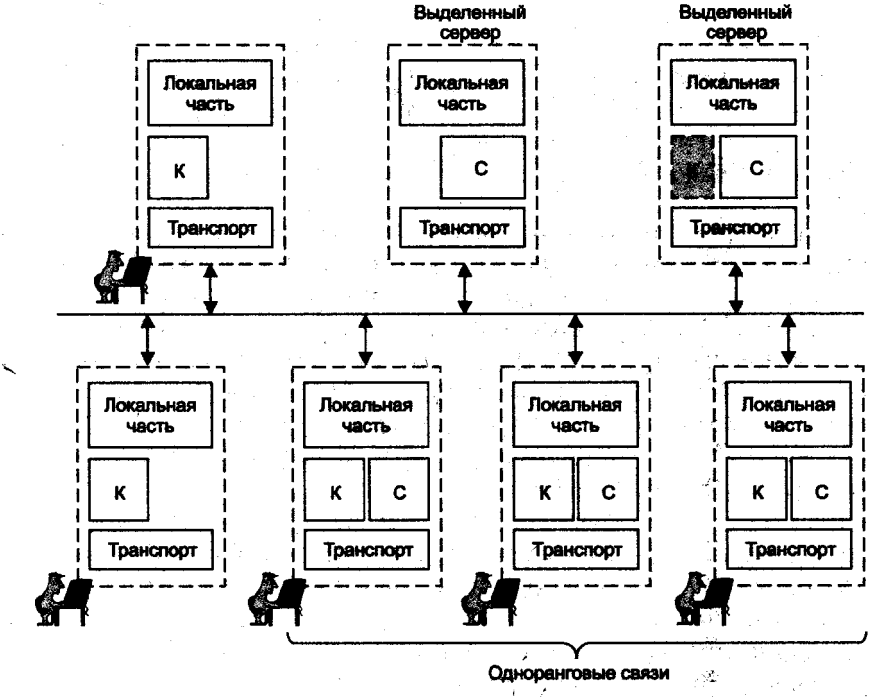


Рис. 2.6. Гибридная сеть

Требования к современным операционным системам

Главным требованием, предъявляемым к операционной системе, является выполнение ею основных функций эффективного управления ресурсами и обеспечение удобного интерфейса для пользователя и прикладных программ. Современная ОС, как правило, должна поддерживать мультипрограммную обработку, виртуальную память, свопинг, многооконный графический интерфейс пользователя, а также выполнять многие другие необходимые функции и услуги. Помимо этих требований функциональной полноты, к операционным системам предъявляются не менее важные эксплуатационные требования, которые перечислены далее.

- *Расширяемость*. В то время как аппаратная часть компьютера устаревает за несколько лет, полезная жизнь операционных систем может измеряться десятилетиями. Примером может служить ОС Unix. Поэтому операционные системы всегда изменяются со временем эволюционно, и эти изменения более значимы, чем изменения аппаратных средств. Изменения ОС обычно заключаются в приобретении ею новых свойств, например поддержке новых типов внешних устройств или новых сетевых технологий. Если код ОС написан таким образом, что дополнения и изменения могут вноситься без нарушения целостности системы, то такую ОС называют расширяемой. Расширяемость достигается за счет модульной структуры ОС, при которой программы строятся из набора отдельных модулей, взаимодействующих только через функциональный интерфейс.
- *Переносимость*. В идеале код ОС должен легко переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которые различаются не только типом процессора, но и способом организации всей аппаратуры компьютера) одного типа на аппаратную платформу другого типа. Переносимые ОС имеют несколько вариантов реализации для разных платформ, такое свойство ОС называют также **многоплатформенностью**.
- *Совместимость*. Существует несколько «долгоживущих» популярных операционных систем (разновидности Unix/Linux, Windows, NetWare), для которых наработана широкая номенклатура популярных приложений. Для пользователя, переходящего по тем или иным причинам с одной ОС на другую, очень привлекательна возможность запуска в новой операционной системе привычного приложения. Если ОС имеет средства для выполнения прикладных программ, написанных для других операционных систем, то про нее говорят, что она обладает совместимостью с этими ОС. Следует отличать совместимость на уровне двоичных кодов от совместимости на уровне исходных текстов. Понятие совместимости включает также поддержку пользовательских интерфейсов других ОС.
- *Надежность* и *отказоустойчивость*. Система должна быть защищена как от внутренних, так и от внешних ошибок, сбоев и отказов. Ее действия должны

быть всегда предсказуемыми, а приложения не должны быть способны нанести вред ОС. Надежность и отказоустойчивость ОС, прежде всего, определяется архитектурными решениями, положенными в ее основу, а также качеством ее реализации (отлаженностью кода). Кроме того, важно, включает ли ОС программную поддержку аппаратных средств обеспечения отказоустойчивости, таких, например, как дисковые массивы или источники бесперебойного питания.

- *Безопасность.* Современная ОС должна защищать данные и другие ресурсы вычислительной системы от несанкционированного доступа. Чтобы ОС обладала свойством безопасности, она должна, как минимум, иметь в своем составе средства аутентификации — определения легальности пользователей, авторизации — предоставления легальным пользователям дифференцированных прав доступа к ресурсам, аудита — фиксации всех «подозрительных» для безопасности системы событий. Свойство безопасности особенно важно для сетевых ОС. В таких ОС к задаче контроля доступа добавляется задача защиты данных, передаваемых по сети.
- *Производительность.* Операционная система должна обладать настолько хорошим быстродействием и временем реакции, насколько это позволяет аппаратная платформа. На производительность ОС влияет много факторов, среди которых основными являются архитектура ОС, многообразие функций, качество программирования кода, возможность исполнения ОС на высокопроизводительной (многопроцессорной) платформе.

Выводы

- ОС — это комплекс взаимосвязанных программ, предназначенный для повышения эффективности аппаратуры компьютера путем рационального управления его ресурсами, а также для обеспечения удобства пользователя за счет предоставления ему расширенной виртуальной машины.
- К числу основных ресурсов, управление которыми осуществляет ОС, относятся процессоры, основная память, таймеры, наборы данных, диски, накопители на магнитных лентах, принтеры, сетевые устройства и некоторые другие. Ресурсы распределяются между процессами. Для решения задач управления ресурсами разные ОС используют различные алгоритмы, особенности которых, в конечном счете, и определяют облик ОС.
- Наиболее важными подсистемами ОС являются подсистемы управления процессами, памятью, файлами и внешними устройствами, а также подсистемы пользовательского интерфейса, защиты данных и администрирования.
- Прикладному программисту возможности ОС доступны в виде набора функций, составляющих интерфейс прикладного программирования (API).
- Термин «сетевая операционная система» используется в двух смыслах: во-первых, как совокупность ОС всех компьютеров сети и, во-вторых, как ОС отдельного компьютера, способного работать в сети.

- К основным функциональным компонентам сетевой ОС относятся средства управления локальными ресурсами и сетевые средства. Последние, в свою очередь, можно разделить на три компонента: средства предоставления локальных ресурсов и услуг в общее пользование (серверная часть ОС), средства запроса доступа к удаленным ресурсам и услугам (клиентская часть ОС, или редиректор) и транспортные средства ОС (совместно с коммуникационной системой обеспечивают передачу сообщений между компьютерами сети).
- Совокупность серверной и клиентской частей, предоставляющих доступ к конкретному типу ресурса компьютера через сеть, называется сетевой службой. Сетевая служба предоставляет пользователям сети набор услуг — сетевой сервис. Каждая служба связана с определенным типом сетевых ресурсов и/или определенным способом доступа к этим ресурсам. Сетевые службы могут быть либо встроены в ОС, либо реализованы в виде программной оболочки.
- В зависимости от того, как распределены функции между компьютерами сети, они могут выступать в трех разных ролях. Компьютер, занимающийся исключительно обслуживанием запросов других компьютеров, играет роль выделенного сервера сети. Компьютер, обращающийся с запросами к ресурсам другой машины, исполняет роль клиентского узла. Компьютер, совмещающий функции клиента и сервера, является одноранговым узлом.
- Одноранговые сети состоят только из одноранговых узлов. При этом все компьютеры в сети имеют потенциально равные возможности. Одноранговые ОС включают как серверные, так и клиентские компоненты сетевых служб. Одноранговые сети проще в организации и эксплуатации, по этой схеме организуется работа в небольших сетях, в которых количество компьютеров не превышает 10–20.
- В сетях с выделенными серверами используются специальные варианты сетевых ОС, оптимизированные для роли либо серверов, либо клиентов. Для серверных ОС характерны поддержка мощных аппаратных платформ, в том числе мультипроцессорных, широкий набор сетевых служб, поддержка большого числа одновременно выполняемых процессов и сетевых соединений, наличие развитых средств защиты и средств централизованного администрирования сети. Клиентские ОС, в общем случае являясь более простыми, должны обеспечивать удобный пользовательский интерфейс и набор редиректоров, позволяющий получать доступ к разнообразным сетевым ресурсам.
- В число требований, предъявляемых сегодня к сетевым ОС, входят: функциональная полнота и эффективность управления ресурсами, модульность и расширяемость, переносимость и многоплатформенность, совместимость на уровне приложений и пользовательских интерфейсов, надежность и отказоустойчивость, безопасность и производительность.

Задачи и упражнения

1. Поясните определение операционной системы как расширенной машины.
2. В соответствии с определением ОС ее главными функциями являются предоставление услуг пользователю и эффективное управление ресурсами компьютера. Какая их этих двух функций должна была доминировать в мультипрограммных ОС времен IBM/360? А в первых ОС для персональных компьютеров?
3. В чем состоит отличие в виртуальных машинах, предоставляемых операционной системой простому пользователю и прикладному программисту?
4. Сравните интерфейс прикладного программиста с операционной системой и интерфейс системного программиста с реальной аппаратурой. Что можно сказать о разнообразии и мощности интерфейсных функций, имеющихся в распоряжении каждого из них?
5. Назовите абстрактно сформулированные задачи ОС по управлению любым типом ресурса. Конкретизируйте эти задачи применительно к процессору, памяти, внешним устройствам.
6. Вставьте пропущенные определения: «Пользователю ... ОС не требуется знать, на каком из компьютеров сети хранятся файлы, с которыми он работает, а пользователю ... ОС эти сведения обычно необходимы».
7. Какие из утверждений верны:
   1) «сетевая операционная система» — это совокупность операционных систем всех компьютеров сети;
   2) «сетевая операционная система» — это операционная система отдельного компьютера, способного работать в сети;
   3) «сетевая операционная система» — это набор сетевых служб, выполненный в виде оболочки.
8. Какой минимум функциональных возможностей надо добавить к локальной ОС, чтобы она стала сетевой?
9. Перечислите основные сетевые службы. Какие из них, как правило, встроены в операционную систему?
10. Какие из утверждений верны:
    1) редиректор — клиентская часть сетевой службы;
    2) редиректор — модуль, входящий в состав клиентской части сетевой службы, распознающий и перенаправляющий запросы к нужному сетевому серверу или локальной ОС.
11. Поясните значение следующих терминов применительно к сетевым ОС: «сервис», «сервер», «клиент», «служба», «оболочка», «услуга», «редиректор». Какие из них употребляются как синонимы?
12. Может ли сетевая оболочка работать поверх сетевой ОС?

13. В каких случаях может оказаться полезным наличие сразу нескольких серверных (клиентских) частей файловых служб?
14. Какие из следующих утверждений верны:
    1) ОС выделенного сервера никогда не содержит клиентских частей сетевых служб;
    2) в одноранговых ОС всегда имеются и клиентские, и серверные части сетевых служб;
    3) в сетях с выделенными серверами могут поддерживаться одноранговые связи.
15. Может ли выделенный сервер обращаться с запросами к ресурсам клиентских станций?
16. Приведите примеры одноранговых ОС и ОС с выделенным сервером.

Глава 3

Архитектура операционной системы

Любая сложная система должна иметь понятную и рациональную структуру, то есть разделяться на части — модули, имеющие вполне законченное функциональное назначение с четко оговоренными правилами взаимодействия. Ясное понимание роли каждого отдельного модуля существенно упрощает работу по модификации и развитию системы. Напротив, сложную систему без хорошей структуры чаще проще разработать заново, чем модернизировать.

Функциональная сложность операционной системы неизбежно приводит к сложности ее архитектуры, под которой понимают структурную организацию ОС на основе различных программных модулей. Обычно в состав ОС входят исполняемые и объектные модули стандартных для данной ОС форматов, библиотеки разных типов, модули исходного текста программ, программные модули специального формата (например, загрузчик ОС, драйверы ввода-вывода), конфигурационные файлы, файлы документации, модули справочной системы и т. д.

Большинство современных операционных систем представляют собой хорошо структурированные модульные системы, способные к развитию, расширению и переносу на новые платформы. Какой-либо единой архитектуры ОС не существует, но существуют универсальные подходы к структурированию ОС.

Ядро и вспомогательные модули ОС

Наиболее общим подходом к структуризации операционной системы является разделение всех ее модулей на две группы:

- ядро — модули, выполняющие основные функции ОС;
- вспомогательные модули ОС.

Модули **ядра** выполняют базовые функции ОС, связанные с управлением процессами, памятью, устройствами ввода-вывода и т. п. Именно ядро занимается переключением контекстов, загрузкой/выгрузкой страниц, обработкой пре-

рываний. Непосредственное выполнение такого рода действий недоступно для приложений. При необходимости они могут обращаться к ядру с системными вызовами, используя для этого имеющийся в их распоряжении интерфейс прикладного программирования — API.

Функции, отнесенные в ведение ядра, являются наиболее часто используемыми функциями операционной системы, поэтому скорость их выполнения определяет производительность системы в целом. Для обеспечения высокой скорости работы ОС все модули ядра или большая их часть постоянно находятся в оперативной памяти, то есть являются *резидентными*.

Обычно ядро оформляется в виде программного модуля некоторого специального формата, отличающегося от формата пользовательских приложений.

Ядро является движущей силой всех вычислительных процессов в компьютерной системе, и крах ядра равносилен краху всей системы. Поэтому разработчики операционной системы уделяют особое внимание надежности кодов ядра.

ПРИМЕЧАНИЕ Термин «ядро» в разных ОС трактуется по-разному. Одним из определяющих свойств ядра является работа в привилегированном режиме. Этот вопрос рассматривается в следующем разделе.

Вспомогательные модули ОС выполняют весьма полезные, но менее обязательные функции. Например, к таким модулям могут быть отнесены программы архивирования данных на магнитной ленте, дефрагментации диска, текстового редактора. Вспомогательные модули ОС оформляются либо в виде приложений, либо в виде библиотек процедур.

Поскольку некоторые компоненты ОС оформлены как обычные приложения, то есть в виде исполняемых модулей стандартного для данной ОС формата, то часто бывает очень сложно провести четкую грань между операционной системой и приложениями (рис. 3.1).

Решение о том, должна ли какая-либо программа стать частью ОС или нет, принимает производитель ОС. Среди многих факторов, способных повлиять на это решение, немаловажными являются перспективы того, будет ли программа иметь массовый спрос у потенциальных пользователей данной ОС.

Некоторая программа может существовать определенное время как пользовательское приложение, а потом стать частью ОС, и наоборот. Ярким примером такого изменения статуса программы является веб-браузер компании Microsoft, который сначала поставлялся как отдельное приложение, затем стал частью операционных систем Windows.

Вспомогательные модули ОС обычно подразделяются на следующие группы:

- *утилиты* — программы, решающие отдельные задачи управления и сопровождения компьютерной системы, такие, например, как программы сжатия дисков, архивирования данных на магнитную ленту;
- *системные обрабатывающие программы* — текстовые или графические редакторы, компиляторы, компоновщики, отладчики;

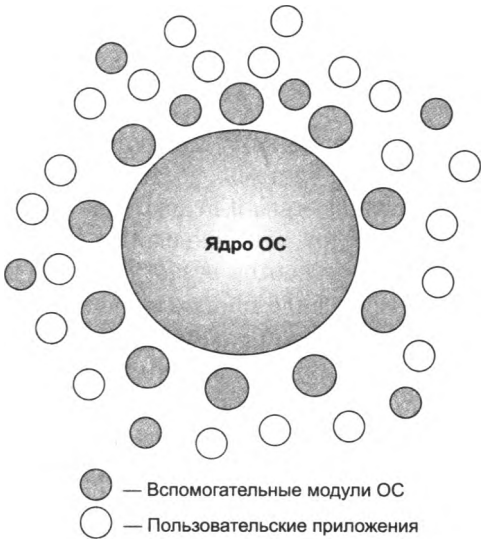


Рис. 3.1. Нечеткость границы между ОС и приложениями

- *программы предоставления пользователю дополнительных услуг* — специальный вариант пользовательского интерфейса, калькулятор и даже игры;
- *библиотеки процедур* различного назначения, упрощающие разработку приложений, например библиотека математических функций, функций ввода-вывода и т. д.

Как и обычные приложения, для выполнения своих задач утилиты, обрабатывающие программы и библиотеки ОС, обращаются к функциям ядра посредством системных вызовов (рис. 3.2).

Разделение операционной системы на ядро и модули-приложения обеспечивает легкую расширяемость ОС. Чтобы добавить новую высокоуровневую функцию, достаточно разработать новое приложение и при этом не требуется модифицировать важные функции, образующие ядро системы. Что же касается внесения изменений в функции ядра, то это может оказаться гораздо сложнее, причем сложность зависит от структурной организации самого ядра. В некоторых случаях каждое исправление ядра может потребовать его полной перекомпиляции.

Модули ОС, оформленные в виде утилит, системных обрабатывающих программ и библиотек, обычно загружаются в оперативную память только на время выполнения своих функций, то есть являются *транзитными*. Постоянно в оперативной памяти располагаются только самые необходимые коды ОС, составляющие ее ядро. Такая организация ОС экономит оперативную память компьютера.

Важным свойством архитектуры ОС, основанной на ядре, является возможность защиты кодов и данных операционной системы за счет выполнения функций ядра в привилегированном режиме.

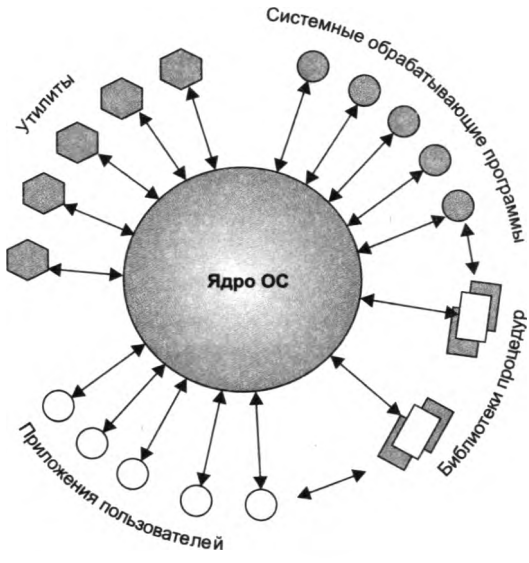


Рис. 3.2. Взаимодействие между ядром и вспомогательными модулями ОС

Ядро в привилегированном режиме

Для надежного управления ходом выполнения приложений операционная система должна иметь по отношению к приложениям определенные привилегии. Иначе некорректно работающее приложение может вмешаться в работу ОС и, например, разрушить часть ее кодов. Все усилия разработчиков операционной системы окажутся напрасными, если их решения воплощены в незащищенные от приложений модули системы, какими бы элегантными и эффективными эти решения ни были. Операционная система должна обладать исключительными полномочиями также для того, чтобы играть роль арбитра в споре приложений за ресурсы компьютера в мультипрограммном режиме. Ни одно приложение не должно иметь возможности без ведома ОС получать дополнительные области памяти, занимать процессор дольше разрешенного операционной системой периода времени, непосредственно управлять совместно используемыми внешними устройствами.

Обеспечить привилегии операционной системе невозможно без специальных средств аппаратной поддержки. Аппаратура компьютера должна поддерживать как минимум два режима работы — **пользовательский** (user mode) и **привилегированный**, который также называют **режимом ядра** (kernel mode), или **супервизора** (supervisor mode). Подразумевается, что операционная система или некоторые ее части работают в привилегированном режиме, а приложения — в пользовательском режиме.

Так как основные функции ОС выполняются ядром, то чаще всего именно ядро становится той частью ОС, которая работает в привилегированном режиме

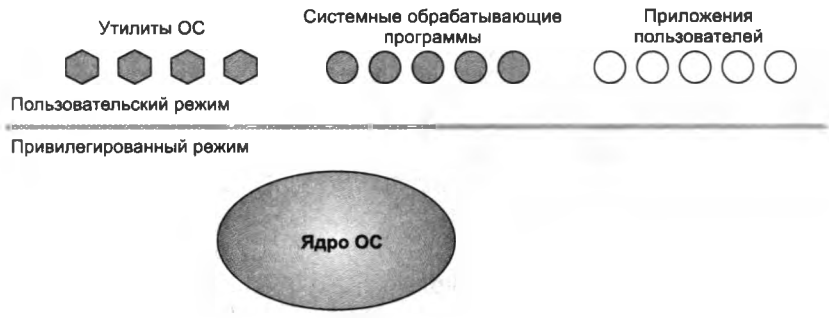


Рис. 3.3. Архитектура операционной системы с ядром в привилегированном режиме

(рис. 3.3). Иногда это свойство — работа в привилегированном режиме — служит основным определением понятия «ядро».

Приложения ставятся в подчиненное положение за счет запрета для них выполнения в пользовательском режиме некоторых критичных команд (инструкций), связанных с переключением процессора с задачи на задачу, управлением устройствами ввода-вывода, доступом к механизмам распределения и защиты памяти. Выполнение некоторых команд в пользовательском режиме запрещается безусловно (очевидно, что к таким командам относится команда перехода в привилегированный режим), тогда как другие запрещается выполнять только при определенных условиях. Например, команды ввода-вывода могут быть запрещены приложениям при доступе к контроллеру жесткого диска, который хранит данные, общие для ОС и всех приложений, но разрешены при доступе к последовательному порту, выделенному в монопольное владение определенного приложения. Важно, что условия разрешения выполнения критичных команд находятся под полным контролем ОС, и этот контроль обеспечивается за счет набора команд, безусловно запрещенных для пользовательского режима.

Аналогичным образом обеспечиваются привилегии ОС при доступе к памяти. Например, выполнение команды доступа к памяти для приложения разрешается, если она обращается к области памяти, отведенной данному приложению операционной системой, и запрещается при обращении к областям памяти, занимаемым ОС или другими приложениями. Полный контроль ОС над доступом к памяти достигается за счет того, что команды конфигурирования механизмов защиты памяти (например, изменения ключей защиты памяти в мэйнфреймах IBM или указателя таблицы дескрипторов памяти в процессорах Pentium) разрешается выполнять только в привилегированном режиме.

Очень важно, что механизмы защиты памяти используются операционной системой не только для защиты своих областей памяти от приложений, но и для защиты областей памяти, выделенных ОС какому-либо приложению, от остальных приложений. Говорят, что каждое приложение работает в своем **адресном пространстве**. Это свойство позволяет локализовать некорректно работающее приложение в собственной области памяти, так что его ошибки не оказывают влияния на остальные приложения и операционную систему.

Между количеством уровней привилегий, реализуемых аппаратно, и количеством уровней привилегий, поддерживаемых ОС, нет прямого соответствия. Так, в свое время на базе четырех уровней, поддерживаемых процессорами компании Intel, операционная система OS/2 реализовала трехуровневую систему привилегий, а операционные системы семейств Windows NT и Unix — двухуровневую. В то же время если аппаратура поддерживает хотя бы два уровня привилегий, то ОС может на этой основе создать программным способом сколь угодно развитую систему защиты. Эта система может, например, поддерживать несколько уровней привилегий, образующих иерархию. Наличие нескольких уровней привилегий позволяет более тонко распределять полномочия как между модулями операционной системы, так и между самими приложениями. Появление внутри операционной системы более привилегированных и менее привилегированных частей дает возможность повысить устойчивость ОС к внутренним ошибкам программных кодов, так как такие ошибки будут распространяться только внутри модулей с определенным уровнем привилегий. Дифференциация привилегий в среде прикладных модулей позволяет строить сложные прикладные комплексы, в которых часть более привилегированных модулей может, например, получать доступ к данным менее привилегированных модулей и управлять их выполнением.

На основе двух режимов привилегий процессора ОС может построить сложную систему индивидуальной защиты ресурсов, примером которой является типичная система защиты файлов и каталогов. Такая система позволяет задать для любого пользователя определенные права доступа к каждому из файлов и каталогов.

Повышение устойчивости операционной системы, обеспечиваемое переходом ядра в привилегированный режим, достигается за счет некоторого замедления выполнения системных вызовов. Системный вызов инициирует переключение процессора из пользовательского режима в привилегированный, а при возврате к приложению — переключение из привилегированного режима в пользовательский (рис. 3.4). Во всех типах процессоров из-за дополнительной двукратной задержки переключения переход на процедуру со сменой режима выполняется медленнее, чем вызов процедуры без смены режима.

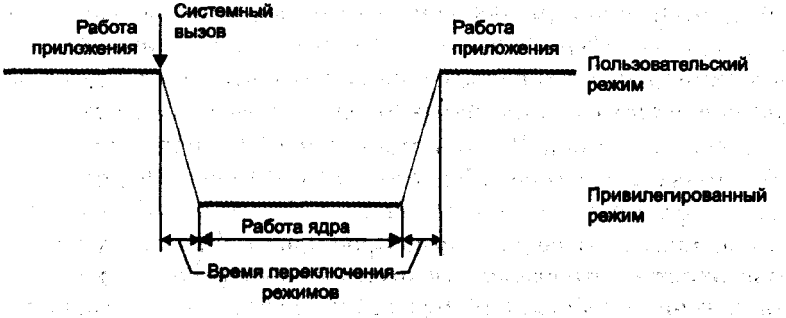


Рис. 3.4. Смена режимов при выполнении системного вызова к привилегированному ядру

Архитектура ОС, основанная на привилегированном ядре и приложениях пользовательского режима, стала, по существу, классической. Ее используют большинство популярных операционных систем, в том числе семейства ОС Windows NT[1], Unix/Linux, Mac OS X, операционные системы мэйнфреймов.

Однако в некоторых случаях разработчики ОС отступают от этого классического варианта архитектуры, организуя работу ядра и приложений в одном и том же режиме. Так, в известных специализированных ОС NetWare 3.x и 4.x компании Novell привилегированный режим процессоров используется как для работы ядра, так и для работы специфических приложений — загружаемых модулей NLM (рис. 3.5). При таком построении ОС обращения приложений к ядру выполняются быстрее, так как нет переключения режимов, однако при этом отсутствует надежная аппаратная защита памяти, занимаемой модулями ОС, от некорректно работающего приложения. Разработчики компании Novell пошли на такое потенциальное снижение надежности в этих своих операционных системах, поскольку ограниченный набор специализированных приложений для них позволял компенсировать этот архитектурный недостаток за счет тщательной отладки каждого приложения[2].

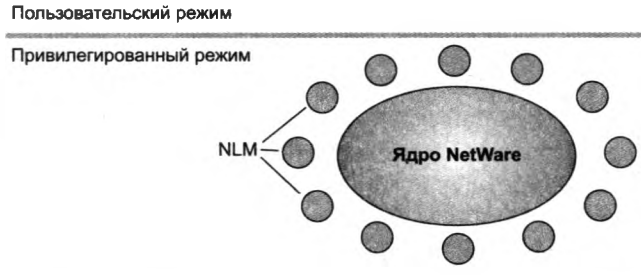


Рис. 3.5. Упрощенная архитектура операционной системы NetWare

Аналогичный подход используется в самой популярной ОС маршрутизаторов — Cisco IOS (Internetwork Operating System). Эта специализированная ОС также состоит из ядра и приложений, которые выполняются в одном и том же режиме процессора. При этом время переключения системы между ядром и приложениями сокращается, а это очень важно для системы реального времени IOS, главной задачей которой является обработка поступающих в маршрутизатор по входным интерфейсам пакетов с минимальными задержками.

В одном режиме работают также ядро и приложения тех операционных систем, которые разработаны для процессоров, вообще не поддерживающих привилегированного режима работы. Наиболее популярным процессором такого типа был процессор Intel 8088/86, послуживший основой для персональных

[1] Под семейством Windows NT здесь и далее имеется в виду следующий ряд операционных систем компании Microsoft, имеющих очень близкую архитектуру: Windows NT 3.1, Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003, Windows Vista.

[2] В последних версиях Novell NetWare 5.x и 6.x разработчики отказались от этого подхода.

компьютеров компании IBM. Операционная система MS-DOS, разработанная компанией Microsoft для этих компьютеров, состояла из двух модулей msdos.sys и io.sys, составлявших ядро системы (хотя название «ядро» для этих модулей не употреблялось, по своей сути они являлись ядром), к которым с системными вызовами обращались командный интерпретатор command.com, системные утилиты и приложения. Архитектура MS-DOS соответствует архитектуре ОС, приведенной на рис. 3.2. Некорректно написанные приложения вполне могли разрушить основные модули MS-DOS, что иногда и происходило, но область использования MS-DOS (и многих подобных ей ранних операционных систем для персональных компьютеров, таких как MSX, CP/M) не предъявляла высоких требований к надежности ОС.

Многослойная структура ОС

Вычислительную систему, работающую под управлением ОС на основе ядра, можно рассматривать как состоящую из трех иерархически расположенных слоев: нижний слой образует аппаратура, промежуточный — ядро, а утилиты, обрабатывающие программы и приложения, составляют верхний слой системы (рис. 3.6). Слоистую структуру вычислительной системы принято изображать в виде системы концентрических окружностей, иллюстрируя тот факт, что каждый слой может взаимодействовать только со смежными слоями. Действительно, при такой организации ОС приложения не могут непосредственно взаимодействовать с аппаратурой, а только через слой ядра.

Рис. 3.6. Трехслойная схема вычислительной системы

Многослойный подход является универсальным и эффективным способом декомпозиции сложных систем любого типа, в том числе программных. В соответствии с этим подходом система состоит из иерархии слоев. Каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс (рис. 3.7). На основе функций нижележащего слоя следующий (вверх по иерархии) слой строит свои функции — более сложные и более мощные, которые, в свою очередь, оказываются примитивами для создания еще более мощных функций вышележащего слоя. Строгие правила касаются только взаимодействия между слоями системы, а между

модулями внутри слоя связи могут быть произвольными. Отдельный модуль может выполнить свою работу либо самостоятельно, либо обратиться к другому модулю своего слоя, либо обратиться за помощью к нижележащему слою через межслойный интерфейс.

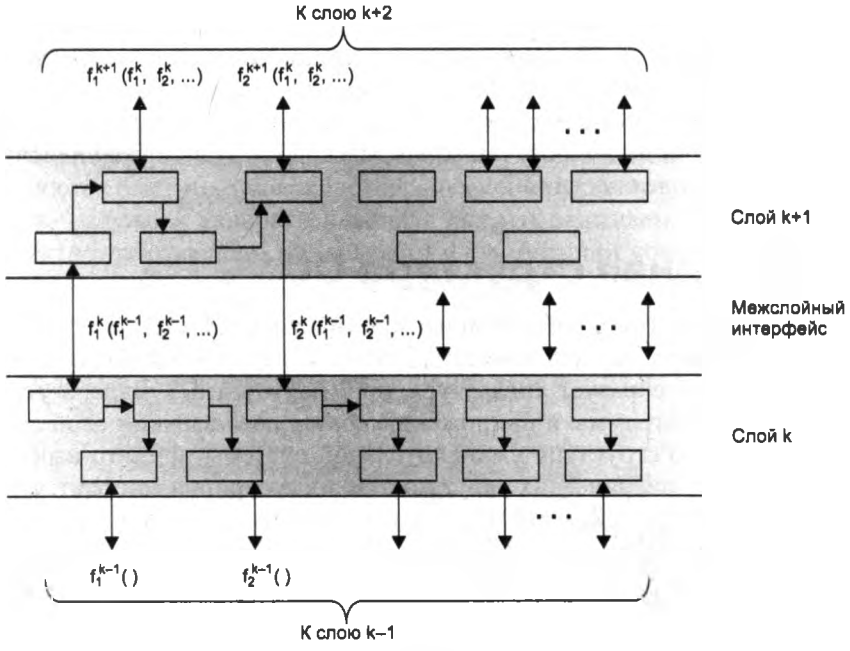


Рис. 3.7. Концепция многослойного взаимодействия

Такая организация системы имеет много достоинств. Она существенно упрощает разработку системы, так как позволяет сначала определить «сверху вниз» функции слоев и межслойные интерфейсы, а затем при детальной реализации постепенно наращивать возможности функций слоев, двигаясь «снизу вверх». Кроме того, при модернизации системы можно изменять модули внутри слоя без необходимости производить какие-либо изменения в остальных слоях, если при этих внутренних изменениях межслойные интерфейсы остаются в силе.

Поскольку ядро представляет собой сложный многофункциональный комплекс, то многослойный подход обычно распространяется и на структуру ядра.

Ядро может состоять из следующих слоев (рис. 3.8).

- *Средства аппаратной поддержки ОС.* До сих пор об операционной системе говорилось как о комплексе программ, но, вообще говоря, часть функций ОС может выполняться и аппаратными средствами. Поэтому иногда можно встретить определение операционной системы как совокупности программных и аппаратных средств, что и отражено на рис. 3.8. К операционной системе относят, естественно, не все аппаратные устройства компьютера, а только

средства аппаратной поддержки ОС, то есть те, которые прямо участвуют в организации вычислительных процессов: средства поддержки привилегированного режима, систему прерываний, средства переключения контекстов процессов, средства защиты областей памяти и т. п.

Рис. 3.8. Многослойная структура ядра ОС

- *Машинно-зависимые компоненты ОС.* Этот слой образуют программные модули, в которых отражается специфика аппаратной платформы компьютера. В идеале этот слой полностью экранирует вышележащие слои ядра от особенностей аппаратуры, что позволяет разрабатывать вышележащие слои на основе машинно-независимых модулей, существующих в единственном экземпляре для всех типов аппаратных платформ, поддерживаемых данной ОС. Linux, Unix, Mac OS, операционные системы семейства Windows NT — во всех этих ОС имеется четко определенный слой программных модулей, экранирующих особенности аппаратуры[1].
- *Базовые механизмы ядра.* Этот слой выполняет наиболее примитивные операции ядра, такие как программное переключение контекстов процессов, диспетчеризацию прерываний, перемещение страниц из памяти на диск и обратно и т. п. Модули данного слоя не принимают решений о распределении ресурсов — они только отрабатывают принятые «на верху» решения, что и дает повод называть их исполнительными механизмами для модулей верхних слоев. Например, решение о том, что в данный момент нужно прервать выполнение текущего процесса *A* и начать выполнение процесса *B*, прини-

[1] В ОС семейства Windows NT этот слой программного обеспечения носит название Hardware Abstraction Layer (HAL). Еще раз подчеркнем, что под этим семейством в данной книге подразумевается следующий ряд архитектурно близких ОС: Windows NT, Windows 2000, Windows XP Professional, Windows Server 2003 и Vista.

мается менеджером процессов на вышележащем слое, а слою базовых механизмов передается только директива о том, что нужно выполнить переключение контекста текущего процесса на контекст процесса *B*.

- *Менеджеры ресурсов.* Этот слой состоит из мощных функциональных модулей, реализующих стратегические задачи по управлению основными ресурсами вычислительной системы. Обычно на данном слое работают менеджеры (называемые также диспетчерами) процессов, ввода-вывода, файловой системы и оперативной памяти. Разбиение на менеджеры может быть и несколько иным, например, менеджер файловой системы иногда объединяют с менеджером ввода-вывода, а функции управления доступом пользователей к системе в целом и ее отдельным объектам поручают отдельному менеджеру безопасности. Каждый из менеджеров ведет учет свободных и используемых ресурсов определенного типа и планирует их распределение в соответствии с запросами приложений. Например, менеджер виртуальной памяти управляет перемещением страниц из оперативной памяти на диск и обратно. Менеджер должен отслеживать интенсивность обращений к страницам, время пребывания их в памяти, состояния процессов, использующих данные, и многие другие параметры, на основании которых он время от времени принимает решения о том, какие страницы необходимо выгрузить и какие — загрузить. Для исполнения принятых решений менеджер обращается к нижележащему слою базовых механизмов с запросами о загрузке (выгрузке) конкретных страниц. Внутри слоя менеджеров существуют тесные взаимные связи, отражающие тот факт, что для выполнения процессу нужен доступ одновременно к нескольким ресурсам: процессору, области памяти, возможно к определенному файлу или устройству ввода-вывода. Например, при создании процесса менеджер процессов обращается к менеджеру памяти, который должен выделить процессу определенную область памяти для его кодов и данных.
- *Интерфейс системных вызовов.* Этот слой является самым верхним слоем ядра и взаимодействует непосредственно с приложениями и системными утилитами, образуя прикладной программный интерфейс операционной системы. API-функции, обслуживающие системные вызовы, предоставляют доступ к ресурсам системы в удобной и компактной форме без указания деталей их физического расположения. Например, в операционной системе Unix с помощью системного вызова `fd = open("/doc/a.txt", O_RDONLY)` приложение открывает файл a.txt, хранящийся в каталоге /doc, а с помощью системного вызова `read(fd, buffer, count)` читает некоторое количество байтов из этого файла в область своего адресного пространства, имеющую имя `buffer`. Для осуществления таких комплексных действий системные вызовы обычно обращаются за помощью к функциям слоя менеджеров ресурсов, причем для выполнения одного системного вызова может понадобиться несколько таких обращений.

Приведенное разбиение ядра ОС на слои является достаточно условным. В реальной системе количество слоев и распределение функций между ними может быть иным. В системах, предназначенных для аппаратных платформ от-

ного типа, слой машинно-зависимых модулей может сливаться со слоем базовых механизмов и, частично, со слоем менеджеров ресурсов. Не всегда оформляются в отдельный слой базовые механизмы — в этом случае менеджеры ресурсов не только планируют использование ресурсов, но и самостоятельно реализуют свои планы.

Возможна и противоположная картина, когда ядро состоит из большего количества слоев. Например, менеджеры ресурсов, составляя определенный слой ядра, в свою очередь, могут обладать многослойной структурой. Прежде всего, это относится к менеджеру ввода-вывода, нижний слой которого составляют драйверы устройств, например драйвер жесткого диска или драйвер сетевого адаптера, а верхние слои — драйверы файловых систем или протоколов сетевых служб, имеющие дело с логической организацией информации.

Способ взаимодействия слоев в реальной ОС также может отклоняться от описанной схемы. Для ускорения работы ядра в некоторых случаях происходит непосредственное обращение с верхнего слоя к функциям нижних слоев, минуя промежуточные. Типичным примером такого «неправильного» взаимодействия является начальная стадия обработки системного вызова. На многих аппаратных платформах для реализации системного вызова используется команда программного прерывания. Этим приложение фактически вызывает модуль первичной обработки прерываний, который находится в слое базовых механизмов, а уже этот модуль вызывает нужную функцию из слоя системных вызовов. Сами функции системных вызовов также иногда нарушают субординацию иерархических слоев, обращаясь прямо к базовым механизмам ядра.

Выбор количества слоев ядра является ответственным и сложным делом: увеличение числа слоев ведет к некоторому замедлению работы ядра за счет дополнительных накладных расходов на межслойное взаимодействие, а уменьшение числа слоев ухудшает расширяемость и логичность системы. Обычно операционные системы, прошедшие долгий путь эволюционного развития, например многие ранние версии Unix, имеют неупорядоченное ядро с небольшим числом четко выделенных слоев, а у сравнительно «молодых» операционных систем, таких как Linux, ядро структурировано в гораздо большей степени.

Аппаратная зависимость и переносимость ОС

Многие операционные системы успешно работают на различных аппаратных платформах без существенных изменений в своем составе. Во многом это объясняется тем, что, несмотря на различия в деталях, средства аппаратной поддержки ОС большинства компьютеров приобрели сегодня много типовых черт, а именно эти средства в первую очередь влияют на работу компонентов операционной системы. В результате в ОС можно выделить достаточно компактный слой машинно-зависимых компонентов ядра и сделать остальные слои ОС общими для разных аппаратных платформ.

Типовые средства аппаратной поддержки ОС

Четкой границы между программной и аппаратной реализацией функций ОС не существует — решение о том, какие функции ОС будут выполняться программно, а какие аппаратно, принимаются разработчиками аппаратного и программного обеспечения компьютера. Тем не менее практически все современные аппаратные платформы имеют некоторый типичный набор средств аппаратной поддержки ОС, в который входят следующие компоненты:

- средства поддержки привилегированного режима;
- средства трансляции адресов;
- средство переключения процессов;
- система прерываний;
- системный таймер;
- средства защиты областей памяти.

Средства поддержки привилегированного режима обычно основаны на системном регистре процессора, часто называемом «словом состояния» машины или процессора. Этот регистр содержит некоторые признаки, определяющие режимы работы процессора, в том числе признак текущего режима привилегий. Смена режима привилегий выполняется за счет изменения слова состояния машины в результате прерывания или выполнения привилегированной команды. Число градаций привилегированности может быть разным у разных типов процессоров, наиболее часто используются два уровня (ядро-пользователь) или четыре (0-1-2-3 у процессоров Intel Pentium). В обязанности средств поддержки привилегированного режима входит проверка допустимости выполнения активной программой команд процессора при текущем уровне привилегированности.

Средства трансляции адресов выполняют операции преобразования виртуальных адресов, которые содержатся в кодах процесса, в адреса физической памяти. Таблицы, предназначенные при трансляции адресов, обычно имеют большой объем, поэтому для их хранения используются области оперативной памяти, а аппаратура процессора содержит только указатели на эти области. Средства трансляции адресов применяют данные указатели для доступа к элементам таблиц и аппаратного выполнения алгоритма преобразования адреса, что значительно ускоряет процедуру трансляции по сравнению с ее чисто программной реализацией.

Средства переключения процессов предназначены для быстрого сохранения контекста приостанавливаемого процесса и восстановления контекста процесса, который становится активным. Содержимое контекста обычно включает содержимое всех регистров общего назначения процессора, регистра флагов операций (то есть флагов нуля, переноса, переполнения и т. п.), а также тех системных регистров и указателей, которые связаны с отдельным процессом, а не операционной системой, например указателя на таблицу трансляции адресов процесса. Для хранения контекстов приостановленных процессов обычно

используются области оперативной памяти, которые поддерживаются указателями процессора. Переключение контекста выполняется по определенным командам процессора, например по команде перехода на новую задачу. Такая команда вызывает автоматическую загрузку данных из сохраненного контекста в регистры процессора, после чего процесс продолжается с прерванного ранее места.

Система прерываний позволяет компьютеру реагировать на внешние события, синхронизировать выполнение процессов и работу устройств ввода-вывода, быстро переходить с одной программы на другую. Механизм прерываний нужен для того, чтобы оповестить процессор о возникновении в вычислительной системе некоторого непредсказуемого события или события, которое не синхронизировано с циклом работы процессора. Примерами таких событий могут служить: завершение операции ввода-вывода внешним устройством (например, запись блока данных контроллером диска), некорректное завершение арифметической операции (например, переполнение регистра), истечение интервала астрономического времени. При возникновении условий прерывания его источник (контроллер внешнего устройства, таймер, арифметический блок процессора и т. п.) выставляет определенный электрический сигнал. Этот сигнал прерывает выполнение процессором последовательности команд, задаваемой исполняемым кодом, и вызывает автоматический переход на заранее определенную процедуру, называемую **процедурой обработки прерываний**. В большинстве моделей процессоров отрабатываемый аппаратурой переход на процедуру обработки прерываний сопровождается заменой слова состояния машины (или даже всего контекста процесса), что позволяет одновременно с переходом по нужному адресу выполнить переход в привилегированный режим. После завершения обработки прерывания обычно происходит возврат к исполнению прерванного кода.

Прерывания играют важнейшую роль в работе любой операционной системы, являясь ее «движущей силой». Действительно, большая часть действий ОС инициируется прерываниями различного типа. Даже системные вызовы от приложений выполняются на многих аппаратных платформах с помощью специальной инструкции прерывания, вызывающей переход к выполнению соответствующих процедур ядра (например, команда `int` в процессорах Intel или `SVC` в мэйнфреймах IBM).

Системный таймер, часто реализуемый в виде быстродействующего регистра-счетчика, необходим операционной системе, чтобы выдерживать интервалы времени. Для этого в регистр таймера программно загружается значение требуемого интервала в условных единицах, из которого затем автоматически с определенной частотой начинает вычитаться по единице. Частота «тиков» таймера, как правило, тесно связана с частотой тактового генератора процессора. (Не следует путать таймер ни с тактовым генератором, который вырабатывает сигналы, синхронизирующие все операции в компьютере, ни с системными часами — работающей на батареях электронной схеме, — которые ведут независимый отсчет времени и календарной даты.) При достижении нулевого значения

счетчика таймер инициирует прерывание, которое обрабатывается процедурой операционной системы. Прерывания от системного таймера используются ОС, в первую очередь, для слежения за тем, как отдельные процессы расходуют время процессора. Например, в системе разделения времени при обработке очередного прерывания от таймера планировщик процессов может принудительно передать управление другому процессу, если данный процесс исчерпал выделенный ему квант времени.

Средства защиты областей памяти обеспечивают на аппаратном уровне проверку возможности программного кода осуществлять с данными определенной области памяти такие операции, как чтение, запись или выполнение (при передаче управления). Если аппаратура компьютера поддерживает механизм трансляции адресов, то средства защиты областей памяти встраиваются в этот механизм. Функции аппаратуры по защите памяти обычно состоят в сравнении уровней привилегий текущего кода процессора и сегмента памяти, к которому производится обращение.

Машинно-зависимые компоненты ОС

Одна и та же операционная система не может без каких-либо изменений устанавливаться на компьютерах, отличающихся типом процессора и/или способом организации всей аппаратуры. В модулях ядра ОС не могут не отразиться такие особенности аппаратной платформы, как количество типов прерываний и формат таблицы ссылок на процедуры обработки прерываний, состав регистров общего назначения и системных регистров, состояние которых нужно сохранять в контексте процесса, особенности подключения внешних устройств и многие другие.

Однако опыт разработки операционных систем показывает: ядро можно спроектировать таким образом, чтобы только часть модулей были *машинно-зависимыми*, а остальные не зависели от особенностей аппаратной платформы. В хорошо структурированном ядре машинно-зависимые модули локализованы и образуют программный слой, естественно примыкающий к слою аппаратуры, как это и показано на рис. 3.8. Такая локализация машинно-зависимых модулей существенно упрощает перенос операционной системы на другую аппаратную платформу.

Объем машинно-зависимых компонентов ОС зависит от того, насколько велики отличия в аппаратных платформах, для которых разрабатывается ОС. Например, ОС, построенная на 32-разрядных адресах, для переноса на машину с 16-разрядными адресами должна быть практически переписана заново. Одно из наиболее очевидных отличий — несовпадение системы команд процессоров — преодолевается достаточно просто. Операционная система программируется на языке высокого уровня, а затем соответствующим компилятором вырабатывается код для конкретного типа процессора. Однако во многих случаях различия в организации аппаратуры компьютера лежат гораздо глубже и преодолеть их таким образом не удается. Например, однопроцессорный и двухпроцессорный компьютер требуют применения в ОС совершенно разных алгорит-

мов распределения процессорного времени. Аналогично, отсутствие аппаратной поддержки виртуальной памяти приводит к принципиальному различию в реализации подсистемы управления памятью. В таких случаях не обойтись без внесения в код операционной системы специфики аппаратной платформы, для которой эта ОС предназначается.

Для уменьшения количества машинно-зависимых модулей производители операционных систем обычно ограничивают универсальность машинно-независимых модулей. Это означает, что их независимость носит условный характер и распространяется только на несколько типов процессоров и созданных на основе этих процессоров аппаратных платформ.

Особое место среди модулей ядра занимают низкоуровневые драйверы внешних устройств. С одной стороны, эти драйверы, как и высокоуровневые драйверы, входят в состав менеджера ввода-вывода, то есть принадлежат слою ядра, занимающему достаточно высокое место в иерархии слоев. С другой стороны, низкоуровневые драйверы отражают все особенности управляемых внешних устройств, поэтому их можно отнести к слою машинно-зависимых модулей. Такая двойственность низкоуровневых драйверов еще раз подтверждает схематичность модели ядра со строгой иерархией слоев.

Для компьютеров на основе процессоров Intel Pentium разработка экранирующего машинно-зависимого слоя ОС несколько упрощается за счет встроенной в постоянную память компьютера базовой системы ввода-вывода — BIOS. BIOS содержит драйверы для всех устройств, входящих в базовую конфигурацию компьютера: жестких и гибких дисков, клавиатуры, дисплея и т. д. Эти драйверы выполняют весьма примитивные операции с управляемыми устройствами, например чтение группы секторов данных с определенной дорожки диска, но за счет этих операций экранируются различия аппаратных платформ персональных компьютеров и серверов на процессорах Intel разных производителей. Разработчики операционной системы могут пользоваться слоем драйверов BIOS как частью машинно-зависимого слоя ОС, а могут и заменить все или часть драйверов BIOS компонентами ОС.

Переносимость операционной системы

> ОС называют **переносимой** (portable), или **мобильной**, если ее код может быть сравнительно легко перенесен с процессора одного типа на процессор другого типа и с аппаратной платформы одного типа на аппаратную платформу другого типа.

Хотя ОС часто описываются либо как переносимые, либо как непереносимые, мобильность — это не бинарное состояние, а понятие степени. Вопрос не в том, может ли быть система перенесена, а в том, насколько легко можно это сделать. Для того чтобы обеспечить свойство мобильности ОС, разработчики должны следовать следующим правилам.

- Большая часть кода ОС должна быть написана на языке высокого уровня, трансляторы которого имеются на всех машинах, куда предполагается переносить

систему. Большинство переносимых ОС написано на языке C, который имеет много особенностей, полезных для разработки кодов операционной системы, и компиляторы которого широко доступны. Программа, написанная на ассемблере, является переносимой только в тех случаях, когда перенос операционной системы планируется на компьютер, обладающий той же системой команд. В остальных случаях ассемблер используется только для тех непереносимых частей системы, которые должны непосредственно взаимодействовать с аппаратурой (например, обработчиков прерываний), или для частей, которые требуют максимальной скорости (как, например, целочисленная арифметика повышенной точности).

- Объем машинно-зависимых частей кода, которые непосредственно взаимодействуют с аппаратными средствами, должен быть по возможности минимизирован. Так, например, следует всячески избегать прямого манипулирования регистрами и другими аппаратными средствами процессора. Для осуществления всех необходимых действий по управлению аппаратурой должен быть написан набор аппаратно-зависимых функций. Каждый раз, когда какому-либо модулю ОС требуется выполнить некоторое действие, связанное с аппаратурой, он должен использовать соответствующую функцию из имеющегося набора. Когда ОС переносится, то требуется переписать только эти функции.
- Аппаратно-зависимый код должен быть надежно изолирован в нескольких модулях, а не распределяться по всей системе. Изоляции подлежат все части ОС, которые отражают специфику как процессора, так и аппаратной платформы

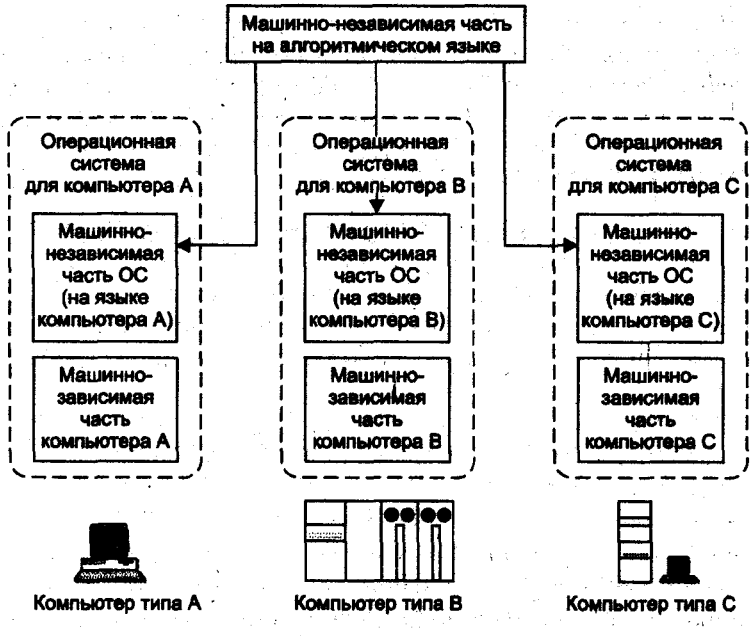


Рис. 3.9. Перенос операционной системы на разные аппаратные платформы

в целом. Низкоуровневые компоненты ОС, имеющие доступ к процессорно-зависимым структурам данных и регистрам, должны быть оформлены в виде компактных модулей, которые могут быть заменены аналогичными модулями для других процессоров.

В идеале слой машинно-зависимых компонентов ядра полностью экранирует остальную часть ОС от конкретных деталей аппаратной платформы (кэши, контроллеры прерываний ввода-вывода и т. п.), по крайней мере, для того набора платформ, который поддерживает данная ОС. В результате происходит подмена реальной аппаратуры некой унифицированной виртуальной машиной, одинаковой для всех вариантов аппаратной платформы. Все слои операционной системы, которые лежат выше слоя машинно-зависимых компонентов, могут быть написаны для управления именно этой виртуальной аппаратурой. Таким образом, у разработчиков появляется возможность создать один вариант машинно-независимой части ОС (включая компоненты ядра, утилиты, системные обрабатывающие программы) для всего набора поддерживаемых платформ (рис. 3.9).

Микроядерная архитектура

Концепция

Микроядерная архитектура является альтернативой классическому варианту построения операционной системы. Под классической архитектурой в данном случае понимается рассмотренная ранее структурная организация ОС, в соответствии с которой *все* основные функции операционной системы, составляющие многослойное ядро, выполняются в привилегированном режиме. При этом некоторые вспомогательные функции ОС оформляются в виде приложений и выполняются в пользовательском режиме наряду с обычными пользовательскими программами (становясь системными утилитами или обрабатывающими программами). Каждое приложение пользовательского режима работает в собственном адресном пространстве и защищено тем самым от какого-либо вмешательства других приложений. Код ядра, выполняемый в привилегированном режиме, имеет доступ к областям памяти всех приложений, но сам полностью от них защищен. Приложения обращаются к ядру с запросами на выполнение системных функций.

> В *микроядерных ОС* в привилегированном режиме остается работать только очень небольшая часть ОС, называемая **микроядром**. Все остальные высокоуровневые функции ядра оформляются в виде приложений, работающих в пользовательском режиме.

Микроядро защищено от остальных частей ОС и приложений (рис. 3.10). В состав микроядра обычно входят машинно-зависимые модули, а также модули, выполняющие базовые (но не все!) функции ядра по управлению процессами, обработке прерываний, управлению виртуальной памятью, пересылке

сообщений и управлению устройствами ввода-вывода, связанные с загрузкой или чтением регистров устройств. Набор функций микроядра обычно соответствует функциям слоя базовых механизмов обычного ядра. Такие функции операционной системы трудно, если не невозможно, выполнить в пользовательском пространстве.

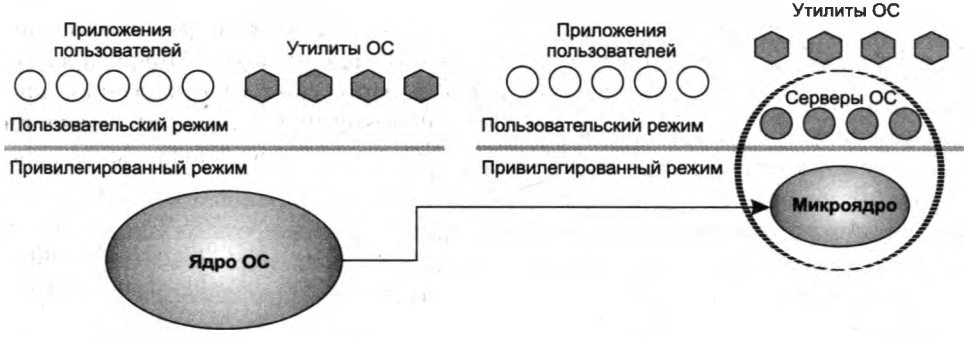


Рис. 3.10. Перенос основного объема функций ядра в пользовательское пространство

Все остальные более высокоуровневые функции ядра оформляются в виде приложений, работающих в пользовательском режиме. Однозначного решения о том, какие из системных функций нужно оставить в привилегированном режиме, а какие перенести в пользовательский, не существует. В общем случае многие менеджеры ресурсов, являющиеся неотъемлемыми частями обычного ядра — файловая система, подсистемы управления виртуальной памятью и процессами, менеджер безопасности и т. п., — становятся «периферийными» модулями, работающими в пользовательском режиме.

Работающие в пользовательском режиме менеджеры ресурсов имеют принципиальные отличия от традиционных утилит и обрабатывающих программ операционной системы, хотя при микроядерной архитектуре все эти программные компоненты также оформлены в виде приложений. Утилиты и обрабатывающие программы вызываются в основном пользователями. Ситуации, когда одному приложению требуется выполнение функции (процедуры) другого приложения, возникают крайне редко. Поэтому в операционных системах с классической архитектурой отсутствует механизм, с помощью которого одно приложение могло бы вызывать функции другого.

Совсем другая ситуация возникает, когда в форме приложения оформляется часть операционной системы. По определению, основным назначением такого приложения является обслуживание запросов других приложений, например создание процесса, выделение памяти, проверка прав доступа к ресурсу и т. д. Именно поэтому менеджеры ресурсов, вынесенные в пользовательский режим, называются **серверами** ОС, то есть модулями, основным назначением которых является обслуживание запросов локальных приложений и других модулей ОС. Очевидно, что для реализации микроядерной архитектуры необходимым условием является наличие в операционной системе удобного и эффективного

механизма вызова процедур одного процесса из другого процесса. Поддержка такого механизма и является одной из главных задач микроядра.

Схематично механизм обращения к функциям ОС, оформленным в виде серверов, выглядит следующим образом (рис. 3.11). Клиент, которым может быть либо прикладная программа, либо другой компонент ОС, запрашивает выполнение некоторой функции у соответствующего сервера, посылая ему сообщение. Непосредственная передача сообщений между приложениями невозможна, так как их адресные пространства изолированы друг от друга. Микроядро, выполняющееся в привилегированном режиме, имеет доступ к адресным пространствам каждого из этих приложений и поэтому может работать в качестве посредника. Микроядро сначала передает сообщение, содержащее имя и параметры вызываемой процедуры нужному серверу, затем сервер выполняет запрошенную операцию, после чего ядро возвращает результаты клиенту с помощью другого сообщения. Таким образом, работа микроядерной операционной системы соответствует известной модели клиент-сервер, в которой роль транспортных средств исполняет микроядро.

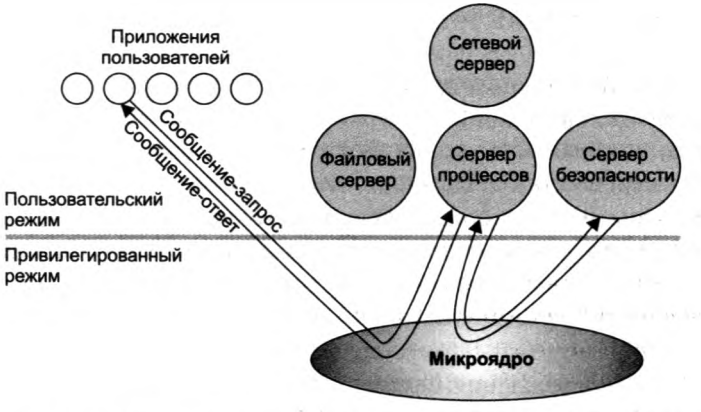


Рис. 3.11. Реализация системного вызова в микроядерной архитектуре

Преимущества и недостатки микроядерной архитектуры

Микроядерная архитектура в высокой степени удовлетворяет большинству требований, предъявляемых к современным ОС: она способствует переносимости, расширяемости, повышению надежности системы и создает хорошие предпосылки для поддержки распределенных приложений. За эти достоинства приходится платить снижением производительности, и это является основным недостатком микроядерной архитектуры.

Высокая степень *переносимости* обусловлена тем, что весь машинно-зависимый код изолирован в микроядре, поэтому для переноса системы на новый процессор требуется меньше изменений и все они логически сгруппированы вместе.

Расширяемость присуща микроядерной ОС в очень высокой степени. В традиционных системах даже при наличии многослойной структуры нелегко удалить один слой и поменять его на другой по причине множественности и размытости интерфейсов между слоями. Добавление новых функций и изменение существующих требует хорошего знания операционной системы и больших затрат времени. В то же время ограниченный набор четко определенных интерфейсов микроядра открывает путь к упорядоченному росту и эволюции ОС. Добавление новой подсистемы требует разработки нового приложения, что никак не затрагивает целостность микроядра. Микроядерная структура позволяет не только добавлять, но и сокращать число компонентов операционной системы, что также бывает очень полезно. Например, не всем пользователям нужны средства безопасности или поддержки распределенных вычислений, а удаление их из традиционного ядра чаще всего невозможно. Обычно традиционные операционные системы позволяют динамически добавлять в ядро или удалять из ядра только драйверы внешних устройств — ввиду частых изменений в конфигурации подключенных к компьютеру внешних устройств подсистема ввода-вывода ядра допускает загрузку и выгрузку драйверов «на ходу», но для этого она разрабатывается особым образом (например, среда STREAMS в Unix). При микроядерном подходе *конфигурируемость* ОС не вызывает никаких проблем и не требует особых мер — достаточно изменить файл с параметрами начальной конфигурации системы или же остановить не нужные больше серверы в ходе работы обычными для остановки приложений средствами.

Использование микроядерной модели повышает *надежность* ОС. Каждый сервер выполняется в виде отдельного процесса в своей собственной области памяти и, таким образом, защищен от других серверов операционной системы, что не наблюдается в традиционной ОС, где все модули ядра могут влиять друг на друга. И если отдельный сервер терпит крах, то он может быть перезапущен без останова или повреждения остальных серверов ОС. Более того, поскольку серверы выполняются в пользовательском режиме, они не имеют непосредственного доступа к аппаратуре и не могут модифицировать память, в которой хранится и работает микроядро. Другим потенциальным источником повышения надежности ОС является уменьшенный объем кода микроядра по сравнению с традиционным ядром — это снижает вероятность появления ошибок программирования.

Модель с микроядром хорошо подходит для поддержки **распределенных вычислений**, так как использует механизмы, аналогичные сетевым: взаимодействие клиентов и серверов путем обмена сообщениями. Серверы микроядерной ОС могут работать как на одном, так и на разных компьютерах. В этом случае при получении сообщения от приложения микроядро может обработать его самостоятельно и передать локальному серверу или же переслать по сети микроядру, работающему на другом компьютере. Переход к распределенной обработке требует минимальных изменений в работе операционной системы — просто локальный транспорт заменяется сетевым.

Производительность. При классической организации ОС (рис. 3.12, *а*) выполнение системного вызова сопровождается двумя переключениями режимов, а при микроядерной организации (рис. 3.12, *б*) — четырьмя. Таким образом, операционная система на основе микроядра при прочих равных условиях всегда будет менее производительной, чем ОС с классическим ядром.

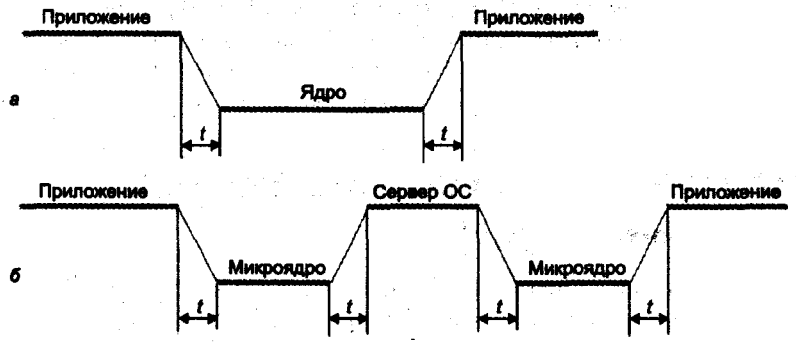


Рис. 3.12. Смена режимов при выполнении системного вызова

Серьезность этого недостатка хорошо иллюстрирует история Windows NT. В первых версиях 3.1 и 3.5 диспетчер окон, графическая библиотека и высокоуровневые драйверы графических устройств входили в состав сервера пользовательского режима, и вызов функций этих модулей осуществлялся в соответствии с микроядерной схемой. Однако очень скоро разработчики Windows NT поняли, что такой механизм обращений к часто используемым функциям графического интерфейса существенно замедляет работу приложений, делая данную операционную систему уязвимой в условиях острой конкуренции. В результате в версию Windows NT 4.0 были внесены существенные изменения — все перечисленные модули были перенесены в ядро, что отдалило эту ОС от идеальной микроядерной архитектуры, но зато резко повысило ее производительность.

Этот пример иллюстрирует главную проблему, с которой сталкиваются разработчики операционной системы, решившие применить микроядерный подход, — что включать в микроядро, а что выносить в пользовательское пространство. В идеальном случае микроядро может состоять только из средств передачи сообщений, средств взаимодействия с аппаратурой, в том числе средств доступа к механизмам привилегированной защиты. Однако многие разработчики не всегда жестко придерживаются принципа минимизации функций ядра, часто жертвуя этим ради повышения производительности. В результате реализации микроядерных ОС образуют некоторый спектр, на одном краю которого находятся системы с минимально возможным микроядром (например, Unix-подобная операционная система реального времени QNX), а на другом — современные системы семейства Windows NT, в которых микроядро нагружено выполнением достаточно большого объема функций.

В самое последнее время идея микроядерного подхода снова начала привлекать повышенное внимание. Это связано с тем, что многие пользователи стали отдавать предпочтение надежности в вечном споре «производительность-надежность». Кроме того, быстродействие процессоров и элементов памяти постоянно растет, так что стало позволительным выделить какую-то часть производительности на накладные расходы, связанные с работой микроядра, получив за счет этого более надежно работающую систему. Все это возродило интерес к данной архитектурной идее, так что можно ожидать в ближайшем будущем появления новых универсальных ОС, работающих на микроядре.

Совместимость и множественные прикладные среды

В то время как многие архитектурные особенности операционных систем непосредственно касаются только системных программистов, концепция **множественных прикладных сред** непосредственно связана с нуждами конечных пользователей — возможностью операционной системы выполнять приложения, написанные для других операционных систем. Такое свойство операционной системы называется **совместимостью**.

Двоичная совместимость и совместимость исходных текстов

Необходимо различать совместимость на двоичном уровне и совместимость на уровне исходных текстов. Приложения обычно хранятся в ОС в виде исполняемых файлов, содержащих двоичные образы кодов и данных. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение в среде другой ОС.

Совместимость на уровне исходных текстов требует наличия соответствующего компилятора в составе программного обеспечения компьютера, на котором предполагается выполнять данное приложение, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый исполняемый модуль.

Совместимость на уровне исходных текстов важна в основном для разработчиков приложений, в распоряжении которых эти исходные тексты всегда имеются. Но для конечных пользователей практическое значение имеет только двоичная совместимость, так как только в этом случае они могут применять один и тот же коммерческий продукт, поставляемый в виде двоичного исполняемого кода, в различных операционных средах и на различных машинах. Для пользователя, купившего в свое время пакет (например, Microsoft Office 95) для Windows 95, важно, чтобы он мог запускать этот полюбившийся ему пакет без каких бы то ни было изменений на своей машине после установки на ней, например, Windows Vista. Необходимость решения проблемы совместимости

может возникнуть и у пользователей системы Linux, поскольку, для MS Windows разработана масса очень удобных приложений, многие из которых не перенесены в среду Linux.

Обладает ли новая ОС двоичной совместимостью или совместимостью исходных текстов с существующими операционными системами, зависит от многих факторов. Самый главный из них — архитектура процессора, на котором работает новая ОС. Если процессор использует тот же набор команд (возможно с некоторыми добавлениями) и тот же диапазон адресов, тогда двоичная совместимость может быть достигнута просто. Для этого достаточно соблюдение следующих условий:

- вызовы API-функций, которые содержит приложение, должны поддерживаться данной ОС;
- внутренняя структура исполняемого файла приложения должна соответствовать структуре исполняемых файлов данной ОС.

Гораздо сложнее достичь двоичной совместимости операционным системам, предназначенным для выполнения на процессорах, имеющих разные архитектуры. Помимо соблюдения приведенных условий, необходимо организовать *эмуляцию* двоичного кода.

Пусть, например, требуется выполнить Windows-программу для IBM PC-совместимого компьютера на компьютере Macintosh. Компьютер Macintosh построен на основе процессора Motorola, а компьютер IBM PC — на основе процессора Intel. Процессор Motorola имеет архитектуру (систему команд, состав регистров и т. п.), отличную от архитектуры процессора Intel, поэтому ему непонятен двоичный код Windows-программы, содержащей команды этого процессора. Для того чтобы компьютер Macintosh смог интерпретировать машинные инструкции, которые ему изначально непонятны, на нем должно быть установлено специальное программное обеспечение — эмулятор.

Эмулятор — это программа, которая последовательно, одну за другой считывает из эмулируемой программы двоичные инструкции одного процессора, анализирует, какие действия необходимо выполнить по этой инструкции, а затем выполняет эквивалентную подпрограмму, написанную в инструкциях другого процессора.

Так как к тому же у процессора Motorola нет в точности таких же регистров, флагов и внутреннего арифметико-логического устройства, как у процессора Intel, он должен также имитировать (эмулировать) все эти элементы с использованием своих регистров или памяти. Состояние эмулируемых регистров и флагов после выполнения каждой команды должно быть абсолютно таким же, как и в реальном процессоре Intel.

Это простая, но очень медленная работа, так как одна команда процессора Intel исполняется значительно быстрее, чем эмулирующая его последовательность команд процессора Motorola.

Трансляция библиотек

Выходом в таких случаях является использование так называемых **прикладных программных сред**. Одной из составляющих, формирующих прикладную программную среду, является набор функций интерфейса прикладного программирования (API), которые операционная система предоставляет своим приложениям. Для сокращения времени на выполнение чужих программ прикладные среды имитируют обращения к библиотечным функциям.

Эффективность этого подхода связана с тем, что большинство сегодняшних программ работают под управлением графических интерфейсов пользователя (Graphical User Interface, GUI) типа Windows, Mac или Unix Motif, при этом приложения тратят большую часть времени, производя некоторые хорошо предсказуемые действия. Они непрерывно выполняют вызовы библиотек GUI для манипулирования окнами и для других связанных с GUI действий. Сегодня в типичных программах 60–80 % времени тратится на выполнение GUI-функций и других библиотечных вызовов ОС. Именно это свойство приложений позволяет прикладным средам компенсировать большие затраты времени на покомандное эмулирование программы. Тщательно спроектированная программная прикладная среда имеет в своем составе библиотеки, имитирующие внутренние библиотеки GUI, но написанные на «родном» коде. Таким образом, достигается существенное ускорение выполнения программ с API другой операционной системы. Иногда такой подход называют трансляцией, что позволяет отличать его от более медленного процесса эмулирования кода по одной команде за раз.

Например, для Windows-программы, работающей на Macintosh, при интерпретации команд процессора Intel 80x86 производительность может быть очень низкой. Но когда производится вызов GUI-функции открытия окна, модуль ОС, реализующий прикладную среду Windows, может перехватить этот вызов и перенаправить его на перекомпилированную для процессора Motorola 680x0 подпрограмму открытия окна. В результате на таких фрагментах кода скорость работы программы может достичь (а возможно, и превзойти) скорость работы на своем «родном» процессоре.

Чтобы программа, написанная для одной ОС, могла быть выполнена в рамках другой ОС, недостаточно лишь обеспечить совместимость API. Концепции, положенные в основу разных ОС, могут входить в противоречие друг с другом. Например, в одной операционной системе приложению может быть разрешено непосредственно управлять устройствами ввода-вывода, в другой эти действия являются прерогативой ОС. Каждая операционная система имеет собственные механизмы защиты ресурсов, свои алгоритмы обработки ошибок и исключительных ситуаций, особую структуру процесса и схему управления памятью, свою семантику доступа к файлам и графический пользовательский интерфейс. Для обеспечения совместимости необходимо организовать бесконфликтное сосуществование в рамках одной ОС нескольких способов управления ресурсами компьютера.

Способы реализации прикладных программных сред

Создание полноценной прикладной среды, полностью совместимой со средой другой операционной системы, является достаточно сложной задачей, тесно связанной со структурой операционной системы. Существуют различные варианты построения множественных прикладных сред, отличающиеся как особенностями архитектурных решений, так и функциональными возможностями, обеспечивающими различную степень переносимости приложений.

Во многих версиях ОС Unix транслятор прикладных сред реализуется в виде обычного приложения. В операционных системах, построенных с использованием микроядерной концепции, таких как, например, Windows NT или Workplace OS, прикладные среды выполнялись в виде серверов пользовательского режима. А в OS/2 с ее более простой архитектурой средства организации прикладных сред были встроены глубоко в операционную систему.

Один из наиболее очевидных вариантов реализации множественных прикладных сред основывается на стандартной многоуровневой структуре ОС. На рис. 3.13 операционная система OS1 поддерживает, помимо своих «родных» приложений, приложения операционных систем OS2 и OS3. Для этого в ее составе имеются специальные приложения — прикладные программные среды, — которые транслируют интерфейсы API «чужих» операционных систем OS2 и OS3 в API своей «родной» операционной системы — OS1. Так, например, в случае если бы в качестве OS2 выступала операционная система Unix, а в качестве OS1 — OS/2, то для выполнения системного вызова создания процесса `fork()` в Unix-приложении программная среда должна была бы обратиться к ядру операционной системы OS/2 с системным вызовом `DosExecPgm()`.

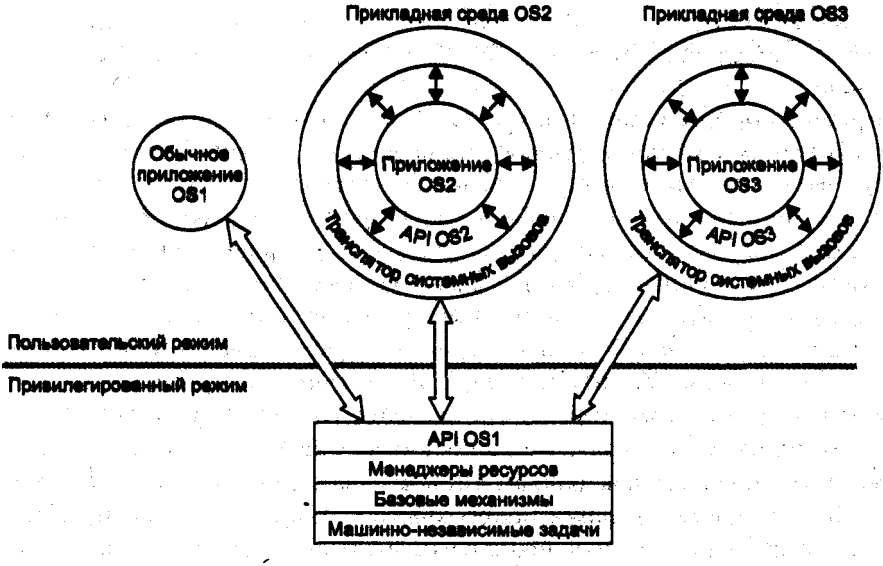


Рис. 3.13. Прикладные программные среды, транслирующие системные вызовы

К сожалению, поведение почти всех функций, составляющих API одной ОС, как правило, существенно отличается от поведения соответствующих функций другой ОС. Например, чтобы функция создания процесса `DosExecPgm()` системы OS/2 полностью соответствовала функции создания процесса `fork()` в Unix-подобных системах, ее нужно было бы изменить, чтобы она поддерживала возможность копирования адресного пространства родительского процесса в пространство дочернего процесса. (При нормальном же поведении этой функции память процесса-потомка инициализируется на основе данных нового исполняемого файла.)

В другом варианте реализации множественных прикладных сред операционная система имеет несколько равноправных прикладных программных интерфейсов. В приведенном на рис. 3.14 примере операционная система поддерживает приложения, написанные для OS1, OS2 и OS3. Для этого непосредственно в пространстве ядра системы размещены прикладные программные интерфейсы всех этих ОС: API OS1, API OS2 и API OS3.

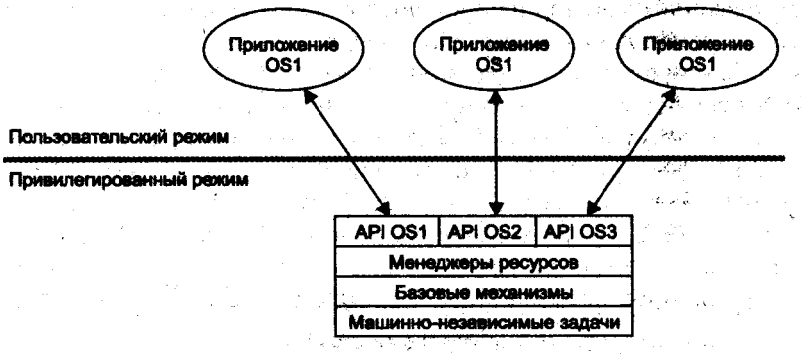


Рис. 3.14. Реализация совместимости на основе нескольких равноправных интерфейсов API

В этом варианте функции уровня API обращаются к функциям нижележащего уровня ОС, которые должны поддерживать все три в общем случае несовместимые прикладные среды. В разных ОС по-разному осуществляется управление системным временем, используются разные форматы времени дня, на основании собственных алгоритмов разделяется процессорное время и т. д. Функции каждого интерфейса API реализуются ядром с учетом специфики соответствующей ОС, даже если они имеют аналогичное назначение. Например, как уже было сказано, функция создания процесса работает по-разному для Unix-приложения и приложения OS/2. Аналогично, при завершении процесса ядру также необходимо определять, к какой ОС относится данный процесс. Если этот процесс был создан по запросу Unix-приложения, то в ходе его завершения ядро должно послать родительскому процессу сигнал, как это делается в ОС Unix. А при завершении процесса OS/2, созданного с параметром `EXEC_SYNC`, ядро должно отметить, что идентификатор процесса не может быть повторно использован другим процессом OS/2. Для того чтобы ядро могло вы-

брать нужный вариант реализации системного вызова, каждый процесс должен передавать в ядро набор идентифицирующих характеристик.

Еще один способ построения множественных прикладных сред основан на микроядерном подходе. При этом очень важно отделить базовые, общие для всех прикладных сред механизмы операционной системы от специфических для каждой из прикладных сред высокоуровневых функций, решающих стратегические задачи.

В соответствии с микроядерной архитектурой все функции ОС реализуются микроядром и серверами пользовательского режима. Важно, что каждая прикладная среда оформляется в виде отдельного сервера пользовательского режима и не включает базовых механизмов (рис. 3.15). Приложения, используя API, обращаются с системными вызовами к соответствующей прикладной среде через микроядро. Прикладная среда обрабатывает запрос, выполняет его (возможно, обращаясь для этого за помощью к базовым функциям микроядра) и отсылает приложению результат. В ходе выполнения запроса прикладной среде приходится, в свою очередь, обращаться к базовым механизмам ОС, реализуемым микроядром и другими серверами ОС.

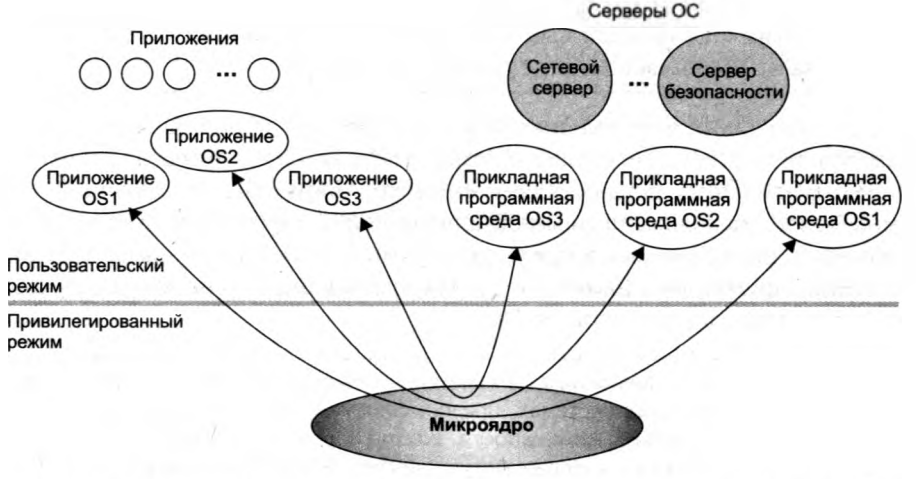


Рис. 3.15. Микроядерный подход к реализации множественных прикладных сред

Такому подходу к конструированию множественных прикладных сред присущи все достоинства и недостатки микроядерной архитектуры, в частности:

- очень просто можно добавлять и исключать прикладные среды, что является следствием хорошей расширяемости микроядерных ОС;
- надежность и стабильность выражаются в том, что при отказе одной из прикладных сред все остальные сохраняют работоспособность;
- низкая производительность микроядерных ОС сказывается на скорости работы прикладных сред, а значит, и на скорости выполнения приложений.

Создание в рамках одной операционной системы нескольких прикладных сред для выполнения приложений различных ОС представляет собой путь, который позволяет иметь единственную версию программы и переносить ее между операционными системами. Множественные прикладные среды обеспечивают совместимость на двоичном уровне данной ОС с приложениями, написанными для других ОС. В результате пользователи получают большую свободу выбора операционных систем и более легкий доступ к качественному программному обеспечению.

Система виртуальных машин

А теперь давайте посмотрим на описанные ранее схемы обеспечения совместимости приложений под несколько другим углом. Если программную среду, позволяющую выполнять приложения определенной ОС, рассматривать как операционную систему, то мы придем к достаточно давно известной концепции системы виртуальных машин.

> **Система виртуальных машин (СВМ)** — это такой вариант организации вычислительного процесса, при котором на одном компьютере одновременно выполняются несколько копий одной и той же или нескольких разных ОС. Каждая из этих ОС, называемых «гостевыми», функционирует так, как если бы она одна выполнялась на отдельном компьютере.

Количество таких виртуальных компьютеров определяется доступными ресурсами физически существующего компьютера, на базе которого строится система. Виртуализацию осуществляет «монитор виртуальных машин» — программный слой, расположенный между аппаратной платформой и «гостевыми» ОС. Приложения, выполняемые в среде одной «гостевой» ОС, никак не влияют на приложения другой ОС. Более того, крах «гостевой» ОС не вызывает серьезных последствий для оставшихся ОС.

ПРИМЕЧАНИЕ Может показаться, что возможность работы в нескольких разных операционных системах, предоставляемая системой виртуальных машин, напоминает возможности компьютеров с альтернативной загрузкой (multiboot), которые могут работать под управлением нескольких разных ОС, хранящихся в разных разделах загрузочного диска. Однако эти варианты организации вычислительного процесса принципиально различны. В то время как СВМ обеспечивает одновременную работу нескольких ОС, в системах с альтернативной загрузкой работает только одна выбранная пользователем ОС.

Преимущества, которые дает использование системы виртуальных машин, прямо следуют из уже рассмотренных ранее свойств множественных программных сред. Действительно, СВМ позволяет решить проблему совместимости приложений, разработанных для разных ОС, повысить эффективность использования аппаратуры за счет мультипрограммного выполнения нескольких ОС на одном компьютере, повысить надежность путем изоляции возможных оши-

бок в программных кодах, достигаемой выполнением приложений в отдельных операционных средах.

Первой популярной ОС, в которой была в полной мере реализована идея СВМ, стала ОС компании IBM VM/370 для мэйнфрейма System/370 (1972 г.). Сравнительно недавними примерами СВМ являются VMware Workstation на платформе Intel x86 для операционной системы Linux и Virtual PC для Windows (обе выпущены в 1999 году).

Выводы

- Простейшая структуризация ОС состоит в разделении всех компонентов ОС на модули, выполняющие основные функции ОС (ядро), и модули, выполняющие вспомогательные функции ОС. Вспомогательные модули ОС оформляются либо в виде приложений (утилиты и системные обрабатывающие программы), либо в виде библиотек процедур. Вспомогательные модули загружаются в оперативную память только на время выполнения своих функций, то есть являются транзитными. Модули ядра постоянно находятся в оперативной памяти, то есть являются резидентными.
- При аппаратной поддержке режимов с разными уровнями полномочий устойчивость ОС может быть повышена путем выполнения функций ядра в привилегированном режиме, а вспомогательных модулей ОС и приложений — в пользовательском. Это дает возможность защитить коды и данные ОС и приложений от несанкционированного доступа. ОС может выступать в роли арбитра в спорах приложений за ресурсы.
- Ядро, являясь структурным элементом ОС, в свою очередь, может быть логически разложено на следующие слои (начиная с самого нижнего):
  - машинно-зависимые компоненты ОС;
  - базовые механизмы ядра;
  - менеджеры ресурсов;
  - интерфейс системных вызовов.
- В многослойной системе каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс. На основе функций нижележащего слоя следующий вверх по иерархии слой строит свои функции (более сложные и мощные), которые, в свою очередь, оказываются примитивами для создания еще более мощных функций вышележащего слоя. Многослойная организация ОС существенно упрощает разработку и модернизацию системы.
- Любая ОС для решения своих задач взаимодействует с аппаратными средствами компьютера, а именно со средствами поддержки привилегированного режима и трансляции адресов, средствами переключения процессов и защиты областей памяти, системой прерываний и системным таймером. Это

делает ОС машинно-зависимой, привязанной к определенной аппаратной платформе.

- Переносимость ОС может быть достигнута при соблюдении следующих правил. Во-первых, большая часть кода должна быть написана на языке, трансляторы которого имеются на всех компьютерах, куда предполагается переносить систему. Во-вторых, объем машинно-зависимых частей кода, которые непосредственно взаимодействуют с аппаратными средствами, должен быть по возможности минимизирован. В-третьих, аппаратно-зависимый код должен быть надежно локализован в нескольких модулях.
- Микроядерная архитектура является альтернативой классическому варианту построения операционной системы, в соответствии с которым все основные функции операционной системы, составляющие многослойное ядро, выполняются в привилегированном режиме. В микроядерных ОС в привилегированном режиме остается работать только очень небольшая часть ОС, называемая микроядром. Все остальные высокоуровневые функции ядра оформляются в виде приложений, работающих в пользовательском режиме.
- Микроядерные ОС удовлетворяют большинству требований, предъявляемых к современным ОС, обладая переносимостью, расширяемостью, надежностью и создавая хорошие предпосылки для поддержки распределенных приложений. За эти достоинства приходится платить снижением производительности, что является основным недостатком микроядерной архитектуры.
- Прикладная программная среда — совокупность средств ОС, предназначенная для организации выполнения приложений, использующих определенную систему машинных команд, определенный тип API и определенный формат исполняемой программы. Каждая ОС создает как минимум одну прикладную программную среду. Проблема состоит в обеспечении совместимости нескольких программных сред в рамках одной ОС. При построении множественных прикладных сред используются различные архитектурные решения, концепции эмуляции двоичного кода, трансляции API.
- Система виртуальных машин — это такой вариант организации вычислительного процесса, при котором на одном компьютере, работающем под управлением основной (базовой) ОС, создаются один или несколько виртуальных компьютеров, причем на каждом из них можно запустить собственную ОС.

Задачи и упражнения

1. Какие из приведенных терминов являются синонимами:
   1) привилегированный режим;
   2) защищенный режим;
   3) режим супервизора;

4) пользовательский режим;
5) реальный режим;
6) режим ядра.

2. Можно ли, анализируя двоичный код программы, сделать вывод о невозможности ее выполнения в пользовательском режиме?
3. В чем состоят отличия в работе процессора в привилегированном и пользовательском режимах?
4. В идеале микроядерная архитектура ОС требует размещения в микроядре только тех компонентов ОС, которые не могут выполняться в пользовательском режиме. Что заставляет разработчиков операционных систем отходить от этого принципа и расширять ядро за счет перенесения в него функций, которые можно было бы реализовать в виде процессов-серверов?
5. Какие этапы включает разработка варианта мобильной ОС для новой аппаратной платформы?
6. Опишите порядок взаимодействия приложений с ОС, имеющей микроядерную архитектуру.
7. Какими этапами отличается выполнение системного вызова в микроядерной ОС и ОС с монолитным ядром?
8. Может ли программа, эмулируемая на «чужом» процессоре, выполняться быстрее, чем на «родном»?

Глава 4

Процессы и потоки

Важнейшей функцией операционной системы является организация рационального использования всех ее аппаратных и информационных ресурсов. К основным ресурсам могут быть отнесены процессоры, память, внешние устройства, данные и программы. Располагающая одними и теми же аппаратными ресурсами, но управляемая различными ОС, вычислительная система может работать с разной степенью эффективности. Поэтому знание внутренних механизмов операционной системы позволяет косвенно судить о ее эксплуатационных возможностях и характеристиках. Хотя и в однопрограммной ОС необходимо решать задачи управления ресурсами (например, задачу распределения памяти между приложением и ОС), главные сложности на этом пути возникают в мультипрограммных ОС, в которых за ресурсы конкурируют сразу несколько приложений. Именно поэтому большая часть всех проблем, рассматриваемых в этой главе, относится к мультипрограммным системам.

Мультипрограммирование

Мультипрограммирование, или **многозадачность** (multitasking), — это такой вариант организации вычислительного процесса, при котором на одном процессоре попеременно выполняются сразу несколько программ.

Попеременно выполняемые программы совместно используют не только процессор, но и другие ресурсы компьютера: оперативную и внешнюю память, устройства ввода-вывода, данные. Мультипрограммирование призвано повысить эффективность использования вычислительной системы, однако эффективность может пониматься по-разному.

Наиболее характерными критериями эффективности вычислительных систем являются:

- *пропускная способность*, то есть количество задач, выполняемых вычислительной системой в единицу времени;

- *удобство работы пользователей*, заключающееся, в частности, в том, что они имеют возможность интерактивно работать одновременно с несколькими приложениями на одной машине;
- *реактивность системы*, то есть способность системы выдерживать заранее заданные (возможно очень короткие) интервалы времени между запуском программы и получением результата.

В зависимости от выбранного критерия эффективности мультипрограммные ОС делятся на системы пакетной обработки, системы разделения времени и системы реального времени. Каждый тип ОС имеет специфические внутренние механизмы и особые области применения. Некоторые операционные системы могут поддерживать одновременно несколько режимов, например часть задач может выполняться в режиме пакетной обработки, а часть — в режиме реального времени или в режиме разделения времени.

Мультипрограммирование в системах пакетной обработки

При использовании мультипрограммирования для повышения пропускной способности компьютера главной целью является минимизация простоев всех устройств компьютера и, прежде всего, центрального процессора. Такие простои могут возникать из-за приостановки задачи по ее внутренним причинам, связанным, например, с ожиданием ввода данных для обработки. Данные могут храниться на диске или же поступать от пользователя, работающего за терминалом, а также от измерительной аппаратуры, установленной на внешних технических объектах. При такого рода блокировке выполняемой задачи естественным решением, ведущим к повышению эффективности использования процессора, является переключение процессора на выполнение другой задачи, у которой есть данные для обработки. Такая концепция мультипрограммирования положена в основу так называемых пакетных систем.

Системы пакетной обработки предназначены для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является *максимальная пропускная способность*.

Для достижения этой цели в системах пакетной обработки используется следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие разные требования к ресурсам так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины. Например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом.

Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается «выгодное» задание. Следовательно, в вычислительных системах, работающих под управлением пакетных ОС, невозможно гарантировать выполнение того или иного задания в течение определенного периода времени.

Рассмотрим более детально совмещение во времени операций ввода-вывода и вычислений.

Такое совмещение может достигаться разными способами. Один из них характерен для компьютеров, имеющих специализированный процессор ввода-вывода. В компьютерах класса мэйнфреймов такие процессоры называют каналами. Обычно канал имеет систему команд, отличающуюся от системы команд центрального процессора. Эти команды специально предназначены для управления внешними устройствами, например, «проверить состояние устройства», «установить магнитную головку», «установить начало листа», «напечатать строку». Канальные программы могут храниться в той же оперативной памяти, что и программы центрального процессора. В системе команд центрального процессора предусматривается специальная инструкция, с помощью которой каналу передаются параметры и указания на то, какую программу ввода-вывода он должен выполнить. Начиная с этого момента центральный процессор и канал могут работать параллельно (рис. 4.1, *а*).

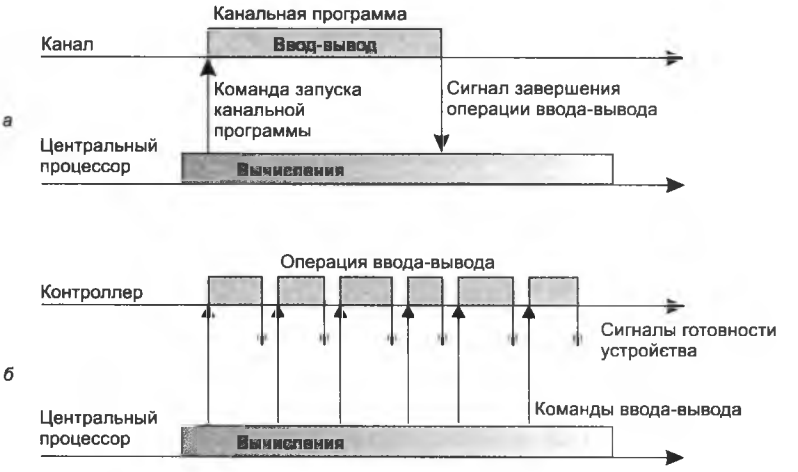


Рис. 4.1. Параллельное выполнение вычислений и операций ввода-вывода

Другой способ совмещения вычислений с операциями ввода-вывода реализуется в компьютерах, в которых внешние устройства управляются не процессором ввода-вывода, а контроллерами. Каждое внешнее устройство (или группа внешних устройств одного типа) имеет собственный контроллер, который автономно отрабатывает команды, поступающие от центрального процессора. При этом контроллер и центральный процессор работают асинхронно. Поскольку многие внешние устройства имеют электромеханические узлы, контроллер вы-

полняет свои команды управления устройствами существенно медленнее, чем центральный процессор — свои. Это обстоятельство используется для организации параллельного выполнения вычислений и операций ввода-вывода: в промежутке между передачей команд контроллеру центральный процессор может выполнять вычисления (рис. 4.1, *б*). Контроллер может сообщить центральному процессору о том, что он готов принять следующую команду, либо сигналом прерывания, либо центральный процессор узнает об этом, периодически опрашивая состояние контроллера.

Максимальный эффект ускорения достигается при наиболее полном перекрытии вычислений и ввода-вывода. Рассмотрим случай, когда процессор выполняет только одну задачу. В этой ситуации степень ускорения зависит от природы данной задачи и от того, насколько тщательно был выявлен возможный параллелизм при ее программировании. В задачах, в которых преобладают либо вычисления, либо ввод-вывод, ускорение почти отсутствует. Параллелизм в рамках одной задачи невозможен также, когда для продолжения вычислений необходимо полное завершение операции ввода-вывода, например, когда дальнейшие вычисления зависят от вводимых данных. В таких случаях неизбежны простои центрального процессора или канала.

Если же в системе выполняется одновременно несколько задач, появляется возможность совмещения вычислений одной задачи с вводом-выводом другой. Пока одна задача ожидает какого-либо события (заметим, что таким событием в мультипрограммной системе может быть не только завершение ввода-вывода, но и, например, наступление определенного момента времени, разблокирование файла или загрузка с диска недостающей страницы программы), процессор не простаивает, как это происходит при последовательном выполнении программ, а выполняет другую задачу.

Общее время выполнения смеси задач часто оказывается меньше, чем их суммарное время последовательного выполнения (рис. 4.2, *а*). Однако выполнение отдельной задачи в мультипрограммном режиме может занять больше времени, чем при монопольном выделении процессора этой задаче. Действительно, при совместном использовании процессора в системе могут возникать ситуации, когда задача не выполняется из-за того, что процессор занят выполнением другой задачи. В таких случаях задача, завершившая ввод-вывод, *готова* выполняться, но вынуждена ждать освобождения процессора, и это удлиняет срок ее выполнения. Так, из рис. 4.2 видно, что в однопрограммном режиме задача *A* выполняется за 6 единиц времени, а в мультипрограммном — за 7. Задача *B* также вместо 5 единиц времени выполняется за 6. Но зато время выполнения обоих задач в мультипрограммном режиме составляет всего 8 единиц, что на 3 единицы меньше, чем при последовательном выполнении.

В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит по инициативе самой активной задачи, например, когда она отказывается от процессора из-за необходимости выполнить операцию ввода-вывода. Поэтому существует высокая вероятность того, что одна задача может надолго занять процессор, и выполнение интерактивных

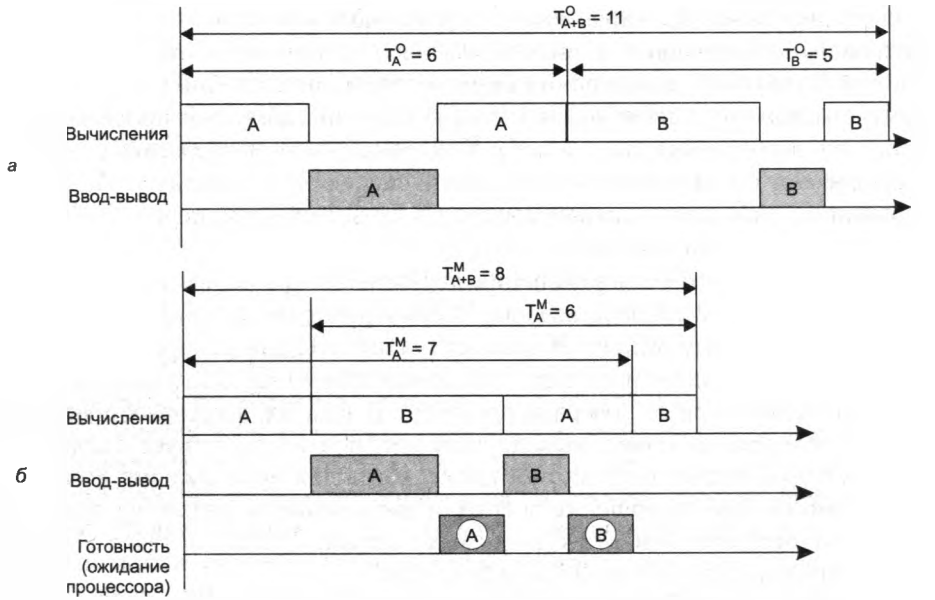


Рис. 4.2. Время выполнения двух задач: в однопрограммной системе (*а*), в мультипрограммной системе (*б*)

задач станет невозможным. Взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок повышает эффективность функционирования аппаратуры, но снижает эффективность работы пользователя.

Мультипрограммирование в системах разделения времени

Система разделения времени — это такая форма организации вычислительного процесса, при которой сразу несколько пользователей одновременно работают на компьютере, причем каждому из них кажется, что он получил компьютер в полное свое распоряжение. Главной целью и критерием эффективности систем разделения времени является *обеспечение удобства и эффективности работы пользователей.*

Другими словами, в системах разделения времени пользователям (в частном случае — одному пользователю) предоставляется возможность интерактивной работы сразу с несколькими приложениями. Для этого каждое приложение должно регулярно получать возможность «общения» с пользователем. Понятно, что в пакетных системах возможности диалога пользователя с приложением весьма ограничены.

В системах разделения времени эта проблема решается за счет того, что ОС принудительно периодически приостанавливает приложения, не дожидаясь, когда они добровольно освободят процессор. Всем приложениям попеременно выделяется квант процессорного времени, таким образом, пользователи, запустившие программы на выполнение, получают возможность поддерживать с ними диалог.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки — изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю в этом случае предоставляется терминал, с которого он может вести диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго, и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что каждый из них единолично использует машину.

Ясно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая «выгодна» системе. Кроме того, производительность системы снижается из-за возросших накладных расходов вычислительной мощности на более частое переключение процессора с задачи на задачу. Это вполне соответствует тому, что критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя. Тем не менее мультипрограммное выполнение интерактивных приложений повышает пропускную способность компьютера (пусть и не в такой степени, как пакетные системы) по сравнению с однопрограммной обработкой. Аппаратура загружается лучше, поскольку в то время, пока одно приложение ждет сообщения пользователя, процессор может заняться другими приложениями.

Мультипрограммирование в системах реального времени

Еще одна разновидность мультипрограммирования используется в **системах реального времени**, предназначенных для управления от компьютера различными техническими объектами (например, станком, спутником, научной экспериментальной установкой) или технологическими процессами (например, доменным процессом, гальванической линией). Управляющая система должна собирать информацию о состоянии управляемого объекта, которое в общем случае изменяется непредсказуемым образом. При наступлении того или иного события, например, когда температура раствора в гальванической ванне достигает заданного уровня, система должна выдавать соответствующее управляющее воздействие на исполнительный механизм, например нагревательный элемент.

Очень важно, что во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная управляющая объектом программа. В противном случае может произойти авария: спутник выйдет из зоны видимости, экспериментальные данные, поступающие с датчиков, будут потеряны, толщина гальванического покрытия не будет соответствовать норме. Таким образом, критерием эффективности здесь является способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (управляющего воздействия). Это время называется **временем реакции системы**, а соответствующее свойство системы — реактивностью. Требования ко времени реакции зависят от специфики управляемого процесса. Контроллер робота может требовать от встроенного компьютера ответ в течение менее 1 мс, в то время как при моделировании полета ответ даже в 40 мс может быть вполне приемлемым. Таким образом, мы пришли к следующему определению.

Операционная система реального времени — это система, предназначенная для управления физическими объектами (процессами), которая способна *обеспечить предсказуемое время реакции* в ответ на изменение состояния управляемого объекта (процесса).

В системах реального времени мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется по прерываниям (исходя из текущего состояния объекта) или в соответствии с расписанием плановых работ.

Способность аппаратуры компьютера и ОС к быстрому ответу зависит в основном от скорости переключения с одной задачи на другую и, в частности, от скорости обработки сигналов прерывания. Если при возникновении прерывания процессор должен опросить сотни потенциальных источников прерывания, то реакция системы будет слишком медленной. Время обработки прерывания в системах реального времени часто определяет требования к классу процессора даже при небольшой его загрузке.

В системах реального времени не стремятся максимально загружать все устройства, наоборот, при проектировании программного управляющего комплекса обычно закладывается некоторый «запас» вычислительной мощности на случай пиковой нагрузки. Статистические аргументы о низкой вероятности возникновения пиковой нагрузки, основанные на том, что вероятность одновременного возникновения большого количества независимых событий очень мала, ко многим ситуациям в системах управления неприменимы. Например, в системе управления атомной электростанцией в случае возникновения крупной аварии атомного реактора многие аварийные датчики сработают одновременно и создадут коррелированную нагрузку. Если система реального времени не спроектирована для поддержки пиковой нагрузки, то может случиться так, что система не справится с работой именно тогда, когда она нужна в наибольшей степени.

Мультипроцессорная обработка

Мультипроцессорная обработка — это такой способ организации вычислительного процесса в системах с несколькими процессорами, при котором несколько задач (процессов, потоков) могут одновременно выполняться на разных процессорах системы.

Концепция мультипроцессирования не нова, она известна с 70-х годов, но до середины 80-х доступных многопроцессорных систем не существовало. Однако к настоящему времени стало обычным включение нескольких процессоров в архитектуру даже персонального компьютера. Более того, многопроцессорность теперь является одним из необходимых требований, которые предъявляются к компьютерам, используемым в качестве центрального сервера более-менее крупной сети.

ПРИМЕЧАНИЕ Не следует путать мультипроцессорную обработку с мультипрограммной. В мультипрограммных системах параллельная работа разных устройств позволяет одновременно выполнять несколько программ, но при этом в процессоре в каждый момент времени выполняется только одна программа. То есть в этом случае несколько задач выполняются попеременно на одном процессоре, создавая лишь видимость параллельного выполнения. В мультипроцессорных же системах несколько задач выполняются действительно одновременно, так как имеется несколько обрабатывающих устройств — процессоров. Конечно, мультипроцессирование вовсе не исключает мультипрограммирования: на каждом из процессоров может попеременно выполняться некоторый закрепленный за данным процессором набор задач.

Мультипроцессорная организация системы приводит к усложнению всех алгоритмов управления ресурсами, например, требуется планировать процессы не для одного, а для нескольких процессоров, что гораздо сложнее. Сложности заключаются и в возрастании числа конфликтов по обращению к устройствам ввода-вывода, данным, общей памяти и совместно используемым программам. Необходимо предусмотреть эффективные средства блокировки при доступе к разделяемым информационным структурам ядра. Все эти проблемы должна решать операционная система путем синхронизации процессов, ведения очередей и планирования ресурсов. Более того, сама операционная система должны быть спроектирована так, чтобы уменьшить существующие взаимозависимости между собственными компонентами.

В наши дни становится общепринятым введение в ОС функций поддержки мультипроцессорной обработки данных. Такие функции имеются практически во всех популярных ОС, в частности во всех актуальных версиях Windows и Unix.

Мультипроцессорные системы часто характеризуют либо как симметричные, либо как несимметричные. При этом следует четко определять, к какому

аспекту мультипроцессорной системы относится эта характеристика — к типу *архитектуры* или к *способу организации вычислительного процесса.*

Симметричная архитектура мультипроцессорной системы предполагает однородность всех процессоров и единообразие включения процессоров в общую схему мультипроцессорной системы. Традиционные симметричные мультипроцессорные конфигурации разделяют одну большую память между всеми процессорами.

В симметричных архитектурах все процессы пользуются одной и той же схемой отображения памяти. Они могут очень быстро обмениваться данными, так что обеспечивается достаточно высокая производительность для тех приложений (например, при работе с базами данных), в которых несколько задач должны активно взаимодействовать между собой.

В **асимметричной архитектуре** разные процессоры могут отличаться как своими характеристиками (производительностью, надежностью, системой команд и т. д., вплоть до модели микропроцессора), так и функциональной ролью, которая поручается им в системе. Например, одни процессоры могут предназначаться для работы в качестве основных вычислителей, другие — для управления подсистемой ввода-вывода, третьи — еще для каких-то особых целей.

Функциональная неоднородность в асимметричных архитектурах влечет за собой структурные отличия во фрагментах системы, содержащих разные процессоры системы. Например, они могут отличаться схемами подключения процессоров к системной шине, набором периферийных устройств и способами взаимодействия процессоров с устройствами.

Другим аспектом мультипроцессорных систем, который может характеризоваться симметрией или ее отсутствием, является *способ организации вычислительного процесса.* Последний, как известно, определяется и реализуется операционной системой.

Асимметричное мультипроцессирование является наиболее простым способом организации вычислительного процесса в системах с несколькими процессорами. Этот способ часто называют также «ведущий-ведомый».

Функционирование системы по принципу «ведущий-ведомый» предполагает выделение одного из процессоров в качестве «ведущего», на котором работает операционная система и который управляет всеми остальными «ведомыми» процессорами. То есть ведущий процессор берет на себя функции распределения задач и ресурсов, а ведомые процессоры работают только как обрабатывающие устройства и никаких действий по организации работы вычислительной системы не выполняют.

Так как операционная система работает только на одном процессоре и функции управления полностью централизованы, то такая операционная система оказывается не намного сложнее ОС однопроцессорной системы.

Асимметричная организация вычислительного процесса может быть реализована как для симметричной мультипроцессорной архитектуры, в которой все

процессоры аппаратно неразличимы, так и для несимметричной, для которой характерна неоднородность процессоров, их специализация на аппаратном уровне.

В архитектурно асимметричных системах на роль ведущего процессора может быть назначен наиболее надежный и производительный процессор. Если в наборе процессоров имеется специализированный процессор, ориентированный, например, на матричные вычисления, то при планировании процессов операционная система, реализующая асимметричное мультипроцессирование, должна учитывать специфику этого процессора. Такая специализация снижает надежность системы в целом, так как процессоры не являются взаимозаменяемыми.

Симметричное мультипроцессирование как способ организации вычислительного процесса может быть реализовано в системах только с симметричной мультипроцессорной архитектурой. Напомним, что в таких системах процессоры работают с общими устройствами и разделяемой основной памятью.

Симметричное мультипроцессирование реализуется общей для всех процессоров операционной системой. При симметричной организации все процессоры равноправно участвуют и в управлении вычислительным процессом, и в выполнении прикладных задач. Например, сигнал прерывания от принтера, который распечатывает данные прикладного процесса, выполняемого на некотором процессоре, может быть обработан совсем другим процессором. Разные процессоры могут в какой-то момент одновременно обслуживать как разные, так и одинаковые модули общей операционной системы. Для этого программы операционной системы должны обладать свойством повторной входимости (**реентерабельности**).

Операционная система является полностью децентрализованной. Модули ОС выполняются на любом доступном процессоре. Как только процессор завершает выполнение очередной задачи, он передает управление планировщику задач. Планировщик выбирает из общей для всех процессоров системной очереди задачу, которая будет выполняться на данном процессоре следующей. Все ресурсы выделяются для каждой выполняемой задачи по мере возникновения в них потребностей и никак не закрепляются за процессором. При таком подходе все процессоры работают с одной и той же динамически выравниваемой нагрузкой. В решении одной задачи могут участвовать сразу несколько процессоров, если она допускает такое распараллеливание, например, путем представления в виде нескольких потоков.

В случае отказа одного из процессоров симметричные системы, как правило, сравнительно просто реконфигурируются, что является их большим преимуществом перед плохо реконфигурируемыми асимметричными системами.

Симметричный и асимметричный способы организации вычислительного процесса в мультипроцессорной системе не связаны напрямую с симметричным или асимметричным вариантом архитектуры, они определяются типом операционной системы. Так, в симметричных архитектурах вычислительный процесс может быть организован как симметричным образом, так и асимметричным. Однако асимметричная архитектура непременно влечет за собой и асимметричный способ организации вычислений.

Планирование процессов и потоков

Одной из основных подсистем мультипрограммной ОС, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами и потоками, которая занимается их созданием и уничтожением, поддерживает взаимодействие между ними, а также распределяет процессорное время между несколькими одновременно существующими в системе процессами и потоками.

Подсистема управления процессами и потоками ответственна за *обеспечение процессов необходимыми ресурсами.* ОС поддерживает в памяти специальные информационные структуры, в которые записывает, какие ресурсы выделены каждому процессу. Она может назначить процессу ресурсы в единоличное или совместное с другими процессами пользование. Некоторые из ресурсов выделяются процессу при его создании, другие — динамически по запросам во время выполнения. Ресурсы могут быть приписаны процессу на все время его жизни или только на определенный период. При выполнении этих функций подсистема управления процессами взаимодействует с другими подсистемами ОС, ответственными за управление ресурсами, такими как подсистема управления памятью, подсистема ввода-вывода, файловая система.

Когда в системе одновременно выполняется несколько независимых задач, то возникают дополнительные проблемы. Хотя потоки появляются и выполняются асинхронно, у них может возникнуть необходимость во взаимодействии, например, при обмене данными. Согласование скоростей потоков также очень важно для предотвращения эффекта «гонок» (когда несколько потоков пытаются изменить один и тот же файл), взаимных блокировок или других коллизий, которые возникают при совместном использовании ресурсов. *Синхронизация* потоков является одной из важных функций подсистемы управления процессами и потоками.

Каждый раз, когда процесс завершается, ОС предпринимает шаги, чтобы «зачистить следы» его пребывания в системе. Подсистема управления процессами закрывает все файлы, с которыми работал процесс, освобождает области оперативной памяти, отведенные под коды, данные и системные информационные структуры процесса. Выполняется коррекция всевозможных очередей ОС и списков ресурсов, в которых имелись ссылки на завершаемый процесс.

Понятия «процесс» и «поток»

Чтобы поддерживать мультипрограммирование, ОС должна определить и оформить для себя те внутренние единицы работы, между которыми будет разделяться процессор и другие ресурсы компьютера. В настоящее время в большинстве операционных систем определены два типа единиц работы. Более крупная единица работы, обычно носящая название **процесс**, или **задача**, требует для своего выполнения нескольких более мелких работ, для обозначения которых используют термины **поток**, или **нить**.

ПРИМЕЧАНИЕ При использовании этих терминов часто возникают сложности. Это происходит в силу нескольких причин. Во-первых, из-за специфики различных ОС, когда совпадающие по сути понятия получили разные названия, например, задача (task) в OS/2, OS/360 и процесс (process) в Unix, NetWare и семействе ОС Windows. Во-торых, по мере развития системного программирования и методов организации вычислений некоторые из этих терминов получили новое смысловое значение, особенно это касается понятия «процесс», который уступил многие свои свойства новому понятию «поток». В-третьих, терминологические сложности порождаются наличием нескольких вариантов перевода англоязычных терминов на русский язык. Например, термин «thread» переводится как «нить», «поток», «облегченный процесс», «мини-задача» и др. Далее в книге в качестве названий единиц работы ОС используются термины «процесс» и «поток». В тех же случаях, когда различия между этими понятиями не играют существенной роли, они объединяются под обобщенным термином «задача».

Итак, в чем же состоят принципиальные отличия в понятиях «процесс» и «поток»?

Очевидно, что любая работа вычислительной системы заключается в выполнении некоторой программы. Поэтому и с процессом, и с потоком связывается определенный программный код, который для этих целей оформляется в виде исполняемого модуля. Чтобы этот программный код мог быть выполнен, его необходимо загрузить в оперативную память, возможно, выделить некоторое место на диске для хранения данных, предоставить доступ к устройствам ввода-вывода, например к последовательному порту для получения данных по подключенному к этому порту модему и т. д. В ходе выполнения программе может также понадобиться доступ к информационным ресурсам, например файлам, портам TCP/UPD, семафорам. И, конечно же, невозможно выполнение программы без предоставления ей процессорного времени, то есть времени, в течение которого процессор выполняет коды данной программы.

В операционных системах, где существуют и процессы, и потоки, процесс рассматривается операционной системой как *заявка на потребление всех видов ресурсов* кроме одного — процессорного времени. Этот последний важнейший ресурс распределяется операционной системой между другими единицами работы — потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд.

В простейшем случае процесс состоит из одного потока, и именно таким образом трактовалось понятие «процесс» до середины 80-х годов (например, в ранних версиях Unix) и в таком же виде оно сохранилось в некоторых современных ОС. В таких системах понятие «поток» полностью поглощается понятием «процесс», то есть остается только одна единица работы и потребления ресурсов — процесс. Мультипрограммирование осуществляется в таких ОС на уровне процессов.

Для того чтобы процессы не могли вмешаться в распределение ресурсов, а также не могли повредить коды и данные друг друга, важнейшей задачей ОС

является *изоляция одного процесса от другого*. Для этого операционная система обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

Виртуальное адресное пространство процесса — это совокупность адресов, которыми может манипулировать программный модуль процесса. Операционная система отображает виртуальное адресное пространство процесса на отведенную процессу физическую память.

Содержимое назначенного процессу виртуального адресного пространства, то есть коды команд, исходные и промежуточные данные, а также результаты вычислений, называют **образом процесса**.

При необходимости взаимодействия процессы обращаются к операционной системе, которая, выполняя функции посредника, предоставляет им *средства межпроцессной связи* — конвейеры, почтовые ящики, разделяемые секции памяти и некоторые другие.

Однако в системах, в которых отсутствует понятие потока, возникают проблемы при организации параллельных вычислений в рамках процесса. А такая необходимость может возникать. Действительно, при мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем в однопрограммном режиме (всякое разделение ресурсов только замедляет работу одного из участников из-за дополнительных затрат времени на ожидание освобождения ресурса). Однако приложение, выполняемое в рамках одного процесса, может обладать внутренним параллелизмом, который в принципе мог бы позволить ускорить его выполнение. Если, например, в программе предусмотрено обращение к внешнему устройству, то на время этой операции можно не блокировать выполнение всего процесса, а продолжить вычисления по другой ветви программы. Параллельное выполнение нескольких работ в рамках одного интерактивного приложения повышает эффективность работы пользователя. Так, при работе с текстовым редактором желательно иметь возможность совмещать набор нового текста с такими продолжительными по времени операциями, как переформатирование значительной части текста, печать документа или его сохранение на локальном или удаленном диске. Еще одним примером необходимости распараллеливания является сетевой сервер баз данных. В этом случае параллелизм желателен как для обслуживания различных запросов к базе данных, так и для более быстрого выполнения отдельного запроса за счет одновременного просмотра различных записей базы.

Потоки возникли в операционных системах как *средство распараллеливания вычислений*. Конечно, задача распараллеливания вычислений в рамках одного приложения может быть решена и традиционными способами.

Во-первых, прикладной программист может взять на себя сложную задачу организации параллелизма, выделив в приложении некоторую подпрограмму-диспетчер, которая периодически передает управление той или иной ветви вычислений. При этом программа получается логически весьма запутанной

с многочисленными передачами управления, что существенно затрудняет ее отладку и модификацию.

Во-вторых, решением является создание в рамках одного приложения нескольких процессов для каждой из параллельных работ. Однако использование для создания процессов стандартных средств ОС не позволяет учесть тот факт, что эти процессы решают единую задачу, а значит, имеют много общего между собой — они могут работать с одними и теми же данными, использовать один и тот же кодовый сегмент, наделяться одними и теми же правами доступа к ресурсам вычислительной системы. Так, если в примере с сервером баз данных создавать отдельные процессы для каждого запроса, поступающего из сети, то все процессы будут выполнять один и тот же программный код и реализовывать поиск в записях общих для всех процессов файлов данных. А операционная система при таком подходе будет рассматривать эти процессы наравне со всеми остальными процессами и с помощью универсальных механизмов обеспечивать их изоляцию друг от друга. В данном случае все эти достаточно громоздкие механизмы применяются явно не по назначению, выполняя не только бесполезную, но и «вредную» работу, затрудняющую обмен данными между различными частями приложения. Кроме того, на создание каждого процесса ОС тратит определенные системные ресурсы, которые в данном случае неоправданно дублируются — каждому процессу выделяется собственное виртуальное адресное пространство, физическая память, закрепляются устройства ввода-вывода и т. п.

Все это сделало необходимым разработку более совершенного способа организации вычислений, которым и стала многопоточная обработка.

Многопоточная обработка (multithreading) представляет собой механизм распараллеливания вычислений, который учитывает тесные связи между отдельными ветвями вычислений одного и того же приложения.

При этом вводится новая единица работы — поток выполнения, а понятие «процесс» в значительной степени меняет смысл. Понятию «поток» соответствует последовательный переход процессора от одной команды программы к другой команде. ОС распределяет процессорное время между потоками. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

ПРИМЕЧАНИЕ Заметим, что в однопрограммных системах не возникает необходимости введения понятия, обозначающего единицу работы, так как там не существует проблемы разделения ресурсов.

Создание потоков требует от ОС меньших накладных расходов, чем создание процессов. В отличие от процессов, которые принадлежат разным, вообще говоря, конкурирующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, нежели процессы в традиционной мультипрограммной системе. Все потоки одного процесса используют общие файлы, таймеры, устройства,

одну и ту же область оперативной памяти, одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому виртуальному адресу процесса, один поток может использовать стек другого потока. Между потоками одного процесса нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Чтобы организовать взаимодействие и обмен данными, потокам вовсе не требуется обращаться к ОС, им достаточно иметь общую память — один поток записывает данные, а другой читает их. В то же время потоки разных процессов по-прежнему хорошо защищены друг от друга.

Итак, мультипрограммирование более эффективно на уровне потоков, а не процессов. Каждый поток имеет собственные счетчик команд и стек. Задача, оформленная в виде нескольких потоков в рамках одного процесса, может быть выполнена быстрее за счет псевдопараллельного (или параллельного в мультипроцессорной системе) выполнения ее отдельных частей. Например, если электронная таблица была разработана с учетом возможностей многопоточной обработки, то пользователь может запросить пересчет своего рабочего листа и одновременно продолжать заполнять таблицу. Особенно эффективно можно использовать многопоточность для выполнения распределенных приложений, например многопоточный сервер может параллельно выполнять запросы сразу нескольких клиентов.

Использование потоков связано не только со стремлением повысить производительность системы за счет параллельных вычислений, но и с целью создания более читабельных, логичных программ. Введение нескольких потоков выполнения упрощает программирование. Например, в задачах типа «писатель-читатель» один поток выполняет запись в буфер, а другой считывает записи из него. Поскольку они разделяют общий буфер, не стоит их делать отдельными процессами. Другой пример использования потоков — управление сигналами, такими как прерывание с клавиатуры (`del` или `break`). Вместо обработки сигнала прерывания один поток назначается для постоянного ожидания поступления сигналов. Таким образом, применение потоков может сократить потребность в прерываниях пользовательского уровня. В этих примерах важно не столько параллельное выполнение, сколько понятность программы.

Наибольший эффект от введения многопоточной обработки достигается в мультипроцессорных системах, в которых потоки, в том числе принадлежащие одному процессу, могут выполняться на разных процессорах действительно параллельно (а не псевдопараллельно).

Создание процессов и потоков

Создать процесс — это, прежде всего, означает создать **описатель процесса**, в качестве которого выступает одна или несколько информационных структур, содержащих все сведения о процессе, необходимые операционной системе для управления им. В число таких сведений могут входить, например, идентификатор процесса, данные о расположении в памяти исполняемого модуля, степень привилегированности процесса (приоритет и права доступа) и т. п. Примерами

описателей процесса являются **блок управления задачей** (Task Control Block, TCB) в OS/360, **управляющий блок процесса** (Process Control Block, PCB) в OS/2, **дескриптор процесса** в Unix, **объект-процесс** (object-process) в ОС семейства Windows NT.

Создание описателя процесса знаменует собой появление в системе еще одного претендента на вычислительные ресурсы. Начиная с этого момента при распределении ресурсов ОС должна принимать во внимание потребности нового процесса.

Создание процесса включает загрузку кодов и данных исполняемой программы данного процесса с диска в оперативную память. Для этого ОС должна обнаружить местоположение такой программы на диске, перераспределить оперативную память и выделить память исполняемой программе нового процесса. Затем необходимо считать программу в выделенные для нее участки памяти и, возможно, изменить параметры программы в зависимости от размещения в памяти. В системах с виртуальной памятью в начальный момент может загружаться только часть кодов и данных процесса, с тем чтобы «подкачивать» остальные по мере необходимости. Существуют системы, в которых на этапе создания процесса не требуется непременно загружать коды и данные в оперативную память, вместо этого исполняемый модуль копируется из того каталога файловой системы, в котором он изначально находился, в область подкачки — специальную область диска, отведенную для хранения кодов и данных процессов. При выполнении всех этих действий подсистема управления процессами тесно взаимодействует с подсистемой управления памятью и файловой системой.

В многопоточной системе при создании процесса ОС создает для каждого процесса как минимум один поток выполнения. При создании потока так же, как и при создании процесса, операционная система генерирует специальную информационную структуру — описатель потока, который содержит идентификатор потока, данные о правах доступа и приоритете, о состоянии потока и другую информацию. В исходном состоянии поток (или процесс, если речь идет о системе, в которой понятие «поток» не определяется) находится в приостановленном состоянии. Момент выборки потока на выполнение выбирается в соответствии с принятым в данной системе правилом предоставления процессорного времени и с учетом всех существующих в данный момент потоков и процессов. В случае если коды и данные процесса находятся в области подкачки, необходимым условием активизации потока процесса является также наличие места в оперативной памяти для загрузки его исполняемого модуля.

Во многих системах поток может обратиться к ОС с запросом на создание так называемых потоков-потомков. В разных ОС по-разному строятся отношения между потоками-потомками и их родителями. Например, в одних ОС выполнение родительского потока синхронизируется с его потомками, в частности после завершения родительского потока ОС может снять с выполнения всех его потомков. В других системах потоки-потомки могут выполняться асинхронно по отношению к родительскому потоку. Потомки, как правило, наследуют

многие свойства родительских потоков. Во многих системах порождение потомков является основным механизмом создания процессов и потоков.

Рассмотрим в качестве примера создание процессов в одной из самых «заслуженных» ОС – Unix System V Release 4. В этой системе потоки не поддерживаются, а в качестве единицы управления и потребления ресурсов выступает процесс. При управлении процессами операционная система использует два основных типа информационных структур: дескриптор процесса и контекст процесса.

Дескриптор процесса – это информационная структура, содержащая информацию о процессе. Эта информация необходима ядру ОС в течение всего жизненного цикла процесса независимо от того, находится он в активном или пассивном состоянии, загружен образ процесса в оперативную память или вытеснен на диск.

Дескрипторы отдельных процессов объединены в список, образующий таблицу процессов. Память для таблицы процессов отводится динамически в области ядра. На основании информации, содержащейся в таблице процессов, операционная система осуществляет планирование и синхронизацию процессов. В дескрипторе прямо или косвенно (через указатели на связанные с процессом структуры) содержится информация о состоянии процесса, о расположении образа процесса в оперативной памяти и на диске, о значении отдельных составляющих приоритета, а также о его итоговом значении – глобальном приоритете, об идентификаторе пользователя, создавшего процесс, о родственных процессах, о событиях, осуществления которых ожидает данный процесс, и некоторая другая информация.

Контекст процесса содержит менее оперативную, но более объемную часть информации о процессе, необходимую операционной системе для возобновления выполнения процесса с прерванного места.

Контекст включает содержимое регистров процессора, коды ошибок выполняемых процессором системных вызовов, информацию обо всех открытых данным процессом файлах и незавершенных операциях ввода-вывода и другие данные, характеризующие состояние вычислительной среды в момент прерывания. Контекст, так же как и дескриптор процесса, доступен только программам ядра, то есть находится в виртуальном адресном пространстве операционной системы, однако он хранится не в области ядра, а непосредственно примыкает к образу процесса и перемещается вместе с ним, если это необходимо, из оперативной памяти на диск.

Порождение процессов в системе Unix происходит в результате выполнения системного вызова `fork`. Операционная система строит образ порожденного процесса, являющийся точной копией образа породившего процесса, то есть дублируются дескриптор, контекст и образ процесса. Сегмент данных и сегмент стека родительского процесса копируются в новое место, образуя сегменты дан-

ных и стека процесса-потомка. Процедурный сегмент копируется только тогда, когда он не является разделяемым. В противном случае процесс-потомок становится еще одним процессом, разделяющим данный процедурный сегмент.

После выполнения системного вызова `fork` оба процесса продолжают выполнение с одной и той же точки. Чтобы процесс мог узнать, является он родительским процессом или дочерним, системный вызов `fork` возвращает в качестве своего значения в породивший процесс идентификатор порожденного процесса, а в порожденный процесс — `NULL`. Типичное разветвление на языке С записывается так:

```
if( fork() ) { действия родительского процесса }
  else { действия порожденного процесса }
```

Идентификатор потомка может быть присвоен переменной, входящей в контекст родителя. Так как контекст процесса наследуется его потомками, то потомки могут узнать идентификаторы своих старших «братьев», таким образом, сумма знаний наследуется при порождении и может быть распространена между родственными процессами. На независимости идентификатора процесса от выполняемой процессом программы построен механизм, позволяющий процессу перейти к выполнению другой программы с помощью системного вызова `exec`.

Таким образом, в Unix порождение нового процесса происходит в два этапа — сначала создается копия процесса-родителя, затем у нового процесса производится замена кодового сегмента заданным.

Планирование и диспетчеризация потоков

На протяжении существования процесса выполнение его потоков может быть многократно прервано и продолжено. (В системе, не поддерживающей потоки, все сказанное далее о планировании и диспетчеризации относится к процессу в целом.) Переход от выполнения одного потока к выполнению другого осуществляется в результате планирования и диспетчеризации.

> Работа по определению того, в какой момент необходимо прервать выполнение текущего активного потока и какому потоку предоставить возможность выполняться, называется **планированием**.

Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. При планировании могут приниматься во внимание приоритеты потоков, время их ожидания в очереди, накопленное время выполнения, интенсивность обращений к вводу-выводу и другие факторы. ОС планирует выполнение потоков независимо от того, принадлежат они одному или разным процессам. Так, например, после выполнения потока некоторого процесса ОС может выбрать для выполнения другой поток того же процесса или же назначить к выполнению поток другого процесса.

Планирование потоков, по существу, требует решения двух задач:

- определение момента времени для смены текущего активного потока;
- выбор для выполнения потока из очереди готовых потоков.

Существует множество различных алгоритмов планирования потоков, посвоему решающих каждую из этих задач. Алгоритмы планирования могут преследовать различные цели и обеспечивать разное качество мультипрограммирования. Например, в одном случае выбирается такой алгоритм планирования, при котором гарантируется, что ни один поток/процесс не будет занимать процессор дольше определенного времени, в другом случае целью является максимально быстрое выполнение «коротких» задач, в третьем — преимущественное право занять процессор получают потоки интерактивных приложений. Именно особенности реализации механизма планирования потоков в наибольшей степени определяют специфику операционной системы, в частности является ли она системой пакетной обработки, системой разделения времени или системой реального времени.

В большинстве операционных систем универсального назначения планирование осуществляется *динамически* (on-line), то есть решения принимаются во время работы системы на основе анализа текущей ситуации. ОС работает в условиях неопределенности — потоки и процессы появляются в случайные моменты времени и также непредсказуемо завершаются. Динамические планировщики могут гибко приспосабливаться к изменяющейся ситуации и не делают никаких предположений относительно мультипрограммной смеси. Для того чтобы оперативно найти в условиях такой неопределенности оптимальный в некотором смысле порядок выполнения задач, операционная система должна затрачивать значительные усилия.

Другой тип планирования — статический. Он может быть использован в специализированных системах, в которых весь набор одновременно выполняемых задач определен заранее, например в системах реального времени.

Планировщик называется *статическим*, или *предварительным*, если он принимает решения о планировании не во время работы системы, а заранее (off-line). Соотношение между динамическим и статическим планировщиками аналогично соотношению между диспетчером железной дороги, который пропускает поезда строго по предварительно составленному расписанию, и регулировщиком на перекрестке автомобильных дорог, не оснащенном светофорами, который решает, какую машину остановить, а какую пропустить, в зависимости от ситуации на перекрестке.

Результатом работы статического планировщика является таблица, называемая расписанием, в которой указывается, какому потоку/процессу, когда и на какое время должен быть предоставлен процессор. Для построения расписания планировщику нужны как можно более полные предварительные знания о характеристиках набора задач, например о максимальном времени выполнения каждой задачи, ограничениях предшествования, ограничениях по взаимному исключению, предельным срокам и т. д.

После того как расписание готово, оно может использоваться операционной системой для переключения потоков и процессов. При этом накладные расходы ОС на исполнение расписания оказываются значительно меньшими, чем при ди-

намическом планировании, и сводятся лишь к диспетчеризации потоков/ процессов.

Диспетчеризация заключается в реализации найденного в результате планирования (динамического или статистического) решения, то есть в переключении процессора с одного потока на другой.

Прежде чем прервать выполнение потока, ОС запоминает его контекст, с тем чтобы впоследствии использовать эту информацию для последующего возобновления выполнения данного потока. Контекст отражает, во-первых, состояние аппаратуры компьютера в момент прерывания потока: значение счетчика команд, содержимое регистров общего назначения, режим работы процессора, флаги, маски прерываний и другие параметры. Во-вторых, контекст включает параметры операционной среды, а именно ссылки на открытые файлы, данные о незавершенных операциях ввода-вывода, коды ошибок выполняемых данным потоком системных вызовов и т. д.

Диспетчеризация сводится к следующему:

- сохранение контекста текущего потока, который требуется сменить;
- загрузка контекста нового потока, выбранного в результате планирования;
- запуск нового потока на выполнение.

Поскольку операция переключения контекстов существенно влияет на производительность вычислительной системы, программные модули ОС выполняют диспетчеризацию потоков совместно с аппаратными средствами процессора.

В контексте потока можно выделить часть, общую для всех потоков данного процесса (ссылки на открытые файлы), и часть, относящуюся только к данному потоку (содержимое регистров, счетчик команд, режим процессора). Переменные глобального контекста доступны для всех потоков, созданных в рамках одного процесса. Переменные локального контекста доступны только для кодов определенного потока, аналогично локальным переменным функции. Очевидно, что такая иерархическая организация контекстов ускоряет переключение потоков, так как при переключении с потока на поток в пределах одной группы нет необходимости заменять контексты групп или глобальные контексты, достаточно заменить контексты потоков, которые имеют меньший объем. Аналогично, при переключении с потока одной группы на поток другой группы в пределах одного процесса глобальный контекст не изменяется, а изменяется лишь контекст группы. Переключение же глобальных контекстов происходит только при переходе с потока одного процесса на поток другого процесса.

ПРИМЕЧАНИЕ В различных ОС можно встретить компоненты ОС, имеющие названия планировщик (scheduler) и диспетчер (dispatcher). Не следует однозначно судить о функциональном назначении этих компонентов по их названиям, то есть считать, что планировщик выполняет планирование, а диспетчер — диспетчеризацию в том смысле, в котором эти функции были определены ранее. Чаще всего оба названия используются для обозначения компонентов, которые занимаются планированием.

Состояния потока

Операционная система выполняет планирование потоков, принимая во внимание их состояние. В мультипрограммной системе поток может находиться в одном из трех основных состояний:

- *выполнение* — активное состояние потока, во время которого поток обладает всеми необходимыми ресурсами и непосредственно обрабатывается процессором;
- *ожидание* — пассивное состояние потока, находясь в котором поток заблокирован по своим внутренним причинам (ждет осуществления некоторого события, например завершения операции ввода-вывода, получения сообщения от другого потока или освобождения какого-либо необходимого ему ресурса);
- *готовность* — также пассивное состояние потока, но в этом случае поток заблокирован в связи с внешним по отношению к нему обстоятельством (имеет все требуемые для него ресурсы, готов выполняться, однако процессор занят выполнением другого потока).

ПРИМЕЧАНИЕ Состояния выполнения и ожидания могут быть отнесены и к задачам, выполняющимся в однопрограммном режиме, а вот состояние готовности характерно только для режима мультипрограммирования.

В течение своей жизни каждый поток переходит из одного состояния в другое в соответствии с алгоритмом планирования потоков, принятым в данной операционной системе.

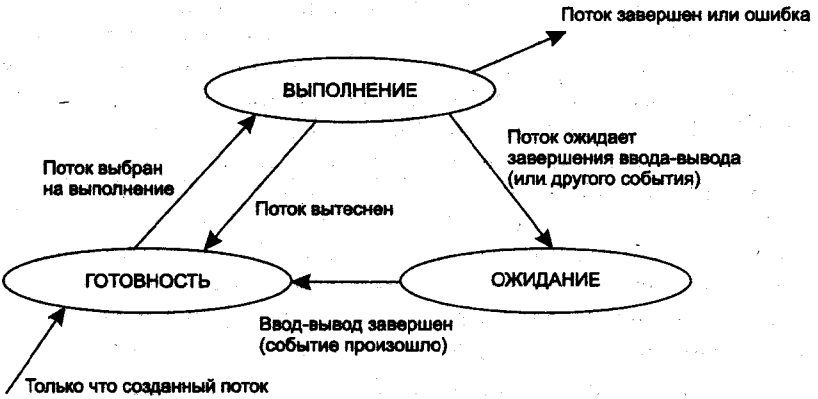


Рис. 4.3. Граф состояний потока в многозадачной среде

Рассмотрим типичный граф состояния потока (рис. 4.3). Только что созданный поток находится в состоянии готовности, он готов к выполнению и стоит в очереди к процессору. Когда в результате планирования подсистема управления потоками принимает решение об активизации данного потока, он перехо-

дит в состояние выполнения и находится в нем до тех пор, пока либо сам освободит процессор, перейдя в состояние ожидания какого-нибудь события, либо будет принудительно «вытеснен» из процессора, например, вследствие исчерпания отведенного данному потоку кванта процессорного времени. В последнем случае поток возвращается в состояние готовности. В это же состояние поток переходит из состояния ожидания, после того как ожидаемое событие произойдет.

В состоянии выполнения в однопроцессорной системе может находиться не более одного потока, а в каждом из состояний ожидания и готовности — несколько потоков. Эти потоки образуют очереди соответственно ожидающих и готовых потоков. Очереди потоков организуются путем объединения в списки описателей отдельных потоков. Таким образом, каждый описатель потока, кроме всего прочего, содержит, по крайней мере, один указатель на другой описатель, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать потоки, переводить потоки из одного состояния в другое. Если предположить, что на рис. 4.4 показана очередь готовых потоков, то запланированный порядок выполнения выглядит так: *A, B, E, D, C.*

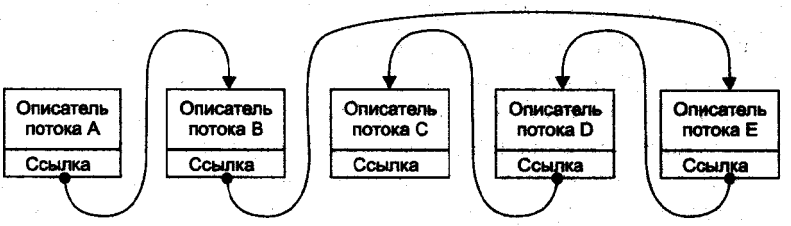


Рис. 4.4. Очередь потоков

Вытесняющие и не вытесняющие алгоритмы планирования

С самых общих позиций все множество алгоритмов планирования можно разделить на два класса: вытесняющие и не вытесняющие.

- *Не вытесняющие* (non-preemptive) алгоритмы основаны на том, что активному потоку позволяется выполняться, пока он сам, по собственной инициативе, не отдаст управление операционной системе, чтобы та выбрала из очереди другой готовый к выполнению поток.
- *Вытесняющие* (preemptive) алгоритмы — это такие алгоритмы планирования потоков, в которых решение о переключении процессора с выполнения одного потока на выполнение другого принимается операционной системой, а не активной задачей.

Основным различием между вытесняющими и не вытесняющими алгоритмами является степень централизации механизма планирования потоков. При вытесняющем мультипрограммировании функции планирования потоков

целиком сосредоточены в операционной системе, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться одновременно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активного потока, запоминает его контекст, выбирает из очереди готовых потоков следующий, запускает новый поток на выполнение, загружая его контекст.

При не вытесняющем мультипрограммировании механизм планирования распределен между операционной системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения очередного цикла своего выполнения и только затем передает управление ОС с помощью какого-либо системного вызова. ОС формирует очереди потоков и выбирает в соответствии с некоторым правилом (например, с учетом приоритетов) следующий поток на выполнение. Такой механизм создает проблемы как для пользователей, так и для разработчиков приложений.

Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением (а не пользователем). Если приложение тратит слишком много времени на выполнение какой-либо работы, например на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу, например на текстовый редактор, в то время как форматирование продолжалось бы в фоновом режиме.

Поэтому разработчики приложений для операционной среды с невытесняющей многозадачностью вынуждены, возлагая на себя часть функций планировщика, создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Например, программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени между задачами работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить «дружественное» отношение своей программы к другим выполняемым одновременно с ней программам. Для этого в программе должны быть предусмотрены частые передачи управления операционной системе. Крайним проявлением «недружественности» приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм имеет возможность снять зависшую задачу с выполнения.

Однако распределение функций планирования потоков между системой и приложениями не всегда является недостатком, при определенных условиях это может стать преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Так как разработчик сам определяет в программе момент возвращения управления, исключается возмож-

ность нерационального прерывания программ в «неудобные» для них моменты времени. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждого цикла выполнения использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит данные. Существенным преимуществом невытесняющего планирования является более высокая скорость переключения с потока на поток.

ПРИМЕЧАНИЕ Понятия вытесняющих и не вытесняющих алгоритмов планирования иногда отождествляют с понятиями приоритетных и бесприоритетных дисциплин, что возможно связано со сходным звучанием соответствующих англоязычных терминов preemptive и non-preemptive. Однако это совершенно неверно, так как приоритеты в том и другом случаях могут как использоваться, так и не использоваться.

Почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений (в том числе в различных версиях операционных систем семейств Unix/Linux и Windows NT), реализованы вытесняющие алгоритмы планирования потоков (процессов).

Примерами применения эффективного невытесняющего планирования являются популярные файловые серверы 90-х годов Novell NetWare 3.x и 4.x, в которых, в значительной степени благодаря такому планированию, была достигнута высокая скорость выполнения файловых операций. Рассмотрим реализацию невытесняющего планирования в этих ОС более подробно.

Чтобы не занимать процессор слишком долго, поток в NetWare 3.x и 4.x сам передает управление планировщику ОС, используя следующие системные вызовы:

- `ThreadSwitch` — поток, вызвавший эту функцию, считает себя готовым к немедленному выполнению, но отдает управления для того, чтобы могли выполняться и другие потоки;
- `ThreadSwitchWithDelay` — функция аналогична предыдущей, но поток считает, что будет готов к выполнению только через определенное количество переключений с потока на поток;
- `Delay` — функция, аналогичная предыдущей, но задержка дается в миллисекундах;
- `ThreadSwitchLowPriority` — функция передачи управления, отличающаяся от `ThreadSwitch` тем, что поток просит поместить его в очередь готовых к выполнению, но низкоприоритетных потоков.

Планировщик использует несколько очередей готовых потоков (рис. 4.5). Только что созданный поток попадает в конец очереди `RunList`, которая содержит готовые к выполнению потоки. После отказа от процессора поток попадает в ту или иную очередь в зависимости от того, какой из системных вызовов был использован при передаче управления. Поток поступает в конец очереди `RunList` при вызове `ThreadSwitch`, в конец очереди `DelayedWorkToDoList` при

вызове `ThreadSwitchWithDelay` или `Delay` и в конец очереди `LowPriorityRunList` при вызове `ThreadSwitchLowPriority`.

После того как выполнявшийся процессором поток завершит свой очередной цикл выполнения, отдав управление с помощью одного из вызовов передачи управления (или вызова ожидания на семафоре), планировщик выбирает для выполнения стоящий первым в очереди `RunList` поток и запускает его.

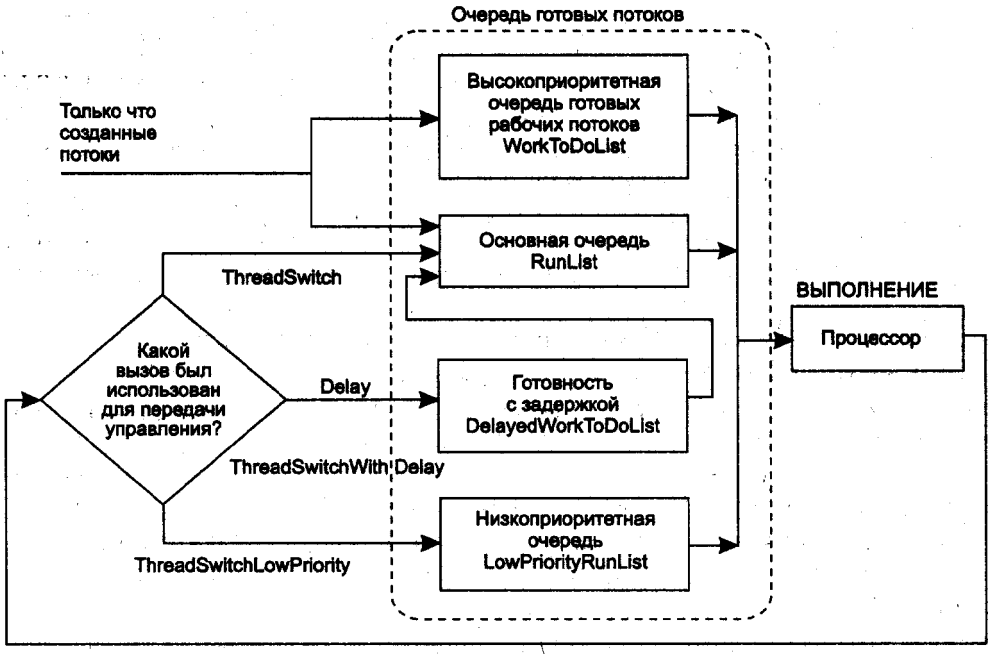


Рис. 4.5. Схема планирования потоков в NetWare 3.x и 4.x

Потоки, находящиеся в очереди `DelayedWorkToDoList`, после завершения условия ожидания перемещаются в конец очереди `RunList`.

Потоки, находящиеся в очереди `LowPriorityRunList`, запускаются на выполнение только в том случае, если очередь `RunList` пуста. Обычно в эту очередь назначаются потоки, выполняющие несрочную фоновую работу.

Очередь `WorkToDoList` является в системе самой приоритетной. В эту очередь попадают так называемые рабочие потоки. В рассматриваемых ОС NetWare, как и в некоторых других ОС, вместо создания нового потока для выполнения определенной работы может быть использован уже существующий системный поток. Пул рабочих потоков создается при старте системы для системных целей и выполнения срочных работ. Рабочие потоки ОС обладают наивысшим приоритетом, то есть поступают на выполнение перед потоками из очереди `RunList`. Планировщик разрешает выполниться подряд только определенному количеству потоков из очереди `WorkToDoList`, а затем запускает поток из очереди `RunList`.

Описанный невытесняющий механизм организации многопоточной работы в ОС NetWare 3.x и NetWare 4.x потенциально более производителен, чем вытесняющий, так как отличается небольшими накладными расходами ОС на диспетчеризацию потоков за счет простых алгоритмов планирования и иерархии контекстов. Но для достижения высокой производительности к разработчикам приложений для ОС, подобных NetWare3.x и NetWare 4.x, предъявляются более высокие требования, так как распределение процессорного времени между различными приложениями зависит, в конечном счете, от искусства программиста. Еще одной причиной отказа разработчиков Novell от использования этого алгоритма планирования в своих ОС является то, что с увеличением вычислительной мощности микропроцессоров эффективность планирования процессов стала все меньше и меньше влиять на производительность системы в целом. С появлением микропроцессора Pentium сложность разработки приложений для ОС NetWare окончательно перевесила то преимущество, которое имела эта система в производительности. Поэтому компания Novell в следующих версиях NetWare 5.x и 6.x отказалась от вытесняющей многозадачности.

Алгоритмы планирования, основанные на квантовании

В основе многих вытесняющих алгоритмов планирования лежит концепция *квантования*.

Квант — ограниченный непрерывный период процессорного времени, поочередно предоставляемый для выполнения каждому потоку.

Смена активного потока происходит, если:

- поток завершился и покинул систему;
- произошла ошибка;
- поток перешел в состояние ожидания;
- исчерпан квант процессорного времени, отведенный данному потоку.

Поток, который исчерпал свой квант, переводится в состояние готовности и ожидает, когда ему будет предоставлен новый квант процессорного времени. При этом на выполнение в соответствии с определенным правилом выбирается новый поток из очереди готовых. Граф состояний потока, изображенный на рис. 4.6, соответствует алгоритму планирования, основанному на квантовании.

Кванты, выделяемые потокам, могут быть одинаковыми для всех потоков или различными. Рассмотрим, например, случай, когда всем потокам предоставляются кванты одинаковой длины q (рис. 4.7). Если в системе имеется n потоков, то время, которое поток проводит в ожидании следующего кванта, можно грубо оценить как $q(n-1)$. Чем больше потоков в системе, тем больше время ожидания и тем меньше возможностей вести одновременную интерактивную работу нескольким пользователям. Но если величина кванта выбрана очень небольшой, то значение произведения $q(n-1)$ все равно будет достаточно

малым для того, чтобы пользователь не ощущал дискомфорта от присутствия в системе других пользователей. Типичное значение кванта в системах разделения времени составляет десятки миллисекунд.

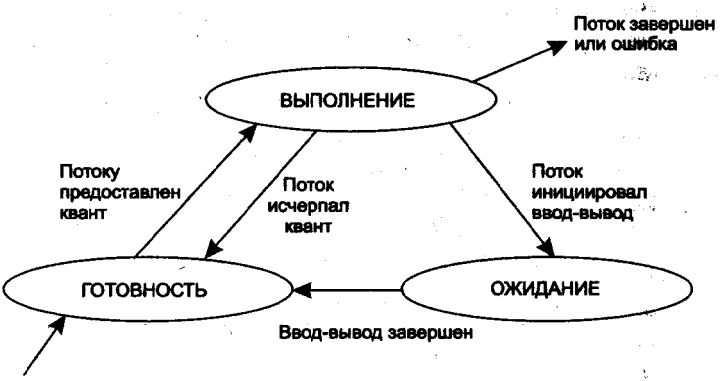


Рис. 4.6. Граф состояний потока в системе с квантованием

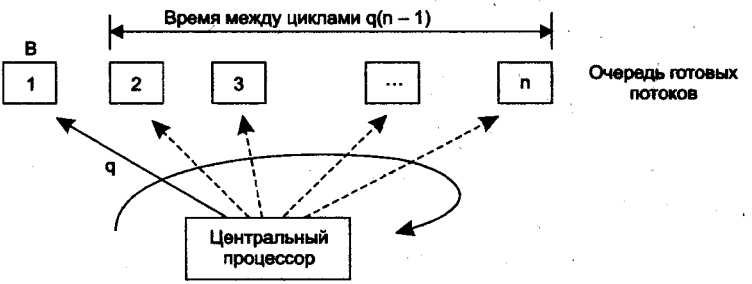


Рис. 4.7. Иллюстрация расчета времени ожидания в очереди

Если квант короткий, то суммарное время, которое проводит поток в ожидании процессора, прямо пропорционально времени, требуемому для его выполнения (то есть времени, которое потребовалось бы для выполнения этого потока при монопольном использовании вычислительной системы). Действительно, поскольку время ожидания между двумя циклами выполнения равно $q(n-1)$, а количество циклов B/q, где B — требуемое время выполнения, то $W = B(n-1)$. Заметим, что эти соотношения представляют собой весьма грубые оценки, основанные на предположении, что B значительно превышает q. При этом не учитывается, что потоки могут использовать кванты не полностью, что часть времени они могут тратить на ввод-вывод, что количество потоков в системе может динамически меняться и т. д.

Чем больше квант, тем выше вероятность того, что потоки завершатся в результате первого же цикла выполнения, и тем менее явной становится зависимость времени ожидания потоков от времени их выполнения. При достаточно большом кванте алгоритм квантования вырождается в алгоритм последовательной обработки, присущий однопрограммным системам, при котором время ожидания задачи

в очереди вообще никак не зависит от требуемой для этой задачи длительности вычислений.

Кванты, выделяемые одному потоку, могут быть фиксированной величины, а могут и изменяться в разные периоды жизни потока. Пусть, например, первоначально каждому потоку назначается достаточно большой квант, а величина каждого следующего кванта уменьшается до некоторой заранее заданной величины. В таком случае преимущество получат короткие задачи, которые успевают выполняться в течение первого кванта, а длительные вычисления будут проводиться в фоновом режиме. Можно представить себе алгоритм планирования, в котором каждый следующий квант, выделяемый определенному потоку, больше предыдущего. Такой подход позволяет снизить накладные расходы на переключение задач в том случае, когда сразу несколько задач выполняют длительные вычисления.

Потоки получают для выполнения квант времени, но некоторые из них используют его не полностью, например, из-за необходимости выполнить ввод или вывод данных. В результате возникает ситуация, когда потоки с интенсивными обращениями к вводу-выводу задействуют только небольшую часть выделенного им процессорного времени. Алгоритм планирования может исправить эту «несправедливость». В качестве компенсации за неиспользованные полностью кванты потоки получают привилегии при последующем обслуживании. Для этого планировщик создает две очереди готовых потоков (рис. 4.8). Очередь 1 образована потоками, которые пришли в состояние готовности в результате исчерпания кванта времени, а очередь 2 — потоками, у которых завершилась операция ввода-вывода. При выборе потока для выполнения прежде всего просматривается очередь 2, и только если она пуста, квант выделяется потоку из очереди 1.

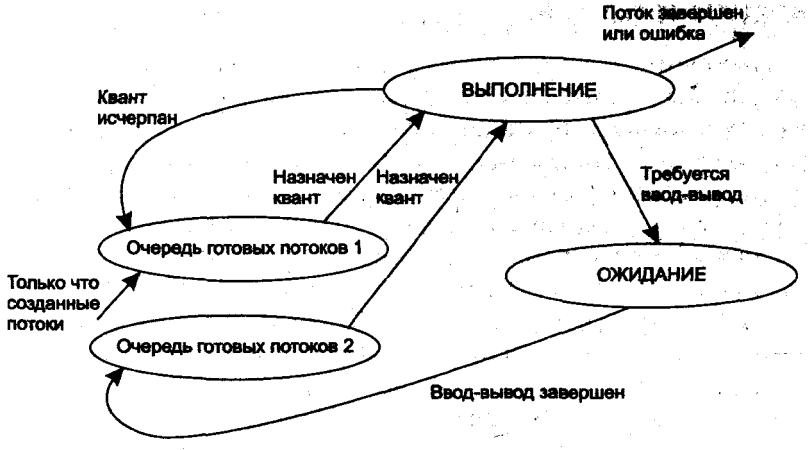


Рис. 4.8. Квантование с предпочтением потоков, интенсивно обращающихся к вводу-выводу

Многозадачные ОС теряют некоторое количество процессорного времени для выполнения вспомогательных работ во время переключения контекстов

задач. При этом запоминаются и восстанавливаются регистры, флаги и указатели стека, а также проверяется статус задач для передачи управления. Затраты на эти вспомогательные действия не зависят от величины кванта времени, поэтому чем больше квант, тем меньше суммарные накладные расходы, связанные с переключением потоков.

ПРИМЕЧАНИЕ В алгоритмах, основанных на квантовании, какую бы цель они ни преследовали (предпочтение коротких или длинных задач, компенсация недоиспользованного кванта или минимизация накладных расходов, связанных с переключениями), не используется никакой предварительной информации о задачах. При поступлении задачи на обработку ОС не имеет никаких сведений о том, является она короткой или длинной, насколько интенсивными будут ее запросы к устройствам ввода-вывода, насколько важно ее быстрое выполнение и т. д. Дифференциация обслуживания при квантовании базируется на «истории существования» потока в системе.

Алгоритмы планирования, основанные на приоритетах

Еще одной важной концепцией, лежащей в основе многих вытесняющих алгоритмов планирования, является *приоритетное обслуживание*. Приоритетное обслуживание предполагает наличие у потоков некоторой изначально известной характеристики — приоритета, на основании которого определяется порядок их выполнения.

Приоритет — это число, характеризующее степень привилегированности потока при использовании ресурсов вычислительной машины, в частности процессорного времени: чем выше приоритет, тем выше привилегии, тем меньше времени будет проводить поток в очередях.

Приоритет может выражаться целым или дробным, положительным или отрицательным значением. В некоторых ОС принято, что приоритет потока тем выше, чем больше (в арифметическом смысле) число, обозначающее приоритет. В других системах наоборот, чем меньше число, тем выше приоритет.

В большинстве операционных систем, поддерживающих потоки, приоритет потока непосредственно связан с приоритетом процесса, в рамках которого выполняется данный поток. Приоритет процесса назначается операционной системой при его создании. Значение приоритета включается в описатель процесса и используется при назначении приоритета потокам этого процесса. При назначении приоритета вновь созданному процессу ОС учитывает, является этот процесс системным или прикладным, каков статус пользователя, запустившего процесс, было ли явное указание пользователя на присвоение процессу определенного приоритета. Поток может быть инициирован не только по команде пользователя, но и в результате выполнения системного вызова другим потоком. В этом случае при назначении приоритета новому потоку ОС должна принимать во внимание параметры системного вызова.

Во многих ОС предусматривается возможность изменения приоритетов в течение жизни потока. Изменение приоритета может происходить по инициативе самого потока, когда он обращается с соответствующим вызовом к операционной системе, или по инициативе пользователя, когда он выполняет соответствующую команду. Кроме того, ОС сама может изменять приоритеты потоков в зависимости от ситуации, складывающейся в системе. В последнем случае приоритеты называются *динамическими*, в отличие от неизменяемых *фиксированных*.

От того, какие приоритеты назначены потокам, существенно зависит эффективность работы всей вычислительной системы. В современных ОС во избежание разбалансировки системы, которая может возникнуть при неправильном назначении приоритетов, возможности пользователей влиять на приоритеты процессов и потоков стараются ограничивать. При этом обычные пользователи, как правило, не имеют права повышать приоритеты своим потокам, это разрешено делать (да и то в определенных пределах) только администраторам. В большинстве же случаев ОС присваивает приоритеты потокам по умолчанию.

В качестве примера рассмотрим схему назначения приоритетов потокам, принятую в ОС семейства Windows NT[1] (рис. 4.9). В этих ОС определено 32 уровня приоритетов и два класса потоков — потоки реального времени и потоки с переменными приоритетами. Диапазон от 1 до 15 включительно отведен для потоков с переменными приоритетами, а от 16 до 31 — для более критичных ко времени потоков реального времени (приоритет 0 зарезервирован для системных целей).

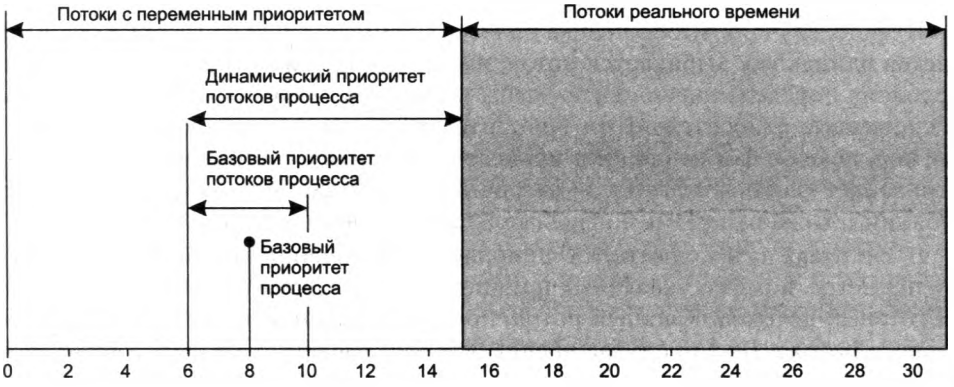


Рис. 4.9. Схема назначения приоритетов в ОС семейства Windows NT

При создании процесса он, в зависимости от класса, получает по умолчанию базовый приоритет в верхней или нижней части диапазона. Базовый приоритет процесса в дальнейшем может быть повышен или понижен операционной

[1] Как уже ранее было сказано, под операционными системами семейства Windows NT здесь подразумевается следующий ряд ОС: Windows NT 3.1, Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003, Windows Vista.

системой. Первоначально поток получает значение базового приоритета из диапазона базового приоритета процесса, в котором он был создан. Пусть, например, значение базового приоритета некоторого процесса равно K. Тогда все потоки данного процесса получат базовые приоритеты из диапазона $[K - 2, K + 2]$. Отсюда видно, что, изменяя базовый приоритет процесса, ОС может влиять на базовые приоритеты его потоков.

С течением времени приоритет потока, относящегося к классу потоков с переменными приоритетами, может отклоняться от базового приоритета потока, причем эти изменения могут быть не связаны с изменениями базового приоритета процесса. ОС может повышать приоритет потока (который в этом случае называется динамическим) в тех случаях, когда поток не полностью использовал отведенный ему квант, или понижать приоритет, если квант был использован полностью. ОС наращивает приоритет дифференцировано в зависимости от того, какого типа событие не дало потоку полностью использовать квант. В частности, ОС повышает приоритет в большей степени потокам, которые ожидают ввода с клавиатуры (интерактивным приложениям), и в меньшей степени потокам, выполняющим дисковые операции. Именно на основе динамических приоритетов осуществляется планирование потоков. Начальной точкой отсчета для динамического приоритета является значение базового приоритета потока. Значение динамического приоритета потока ограничено снизу его базовым приоритетом, верхней же границей является нижняя граница диапазона приоритетов реального времени. Существует две разновидности приоритетного планирования: обслуживание с относительными приоритетами и обслуживание с абсолютными приоритетами.

В обоих случаях выбор потока на выполнение из очереди готовых осуществляется одинаково: выбирается поток, имеющий наивысший приоритет. Однако проблема определения момента смены активного потока решается по-разному. В системах с относительными приоритетами активный поток выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ожидания (или произойдет ошибка, или поток завершится). На рис. 4.10, *а* показан граф состояний потока в системе с относительными приоритетами.

В системах с абсолютными приоритетами выполнение активного потока прерывается, помимо указанных причин, еще при одном условии: если в очереди готовых потоков появился поток, приоритет которого выше приоритета активного потока. В этом случае прерванный поток переходит в состояние готовности (рис. 4.10, *б*).

В системах, в которых планирование осуществляется на основе относительных приоритетов, минимизируются затраты на переключения процессора с одной работы на другую. В то же время здесь могут возникать ситуации, когда одна задача занимает процессор долгое время. Ясно, что для систем разделения времени и реального времени такая дисциплина обслуживания не подходит: интерактивное приложение может ждать своей очереди часами, пока вычислительной задаче не потребуется ввод-вывод. А вот в системах пакетной обработки (в том числе в классической ОС для мэйнфреймов OS/360) относительные приоритеты используются широко.

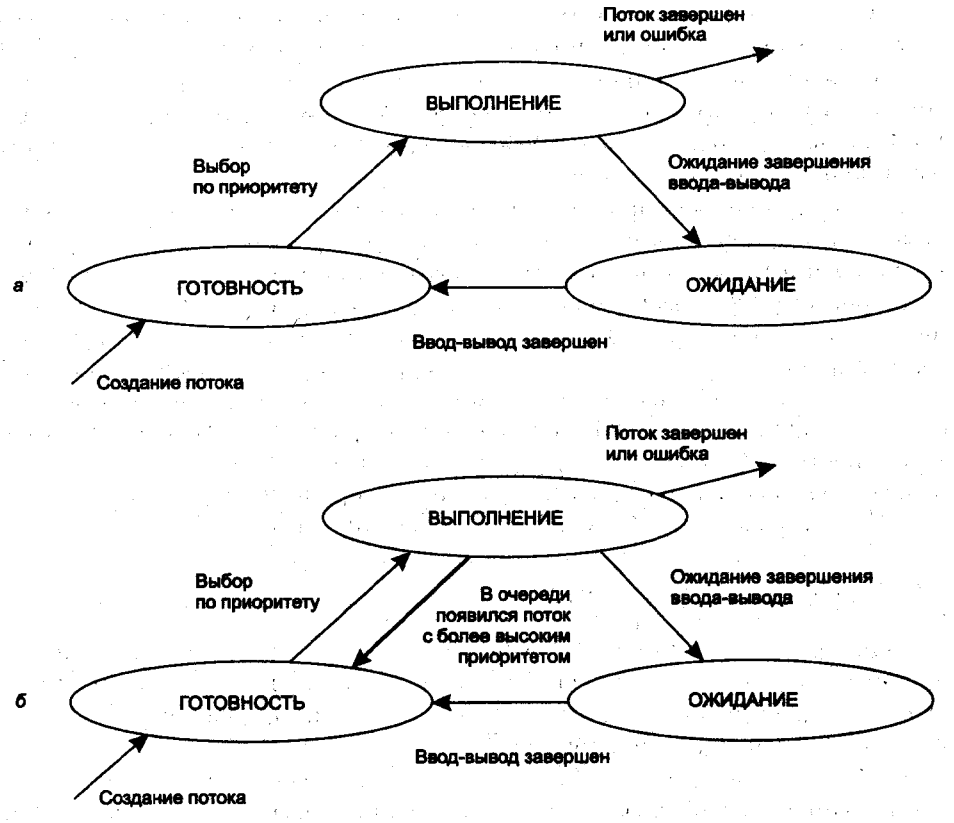


Рис. 4.10. Графы состояний потоков в системах с относительными и абсолютными приоритетами

В системах с абсолютными приоритетами время ожидания потока в очередях может быть сведено к минимуму, если ему назначить самый высокий приоритет. Такой поток будет вытеснять из процессора все остальные потоки (кроме потоков, тоже имеющих наивысший приоритет). Это делает планирование на основе абсолютных приоритетов подходящим для систем управления объектами, в которых важна быстрая реакция на событие.

Смешанные алгоритмы планирования

Во многих операционных системах алгоритмы планирования построены с использованием как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора потока из очереди готовых определяется приоритетами потоков. Именно так реализовано планирование в ОС семейства Windows NT, где квантование сочетается с динамическими абсолютными приоритетами. На выполнение выбирается готовый поток с наивысшим приоритетом. Ему выделяется квант времени. Если во время выполнения в очереди готовых появляется поток с более высоким

приоритетом, то он вытесняет выполняемый поток. Вытесненный поток возвращается в очередь готовых, причем он становится впереди всех остальных потоков, имеющих такой же приоритет.

В качестве примера рассмотрим реализацию алгоритма планирования процессов в операционной системе Unix System V Release 4. В этой ОС реализована вытесняющая многозадачность, основанная на использовании приоритетов и квантования. Понятие «поток» здесь отсутствует. Каждый процесс, в зависимости от задачи, которую он решает, относится к одному из трех определенных в системе приоритетных классов: классу реального времени, классу системных процессов или классу процессов разделения времени. Назначение и обработка приоритетов выполняются для разных классов по-разному. Процессы системного класса, зарезервированные для ядра, применяют стратегию фиксированных приоритетов. Уровень приоритета процессу назначается ядром и никогда не изменяется.

Процессы реального времени также применяют стратегию фиксированных приоритетов, но пользователь может их изменять. Так как при наличии готовых к выполнению процессов реального времени другие процессы не рассматриваются, то процессы реального времени надо тщательно проектировать, чтобы они не захватывали процессор на слишком долгое время. Характеристики планирования процессов реального времени включают две величины: уровень глобального приоритета и квант времени. Для каждого уровня приоритета по умолчанию имеется своя величина кванта времени. Процессу разрешается захватывать процессор на указанный квант времени, а по его истечению планировщик снимает процесс с выполнения.

По умолчанию Unix System V Release 4 относит новые процессы к классу процессов разделения времени. Состав класса процессов разделения времени наиболее неопределенный и часто меняющийся, в отличие от системных процессов и процессов реального времени. Для справедливого распределения времени процессора между процессами в этом классе используется стратегия динамических приоритетов. Величина приоритета, назначаемого процессам разделения времени, вычисляется пропорционально значениям двух частей приоритета: пользовательской и системной. Пользовательская часть приоритета может быть изменена администратором и владельцем процесса, но в последнем случае только в сторону его снижения.

Системная составляющая позволяет планировщику управлять процессами в зависимости от того, как долго они занимают процессор, не уходя в состояние ожидания. У тех процессов, которые отнимают большие периоды процессорного времени без ухода в состояние ожидания, приоритет снижается, а у тех процессов, которые часто уходят в состояние ожидания после короткого периода использования процессора, приоритет повышается. Таким образом, процессам, ведущим себя не «по-джентльменски», дается низкий приоритет. Это означает, что они реже выбираются для выполнения. Подобное ущемление в правах компенсируется тем, что процессам с низким приоритетом даются бóльшие кванты времени, чем процессам с высокими приоритетами. Таким образом, хотя низко-

приоритетный процесс и не работает так часто, как высокоприоритетный, зато, когда он, наконец, выбирается для выполнения, ему отводится больше времени.

Другой пример относится к операционной системе OS/2. Планирование здесь основано на квантовании и абсолютных динамических приоритетах. На множестве потоков определены приоритетные классы — *критический* (time critical), *серверный* (server), *стандартный* (regular) и *остаточный* (idle), в каждом из которых имеется 32 приоритетных уровня. Потоки критического класса имеют наивысший приоритет. В этот класс могут быть отнесены, например, системные потоки, выполняющие задачи управления сетью. Следующий по приоритетности класс предназначен, как это следует из его названия, для потоков, обслуживающих серверные приложения. К стандартному классу могут быть отнесены потоки обычных приложений. Потоки, входящие в остаточный класс, имеют самый низкий приоритет. К этому классу относится, например, поток, выводящий на экран заставку, когда в системе не выполняется никакой работы.

Поток из менее приоритетного класса не может быть выбран для выполнения, пока в очереди более приоритетного класса имеется хотя бы один поток. Внутри каждого класса потоки выбираются также по приоритетам. Потоки, имеющие одинаковое значение приоритета, обслуживаются в циклическом порядке.

Приоритеты могут изменяться планировщиком в следующих случаях.

- Если поток находится в ожидании процессорного времени дольше, чем это задано системной переменной `MAXWAIT`, его приоритет будет автоматически увеличен операционной системой. При этом результирующее значение приоритета не должно превышать нижней границы диапазона приоритетов критического класса.
- Если поток ушел на выполнение операции ввода-вывода, то после ее завершения он получит наивысший для своего класса приоритет.
- Приоритет потока автоматически повышается, когда он поступает на выполнение.

ОС динамически устанавливает величину кванта, отводимому потоку для выполнения. Величина кванта зависит от загрузки системы и интенсивности подкачки. Параметры настройки системы позволяют явно задать границы изменения кванта. В любом случае он не может быть меньше 32 мс и больше 65 536 мс. Если поток был прерван до истечения кванта, то следующий выделенный ему интервал выполнения будет увеличен на время, равное одному периоду таймера (около 32 мс), и так до тех пор, пока квант не достигнет заранее заданного при настройке ОС предела.

Благодаря такому алгоритму планирования ни один поток не будет «забыт» системой и получит достаточно процессорного времени.

Планирование в системах реального времени

В системах реального времени, в которых главным критерием эффективности является обеспечение временных характеристик вычислительного процесса, планирование имеет особое значение. Любая система реального времени должна

реагировать на сигналы управляемого объекта в течение заданных временных ограничений. Необходимость тщательного планирования работ облегчается тем, что в системах реального времени весь набор выполняемых задач известен заранее. Кроме того, часто в системе имеется информация о временах выполнения задач, моментах активизации, предельных допустимых сроках ожидания ответа и т. д. Эти данные могут быть использованы планировщиком для создания статического расписания или для построения адекватного алгоритма динамического планирования.

При разработке алгоритмов планирования для систем реального времени необходимо учитывать, какие последствия в этих системах возникают при несоблюдении временных ограничений. Если эти последствия катастрофичны, как, например, для системы управления полетами или атомной электростанцией, то операционная система реального времени, на основе которой строится управление объектом, называется *жесткой* (hard). Если же последствия нарушения временных ограничений не столь серьезны, то есть сравнимы с той пользой, которую приносит система управления объектом, то система является *мягкой* (soft). Примером мягкой системы реального времени является система резервирования билетов. Если из-за временных нарушений оператору не удается зарезервировать билет, это не очень страшно — можно просто послать запрос на резервирование заново.

В жестких системах реального времени время завершения выполнения каждой из критических задач должно быть гарантировано для всех возможных сценариев работы системы. Такие гарантии могут быть даны либо в результате исчерпывающего тестирования всех возможных сценариев поведения управляемого объекта и управляющих программ, либо в результате построения статического расписания, либо в результате выбора математически обоснованного динамического алгоритма планирования. При построении расписания надо иметь в виду, что для некоторых наборов задач в принципе невозможно найти расписания, в котором бы удовлетворялись заданные временные характеристики. С целью определения возможности существования расписания могут быть использованы различные критерии. Например, простейшим критерием может быть условие, что разность между предельным сроком выполнения задачи (после появления запроса на ее выполнение) и временем ее вычисления (при условии непрерывного выполнения) всегда должна быть положительной. Очевидно, что такой критерий является необходимым, но недостаточным. Точные критерии, гарантирующие наличие расписания, являются очень сложными в вычислительном отношении.

В мягких системах реального времени предполагается, что заданные временные ограничения могут иногда нарушаться, поэтому здесь обычно применяются менее затратные способы планирования.

В зависимости от характера возникновения запросов на выполнение задач полезно разделять их на два типа: *периодические* и *спорадические*. Начиная с момента первоначального запроса, все будущие моменты запроса периодической задачи можно определить заранее путем прибавления к моменту начального за-

проса величины, кратной известному периоду. Время поступления запросов на выполнение спорадических задач заранее не известно.

ПРИМЕЧАНИЕ Ранее было отмечено, что в операционных системах семейства Windows NT, OS/2 и Unix System V Release 4 имеется приоритетный класс реального времени. Однако для потоков/процессов, относящихся к этому классу, каждая из вышеназванных систем не гарантирует соблюдение заданных временных ограничений, а лишь обеспечивает предпочтение в скорости обслуживания. Следовательно, эти ОС пригодны для построения мягких, а не жестких систем реального времени.

Предположим, что имеется периодический набор задач $\{T_i\}$ с периодами p_i, предельными сроками d_i и требованиями ко времени выполнения c_i. Для проверки возможности существования расписания достаточно проанализировать расписание на периоде времени, равном, по крайней мере, наименьшему общему множителю периодов этих задач. Необходимым критерием существования расписания для набора периодических задач является следующее достаточно очевидное утверждение: сумма коэффициентов использования $\mu_i = c_i/p_i$ должна быть меньше или равна k, где k — количество доступных процессоров, то есть:

$$\mu = \Sigma\, c_i/p_i \le k.$$

При выборе алгоритма планирования следует учитывать данные о возможной зависимости задач. Эта зависимость может выступать, например, в виде ограничений на последовательность выполнения задач или их синхронизации, вызванной взаимными исключениями (запрете выполнения некоторых задач в течение определенных периодов времени).

С практической точки зрения алгоритмы планирования зависимых задач более важны, чем независимых. При наличии дешевых микроконтроллеров нет смысла организовывать мультипрограммное выполнение большого количества независимых задач на одном компьютере, так как при этом значительно возрастает сложность программного обеспечения. Обычно одновременно выполняющиеся задачи должны обмениваться информацией и получать доступ к общим данным для достижения общей цели системы, то есть такие задачи являются зависимыми. Поэтому существование некоторого предпочтения последовательности выполнения задач или взаимного исключения — это скорее норма для систем управления реального времени, чем исключение.

Проблема планирования зависимых задач очень сложна, нахождение ее оптимального решения требует больших вычислительных ресурсов, сравнимых с теми, которые требуются для собственно выполнения задач управления. Решение этой проблемы возможно за счет, во-первых, разделения проблемы планирования на две части, чтобы одна часть выполнялась заранее, перед запуском системы, а вторая, более простая часть, — во время работы системы. Предварительный анализ набора задач с взаимными исключениями может состоять, например, в выявлении так называемых запрещенных областей времени, в течение которых нельзя назначать выполнение задач, содержащих критические

секции. Во-вторых, можно сделать ограничивающие предположения о поведении набора задач.

При таком подходе планирование приближается к статическому.

Возвращаясь к планированию независимых задач, рассмотрим классический алгоритм для жестких систем реального времени с одним процессором, разработанный в 1973 году Лью (Liu) и Лейландом (Layland). Алгоритм является динамическим, то есть он основан на вытесняющей многозадачности и относительных статических (неизменяемых в течение жизни задачи) приоритетах.

Алгоритм базируется на следующих предположениях.

- Запросы на выполнение всех задач набора, имеющих жесткие ограничения на время реакции, являются периодическими.
- Все задачи независимы. Между любой парой задач не существует никаких ограничений на предшествование или на взаимное исключение.
- Срок выполнения каждой задачи равен ее периоду p_i.
- Максимальное время выполнения каждой задачи c_i известно и постоянно.
- Время переключения контекста можно игнорировать.
- Максимальный суммарный коэффициент загрузки процессора $\Sigma\ c_i/p_i$ при существовании n задач не превосходит $n\ (21/n - 1)$. Эта величина при стремлении n к бесконечности приблизительно равна ln2, то есть 0,7.

Суть алгоритма состоит в том, что всем задачам назначаются статические приоритеты в соответствии с величиной их периодов выполнения. Задача с самым коротким периодом получает высший приоритет, а задача с наибольшим периодом выполнения — низший. При соблюдении всех ограничений этот алгоритм гарантирует выполнение временных ограничений для всех задач во всех ситуациях.

Если же периоды повторения задач кратны периоду выполнения самой короткой задачи, то требование к максимальному коэффициенту загрузки процессора смягчается — он может доходить до 1.

Существуют также алгоритмы с динамическим изменением приоритетов, которые назначаются в соответствии с такими текущими параметрами задачи, как, например, **конечный срок выполнения** (deadline). При необходимости назначения некоторой задачи на выполнение выбирается та, у которой текущее значение разницы между конечным сроком выполнения и временем, требуемым для ее непрерывного выполнения, является наименьшим.

Моменты перепланирования

Для реализации алгоритма планирования ОС должна получать управление всякий раз, когда в системе происходит событие, требующее перераспределения процессорного времени.

К событиям, приводящим к необходимости перепланирования процессов, могут быть отнесены следующие.

- Прерывание от таймера, сигнализирующее, что *время, отведенное активной задаче на выполнение, закончилось*. Планировщик переводит задачу в состояние готовности и выполняет перепланирование.
- Активная задача выполнила *системный вызов, связанный с запросом на ввод-вывод или на доступ к ресурсу*, который в настоящий момент занят (например, файл данных). Планировщик переводит задачу в состояние ожидания и выполняет перепланирование.
- Активная задача выполнила *системный вызов, связанный с освобождением ресурса*. Планировщик проверяет, не ожидает ли этот ресурс какая-либо задача. Если — да, то эта задача переводится из состояния ожидания в состояние готовности. При этом, возможно, что задача, которая получает ресурс, имеет более высокий приоритет, чем текущая активная задача. После перепланирования более приоритетная задача получает доступ к процессору, вытесняя текущую задачу.
- Внешнее (аппаратное) прерывание[1], которое сигнализирует о *завершении периферийным устройством операции ввода-вывода*, переводит соответствующую задачу в очередь готовых, и выполняется перепланирование.
- Внутреннее прерывание сигнализирует об *ошибке*, которая произошла в результате выполнения активной задачи. Планировщик снимает задачу и выполняет перепланирование.

При возникновении каждого из этих событий планировщик выполняет просмотр очередей и решает вопрос о том, какая задача будет выполняться следующей. Помимо указанных, существует и ряд других событий (часто связанных с системными вызовами), требующих перепланирования. Например, запросы приложений и пользователей на создание новой задачи или повышение приоритета уже существующей задачи создают новую ситуацию, которая требует пересмотра очередей и, возможно, переключения процессора.

На рис. 4.11 показан фрагмент временной диаграммы работы планировщика в системе, где одновременно выполняется четыре потока. В данном случае неважно, по какому правилу выбираются потоки на выполнение и каким образом изменяются их приоритеты. Существенное значение имеют лишь события, вызывающие активизацию планировщика.

Первые четыре цикла работы планировщика, приведенные на рисунке, инициированы прерываниями от таймера по истечении квантов времени (эти события обозначены на рисунке символом T).

Следующая передача управления планировщику осуществлена в результате выполнения потоком 3 системного запроса на ввод-вывод (событие I/O). Планировщик перевел этот поток в состояние ожидания, а затем переключил процессор на поток 2. Поток 2 полностью использовал свой квант, произошло прерывание от таймера, и планировщик активизировал поток 1.

---

[1] Подробнее о прерываниях читайте в следующем разделе

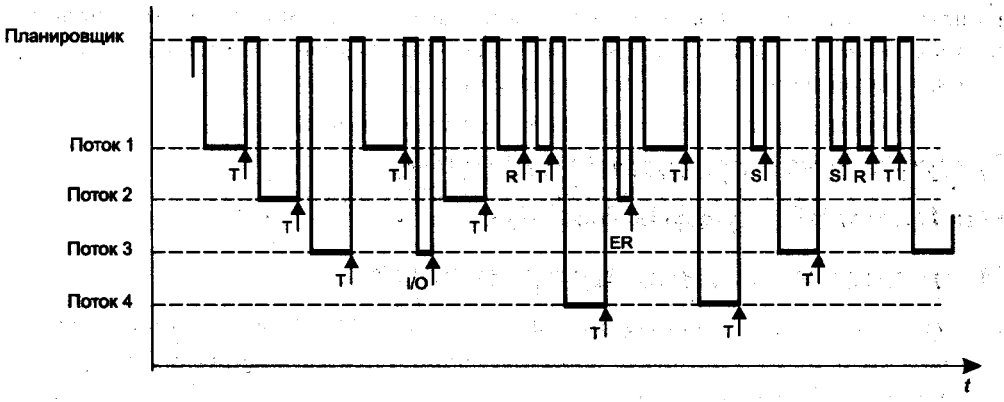


Рис. 4.11. Моменты перепланирования потоков

При выполнении потока 1 произошло событие R — системный вызов, в результате которого освободился некоторый ресурс (например, был закрыт файл). Это событие вызвало перепланирование потоков. Планировщик просмотрел очередь ожидающих потоков и обнаружил, что поток 4 ждет освобождения данного ресурса. Этот поток был переведен в состояние готовности, но поскольку приоритет выполняющегося в данный момент потока 1 выше приоритета потока 4, планировщик вернул процессор потоку 1.

В следующем цикле работы планировщик активизировал поток 4, а затем, после истечения кванта и сигнала от таймера, управление получил поток 2. Этот поток не успел использовать свой квант, так как был снят с выполнения в результате возникшей ошибки (событие ER).

Далее планировщик предоставлял процессорное время потокам 1, 4 и снова 1. Во время выполнения потока 1 произошло прерывание S от внешнего устройства, сигнализирующее о том, что операция передачи данных завершена. Это событие активизировало работу планировщика, в результате которой поток 3, ожидавший завершения ввода-вывода, вытеснил поток 1, так как имел в этот момент более высокий приоритет.

Последний показанный на диаграмме период выполнения потока 1 прерывался несколько раз. Вначале это было прерывание от внешнего устройства (S), затем программное прерывание (R), вызвавшее освобождение ресурса, и, наконец, прерывание от таймера (T). Каждое из этих трех прерываний вызвало перепланирование потоков. В двух первых случаях планировщик оставил выполняться поток 1, так как в очереди не оказалось более приоритетных потоков, а квант времени, выделенный потоку 1, еще не был исчерпан. Переключение потоков было выполнено только по прерыванию от таймера.

В системах реального времени для отработки статического расписания планировщик активизируется по прерываниям от таймера. Эти прерывания пронизывают всю временную ось, возникая через короткие постоянные интервалы времени. После каждого прерывания планировщик просматривает расписание и проверяет, не пора ли переключить задачи. Помимо прерываний от таймера,

в системах реального времени перепланирование задач может происходить по прерываниям от внешних устройств — различного вида датчиков и исполнительных механизмов.

Мультипрограммирование на основе прерываний

Назначение и типы прерываний

Прерывания являются основной движущей силой любой операционной системы. Отключите систему прерываний, и «жизнь» в операционной системе немедленно остановится.

Как верно было замечено[1]: «прерывания названы так весьма удачно, поскольку они прерывают нормальную работу системы». Система прерываний переводит процессор на выполнение потока команд, отличного от того, который выполнялся до сих пор, с последующим возвратом к исходному коду.

ПРИМЕЧАНИЕ Из сказанного можно сделать вывод о том, что механизм прерываний очень похож на механизм выполнения процедур. Это на самом деле так, хотя между этими механизмами имеется важное отличие. Переключение по прерыванию отличается от переключения по команде безусловного или условного перехода, предусмотренной программистом в потоке команд приложения. Переход по команде происходит в заранее определенных программистом точках программы в зависимости от исходных данных, обрабатываемых программой. Прерывание же происходит в произвольной точке потока команд программы, которую программист не может прогнозировать. Прерывание возникает либо в зависимости от внешних по отношению к процессу выполнения программы событий, либо при появлении непредвиденных аварийных ситуаций в процессе выполнения данной программы. Сходство же прерываний с процедурами состоит в том, что в обоих случаях выполняется некоторая подпрограмма, обрабатывающая специальную ситуацию, а затем продолжается выполнение основной ветви программы.

Механизм прерываний поддерживается аппаратными средствами компьютера и программными средствами операционной системы.

В зависимости от источника прерывания делятся на три больших класса:

- внешние;
- внутренние;
- программные.

Внешние прерывания могут возникать в результате:

- действий пользователя или оператора за терминалом;

[1] Скотт Максвелл. Ядро Linux в комментариях. — К.: «ДиаСофт», 2000.

- поступления сигналов от аппаратных устройств (сигналов завершения операций ввода-вывода, вырабатываемых контроллерами внешних устройств компьютера, такими как принтер или накопитель на жестких дисках);
- поступления сигналов от датчиков управляемых компьютером технических объектов.

Внешние прерывания называют также *аппаратными*, отражая тот факт, что прерывание возникает вследствие подачи некоторой аппаратурой (например, контроллером принтера) электрического сигнала, который передается (возможно, проходя через другие блоки компьютера, например контроллер прерываний) на специальный вход прерывания процессора. Данный класс прерываний является *асинхронным* по отношению к потоку инструкций прерываемой программы. Аппаратура процессора работает так, что асинхронные прерывания возникают между выполнением двух соседних инструкций, при этом система после обработки прерывания продолжает выполнение процесса, начиная уже со следующей инструкции.

Внутренние прерывания, называемые также *исключениями* (exeption), происходят *синхронно* выполнению программы при появлении аварийной ситуации в ходе обработки некоторой инструкции программы. Примерами исключений являются деление на нуль, ошибки защиты памяти, обращения по несуществующему адресу, попытки выполнить привилегированную инструкцию в пользовательском режиме и т. п. Исключения возникают непосредственно в ходе выполнения тактов команды («внутри» выполнения).

Программные прерывания отличаются от предыдущих двух классов тем, что они по своей сути не являются «истинными» прерываниями. Программное прерывание возникает при выполнении особой команды процессора, что имитирует прерывание, то есть переход на новую последовательность инструкций. Причины использования программных прерываний вместо обычных инструкций вызова процедур будут изложены далее, после рассмотрения механизма прерываний.

Прерываниям приписывается приоритет, с помощью которого они ранжируются по степени важности и срочности. О прерываниях, имеющих одинаковое значение приоритета, говорят, что они относятся к одному *уровню приоритета* прерываний.

Прерывания обычно обрабатываются модулями операционной системы, так как действия, выполняемые по прерыванию, относятся к управлению разделяемыми ресурсами вычислительной системы: принтером, диском, таймером, процессором и т. п. Процедуры, вызываемые по прерываниям, обычно называют **обработчиками прерываний**, или **процедурами обслуживания прерываний** (Interrupt Service Routine, **ISR**). Аппаратные прерывания обрабатываются драйверами соответствующих внешних устройств, исключения — специальными модулями ядра, программные прерывания — процедурами ОС, обслуживающими системные вызовы. Помимо этих модулей в операционной системе может находиться так называемый диспетчер прерываний, который координирует работу отдельных обработчиков прерываний.

Аппаратная поддержка прерываний

Аппаратная поддержка прерываний осуществляется как на уровне процессора, так и на уровне функциональных блоков компьютера. Функциональный блок компьютера, назначением которого является поддержка прерываний, называется **контроллером прерываний**. Контроллер прерываний является источником сигналов прерываний по отношению к процессору и обработчиком прерываний по отношению к внешним устройствам компьютера.

Аппаратная поддержка прерываний имеет свои особенности, зависящие от типа процессора и других аппаратных компонентов, являющихся источниками сигналов прерывания, таких, например, как контроллеры внешних устройств, шины подключения внешних устройств, контроллеры прерываний. Особенности аппаратной реализации прерываний оказывают влияние на средства программной поддержки прерываний, работающие в составе ОС.

Существует два основных способа, с помощью которых шины выполняют прерывания: *векторный* (vectored) и *опрашиваемый* (polled). В обоих способах процессору предоставляется информация об уровне приоритета прерывания на шине подключения внешних устройств. В случае векторных прерываний в процессор передается также информация о начальном адресе соответствующего обработчика прерываний.

Устройствам, которые используют векторные прерывания, назначается **вектор прерываний**. Он представляет собой электрический сигнал, выставляемый на соответствующие шины процессора и несущий в себе информацию об определенном, закрепленном за данным устройством номере, который идентифицирует соответствующий обработчик прерываний. Этот вектор может быть фиксированным, конфигурируемым (например, с использованием переключателей) или программируемым. Операционная система может предусматривать процедуру регистрации вектора обработки прерываний для определенного устройства, которая связывает некоторую процедуру обслуживания прерываний с определенным вектором. При получении сигнала запроса прерывания процессор выполняет специальный цикл подтверждения прерывания, в котором устройство должно идентифицировать себя. В течение этого цикла устройство отвечает, выставляя на шину вектор прерываний. Затем процессор использует этот вектор для нахождения обработчика данного прерывания. Примером шины подключения внешних устройств, которая поддерживает векторные прерывания, является шина VMEbus.

При использовании опрашиваемых прерываний процессор получает от запросившего прерывание устройства только информацию об уровне приоритета прерывания (например, номере IRQ на шине ISA или номере IPL на шине SBus компьютеров SPARC). С каждым уровнем прерываний может быть связано несколько устройств и, соответственно, несколько программ — обработчиков прерываний. При возникновении прерывания процессор должен определить, какое устройство из тех, что связаны с данным уровнем прерываний, действительно запросило прерывание. Это достигается вызовом всех обработчиков прерываний для данного уровня приоритета, пока один из обработчиков не подтвердит,

что прерывание пришло от обслуживаемого им устройства. Если же с каждым уровнем прерываний связано только одно устройство, то определение нужной программы обработки прерывания происходит немедленно, как и при векторном прерывании. Опрашиваемые прерывания поддерживают шины ISA, EISA, MCA, PCI и Sbus.

Механизм прерываний некоторой аппаратной платформы может сочетать векторный и опрашиваемый типы прерываний. Типичным примером является архитектура персональных компьютеров на основе процессоров Intel Pentium. Шины PCI, ISA, EISA или MCA, используемые в этой платформе в качестве шин подключения внешних устройств, поддерживают механизм опрашиваемых прерываний. Контроллеры периферийных устройств выставляют на шину не вектор, а сигнал запроса прерывания определенного уровня IRQ. Однако в процессоре Pentium система прерываний является векторной. Вектор прерываний в процессор Pentium поставляет контроллер прерываний, который отображает поступающий от шины сигнал IRQ на определенный номер вектора. Вектор прерываний, передаваемый в процессор, представляет собой целое число в диапазоне от 0 до 255, указывающее на одну из 256 программ обработки прерываний, адреса которых хранятся в таблице обработчиков прерываний. В том случае, когда к каждой линии IRQ подключается только одно устройство, процедура обработки прерываний работает так, как если бы система прерываний была чисто векторной, — то есть процедура не выполняет никаких дополнительных опросов для выяснения того, какое именно устройство запросило прерывание. Однако при совместном использовании одного уровня IRQ несколькими устройствами программа обработки прерываний должна действовать в соответствии со схемой опрашиваемых прерываний, то есть дополнительно выполнить опрос всех устройств, подключенных к данному уровню IRQ.

Механизм прерываний чаще всего поддерживает **приоритезацию** и **маскирование** прерываний.

Приоритезация означает, что все источники прерываний делятся на классы и каждому классу назначается свой уровень приоритета запроса на прерывание. Приоритеты могут обслуживаться как *относительные* и *абсолютные*. Обслуживание запросов прерываний по схеме с относительными приоритетами заключается в том, что при одновременном поступлении запросов прерываний из разных классов выбирается запрос, имеющий высший приоритет. Однако в дальнейшем при обслуживании этого запроса процедура обработки прерывания уже не откладывается даже в том случае, когда появляются более приоритетные запросы — решение о выборе нового запроса принимается только в момент завершения обслуживания очередного прерывания. Если же более приоритетным прерываниям разрешается приостанавливать работу процедур обслуживания менее приоритетных прерываний, это означает, что работает схема приоритезации с абсолютными приоритетами.

Если процессор (или компьютер, когда поддержка приоритезации прерываний вынесена во внешний по отношению к процессору блок) работает по схеме с абсолютными приоритетами, то он поддерживает в одном из своих внутрен-

них регистров переменную, фиксирующую уровень приоритета обслуживаемого в данный момент прерывания. При поступлении запроса из определенного класса его приоритет сравнивается с текущим приоритетом процессора, и если приоритет запроса выше, то текущая процедура обслуживания прерываний вытесняется, а при завершении обработки нового прерывания происходит возврат к прерванной процедуре.

Упорядоченное обслуживание запросов прерываний наряду со схемами приоритетной обработки запросов может выполняться **механизмом маскирования запросов**. Собственно говоря, в описанной схеме абсолютных приоритетов тоже выполняется маскирование — при обслуживании некоторого запроса все запросы с равным или более низким приоритетами маскируются, то есть не обслуживаются. Схема маскирования предполагает возможность временного маскирования прерываний любого класса независимо от уровня приоритета.

Обобщенно последовательность действий аппаратных и программных средств по обработке прерывания можно описать следующим образом.

1. При возникновении сигнала (для аппаратных прерываний) или условия (для внутренних прерываний) прерывания происходит первичное аппаратное распознавание типа прерывания. Если прерывания данного типа в настоящий момент запрещены (схемой приоритезации или механизмом маскирования), то процессор продолжает поддерживать естественный ход выполнения команд. В противном случае в зависимости от поступившей в процессор информации (уровень прерывания, вектор прерывания или тип условия внутреннего прерывания) происходит автоматический вызов процедуры обслуживания прерывания, адрес которой находится в специальной таблице операционной системы, размещаемой либо в регистрах процессора, либо в определенном месте оперативной памяти.
2. Автоматически сохраняется некоторая часть контекста прерванного потока, которая позволит ядру возобновить исполнение потока процесса после обработки прерывания. В это подмножество обычно включаются значения счетчика команд, слова состояния машины, хранящего признаки основных режимов работы процессора (пример такого слова — регистр `EFLAGS` в Intel Pentium), а также нескольких регистров общего назначения, которые требуются программе обслуживания прерывания. Может быть сохранен и полный контекст процесса, если ОС обрабатывает данное прерывание со сменой процесса. Однако в общем случае это не обязательно, часто обработка прерываний выполняется без вытеснения текущего процесса[1].
3. Одновременно с загрузкой адреса процедуры обслуживания прерываний в счетчик команд может автоматически выполняться загрузка нового значе-

[1] Решение о перепланировании процессов может быть принято в ходе обработки прерывания, например, если это прерывание от таймера и после наращивания значения системных часов выясняется, что процесс исчерпал выделенный ему квант времени. Однако это совсем не обязательно — прерывание может выполняться и без смены процесса, например, прием очередной порции данных от контроллера внешнего устройства чаще всего происходит в рамках текущего процесса, хотя данные, скорее всего, предназначены другому процессу.

ния слова состояния машины (или другой системной структуры, например селектора кодового сегмента в процессоре Pentium), которое определяет режимы работы процессора при обработке прерывания, в том числе работу в привилегированном режиме. В некоторых моделях процессоров переход в привилегированный режим за счет смены состояния машины при обработке прерывания является единственным способом смены режима. Прерывания практически во всех мультипрограммных ОС обрабатываются в привилегированном режиме модулями ядра, так как при этом обычно нужно выполнить ряд критических операций, от которых зависит жизнеспособность системы — управлять внешними устройствами, перепланировать потоки и т. п.

4. Временно запрещаются прерывания данного типа, чтобы не образовалась очередь вложенных друг в друга потоков одной и той же процедуры. Детали выполнения этой операции зависят от особенностей аппаратной платформы, например, может использоваться механизм маскирования прерываний. Многие процессоры автоматически устанавливают признак запрета прерываний в начале цикла обработки прерывания, в противном случае это делает процедура обслуживания прерываний.
5. После того как прерывание обработано ядром операционной системы, прерванный контекст восстанавливается, и работа потока возобновляется с прерванного места. Часть контекста (например, адрес следующей команды и слово состояния машины) восстанавливается аппаратно по команде возврата из прерываний, а часть — программным способом с помощью явных команд извлечения данных из стека. При возврате из прерывания блокировка повторных прерываний данного типа снимается.

Программные прерывания

Программное прерывание реализует один из способов перехода на подпрограмму с помощью специальной инструкции процессора, такой как `INT` в процессорах Intel Pentium, `trap` в процессорах Motorola, `syscall` в процессорах MIPS или `Ticc` в процессорах SPARC. При выполнении команды программного прерывания процессор отрабатывает ту же последовательность действий, что и при возникновении внешнего или внутреннего прерывания, но только происходит это в предсказуемой точке программы — там, где программист поместил данную команду.

Практически все современные процессоры имеют в системе команд инструкции программных прерываний. Одной из причин появления инструкций программных прерываний в системе команд процессоров является то, что их использование часто приводит к более компактному коду программ по сравнению с применением стандартных команд выполнения процедур. Это объясняется тем, что разработчики процессора обычно резервируют для обработки прерываний небольшое число возможных подпрограмм, так что длина операнда в команде программного прерывания, который указывает на нужную подпрограмму, меньше, чем в команде перехода на подпрограмму. Например, в процессоре x86 предусмотрена возможность применения 256 обработчиков прерыва-

ний, поэтому в инструкции INT операнд имеет длину в один байт (а инструкция INT 3, которая предназначена для вызова отладчика, вся имеет длину в один байт). Значение операнда команды INT просто является индексом в таблице из 256 адресов подпрограмм обслуживания прерываний, один из которых и используется для перехода по команде INT. При использовании команды CALL потребовался бы уже не однобайтовый, а двух- или четырехбайтовый операнд. Другой причиной применения программных прерываний вместо обычных инструкций вызова подпрограмм является возможность смены пользовательского режима на привилегированный одновременно с вызовом процедуры — это свойство программных прерываний поддерживается большинством процессоров.

В результате программные прерывания часто используются для выполнения ограниченного количества вызовов функций ядра операционной системы, то есть системных вызовов.

Диспетчеризация и приоритезация прерываний в ОС

Прерывания выполняют очень полезную для вычислительной системы функцию — они позволяют реагировать на асинхронные по отношению к вычислительному процессу события. В то же время прерывания создают дополнительные трудности для ОС в организации вычислительного процесса. Эти трудности связаны с непредвиденными переходами управления от одной процедуры к другой, возникающими в результате прерываний от контроллеров внешних устройств. Возможно также возникновение в непредвиденные моменты времени исключений, связанных с ошибками во время выполнения инструкций. Усложняют задачу планирования вычислительных работ и запросы на выполнение системных функций (системные вызовы) от пользовательских приложений, выполняемые с помощью программных прерываний. Сами модули ОС также часто вызывают друг друга с помощью программных прерываний, еще больше запутывая картину вычислительного процесса.

Операционная система не может терять контроль над ходом выполнения системных процедур, вызываемых по прерываниям. Она должна упорядочивать их во времени так же, как планировщик упорядочивает многочисленные пользовательские потоки. Кроме того, сам планировщик потоков является системной процедурой, вызываемой по прерываниям (аппаратным — от таймера или контроллера устройства ввода-вывода, или программным — от приложения или модуля ОС). Поэтому правильное планирование процедур, вызываемых по прерываниям, является необходимым условием правильного планирования пользовательских потоков. В противном случае могут возникать, например, такие ситуации, когда операционная система длительное время занимается не требующей мгновенной реакции задачей управления стримером, архивирующим данные, в то время, когда высокоскоростной диск простаивает и тормозит работу многочисленных приложений, обменивающихся данными с этим диском. Еще один пример такой ситуации иллюстрирует рис. 4.12. В данном случае обработчик прерываний принтера блокирует на длительное время обработку

прерывания от таймера, в результате системное время на некоторое время «замирает», и поток 2, критически важный для пользователя, не получает управление в запланированное время. Остроту проблемы несколько смягчает то обстоятельство, что во многих случаях обработка прерывания связана с выполнением всего нескольких операций ввода-вывода и поэтому имеет очень небольшую продолжительность. Тем не менее ОС всегда должна контролировать ситуацию и выполнять критичную работу вовремя, а не полагаться на волю случая.

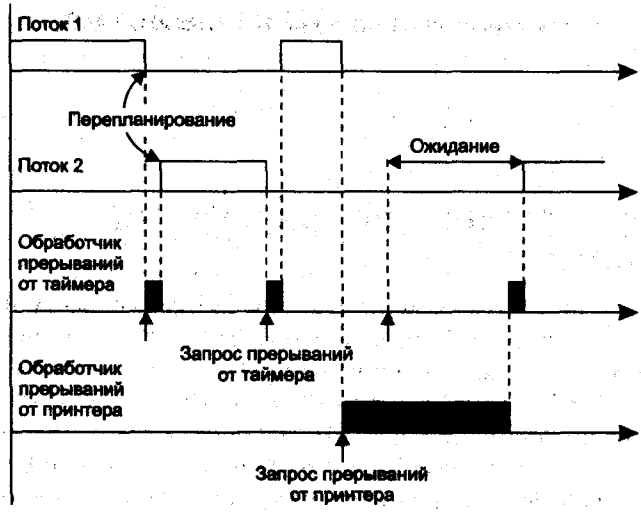


Рис. 4.12. Неупорядоченная обработка прерываний

Для упорядочения работы обработчиков прерываний в операционных системах применяется тот же механизм, что и для упорядочения работы пользовательских процессов — **механизм приоритетных очередей**.

Все источники прерываний обычно делятся на несколько классов, причем каждому классу присваивается приоритет. В операционной системе выделяется программный модуль, который занимается диспетчеризацией обработчиков прерываний. Этот модуль в разных ОС называется по-разному, но для определенности будем его называть **диспетчером прерываний**.

При возникновении прерывания диспетчер прерываний вызывается первым. Он запрещает на небольшое время все прерывания, а затем выясняет причину прерывания. После этого диспетчер сравнивает назначенный данному источнику прерывания приоритет и сравнивает его с текущим приоритетом потока команд, выполняемого процессором. В этот момент времени процессор уже может выполнять инструкции другого обработчика прерываний, также имеющего некоторый ириоритет. Если приоритет нового запроса выше текущего, то выполнение текущего обработчика приостанавливается, и он помещается в соответствующую очередь обработчиков прерываний. В противном случае в очередь помещается обработчик нового запроса.

ПРИМЕЧАНИЕ Приоритет обработчиков прерываний не совпадает в общем случае с приоритетом потоков, выполняемых в обычной последовательности, определяемой планировщиком потоков. По отношению к обработчикам прерываний любой поток, который назначен на выполнение планировщиком, имеет самый низкий приоритет, так что любой запрос на прерывание всегда может прервать выполнение этого потока.

Функции централизованного диспетчера прерываний на примере ОС семейства Windows NT

Некоторые процессоры и контроллеры прерываний компьютера на аппаратном уровне поддерживают приоритезацию запросов на прерывание. Например, в процессорах MIPS существует несколько уровней аппаратных запросов на прерывания и несколько уровней программных запросов. В процессоре имеется внутренняя переменная, называемая уровнем прерываний процессора. Прерывание происходит только в том случае, когда уровень запроса на прерывание выше текущего уровня прерываний процессора. Имеется также привилегированная инструкция, с помощью которой код ядра ОС может изменить уровень прерываний процессора. Необработанные запросы на прерывания должны храниться в контроллерах устройств, чтобы не потеряться и дождаться обслуживания при снижении уровня прерывания процессора, когда он закончит выполнение более срочных работ. При работе на такого рода аппаратной платформе ОС может пользоваться встроенными в процессор средствами приоритезации прерываний для упорядочивания процесса их обработки. Однако такие средства имеются не у всех процессоров, например процессоры семейства Pentium не имеют встроенной переменной для фиксации уровня прерываний исполняемого кода — аппаратные прерывания могут быть либо полностью запрещены, либо полностью разрешены (а наиболее критичные вовсе запретить нельзя).

Для исключения зависимости от аппаратной платформы в некоторых ОС предусмотрены собственные *программные средства поддержки приоритетов прерываний*. Например, диспетчер прерываний в ОС семейства Windows NT (так называемый обработчик ловушки — trap handler) работает с программной моделью прерываний, единой для всех аппаратных платформ, поддерживаемых этими ОС. Все источники прерываний (аппаратных и программных, а также некоторых важных для системы исключений, например, исключения по ошибке шины) делятся на несколько классов, и каждому классу присваивается уровень запроса прерывания (Interrupt Request Level, IRQL). Этот уровень и представляет приоритет данного класса. Операционная система программным способом поддерживает внутреннюю переменную, называемую IRQL выполняемого процессором кода, которая по назначению соответствует уровню прерывания процессора. Если процессор, на котором работает ОС, поддерживает такую переменную, то она используется и IRQL выполняемого кода отображается на нее, в противном случае соответствующие функции процессора эмулируются программно.

Общая схема планирования обработки прерываний выглядит в ОС семейства Windows NT следующим образом. При поступлении в процессор сигнала запроса на прерывание/исключение вызывается диспетчер прерываний, который запоминает информацию об источнике прерывания и анализирует его приоритет. Если приоритет запроса ниже или равен IRQL прерванного кода, то обслуживание этого запроса откладывается, и данные о запросе помещаются в соответствующую очередь запросов, после чего происходит быстрый возврат к прерванному обработчику прерываний. При завершении обработки высокоприоритетного прерывания управление возвращается диспетчеру прерываний, который просматривает очереди отложенных прерываний и выбирает из них наиболее приоритетное. При этом уровень IRQL снижается до уровня выбранного прерывания.

Если же запрос имеет более высокий приоритет, чем IRQL текущего кода, то текущий обработчик прерываний вытесняется и ставится в очередь, соответствующую его значению IRQL, а управление передается новому обработчику, в соответствии с IRQL запроса. После этого уровень IRQL процессора делается равным уровню IRQL принятого на выполнение запроса.

Таким образом, запрос на прерывание принимается диспетчером прерываний всегда, независимо от текущего уровня IRQL выполняемого кода, но диспетчер прерываний не передает его на обработку соответствующей процедуре обслуживания прерываний, а помещает в программную очередь запросов, если в данный момент выполняется более приоритетная процедура обслуживания прерываний. Операционная система имеет полный контроль над ситуацией, не позволяя контроллерам устройств ввода-вывода принимать решения о ходе вычислительного процесса.

Низший уровень IRQL соответствует обычным потокам, назначаемым на выполнение диспетчером потоков (рис. 4.13). Это является некоторым допущением, так как код потоков начинает выполняться процессором не в результате запроса на прерывание, но это допущение хорошо работает, поскольку позволяет любому «настоящему» запросу прерывать код обычного потока.

Рис. 4.13. Диспетчеризация прерываний в ОС семейства Windows NT

Высший уровень в иерархии IRQL отводится таким важным событиям, как исключение по ошибке шины и другим тяжелым аппаратным сбоям, далее рас-

полагается запрос на прерывание по сбою питания, и запрос на межпроцессорное прерывание.

Прерывания от внешних устройств занимают промежуточные уровни IRQL. Конкретное соотношение между приоритетами внешних устройств определяется приоритетами, задаваемыми аппаратной платформой, например уровнем IRQ шины PCI, назначенным устройству.

Особую роль в работе вычислительной системы играет системный таймер: на основании его прерываний обновляются системные часы, определяющие очередной момент вызова планировщика потоков, момент выдачи управляющего воздействия потоком реального времени и многое другое. Ввиду важности немедленной обработки прерываний от таймера ему в ОС семейства Windows NT дан весьма высокий уровень приоритета — более высокий, чем уровень любого устройства ввода-вывода.

В системе очередей диспетчера прерываний несколько очередей отведено для обслуживания отложенных программных прерываний.

Программные прерывания, обслуживающие системные вызовы от приложений, выполняются с низшим уровнем приоритета, что соответствует концепции продолжения одного и того же процесса при выполнении системного вызова, но только в системной фазе. А вот для программных прерываний, исходящих от модулей ядра ОС, отводится более высокий уровень запросов, имеющий двойное название «диспетчерский/DPC».

Этот уровень приоритета называется *диспетчерским*, потому что именно в эту очередь помещаются программные запросы, вызывающие диспетчер потоков. Часто при обработке высокоприоритетных прерываний возникает ситуация, требующая перепланирования потоков. Например, при обработке очередного прерывания от таймера нужно проверить, не исчерпан ли квант, выделенный текущему потоку. Другим примером может служить обработка прерывания от контроллера диска после завершения дисковой операции, которую могут ждать несколько потоков. Во всех таких ситуациях в ОС семейства Windows NT планировщик/диспетчер вызывается высокоуровневыми процедурами ядра не прямо посредством вызова процедуры, а косвенно с помощью программного прерывания. Это дает возможность отделить короткую, но требующую быстрой реакции системы процедуру обслуживания высокоприоритетного прерывания (например, наращивание системных часов), от менее критичной операции перепланирования пользовательских потоков. Прямой вызов планировщика/диспетчера потоков такой возможности бы не дал, и критичные запросы прерывания от контроллеров устройств ввода-вывода вынуждены были бы ждать, пока отработает планировщик. При этом планировщик, возможно, выбрал бы для выполнения другой поток, если бы работал после процедур обслуживания аппаратных прерываний, так как он получил бы свежие сведения о завершении некоторых операций ввода-вывода. Помещение вызова планировщика потоков в очередь позволяет выполнять его только в тех ситуациях, когда в системе отсутствуют ожидающие аппаратные запросы прерываний.

Наличие отдельного уровня для планировщика/диспетчера потоков не означает того, что он всегда вызывается с помощью программных прерываний. В тех случаях, когда он вызывается из кода, имеющего низкий уровень запроса на прерывание (если он выполняется, значит, высокоприоритетные запросы на прерывания отсутствуют), планировщик может быть вызван быстрее путем непосредственного внутрисегментного вызова процедуры. Например, системному вызову, переводящему поток по собственному желанию в состояние ожидания, нет смысла вызывать планировщик потоков по программному прерыванию.

Второе название диспетчерского уровня, DPC, являясь аббревиатурой от Deferred Procedure Call (отложенный вызов процедуры), говорит о том, что на этом уровне ожидают своей очереди отложенные вызовы и других процедур ОС, а не только планировщика/диспетчера. Процедуры ОС могут вызывать друг друга и непосредственно, но при многослойном построении ядра существуют более и менее приоритетные процедуры, и вызов менее приоритетных процедур из более приоритетных с помощью механизма программных прерываний позволяет, как и в случае с планировщиком, упорядочить во времени их выполнение, что оптимизирует работу ОС в целом. Примером процедур ОС, работающих на высоком приоритетном уровне, являются те части драйверов устройств ввода-вывода, которые выполняют короткие, но критичные ко времени реакции действия. В то же время существуют и другие части драйверов, которые выполняют менее срочную, но более объемную работу[1]. В ОС семейства Windows NT такие части драйверов оформляют как процедуры, вызываемые с помощью программных прерываний уровня «диспетчерский/ DPC» (DPC-процедуры), а само программное прерывание выполняет критичная часть драйвера. Естественно, существуют и другие модули ОС, оформляемые подобным образом.

Описанная программная реализация приоритетного обслуживания прерываний приводит к однотипной работе операционных систем семейства Windows NT на различных аппаратных платформах, что упрощает логику работы ОС и ее перенос на новые платформы. Отрицательным следствием такого централизованного подхода является некоторое замедление обработки прерываний, так как вместо непосредственной передачи управления драйверу устройства или обработчику исключений выполняется вызов некоего посредника — диспетчера прерываний. В ОС семейства Unix принят похожий, но менее централизованный подход к ведению и обработке очередей прерываний. Вместо единого диспетчера прерываний его функции выполняют процедуры, обслуживающие каждый приоритетный класс прерываний. Тем не менее общий подход к упорядочиванию обработки прерываний за счет их многоуровневой приоритезации и ведения системы очередей присутствует практически во всех современных операционных системах.

[1] В операционных системах семейства Unix эти части называют соответственно верхними половинами (top half) и нижними половинами (bottom half) обработчика прерываний.

Процедуры обработки прерываний и текущий процесс

Важной особенностью процедур, выполняемых по запросам прерываний, является то, что их работа чаще всего никак не связана с текущим процессом. Например, драйвер диска может получить управление после того, как контроллер диска записал в соответствующие сектора информацию, полученную от процесса *A*, но этот момент времени, скорее всего, не совпадет с периодом очередной итерации выполнения процесса *A* или его потока. В наиболее типичном случае процесс *A* будет находиться в состоянии ожидания завершения операции ввода-вывода (при синхронном режиме выполнения этой операции), и драйвер диска прервет какой-либо другой процесс, например, процесс *B*. В ОС семейства Windows NT процедуры, вызываемые как DPC, также могут работать в контексте процесса, отличающегося от того, для которого они выполняют свои функции.

В некоторых случаях вообще трудно однозначно определить, для какого процесса выполняет работу тот или иной программный модуль ОС, например планировщик потоков. Поэтому для такого рода процедур вводятся ограничения — они не имеют права использовать ресурсы (память, открытые файлы и т. п.), с которыми работает текущий процесс, или же от имени этого процесса запрашивать выделение дополнительных ресурсов. Процедуры обслуживания прерываний работают с ресурсами, которые были выделены им при инициализации соответствующего драйвера или инициализации самой операционной системы. Эти ресурсы принадлежат операционной системе, а не какому-либо процессу. В частности, память выделяется драйверам из системной области, то есть той области, на которую отображаются сегменты из общей части виртуального адресного пространства всех процессов. Поэтому обычно говорят, что процедуры обслуживания прерываний работают вне контекста процесса. Поскольку все подобные процедуры являются частью операционной системы, ответственность за соблюдение этих ограничений несет системный программист. Заставить свои модули выполнять эти ограничения ОС не может.

Хороший пример того, что не бывает правил без исключений, предоставляет нам семейство ОС Windows NT. В этих ОС существуют процедуры обслуживания прерываний, которые всегда выполняются в контексте определенного процесса. Это процедуры, вызываемые с помощью программного прерывания APC (Asynchronous Procedure Call — *асинхронный вызов процедуры*). Для них в диспетчере прерываний предусмотрен свой уровень приоритета IRQL, выше уровня для обычного кода, но ниже уровня DPC. Эти процедуры могут прервать текущий код и выполниться при соблюдении двух условий: текущий код имеет низший уровень приоритета (то есть выполняется обычный код), текущим процессом является вполне определенный процесс, описатель которого был задан в запросе на прерывание для данной APC-процедуры. APC-процедуры могут пользоваться ресурсами текущего процесса, и, собственно, для этого они и были введены. Основное назначение APC-процедур — перемещение данных, полученных драйвером от какого-либо устройства ввода-вывода, из памяти

системной области памяти, куда они помещаются после считывания из регистров контроллера этого устройства, в индивидуальную часть адресного пространства процесса, запросившего операцию ввода-вывода. Такое действие постоянно выполняется системой ввода-вывода, и для его реализации были введены такие специфические процедуры обслуживания прерываний, как APC.

Диспетчеризация прерываний является важной функцией ОС, и эта функция реализована практически во всех мультипрограммных операционных системах. Можно заметить, что в общем случае в операционной системе реализуется двухуровневый механизм планирования работ. Верхний уровень планирования выполняется диспетчером прерываний, который распределяет процессорное время между потоком поступающих запросов на прерывания различных типов — внешних, внутренних и программных. Оставшееся процессорное время распределяется другим диспетчером — диспетчером потоков на основании дисциплин квантования и других, рассмотренных ранее.

Системные вызовы

Системный вызов позволяет приложению обратиться к операционной системе с просьбой выполнить то или иное действие, оформленное как процедура (или набор процедур) кодового сегмента ОС.

Для прикладного программиста операционная система выглядит как некая библиотека, предоставляющая некоторый набор полезных функций, с помощью которых можно упростить прикладную программу или выполнить действия, запрещенные в пользовательском режиме, например обменяться данными с устройством ввода-вывода.

Реализация системных вызовов должна удовлетворять следующим требованиям:

- обеспечивать переключение в привилегированный режим;
- обладать высокой скоростью вызова процедур ОС;
- обеспечивать по возможности единообразное обращение к системным вызовам для всех аппаратных платформ, на которых работает ОС;
- допускать легкое расширение набора системных вызовов;
- обеспечивать контроль со стороны ОС за корректным использованием системных вызовов.

Первое требование для большинства аппаратных платформ может быть выполнено только с помощью механизма программных прерываний. Поэтому будем считать, что остальные требования нужно обеспечить именно для такой реализации системных вызовов. Как это обычно бывает, некоторые из этих требований взаимно противоречивы.

Для обеспечения высокой скорости было бы полезно использовать векторные свойства системы программных прерываний, имеющиеся во многих процессорах, то есть закрепить за каждым системным вызовом определенное значение вектора. Приложение при таком способе вызова непосредственно

указывает в аргументе запроса значение *вектора*, после чего управление немедленно передается требуемой процедуре операционной системы (рис. 4.14, *а*). Однако этот децентрализованный способ передачи управления привязан к особенностям аппаратной платформы, а также не позволяет операционной системе легко модифицировать набор системных вызовов и контролировать их использование. Например, в процессоре Pentium количество системных вызовов определяется количеством векторов прерываний, выделенных для этой цели из общего пула в 256 элементов (часть которых используется под аппаратные прерывания и обработку исключений). Добавление нового системного вызова требует от системного программиста тщательного поиска свободного элемента в таблице прерываний, которого к тому же на каком-то этапе развития ОС может и не оказаться.

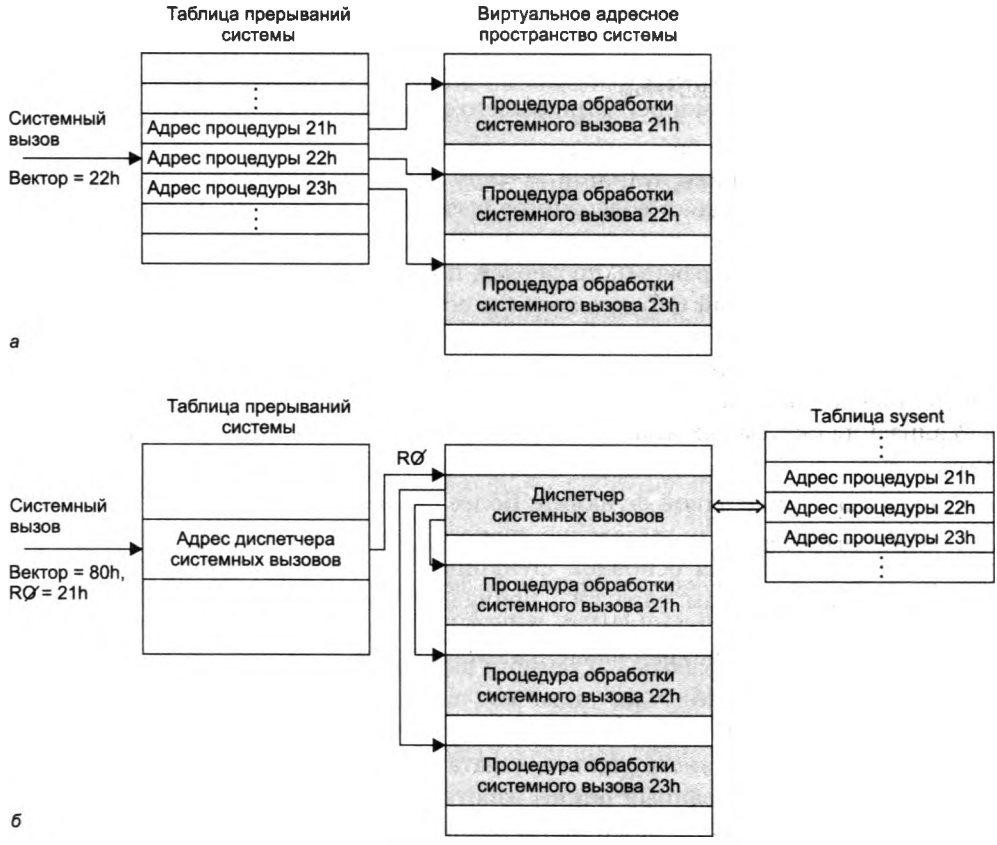


Рис. 4.14. Децентрализованная и централизованная схемы обработки системных вызовов

В большинстве ОС системные вызовы обслуживаются по централизованной схеме, основанной на существовании *диспетчера системных вызовов* (рис. 4.14, *б*). При любом системном вызове приложение выполняет программное прерывание

с определенным и единственным номером вектора. Например, ОС Linux использует для системных вызовов команду `INT 80h`, а ОС семейства Windows NT (при работе на платформе Pentium) — `INT 2Eh`. Перед выполнением программного прерывания приложение тем или иным способом передает операционной системе номер системного вызова, который является индексом в таблице адресов процедур ОС, реализующих системные вызовы (таблица sysent на рис. 4.14, *б*). Способ передачи зависит от реализации, например, номер можно поместить в определенный регистр общего назначения процессора (на рисунке этот регистр условно обозначен *R*0) или передать через стек (в этом случае после прерывания и перехода в привилегированный режим их нужно будет скопировать в системный стек из пользовательского, это действие в некоторых процессорах автоматизировано). Также некоторым способом передаются аргументы системного вызова, они могут помещаться как в регистры общего назначения, так и передаваться через стек или массив, находящийся в оперативной памяти. Массив удобен при большом объеме данных, передаваемых в качестве аргументов, — при этом в регистре общего назначения указывается адрес этого массива.

Диспетчер системных вызовов обычно представляет собой простую программу, которая сохраняет содержимое регистров процессора в системном стеке (поскольку в результате программного прерывания процессор переходит в привилегированный режим), проверяет, попадает ли запрошенный номер вызова в поддерживаемый ОС диапазон (то есть не выходит ли номер за границы таблицы), и передает управление процедуре ОС, адрес которой задан в таблице адресов системных вызовов.

Процедура реализации системного вызова извлекает из системного стека аргументы и выполняет заданное действие. Это действие может быть весьма простым, например чтение значения системных часов, так что системный вызов оформляется в виде одной функции. Более сложные системные вызовы, такие как чтение из файла или выделение процессу дополнительного сегмента памяти, требуют обращения основной функции системного вызова к нескольким внутренним процедурам ядра ОС, принадлежащим различным подсистемам, таким как подсистема ввода-вывода или управления памятью.

После завершения работы системного вызова управление возвращается диспетчеру, при этом он получает также код завершения этого вызова. Диспетчер восстанавливает регистры процессора, помещает в определенный регистр код возврата и выполняет инструкцию возврата из прерывания, которая восстанавливает непривилегированный режим работы процессора.

Для приложения системный вызов внешне ничем не отличается от вызова обычной библиотечной функции языка С, связанной (динамически или статически) с объектным кодом приложения и выполняющейся в пользовательском режиме. И такая ситуация действительно имеет место — для всех системных вызовов в библиотеках, предоставляемых компилятором С, имеются так называемые «заглушки» (в англоязычном варианте используется термин «stub» — остаток, огрызок). Каждая заглушка оформлена как С-функция, при этом она

содержит несколько ассемблерных строк, нужных для выполнения инструкции программного прерывания. Таким образом, пользовательская программа вызывает заглушку, а та, в свою очередь, вызывает процедуру ОС.

Для ускорения выполнения некоторых достаточно простых системных вызовов, которым, к тому же, не нужно действовать в привилегированном режиме, требуемая работа полностью выполняется библиотечной функцией. В данном случае такую функцию несправедливо называть заглушкой. К тому же, такая функция не является системным вызовом, а представляет собой «чистую» библиотечную функцию, выполняющую всю свою работу в пользовательском режиме в виртуальном адресном пространстве процесса, но прикладной программист может об этом и не знать — для него системные вызовы и библиотечные функции выглядят единообразно. Прикладной программист имеет дело с набором функций прикладного программного интерфейса (например, Win32 или POSIX), состоящего из библиотечных функций, часть из которых пользуется для завершения работы системными вызовами, а часть — нет.

Описанный табличный способ организации системных вызовов принят практически во всех операционных системах. Он позволяет легко модифицировать состав системных вызовов, просто добавив в таблицу новый адрес и расширив диапазон допустимых номеров вызовов.

Операционная система может выполнять системные вызовы в синхронном или асинхронном режиме.

Синхронный системный вызов означает, что процесс, сделавший такой вызов, приостанавливается (переводится планировщиком ОС в состояние ожидания) до тех пор, пока системный вызов не выполнит всю требующуюся от него работу (рис. 4.15, *а*). После этого планировщик переводит процесс в состояние готовности, и при очередном выполнении процесс гарантированно может воспользоваться результатами завершившегося к этому времени системного вызова. Синхронные вызовы называются также блокирующими, так как вызвавший системное действие процесс блокируется до его завершения.

Асинхронный системный вызов не приводит к переводу процесса в режим ожидания, после выполнения некоторых начальных системных действий, например запуска операции ввода-вывода, управление возвращается прикладному процессу (рис. 4.15, *б*).

Большинство системных вызовов в операционных системах являются синхронными, так как этот режим избавляет приложение от необходимости выяснения момента появления результата вызова. Вместе с тем, в новых версиях операционных систем количество асинхронных системных вызовов постепенно увеличивается, что дает больше свободы разработчикам сложных приложений. Особенно нужны асинхронные системные вызовы в микроядерных операционных системах, так как в них в пользовательском режиме работает та часть ОС, которой необходимо иметь полную свободу в организации своей работы, а такую свободу дает только асинхронный режим обслуживания вызовов микроядром.

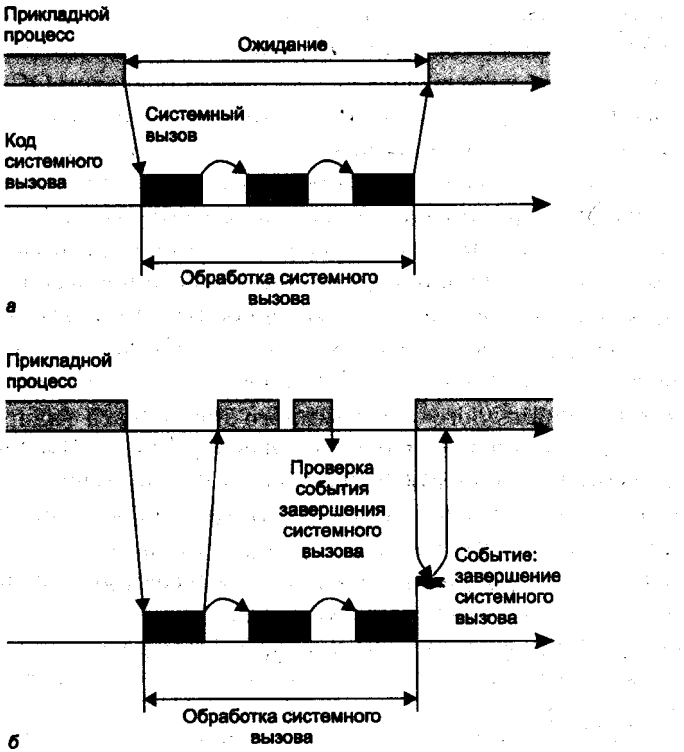


Рис. 4.15. Синхронные и асинхронные системные вызовы

Синхронизация процессов и потоков

Цели и средства синхронизации

Существует достаточно обширный класс средств операционной системы, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Во многих операционных системах эти средства называются средствами межпроцессного взаимодействия (Inter Process Communications, IPC), что лишь отражает историческую первичность понятия «процесс» по отношению к понятию «поток»[1]. Обычно к средствам IPC относят не только средства межпроцессной синхронизации, но и средства межпроцессного обмена данными.

Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с совместным использованием несколькими процессами аппаратных и информационных ресурсов вычислительной системы.

[1] В данном разделе мы будем говорить о синхронизации потоков, имея в виду, что если операционная система не поддерживает потоки, то все сказанное относится к синхронизации процессов.

Выполнение потока в мультипрограммной среде всегда имеет асинхронный характер. Очень сложно с полной определенностью сказать, на каком этапе выполнения будет находиться процесс в определенный момент времени. Даже в однопрограммном режиме не всегда можно точно оценить время выполнения задачи. Это время во многих случаях существенно зависит от значения исходных данных, которые влияют на количество циклов, направления разветвления программы, время выполнения операций ввода-вывода и т. п. Так как исходные данные в разные моменты запуска задачи могут быть разными, то и время выполнения отдельных этапов и задачи в целом является весьма неопределенной величиной. Еще более неопределенным является время выполнения программы в мультипрограммной системе. Моменты прерывания потоков, время нахождения их в очередях к разделяемым ресурсам, порядок выбора потоков для выполнения — все эти события являются результатом стечения многих обстоятельств и могут быть интерпретированы как случайные. В лучшем случае можно оценить вероятностные характеристики вычислительного процесса, например вероятность его завершения за данный период времени.

Таким образом, потоки в общем случае (когда программист не предпринял специальных мер по их синхронизации) протекают независимо, асинхронно друг другу. Это справедливо как по отношению к потокам одного процесса, выполняющим общий программный код, так и по отношению к потокам разных процессов, каждый из которых выполняет собственную программу.

Любое взаимодействие процессов (потоков), связанное с разделением ресурсов или обменом данными, требует их **синхронизации**. Синхронизация заключается в согласовании скоростей процессов (потоков) путем приостановки одного или нескольких из них до наступления некоторого события и последующей активизации при наступлении этого события.

Например, поток-получатель должен обращаться за данными только после того, как они помещены в буфер потоком-отправителем. Если же поток-получатель обратился к данным до момента их поступления в буфер, то он должен быть приостановлен.

При совместном использовании аппаратных ресурсов синхронизация также совершенно необходима. Когда, например, активному потоку требуется доступ к последовательному порту, а с этим портом в монопольном режиме работает другой поток, находящийся в данный момент в состоянии ожидания, то ОС приостанавливает активный поток и не активизирует его до тех пор, пока нужный ему порт не освободится. Часто нужна также синхронизация с событиями, внешними по отношению к вычислительной системе, например реакции на нажатие комбинации клавиш Ctrl+C.

Ежесекундно в системе происходят сотни событий, связанных с распределением и освобождением ресурсов, и ОС должна иметь надежные и производительные средства, которые бы позволяли ей синхронизировать потоки с происходящими в системе событиями.

Для синхронизации потоков прикладных программ программист может использовать как собственные средства и приемы синхронизации, так и средства операционной системы. Например, два потока одного прикладного процесса могут координировать свою работу с помощью доступной для них обоих *глобальной логической переменной*, которая устанавливается в единицу при осуществлении некоторого события, например выработки одним потоком данных, нужных для продолжения работы другого.

Однако во многих случаях более эффективными или даже единственно возможными являются средства синхронизации, предоставляемые операционной системой в форме системных вызовов. Так, потоки, принадлежащие разным процессам, не имеют возможности вмешиваться каким-либо образом в работу друг друга. Без посредничества операционной системы они не могут приостановить друг друга или оповестить о произошедшем событии.

Средства синхронизации используются операционной системой не только для синхронизации прикладных процессов, но и для ее внутренних нужд.

Обычно разработчики операционных систем предоставляют в распоряжение прикладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, кроме того, подобные средства могут быть функционально специализированными, например, средства для синхронизации потоков одного процесса, средства для синхронизации потоков разных процессов при обмене данными и т. д. Часто функциональные возможности разных системных вызовов синхронизации перекрываются, так что для решения одной задачи программист может воспользоваться несколькими вызовами в зависимости от своих личных предпочтений.

Необходимость синхронизации и гонки

Пренебрежение вопросами синхронизации в многопоточной системе может привести к неправильному решению задачи или даже к краху системы. Рассмотрим, например, задачу ведения базы данных клиентов некоторого предприятия (рис. 4.16). Каждому клиенту отводится отдельная запись в базе данных, в которой среди прочих полей имеются поля `Заказ` и `Оплата`. Программа, ведущая базу данных, оформлена как единый процесс, имеющий несколько потоков, в том числе поток *A*, который заносит в базу данных информацию о заказах, поступивших от клиентов, и поток *B*, который фиксирует в базе данных сведения об оплате клиентами выставленных счетов. Оба эти потока совместно работают над общим файлом базы данных, используя однотипные алгоритмы, включающие три шага.

1. Считать из файла базы данных в буфер запись о клиенте с заданным идентификатором.
2. Внести новое значение в поле `Заказ` (для потока *A*) или `Оплата` (для потока *B*).
3. Вернуть модифицированную запись в файл базы данных.

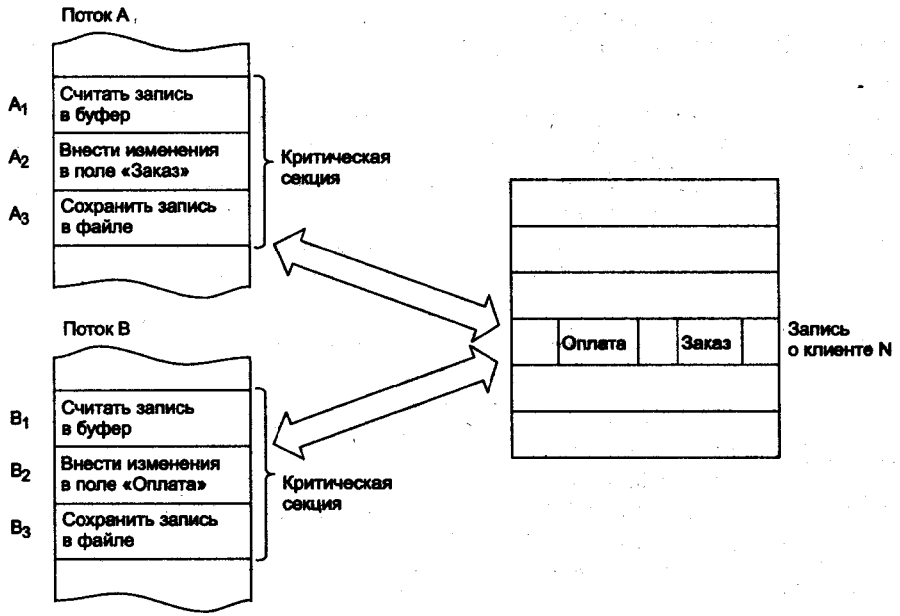


Рис. 4.16. Возникновение гонок при доступе к разделяемым данным

Обозначим соответствующие шаги для потока *A* как A_1, A_2 и A_3, а для потока *B* как B_1, B_2 и B_3. Предположим, что в некоторый момент поток *A* обновляет поле Заказ записи о клиенте *N*. Для этого он считывает эту запись в свой буфер (шаг A_1), модифицирует значение поля Заказ (шаг A_2), но записать запись в базу данных (шаг A_3) не успевает, так как его выполнение прерывается, например, вследствие завершения кванта времени.

Предположим также, что потоку *B* также потребовалось внести сведения об оплате относительно того же клиента *N*. Когда подходит очередь потока *B*, он успевает считать запись в свой буфер (шаг B_1) и выполнить обновление поля Оплата (шаг B_2), а затем прерывается. Заметим, что в буфере у потока *B* находится запись о клиенте *N*, в которой поле Заказ имеет прежнее, не измененное значение.

Когда в очередной раз управление будет передано потоку *A*, то он, продолжая свою работу, сделает запись о клиенте *N* с модифицированным полем Заказ в базу данных (шаг A_3). После прерывания потока *A* и активизации потока *B* последний запишет в базу данных поверх только что обновленной записи о клиенте *N* свой вариант записи, в которой обновлено значение поля Оплата. Таким образом, в базе данных будут зафиксированы сведения о том, что клиент *N* произвел оплату, но информация о его заказе окажется потерянной (рис. 4.17, *а*).

Сложность проблемы синхронизации кроется в нерегулярности возникающих ситуаций. Так, в предыдущем примере можно представить и другое развитие событий: могла быть потеряна информация не о заказе, а об оплате (рис. 4.17, *б*) или, напротив, все исправления могли быть успешно внесены (рис. 4.17, *в*).

Все определяется взаимными скоростями потоков и моментами их прерывания. Поэтому отладка взаимодействующих потоков является сложной задачей. Ситуации подобные той, когда два или более потоков обрабатывают разделяемые данные, а конечный результат зависит от соотношения скоростей потоков, называются **гонками**.

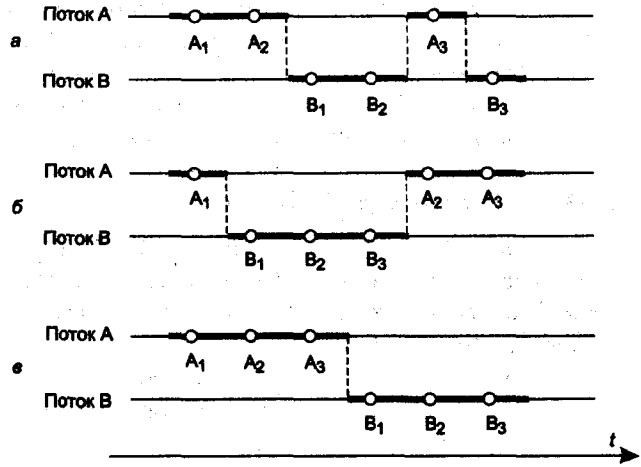


Рис. 4.17. Влияние относительных скоростей потоков на результат решения задачи

Критическая секция

В плане синхронизации потоков важным понятием является понятие «критической секции» программы.

> **Критическая секция** — это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено.

Критическая секция всегда определяется по отношению к определенным **критическим данным**, при несогласованном изменении которых могут возникнуть нежелательные эффекты. В предыдущем примере такими критическими данными являлись записи файла базы данных. Во всех потоках, работающих с критическими данными, должна быть определена критическая секция. Заметим, что в разных потоках критическая секция состоит, в общем случае, из разных последовательностей команд.

Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток, причем неважно, в каком состоянии, активном или приостановленном. Этот прием называют **взаимным исключением**. Операционная система использует разные способы реализации взаимного исключения. Некоторые способы обеспечивают взаим-

ное исключение при вхождении в критическую секцию только потоков одного процесса, в то время как другие могут обеспечить взаимное исключение также для потоков разных процессов.

Самый простой и в то же время самый неэффективный способ обеспечения взаимного исключения заключается в том, что операционная система позволяет потоку запрещать любые прерывания на время его нахождения в критической секции. Однако этот способ практически не применяется, так как опасно доверять управление системой пользовательскому потоку — он может надолго занять процессор, а при крахе потока в критической секции крах потерпит вся система, потому что прерывания никогда не будут разрешены.

Блокирующие переменные

Для синхронизации потоков одного процесса прикладной программист может использовать **глобальные блокирующие переменные**. С этими переменными, к которым все потоки процесса имеют прямой доступ, программист работает, не обращаясь к системным вызовам ОС.

Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. На рис. 4.18 показан фрагмент алгоритма потока, использующего для реализации взаимного исключения доступа к критическим данным D блокирующую переменную $F(D)$. Перед входом в критическую секцию поток проверяет, не работает ли уже какой-нибудь поток с данными D. Если переменная $F(D)$ установлена в 0, то данные заняты, и проверка циклически повторяется. Если же данные свободны, то есть $F(D) = 1$, тогда значение переменной $F(D)$ устанавливается в 0, и поток входит в критическую секцию. После того как поток выполнит все действия с данными D, значение переменной $F(D)$ снова устанавливается равным 1.

Блокирующие переменные могут использоваться не только при доступе к разделяемым данным, но и при доступе к разделяемым ресурсам любого вида.

Если все потоки написаны с учетом вышеописанных соглашений, то взаимное исключение гарантируется. При этом потоки могут быть прерваны операционной системой в любой момент и в любом месте, в том числе в критической секции.

Однако следует заметить, что одно ограничение на прерывания все же имеется. Нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной. Поясним это. Пусть в результате проверки переменной поток определил, что ресурс свободен, но сразу после этого, не успев установить переменную в 0, был прерван. За время его приостановки другой поток занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. Когда управление было возвращено первому потоку, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом, был нарушен принцип взаимного исключения, что потенциально может привести к нежелательным последствиям. Во избежание таких ситуаций в системе

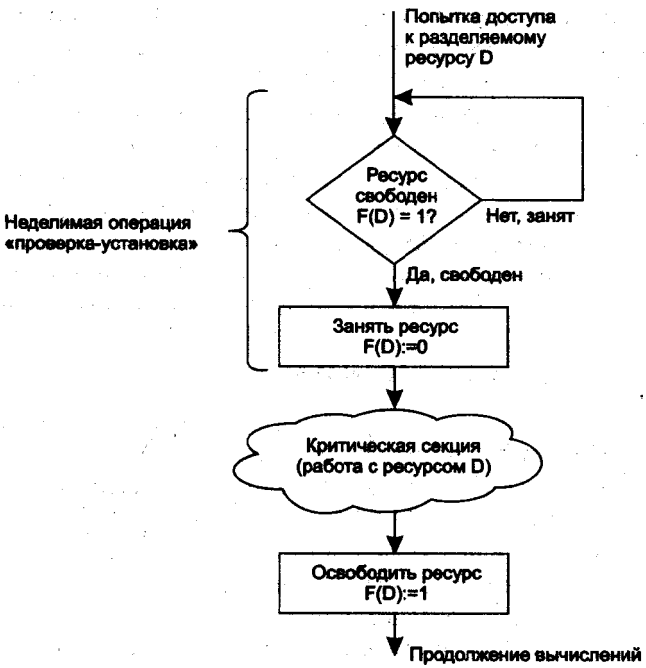


Рис. 4.18. Реализация критических секций с использованием блокирующих переменных

команд многих компьютеров предусмотрена единая, неделимая команда анализа и присвоения значения логической переменной (например, команды `BTC`, `BTR` и `BTS` процессора Pentium). При отсутствии такой команды в процессоре соответствующие действия должны реализовываться специальными системными примитивами[1], которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Реализация взаимного исключения описанным способом имеет существенный недостаток: в течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ к процессору, будет непрерывно опрашивать блокирующую переменную. При этом будет бесполезно тратиться выделяемое потоку процессорное время, которое можно было бы использовать для выполнения какого-нибудь другого потока. Для устранения этого недостатка во многих ОС предусматриваются специальные системные вызовы для работы с критическими секциями.

На рис. 4.19 показано, как с помощью указанных функций реализовано взаимное исключение в операционных системах семейства Windows NT. Перед тем как начать изменение критических данных, поток выполняет системный вызов `EnterCriticalSection()`. В рамках этого вызова сначала выполняется, как и в предыдущем случае, проверка блокирующей переменной, отражающей

[1] Примитив — базовая функция ОС.

состояние критического ресурса. Если системный вызов определил, что ресурс занят, то есть $F(D) = 0$, он, в отличие от предыдущего случая, не выполняет циклический опрос, а переводит поток в состояние ожидания (D) и делает отметку о том, что данный поток должен быть активизирован, когда соответствующий ресурс освободится. Поток, который в это время использует данный ресурс, после выхода из критической секции должен выполнить системную функцию `LeaveCriticalSection()`, в результате чего блокирующая переменная принимает значение, соответствующее свободному состоянию ресурса, то есть $F(D) = 1$, а операционная система просматривает очередь ожидающих этот ресурс потоков и переводит первый поток из очереди в состояние готовности.

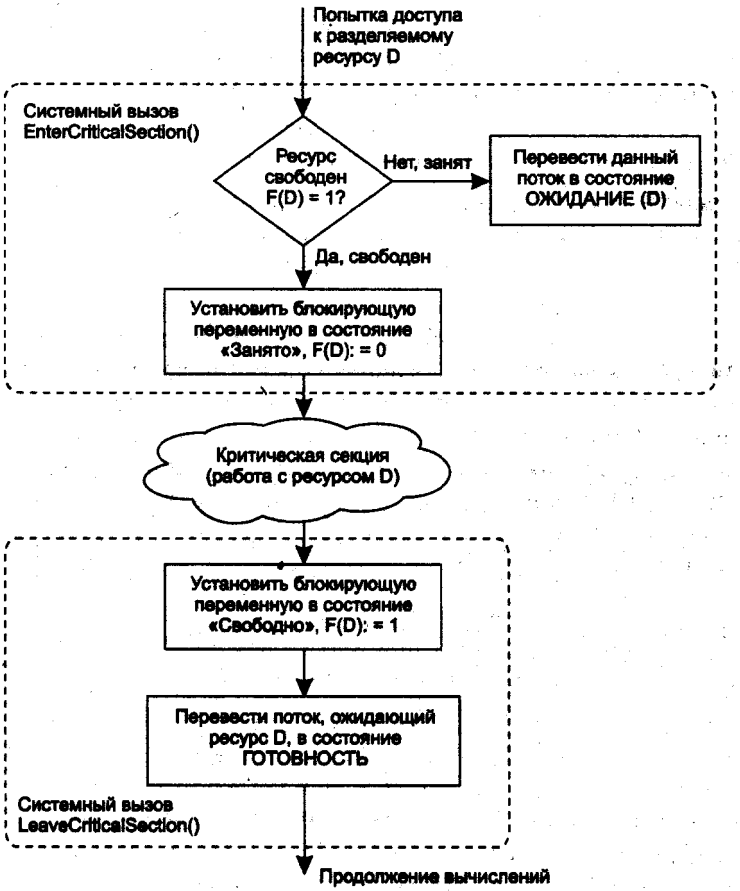


Рис. 4.19. Реализация взаимного исключения с использованием системных функций входа в критическую секцию и выхода из нее

Таким образом, исключается непроизводительная потеря процессорного времени на циклическую проверку освобождения занятого ресурса. Однако в тех случаях, когда объем работы в критической секции небольшой и существует

высокая вероятность в очень скором доступе к разделяемому ресурсу, более предпочтительным может оказаться использование блокирующих переменных. Действительно, в такой ситуации накладные расходы ОС по реализации функции входа в критическую секцию и выхода из нее могут превысить полученную экономию.

Семафоры

Обобщением блокирующих переменных являются так называемые **семафоры Дийкстры**. Вместо двоичных переменных Дийкстра (Dijkstra) предложил использовать переменные, которые могут принимать произвольные целые неотрицательные значения. Такие переменные, используемые для синхронизации вычислительных процессов, получили название семафоров.

Для работы с семафорами вводятся два примитива, традиционно обозначаемых *P* и *V*. Пусть переменная *S* представляет собой семафор. Тогда действия *V*(*S*) и *P*(*S*) определяются следующим образом.

- *V*(*S*) — переменная *S* увеличивается на 1 единым действием. Выборка, наращивание и запоминание не могут быть прерваны. К переменной *S* нет доступа другим потокам во время выполнения этой операции.
- *P*(*S*) — уменьшение *S* на 1, если это возможно. Если S = 0 и невозможно уменьшить *S*, оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию *P*, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также является неделимой операцией.

Никакие прерывания во время выполнения примитивов *V* и *P* недопустимы.

В частном случае, когда семафор *S* может принимать только значения 0 и 1, он превращается в блокирующую переменную, которую по этой причине часто называют двоичным семафором. Операция *P* заключает в себе потенциальную возможность перехода потока, который ее выполняет, в состояние ожидания, в то время как операция *V* может при некоторых обстоятельствах активизировать другой поток, приостановленный операцией *P*.

Рассмотрим использование семафоров на классическом примере взаимодействия двух выполняющихся в режиме мультипрограммирования потоков, один из которых пишет данные в буферный пул, а другой считывает их из буферного пула. Пусть буферный пул состоит из *N* буферов, каждый из которых может содержать одну запись. В общем случае поток-писатель и поток-читатель могут иметь различные скорости и обращаться к буферному пулу с переменной интенсивностью. В один период скорость записи может превышать скорость чтения, в другой — наоборот. Для правильной совместной работы поток-писатель должен приостанавливаться, когда все буферы оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, поток-читатель должен приостанавливаться, когда все буферы пусты, и активизироваться при появлении хотя бы одной записи.

Введем два семафора: e — число пустых буферов и f — число заполненных буферов, причем в исходном состоянии $e = N$, а $f = 0$. Тогда работа потоков с общим буферным пулом может быть описана следующим образом (рис. 4.20).

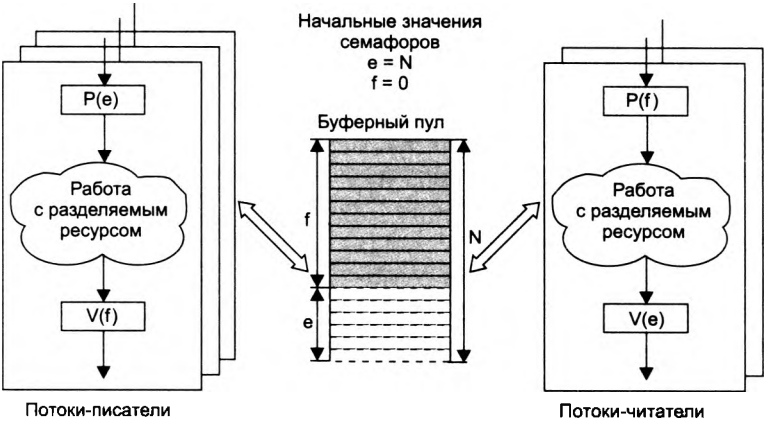


Рис. 4.20. Использование семафоров для синхронизации потоков

Поток-писатель, прежде всего, выполняет операцию $P(e)$, с помощью которой он проверяет, имеются ли в буферном пуле незаполненные буферы. В соответствии с семантикой операции P, если семафор e равен 0 (то есть свободных буферов в данный момент нет), то поток-писатель переходит в состояние ожидания. Если же значением e является положительное число, то он уменьшает число свободных буферов, записывает данные в очередной свободный буфер и после этого наращивает число занятых буферов операцией $V(f)$. Поток-читатель действует аналогичным образом с той разницей, что он начинает работу с проверки наличия заполненных буферов, а после чтения данных наращивает количество свободных буферов.

В данном случае предпочтительнее использовать семафоры вместо блокирующих переменных. Действительно, критическим ресурсом здесь является буферный пул, который может быть представлен как набор идентичных ресурсов — отдельных буферов, а, значит, с буферным пулом могут работать сразу несколько потоков, и именно столько, сколько буферов в нем содержится. Использование двоичной переменной не позволяет организовать доступ к критическому ресурсу более чем одному потоку. Семафор же решает задачу синхронизации более гибко, допуская к разделяемому пулу ресурсов заданное количество потоков. Так, в нашем примере с буферным пулом могут работать максимум N потоков, часть из которых может быть писателями, а часть — читателями.

Таким образом, семафоры позволяют эффективно решать задачу синхронизации доступа к ресурсным пулам, таким, например, как набор идентичных в функциональном назначении внешних устройств (модемов, принтеров, портов), или набор областей памяти одинаковой величины, или набор информаци-

онных структур. Во всех этих и подобных им случаях с помощью семафоров можно организовать доступ к разделяемым ресурсам сразу нескольких потоков.

Семафор может использоваться и в качестве блокирующей переменной. В рассмотренном примере для того, чтобы исключить коллизии при работе с разделяемой областью памяти, будем считать, что запись в буфер и считывание из буфера являются критическими секциями. Взаимное исключение будем обеспечивать с помощью двоичного семафора *b* (рис. 4.21). Оба потока после проверки доступности буферов должны выполнить проверку доступности критической секции.

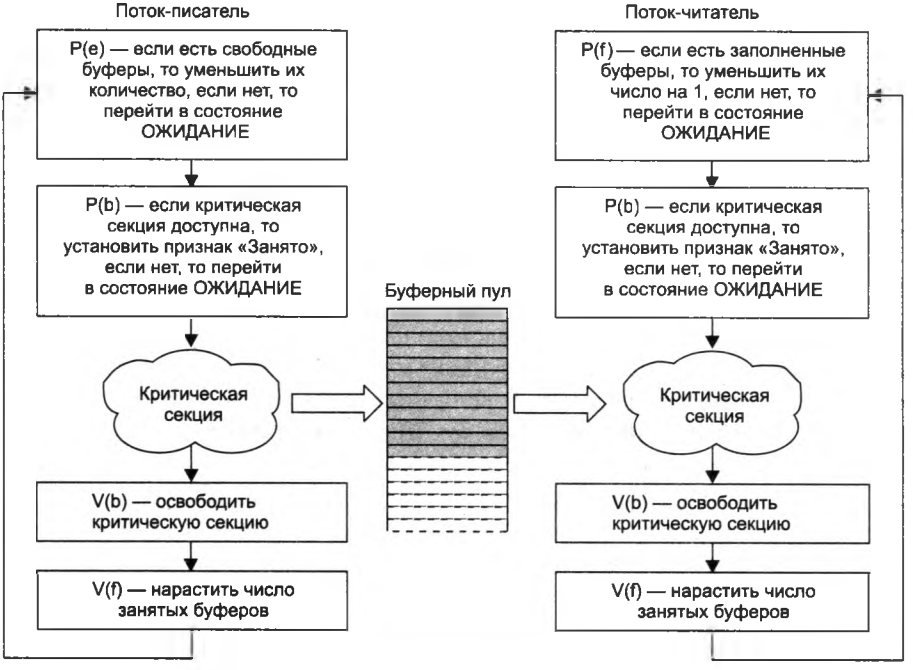


Рис. 4.21. Использование двоичного семафора

Тупики

Приведенный пример позволяет также проиллюстрировать еще одну проблему синхронизации — **взаимные блокировки**, называемые также **дедлоками** (deadlocks), **клинчами** (clinch) или **тупиками**. Покажем, что если переставить местами операции *P*(*e*) и *P*(*b*) в потоке-писателе, то при некотором стечении обстоятельств эти два потока могут взаимно блокировать друг друга.

Итак, пусть поток-писатель начинает свою работу с проверки доступности критической секции — операции *P*(*b*), и пусть он первым войдет в критическую секцию. Выполняя операцию *P*(*e*), он может обнаружить отсутствие свободных буферов и перейти в состояние ожидания. Как уже было показано, из этого состояния его может вывести только поток-читатель, который возьмет очередную

запись из буфера. Но поток-читатель не сможет этого сделать, так как для этого ему потребуется войти в критическую секцию, вход в которую заблокирован потоком-писателем. Таким образом, ни один из этих потоков не может завершить начатую работу, и возникает тупиковая ситуация, которая не может разрешиться без внешнего воздействия.

Рассмотрим еще один пример тупика. Пусть двум потокам, принадлежащим разным процессам и выполняющимся в режиме мультипрограммирования, для выполнения их работы нужно два ресурса, например принтер и последовательный порт. Такая ситуация может возникнуть, например во время работы приложения, задачей которого является распечатка информации, поступающей по модемной связи.

На рис. 4.22, *а* показаны фрагменты соответствующих программ. Поток *A* запрашивает сначала принтер, а затем порт, а поток *B* запрашивает устройства в обратном порядке. Предположим, что после того, как ОС назначила принтер

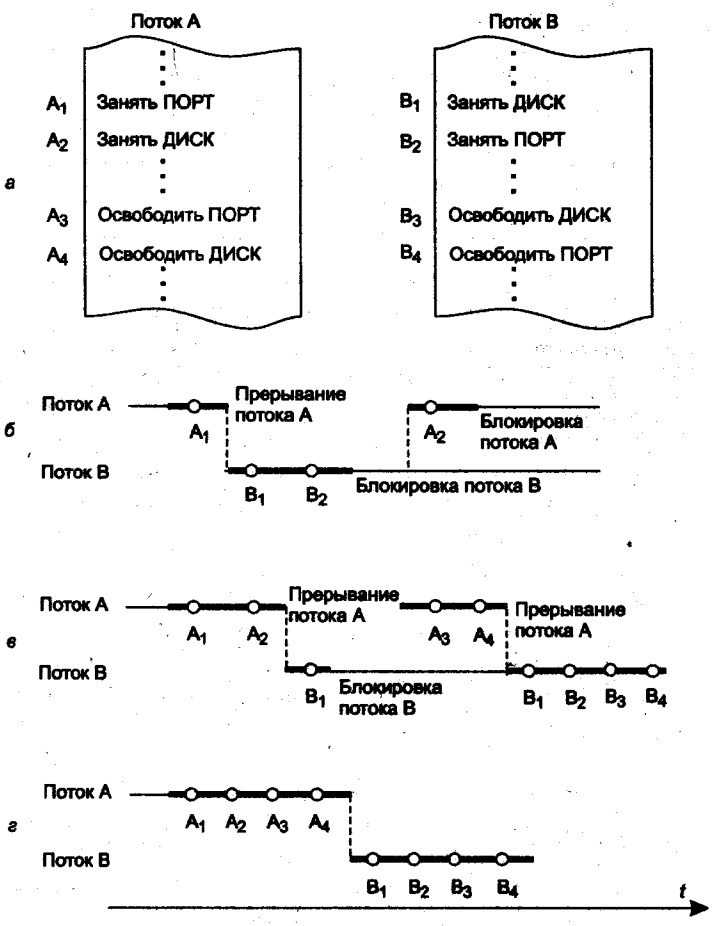


Рис. 4.22. Возникновение взаимных блокировок при выполнении программы

потоку *A* и установила связанную с этим ресурсом блокирующую переменную, поток *A* был прерван. Управление получил поток *B*, который сначала выполнил запрос на получение COM-порта, затем при выполнении следующей команды был заблокирован, так как принтер оказался уже занятым потоком *A*. Управление снова получил поток *A*, который в соответствии со своей программой сделал попытку занять порт и был заблокирован, поскольку порт уже выделен потоку *B*. В таком положении потоки *A* и *B* могут находиться сколь угодно долго.

В зависимости от соотношения скоростей потоков они могут либо взаимно блокировать друг друга (рис. 4.22, *б*), либо образовывать очереди к разделяемым ресурсам (рис. 4.22, *в*), либо совершенно независимо использовать разделяемые ресурсы (рис. 4.22, *г*).

ПРИМЕЧАНИЕ Тупиковые ситуации надо отличать от простых очередей, хотя те и другие возникают при совместном использовании ресурсов и внешне выглядят похоже: поток приостанавливается и ждет освобождения ресурса. Однако очередь — это нормальное явление, неотъемлемый признак высокого коэффициента использования ресурсов при случайном поступлении запросов. Очередь появляется тогда, когда ресурс недоступен в данный момент, но освободится через некоторое время, позволив потоку продолжить выполнение. Тупик же, что понятно из его названия, является в некотором роде неразрешимой ситуацией. Необходимым условием возникновения тупика является потребность потока сразу в нескольких ресурсах.

В рассмотренных примерах тупик был образован двумя потоками, но взаимно блокировать друг друга могут и большее число потоков. На рис. 2.23 показано такое распределение ресурсов R_i между несколькими потоками T_j, которое привело к возникновению взаимных блокировок. Стрелки обозначают потребность потока в ресурсах. Сплошная стрелка означает, что соответствующий ресурс был выделен потоку, а пунктирная стрелка соединяет поток с тем ресурсом, который необходим, но не может быть пока выделен, поскольку занят другим потоком. Например, потоку T_1 для выполнения работы необходимы ресурсы R_1 и R_2, из которых выделен только один — R_1, а ресурс R_2 удерживается потоком T_2. Ни один из четырех показанных на рисунке потоков не может продолжить свою работу, так как не имеет всех необходимых для этого ресурсов.

Невозможность потоков завершить начатую работу из-за взаимных блокировок снижает производительность вычислительной системы. Поэтому проблеме предотвращения тупиков уделяется большое внимание. На тот случай, когда взаимная блокировка все же возникает, система должна предоставить администратору-оператору средства, с помощью которых он смог бы распознать тупик, отличить его от обычной блокировки из-за временной недоступности ресурсов. И наконец, если тупик диагностирован, то нужны средства для снятия взаимных блокировок и восстановления нормального вычислительного процесса.

Тупики могут быть предотвращены на стадии написания программ, то есть программы должны быть написаны таким образом, чтобы тупик не мог возникнуть при любом соотношении взаимных скоростей потоков. Так, если бы в при-

мере, показанном на рис. 4.22, потоки *A* и *B* запрашивали ресурсы в одинаковой последовательности, то тупик был бы в принципе невозможен. Другой более гибкий подход к предотвращению тупиков заключается в том, что ОС каждый раз при запуске задач анализирует их потребности в ресурсах и определяет, может ли в данной мультипрограммной смеси возникнуть тупик. Если да, то запуск новой задачи временно откладывается. Операционная система может также использовать определенные правила при назначении ресурсов потокам, например, ресурсы могут выделяться операционной системой в определенной последовательности, общей для всех потоков.

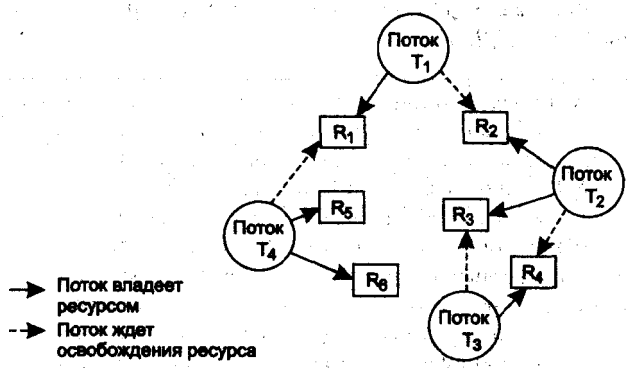


Рис. 4.23. Взаимная блокировка нескольких потоков

В тех же случаях, когда тупиковую ситуацию не удалось предотвратить, важно быстро и точно ее распознать, поскольку блокированные потоки не выполняют никакой полезной работы. Если тупиковая ситуация образована множеством потоков, занимающих массу ресурсов, распознавание тупика является нетривиальной задачей. Существуют формальные программно реализованные методы распознавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам. Анализ этих таблиц позволяет обнаружить взаимные блокировки.

Если же тупиковая ситуация возникла, то не обязательно снимать с выполнения все заблокированные потоки. Можно снять только часть из них, освободив ресурсы, ожидаемые остальными потоками, можно вернуть некоторые потоки в область подкачки, можно совершить «откат» некоторых потоков до так называемой контрольной точки, в которой запоминается вся информация, необходимая для восстановления выполнения программы с данного места. Контрольные точки расставляются в программе в тех местах, после которых возможно возникновение тупика.

Системные синхронизирующие объекты

Рассмотренные механизмы синхронизации, основанные на использовании глобальных переменных процесса, обладают существенным недостатком — они не подходят для синхронизации потоков разных процессов. В таких случаях могут

быть использованы предоставляемые операционной системой *системные синхронизирующие объекты*, которые «видны» всем потокам, даже если они принадлежат разным процессам и работают в разных адресных пространствах.

Примерами таких синхронизирующих объектов ОС являются *системные семафоры*, *мьютексы*, *события*, *таймеры* и другие — их набор зависит от конкретной ОС, которая создает эти объекты по запросам процессов.

Чтобы процессы могли разделять системные синхронизирующие объекты, в разных ОС используются разные методы. Некоторые ОС возвращают указатель на объект. Этот указатель может быть доступен всем родственным процессам, наследующим характеристики общего родительского процесса. В других ОС процессы в запросах на создание объектов синхронизации указывают имена, которые должны быть им присвоены. Далее эти имена используются разными процессами для манипуляций объектами синхронизации. В последнем случае работа с синхронизирующими объектами подобна работе с файлами. Их можно создавать, открывать, закрывать, уничтожать.

Кроме того, для синхронизации могут быть использованы такие «обычные» объекты ОС, как файлы, процессы и потоки.

Синхронизирующие объекты могут находиться в двух состояниях: сигнальном и несигнальном — свободном. Для каждого объекта смысл, вкладываемый в понятие *сигнальное состояние*, зависит от типа объекта. Так, например, поток переходит в сигнальное состояние тогда, когда он завершается. Процесс переходит в сигнальное состояние тогда, когда завершаются все его потоки. Файл переходит в сигнальное состояние в том случае, когда завершается операция ввода-вывода этого файла. Для остальных объектов сигнальное состояние устанавливается в результате выполнения специальных системных вызовов. Приостановка и активизация потоков осуществляются в зависимости от состояния синхронизирующих объектов ОС.

Потоки с помощью специального системного вызова сообщают операционной системе о том, что они хотят синхронизировать свое выполнение с состоянием некоторого объекта. Будем далее называть этот системный вызов `Wait(X)`, где `X` — указатель на объект синхронизации. Системный вызов, с помощью которого поток может перевести объект синхронизации в сигнальное состояние, назовем `Set(X)`.

Поток, выполнивший системный вызов `Wait(X)`, переводится операционной системой в состояние ожидания до тех пор, пока объект `X` не перейдет в сигнальное состояние. Примерами системных вызовов типа `Wait()` и `Set()` являются вызовы `WaitForSingleObject()` и `SetEvent()` в ОС семейства Windows NT, `DosSemWait()` и `DosSemSet()` в OS/2, `sleep()` и `wakeup()` в Unix.

Поток может ожидать установки сигнального состояния не одного объекта, а нескольких. При этом поток может попросить ОС активизировать его при установке либо одного из указанных объектов, либо всех объектов. Поток может в качестве аргумента системного вызова `Wait()` указать также максимальное время, которое он будет ожидать перехода объекта в сигнальное состояние, после чего ОС должна его активизировать в любом случае. Может случиться, что

установки некоторого объекта в сигнальное состояние ожидает сразу несколько потоков. В зависимости от объекта синхронизации в состояние готовности могут переводиться либо все ожидающие это событие потоки, либо один из них.

Синхронизация тесно связана с планированием потоков. Во-первых, любое обращение потока с системным вызовом `Wait(X)` влечет за собой действия в подсистеме планирования — этот поток снимается с выполнения и помещается в очередь ожидающих потоков, а из очереди готовых потоков выбирается и активизируется новый поток. Во-вторых, при переходе объекта в сигнальное состояние (в результате выполнения некоторого потока — либо системного, либо прикладного) ожидающий этот объект поток (или потоки) переводится в очередь готовых к выполнению потоков. В обоих случаях осуществляется перепланирование потоков, при этом если в ОС предусмотрены изменяемые приоритеты и/или кванты времени, то они пересчитываются по правилам, принятым в этой операционной системе.

Рассмотрим несколько примеров, когда в качестве синхронизирующих объектов используются *файлы*, *потоки* и *процессы*.

Пусть программа приложения построена так, что для выполнения запросов, поступающих из сети, основной поток создает вспомогательные серверные потоки. При поступлении от пользователя команды завершения приложения основной поток должен дождаться завершения всех серверных потоков и только после этого завершится сам. Следовательно, процедура завершения должна включать вызов `Wait(X1, X2, ...)`, где `X1`, `X2` — указатели на серверные потоки. В результате выполнения данного системного вызова основной поток будет переведен в состояние ожидания и останется в нем до тех пор, пока все серверные потоки не перейдут в сигнальное состояние, то есть завершатся. После этого ОС переведет основной поток в состояние готовности. При получении доступа к процессору основной поток завершится.

Другой пример. Пусть выполнение некоторого приложения требует последовательных работ-этапов. Для каждого этапа имеется свой отдельный процесс. Сигналом для начала работы каждого следующего процесса является завершение предыдущего. Для реализации такой логики работы необходимо в каждом процессе, кроме первого, предусмотреть выполнение системного вызова `Wait(X)`, в котором синхронизирующим объектом является предшествующий поток.

Объект-файл, переход которого в сигнальное состояние соответствует завершению операции ввода-вывода этого файла, используется в тех случаях, когда поток, инициировавший эту операцию, решает дождаться ее завершения прежде, чем продолжить свои вычисления.

Однако круг событий, с которыми потоку может потребоваться синхронизировать свое выполнение, отнюдь не исчерпывается завершением потока, процесса или операции ввода-вывода. Поэтому в ОС, как правило, имеются и другие, более универсальные объекты синхронизации, такие как событие (event), мьютекс (mutex), системный семафор, сигналы и другие.

Мьютекс, как и **семафор**, обычно используется для управления доступом к данным. В отличие от объектов-потоков, объектов-процессов и объектов-файлов,

которые при переходе в сигнальное состояние переводят в состояние готовности все потоки, ожидающие этого события, объект-мьютекс «освобождает» из очереди ожидающих только один поток. Работа мьютекса хорошо поясняется в терминах «владения». Пусть поток, который, пытаясь получить доступ к критическим данным, выполнил системный вызов `Wait(X)`, где `X` — указатель на мьютекс. Предположим, что мьютекс находится в сигнальном состоянии, в этом случае поток тут же становится его владельцем, устанавливая его в несигнальное состояние, и входит в критическую секцию. После того как поток выполнил работу с критическими данными, он «отдает» мьютекс, устанавливая его в сигнальное состояние. В этот момент мьютекс свободен и не принадлежит ни одному потоку. Если какой-либо поток ожидает его освобождения, то он становится следующим владельцем этого мьютекса, одновременно мьютекс переходит в несигнальное состояние.

Событие (в данном случае слово «событие» используется в узком смысле, как обозначение конкретного вида объектов синхронизации) обычно используется не для доступа к данным, а для того, чтобы оповестить другие потоки о том, что некоторые действия завершены. Пусть, например, в некотором приложении работа организована таким образом, что один поток читает данные из файла в буфер памяти, а другие потоки обрабатывают эти данные, затем первый поток считывает новую порцию данных, а другие потоки снова ее обрабатывают и т. д. В начале работы первый поток устанавливает объект-событие в несигнальное состояние. Все остальные потоки выполнили вызов `Wait(X)`, где `X` — указатель события, и находятся в приостановленном состоянии, ожидая наступления этого события. Как только буфер заполняется, первый поток сообщает об этом операционной системе, выполняя вызов `Set(X)`. Операционная система просматривает очередь ожидающих потоков и активизирует все потоки, которые ждут этого события.

Сигналы дают возможность задаче реагировать на событие, источником которого может быть операционная система или другая задача. Сигналы вызывают прерывание задачи и выполнение заранее предусмотренных действий. Сигналы могут вырабатываться синхронно, то есть как результат работы самого процесса, а могут быть направлены процессу другим процессом, то есть вырабатываться асинхронно.

Синхронные сигналы чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой, например о делении на нуль, об ошибке адресации, о нарушении защиты памяти и т. д.

Примером **асинхронного сигнала** является сигнал с терминала. Во многих ОС предусматривается оперативное снятие процесса с выполнения. Для этого пользователь может нажать некоторую комбинацию клавиш (`Ctrl+C`, `Ctrl+Break`), в результате чего ОС вырабатывает сигнал и направляет его активному процессу. Сигнал может поступить в любой момент выполнения процесса (то есть он является асинхронным), требуя от процесса немедленного завершения работы. В данном случае реакцией на сигнал является безусловное завершение процесса.

В системе может быть определен набор сигналов. Программный код процесса, которому поступил сигнал, может либо проигнорировать его, либо прореагировать на него стандартным действием (например, завершиться), либо выполнить специфические действия, определенные прикладным программистом. В последнем случае в программном коде необходимо предусмотреть специальные системные вызовы, с помощью которых операционная система информируется о том, какую процедуру надо выполнить в ответ на поступление того или иного сигнала.

Сигналы обеспечивают логическую связь между процессами, а также между процессами и пользователями (терминалами). Поскольку посылка сигнала предусматривает знание идентификатора процесса, то взаимодействие посредством сигналов возможно только между родственными процессами, способными получить данные об идентификаторах друг друга.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, блокирующие переменные, семафоры, сигналы и другие аналогичные средства, основанные на разделяемой памяти, оказываются непригодными. В таких системах синхронизация может быть реализована только посредством обмена сообщениями.

Выводы

- Мультипрограммирование, или многозадачность (multitasking), — это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются сразу несколько программ.
- Мультипрограммирование применяется для повышения эффективности вычислительной системы, которая может пониматься как:
  - общая пропускная способность вычислительной системы;
  - удобство работы пользователей, например, возможность интерактивной работы для нескольких пользователей или возможность одновременной работы одного пользователя с несколькими приложениями на одной машине;
  - реактивность системы — то есть способность системы выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением результата.
- В зависимости от выбранного критерия эффективности ОС делятся на системы пакетной обработки, системы разделения времени и системы реального времени.
- Мультипроцессорная обработка — это такой способ организации вычислительного процесса в системах с несколькими процессорами, в котором несколько задач (процессов, потоков) могут одновременно выполняться на разных процессорах системы.

- Основной задачей мультипрограммной операционной системы является распределение ресурсов между процессами и потоками — двумя базовыми единицами работы ОС.
- В операционных системах, в которых существуют как процессы, так и потоки, процесс рассматривается операционной системой как заявка на потребление всех видов ресурсов кроме одного — процессорного времени. Процессорное время распределяется ОС между другими единицами работы — потоками, представляющими собой последовательности команд.
- Потоки возникли в операционных системах как средство распараллеливания вычислений, облегчающее работу программиста. В ОС, не поддерживающей потоков, процесс всегда состоит из одного потока, и программисту приходится самостоятельно решать задачу синхронизации нескольких параллельных ветвей программы.
- Операционная система для реализации мультипрограммирования выполняет планирование и диспетчеризацию потоков (в ОС, не поддерживающих потоков, — диспетчеризацию процессов). Планирование включает определение момента смены текущего потока, а также выбор нового потока для выполнения. Диспетчеризация заключается в реализации найденного в результате планирования решения, то есть в переключении процессора с одного потока на другой.
- Планирование может выполняться динамически, когда решения принимаются во время работы системы на основе анализа текущей ситуации, или статически, если потоки запускаются на выполнение на основании заранее разработанного расписания. Первый способ характерен для универсальных ОС, второй — для специализированных ОС, например ОС реального времени.
- Динамический планировщик ОС может реализовывать различные алгоритмы планирования, которые делятся на такие крупные классы, как вытесняющие и не вытесняющие алгоритмы; алгоритмы квантования, приоритетные алгоритмы. Используемый алгоритм планирования зависит от назначения ОС. Применяются также смешанные алгоритмы, объединяющие достоинства нескольких классов.
- При применении вытесняющих алгоритмов планирования ОС получает полный контроль над вычислительным процессом, а при применении невытесняющих алгоритмов решения принимаются децентрализовано: активный поток определяет момент смены потоков, а ОС выбирает новый поток для выполнения.
- Система прерываний позволяет ОС реагировать на внешние события, происходящие асинхронно по отношению к вычислительному процессу: сигналы готовности устройств ввода-вывода, аварийные сигналы аппаратуры вычислительной системы и т. п.
- В зависимости от источника прерывания делятся на три больших класса:
  - внешние прерывания, связанные с сигналами от внешних устройств;

- внутренние прерывания, возникающие в результате ошибок вычислений;
- программные прерывания, представляющие собой удобный способ вызова процедур операционной системы.

- Механизм прерываний поддерживается аппаратными средствами компьютера и программными средствами операционной системы.
- Существует два основных способа выполнения прерывания: векторный (vectored), когда в процессор передается номер вызываемой процедуры обработки прерывания, и опрашиваемый (polled), когда процессор вынужден последовательно опрашивать потенциальные источники запроса прерывания.
- Для упорядочивания процедур обработки прерываний все источники прерываний распределяются по нескольким приоритетным уровням, а роль арбитра выполняет диспетчер прерываний ОС.
- Системные вызовы, с помощью которых приложения получают обслуживание со стороны ОС, реализуются на основе механизма программных прерываний. Системные вызовы могут выполняться синхронно, когда поток приостанавливается до завершения системного вызова, или асинхронно, когда поток продолжает работу параллельно с системной процедурой, реализующей вызов.
- Для синхронизации процессов и потоков, решающих общие задачи и совместно использующих ресурсы, в операционных системах существуют специальные средства: критические секции, семафоры, мьютексы, события, таймеры. Отсутствие синхронизации может приводить к таким нежелательным последствиям, как гонки и тупики.

Задачи и упражнения

1. Поясните употребление терминов «программа», «процесс», «задача», «поток», «нить».
2. В чем состоит принципиальное отличие состояний ожидания и готовности потока, ведь и в том и в другом он ожидает некоторого события?
3. Мультипрограммные операционные системы принято разделять на системы реального времени, системы разделения времени, системы пакетной обработки. В то же время алгоритмы планирования могут быть основаны на квантовании, относительных приоритетах и абсолютных приоритетах. Предложите для каждого из перечисленных тинов ОС наиболее подходящий, по вашему мнению, тип алгоритма планирования.
4. В какой очереди (ожидающих или готовых) скапливается большее число процессов:
   1) в интерактивных системах разделения времени;
   2) в системах пакетной обработки, решающих «счетные» задачи.
5. Известно, что программа *A* выполняется в монопольном режиме за 10 минут, а программа *B* — за 20 минут, то есть при последовательном выполне-

нии они требуют 30 минут. Если T — время выполнения обеих этих задач в режиме мультипрограммирования, то какое из следующих неравенств справедливо:

1) $T < 10$;
2) $10 < T < 20$;
3) $20 < T < 30$;
4) $T > 30$.

6. Может ли процесс в мультипрограммном режиме выполняться быстрее, чем в монопольном?
7. Чем объясняется потенциально более высокая надежность операционных систем, в которых реализована вытесняющая многозадачность?
8. В каких ОС реализована невытесняющая многозадачность? А вытесняющая многозадачность?
9. При невытесняющем планировании необходимо, чтобы во всех выполняющихся программах были предусмотрены кодовые последовательности, которые передают управление ОС. Эти точки возврата управления прикладной программист должен определить заранее еще до выполнения программы. Можно ли сказать, что в этом случае мы имеем дело со статическим планированием?
10. Приведите пример алгоритма планирования, в результате работы которого процесс, располагая всеми необходимыми ресурсами, может бесконечно долго находиться в системе, не имея возможности завершиться.
11. Могут ли быть применены сразу все перечисленные характеристики к одному алгоритму планирования потоков:
    1) вытесняющий с абсолютными динамическими приоритетами;
    2) невытесняющий с абсолютными фиксированными приоритетами;
    3) невытесняющий с относительными динамическими приоритетами;
    4) вытесняющий с абсолютными фиксированными приоритетами, основанный на квантовании с динамически изменяющейся длиной кванта;
    5) невытесняющий основанный на квантовании с фиксированной длиной кванта.

    Для тех вариантов, которые вы считаете возможными, опишите более подробно алгоритм планирования.
12. Являются ли синонимами термины «планирование процессов» и «диспетчеризация процессов»?
13. Можно ли задачу планирования процессов целиком возложить на приложения?
14. Приведите пример задачи, при программировании которой использование механизма потоков может привести к существенному повышению скорости ее выполнения?

15. Возможно ли существование асимметричной мультипроцессорной ОС для компьютера с симметричной мультипроцессорной архитектурой?
16. Сравните два варианта организации мультипроцессорной обработки. В первом случае процесс (поток), начав выполняться на каком-либо процессоре, при каждой следующей активизации будет назначаться планировщиком на этот же процессор. Во втором варианте процесс (поток) каждый раз, в общем случае, выполняется на произвольно выбранном свободном процессоре. Какой вариант эффективнее в отношении времени выполнения отдельного приложения? В отношении суммарной производительности компьютера?
17. Представьте себе ОС, разработанную для компьютера, в котором отсутствует система прерываний. Какой алгоритм планирования процессов может быть реализован в такой ОС?
18. Охарактеризуйте алгоритмы планирования, реализованные в операционных системах, используя следующие характеристики: вытесняющий/невытесняющий, приоритеты относительные/абсолютные, динамические/фиксированные, кванты фиксированные/динамические, процессы жесткого/мягкого реального времени:
    1) ОС семейства Windows NT;
    2) NetWare 3.x и 4.x;
    3) OS/2.
19. Какие события вызывают перепланирование процессов (потоков)?
20. Поясните разницу между программными и аппаратными прерываниями.
21. Что такое вектор прерываний?
22. Какой тип системы прерываний — векторный или опрашиваемый — реализован в процессоре Pentium?
23. Всегда ли прерывание вызывает необходимость перепланирования процессов?
24. Опишите механизм обработки прерываний в ОС семейства Windows NT.
25. Какими средствами синхронизации процессов располагает современная ОС?
26. Зачем в системе команд многих компьютеров предусмотрена единая, неделимая команда анализа и присвоения значения логической переменной, хотя эти же действия могут быть выполнены с помощью двух разных команд, также обычно присутствующих в системе команд?
27. Представим себе двух студентов, которым нужно поработать с одной и той же книгой, имеющейся в библиотеке в единственном экземпляре. Они одновременно пришли в библиотеку, но один из них сначала пошел в читальный зал, и заняв единственное свободной место, отправился в книжное хранилище, а другой — наоборот начал с того, что получил книгу, а потом пошел в читальный зал искать место. В результате ни один из них не может выполнить работу, так как для этого им не хватает необходимого ресурса. Можно ли считать, что в данном случае произошла взаимная блокировка, или другими словами клинч?

Глава 5
Управление памятью

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Под **памятью** (memory) здесь подразумевается оперативная память компьютера. В отличие от памяти жесткого диска, которую называют **внешней памятью** (storage), оперативной памяти для сохранения информации требуется постоянное электропитание.

Особая роль памяти объясняется тем, что процессор может выполнять инструкции программы только в том случае, если они находятся в памяти. Память распределяется как между модулями прикладных программ, так и между модулями самой операционной системы.

Функции ОС по управлению памятью

В ранних ОС управление памятью сводилось просто к загрузке программы и ее данных с некоторого внешнего накопителя (перфоленты, магнитной ленты или магнитного диска) в память. С появлением мультипрограммирования перед ОС были поставлены новые задачи, связанные с распределением имеющейся памяти между несколькими одновременно выполняющимися программами.

Функциями ОС по *управлению памятью* в мультипрограммной системе являются:

- отслеживание свободной и занятой памяти;
- выделение памяти процессам и освобождение памяти при завершении процессов;
- вытеснение кодов и данных процессов из оперативной памяти на диск (полное или частичное), когда размеры основной памяти недостаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место;
- настройка адресов программы на конкретную область физической памяти;
- защита памяти — запрет выполняемому процессу записывать данные в память, назначенной другому процессу, или читать их оттуда. Эта функция,

как правило, реализуется программными модулями ОС в тесном взаимодействии с аппаратными средствами.

Помимо первоначального выделения памяти процессам при их создании, ОС должна также заниматься динамическим распределением памяти, то есть выполнять запросы приложений во время выполнения на выделение им дополнительной памяти. После того как приложение перестает нуждаться в дополнительной памяти, оно может возвратить ее системе. Выделение памяти случайной длины в случайные моменты времени из общего пула памяти приводит к фрагментации и, вследствие этого, к неэффективному ее использованию. Дефрагментация памяти тоже является функцией операционной системы.

Во время работы операционной системы ей часто приходится создавать новые служебные информационные структуры, такие как описатели процессов и потоков, различные таблицы распределения ресурсов, буферы, используемые процессами для обмена данными, синхронизирующие объекты и т. п. Все эти системные объекты требуют памяти. В некоторых ОС заранее (во время установки) резервируется некоторый фиксированный объем памяти для системных нужд. В других же ОС используется более гибкий подход, при котором память для системных целей выделяется динамически. В таком случае разные подсистемы ОС при создании своих таблиц, объектов, структур и т. п. обращаются к подсистеме управления памятью с запросами.

Типы адресов

Для идентификации переменных и команд на разных этапах жизненного цикла программы используются символьные имена (метки), виртуальные адреса и физические адреса (рис. 5.1).

- **Символьные имена (метки)** присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.
- **Виртуальные, математические,** или **логические, адреса** вырабатывает транслятор, переводящий программу на машинный язык. Поскольку во время трансляции в общем случае неизвестно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам условные виртуальные адреса, обычно считая по умолчанию, что начальным адресом программы будет нулевой адрес.
- **Физические адреса** соответствуют номерам ячеек оперативной памяти, где в действительности расположены или будут расположены переменные и команды.

Совокупность виртуальных адресов процесса называется **виртуальным адресным пространством**.

Диапазон возможных адресов виртуального пространства у всех процессов является одним и тем же. Например, при использовании 32-разрядных виртуальных адресов этот диапазон задается границами 00000000_{16} и $FFFFFFFF_{16}$.

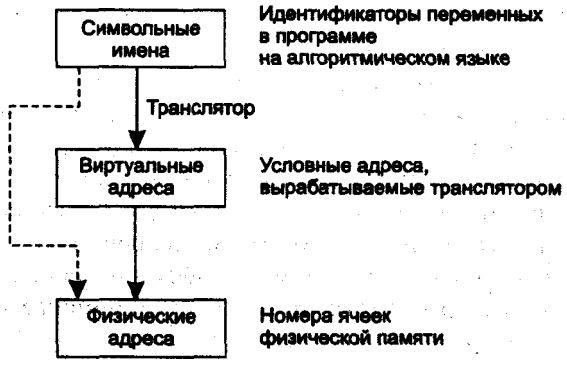


Рис. 5.1. Типы адресов

В то же время каждый процесс имеет собственное виртуальное адресное пространство — транслятор присваивает виртуальные адреса переменным и кодам каждой программе независимо (рис. 5.2).

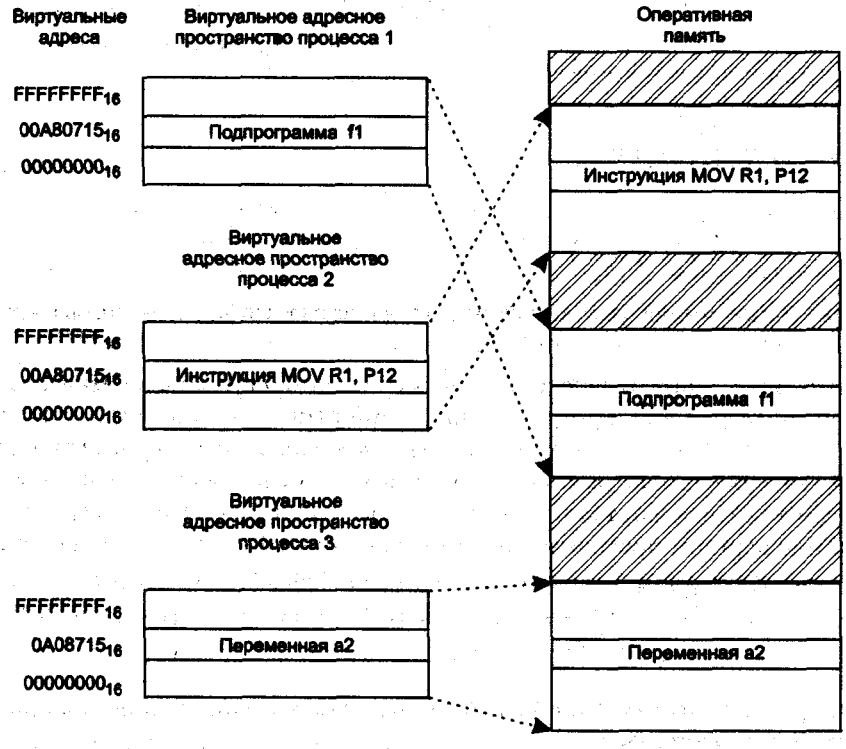


Рис. 5.2. Виртуальные адресные пространства нескольких программ

Совпадение виртуальных адресов переменных и команд различных процессов не приводит к конфликтам, так как в том случае, когда эти переменные од-

новременно присутствуют в памяти, операционная система отображает их на разные физические адреса[1].

В разных операционных системах используются разные способы структуризации виртуального адресного пространства. В одних ОС виртуальное адресное пространство процесса подобно физической памяти представлено в виде непрерывной последовательности виртуальных адресов. Такую структуру адресного пространства называют **линейной**, или **плоской** (flat). При этом виртуальным адресом является единственное число, представляющее собой смещение относительно начала (обычно это значение 000...000) виртуального адресного пространства (рис. 5.3, *а*). Адрес такого типа называют **линейным виртуальным адресом**.

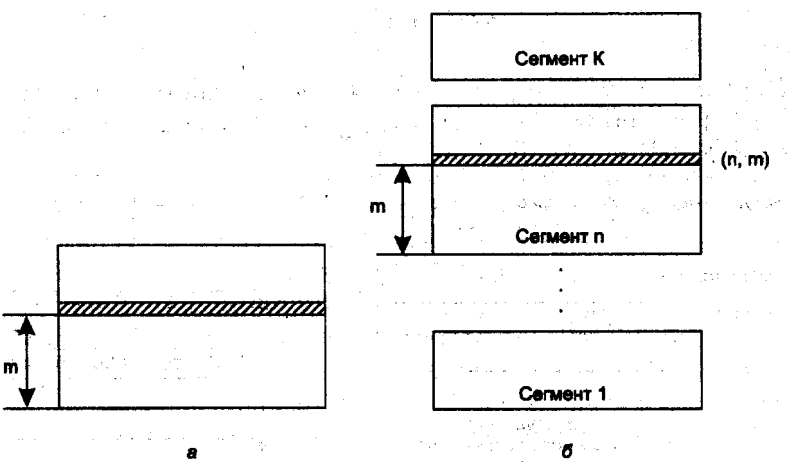


Рис. 5.3. Типы виртуальных адресных пространств: плоское (*а*), сегментированное (*б*)

В других ОС виртуальное адресное пространство делится на части, называемые **сегментами** (или секциями, или областями, или другими терминами). В этом случае говорят, что виртуальное адресное пространство имеет *сегментированную* структуру. Для позиционирования данных здесь помимо линейного виртуального адреса может быть использован виртуальный адрес, учитывающий разбиение на сегменты и представляющий собой пару чисел (*n*, *m*), где *n* определяет сегмент, а *m* — смещение внутри сегмента (рис. 5.3, *б*).

Существуют и более сложные способы структуризации виртуального адресного пространства, когда виртуальный адрес образуется тремя или даже более числами.

Задачей операционной системы является отображение индивидуальных виртуальных адресных пространств всех одновременно выполняющихся процессов

[1] Если необходимо, чтобы несколько процессов разделяли общие данные или коды, операционная система отображает соответствующие участки виртуального адресного пространства этих процессов на один и тот же участок физической памяти (см. раздел «Разделяемые сегменты памяти»).

на общую физическую память. При этом ОС отображает либо все виртуальное адресное пространство, либо только определенную его часть. Процедура преобразования виртуальных адресов в физические должна быть максимально прозрачна для пользователя и программиста.

Существует два принципиально отличающихся подхода к преобразованию виртуальных адресов в физические.

В первом случае замена виртуальных адресов физическими выполняется один раз для каждого процесса во время начальной загрузки программы в память. Специальная системная программа — **перемещающий загрузчик** — на основании имеющихся у нее исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, а также информации, предоставленной транслятором об адресно-зависимых элементах программы, выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, то есть операнды инструкций и адреса переходов имеют те значения, которые выработал транслятор. В наиболее простом случае, когда виртуальная и физическая память процесса представляют собой единые непрерывные области адресов, операционная система выполняет преобразование виртуальных адресов в физические по следующей схеме. При загрузке операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Схема такого преобразования показана на рис. 5.4. Пусть, например, операционная система использует линейно структурированное виртуальное адресное пространство, и пусть некоторая программа, работающая под управлением этой ОС, загружена в физическую память, начиная с физического адреса `S`. Операционная система запоминает значение начального смещения `S` и во время выполнения программы помещает его в специальный регистр процессора. При обращении к памяти виртуальные адреса данной программы преобразуются в физические путем прибавления к ним смещения `S`. Например, при выполнении инструкции `MOV` пересылки данных, находящихся по адресу `VA`, виртуальный адрес `VA` заменяется физическим адресом `VA+S`.

Последний способ является более гибким: в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти, динамическое преобразование виртуальных адресов позволяет перемещать программный код процесса в течение всего периода его выполнения. Однако использование перемещающего загрузчика более экономично, так как в этом случае преобразование каждого виртуального адреса происходит только один раз во время загрузки, а при динамическом преобразовании — при каждом обращении по данному адресу.

В некоторых случаях (обычно в специализированных системах), когда заранее точно известно, в какой области оперативной памяти будет выполняться программа, транслятор выдает исполняемый код сразу в физических адресах.

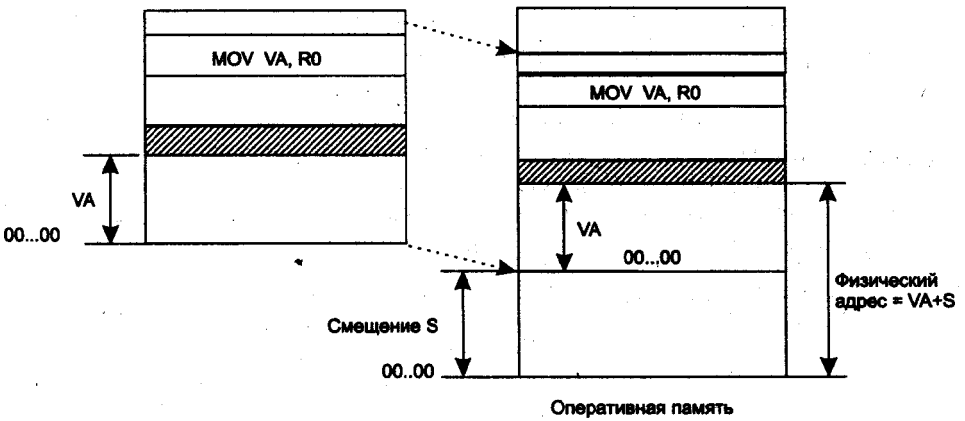


Рис. 5.4. Схема динамического преобразования адресов

Необходимо различать *максимально возможное* виртуальное адресное пространство процесса и *назначенное* (*выделенное*) процессу виртуальное адресное пространство. В первом случае речь идет о максимальном размере виртуального адресного пространства, определяемом архитектурой компьютера, на котором работает ОС, и, в частности, разрядностью его схем адресации (32-разрядная, 64-разрядная и т. п.). Например, при работе на компьютерах с 32-разрядными процессорами Intel Pentium операционная система может предоставить каждому процессу виртуальное адресное пространство размером до 4 Гбайт (2^{32}). Однако это значение представляет собой только потенциально возможный размер виртуального адресного пространства, который редко на практике бывает необходим процессу. Процесс использует только часть доступного ему виртуального адресного пространства.

Назначенное виртуальное адресное пространство представляет собой набор виртуальных адресов, действительно нужных процессу для работы. Эти адреса первоначально назначает программе транслятор на основании текста программы, когда создает кодовый (текстовый) сегмент, а также сегмент или сегменты данных, с которыми работает программа. Затем при создании процесса ОС фиксирует назначенное виртуальное адресное пространство в своих системных таблицах. В ходе своего выполнения процесс может увеличить размер первоначального назначенного виртуального ему адресного пространства, запросив у ОС создания дополнительных сегментов или увеличения размера существующих. В любом случае операционная система обычно следит за корректностью использования процессом виртуальных адресов — процессу не разрешается оперировать виртуальным адресом, выходящим за пределы назначенных ему сегментов.

Максимальный размер виртуального адресного пространства ограничивается только разрядностью адреса, присущей данной архитектуре компьютера, и, как правило, не совпадает с объемом физической памяти, имеющимся в компьютере.

Сегодня для машин универсального назначения типична ситуация, когда объем виртуального адресного пространства превышает доступный объем оперативной памяти. В таком случае операционная система для хранения данных виртуального адресного пространства процесса, не помещающихся в оперативную память, использует внешнюю память, которая в современных компьютерах представлена жесткими дисками (рис. 5.5, *а*). Именно на этом принципе основана **виртуальная память** — наиболее совершенный механизм, используемый в операционных системах для управления памятью.

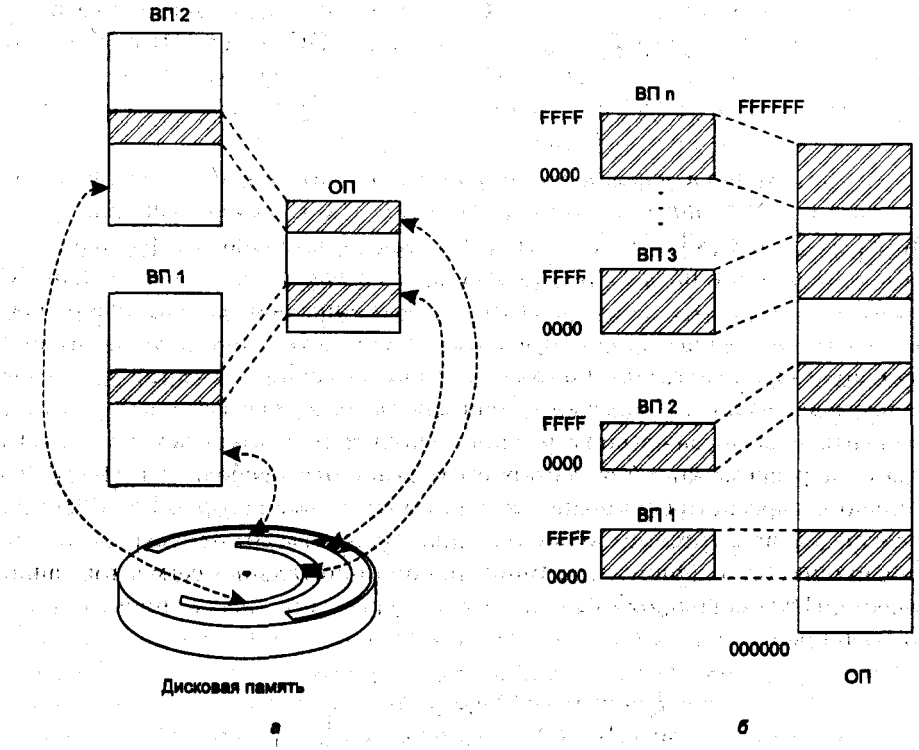


Рис. 5.5. Соотношение объемов виртуального адресного пространства и физической памяти: виртуальное адресное пространство превосходит объем физической памяти (*а*), виртуальное адресное пространство меньше объема физической памяти (*б*)

ПРИМЕЧАНИЕ Необходимо подчеркнуть, что виртуальное адресное пространство и виртуальная память — это различные механизмы, и они не обязательно реализуются в операционной системе одновременно. Можно представить себе ОС, в которой поддерживаются виртуальные адресные пространства для процессов, но отсутствует механизм виртуальной памяти. Это возможно только в том случае, если размер виртуального адресного пространства каждого процесса меньше объема физической памяти.

Однако соотношение объемов виртуальной и физической памяти может быть и обратным. Так, в миникомпьютерах 80-х годов разрядности поля адреса

нередко не хватало для того, чтобы охватить всю имеющуюся оперативную память. Несколько процессов могло быть загружено в память одновременно и целиком (рис. 5.5, *б*).

Во время работы процесса постоянно выполняются переходы от прикладных кодов к кодам ОС, которые вызываются либо явно из прикладных процессов как системные функции, либо как реакция на внешние события или на исключительные ситуации, возникающие при некорректном поведении прикладных кодов. Для того чтобы упростить передачу управления от прикладного кода коду ОС, а также для легкого доступа модулей ОС к прикладным данным (например, для вывода их на внешнее устройство), в большинстве ОС ее сегменты разделяют виртуальное адресное пространство с прикладными сегментами активного процесса. То есть сегменты ОС и сегменты активного процесса образуют единое виртуальное адресное пространство.

Обычно виртуальное адресное пространство процесса делится на две непрерывные части: *системную* и *пользовательскую*. В некоторых ОС (например, ОС семейства Windows NT, OS/2) эти части имеют одинаковый размер — по 2 Гбайта, хотя в принципе деление может и другим, например, 1 Гбайт — для ОС, и 2 Гбайта — для прикладных программ[1]. Часть виртуального адресного пространства каждого процесса, отводимая под сегменты ОС, является идентичной для всех процессов. Поэтому при смене активного процесса заменяется только вторая часть виртуального адресного пространства, содержащая его индивидуальные сегменты, как правило, коды и данные прикладной программы (рис. 5.6). Архитектура современных процессоров отражает эту особенность структуры виртуального адресного пространства, например, в процессорах Intel Pentium существует два типа системных таблиц: одна — для описания сегментов, общих для всех процессов, а другая — для описания индивидуальных сегментов данного процесса. При смене процесса первая таблица остается неизменной, а вторая заменяется новой.

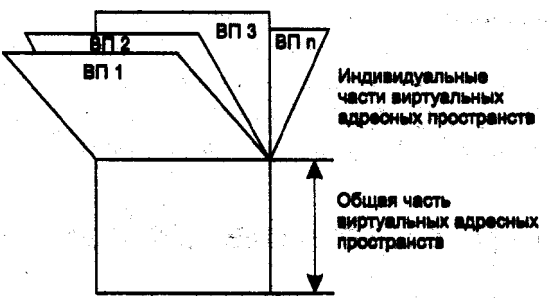


Рис. 5.6. Общая и индивидуальные части виртуальных адресных пространств

[1] Деление виртуального адресного пространства на две непрерывные области не является обязательным — можно представить себе ОС, которая обходится без него, чередуя свои сегменты и сегменты прикладных программ и размещая их в виртуальном адресном пространстве в произвольном порядке.

Описанное назначение двух частей виртуального адресного пространства — для сегментов ОС и для сегментов прикладной программы — является типичным, но не абсолютным. Имеются и исключения из общего правила. В некоторых ОС существуют системные процессы, порожденные для решения внутренних задач ОС. В этих процессах отсутствуют сегменты прикладной программы, но они могут расположить некоторые свои сегменты (сегменты ОС) в общей части виртуального адресного пространства, а некоторые — в индивидуальной части, обычно предназначенной для прикладных сегментов. И, наоборот, в общей, системной части виртуального адресного пространства размещаются сегменты прикладного кода, предназначенные для совместного использования несколькими прикладными процессами.

Механизм страничной памяти в большинстве универсальных операционных систем применяется ко всем сегментам пользовательской части виртуального адресного пространства процесса. Исключения могут составлять специализированные ОС, например ОС реального времени, в которых некоторые сегменты жестко фиксируются в оперативной памяти и, соответственно, никогда не выгружаются на диск — это обеспечивает быструю реакцию определенных приложений на внешние события.

Системная часть виртуальной памяти в ОС любого типа включает область, подвергаемую *страничному вытеснению* (paged), и область, на которую страничное вытеснение не распространяется (non-paged). В невытесняемой области размещаются модули ОС, требующие быстрой реакции и/или постоянного присутствия в памяти, например диспетчер потоков или код, который управляет заменой страниц памяти. Остальные модули ОС подвергаются страничному вытеснению, как и пользовательские сегменты.

Обычно аппаратура накладывает свои ограничения на порядок использования виртуального адресного пространства. Некоторые процессоры (например, MIPS) предусматривают для определенной области системной части адресного пространства особые правила отображения на физическую память. При этом *виртуальный адрес прямо отображается на физический* (последний либо полностью соответствует виртуальному адресу, либо равен его части). Такая особая область памяти не подвергается страничному вытеснению, и поскольку достаточно трудоемкая процедура преобразования адресов исключается, то доступ к располагаемым здесь кодам и данным осуществляется очень быстро.

Алгоритмы распределения памяти

Следует ли назначать каждому процессу одну непрерывную область физической памяти или можно выделять память «кусками»? Должны ли сегменты программы, загруженные в память, находиться на одном месте в течение всего периода выполнения процесса или можно их время от времени сдвигать? Что делать, если сегменты программы не помещается в имеющуюся память? Разные ОС по-разному отвечают на эти и другие базовые вопросы управления памятью. Далее будут рассмотрены наиболее общие подходы к распределению памяти,

которые были характерны для разных периодов развития операционных систем. Некоторые из них сохранили актуальность и широко используются в современных ОС, другие же представляют в основном только познавательный интерес, хотя их и сегодня можно встретить в специализированных системах.

На рис. 5.7 все алгоритмы распределения памяти разделены на два класса: алгоритмы, в которых используется перемещение сегментов процессов между оперативной памятью и диском, и алгоритмы, в которых внешняя память не привлекается.

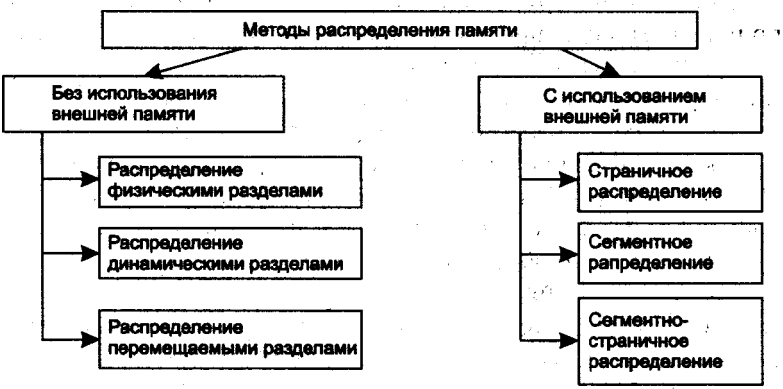


Рис. 5.7. Классификация методов распределения памяти

Фиксированные разделы

Простейший способ управления оперативной памятью состоит в том, что память разбивается на несколько областей фиксированной величины, называемых **разделами**. Такое разбиение может быть выполнено вручную оператором во время старта системы или во время ее установки. После этого границы разделов не изменяются.

Очередной новый процесс, поступивший на выполнение, помещается либо в общую очередь (рис. 5.8, *а*), либо в очередь к некоторому разделу (рис. 5.8, *б*).

При распределении памяти **фиксированными разделами** подсистема управления памятью решает следующие задачи.

- Сравнивает объем памяти, требуемый для вновь поступившего процесса, с размерами свободных разделов и выбирает подходящий раздел.
- Осуществляет загрузку программы в один из разделов и настройку адресов. Уже на этапе трансляции разработчик программы может задать раздел, в котором ее следует выполнять. Это позволяет сразу, без использования перемещающего загрузчика получить машинный код, настроенный на конкретную область памяти.

При очевидном преимуществе — простоте реализации, данный метод имеет существенный недостаток — *жесткость*. Так как в каждом разделе может выполняться только один процесс, то уровень мультипрограммирования заранее

ограничен числом разделов. Независимо от размера программы она будет занимать весь раздел. Так, например, в системе с тремя разделами невозможно выполнять одновременно более трех процессов, даже если им требуется совсем мало памяти. К тому же, разбиение памяти на разделы не позволяет выполнять процессы, программы которых не помещаются ни в один из разделов, но для которых было бы достаточно памяти нескольких разделов.

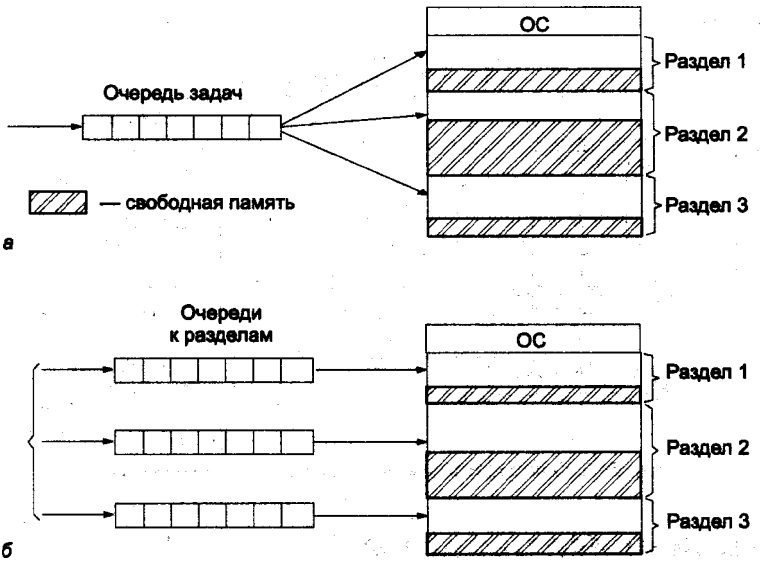


Рис. 5.8. Распределение памяти фиксированными разделами: с общей очередью (*а*), с отдельными очередями (*б*)

Такой способ управления памятью применялся в ранних мультипрограммных ОС. Однако и сейчас метод распределения памяти фиксированными разделами находит применение в системах реального времени в основном благодаря небольшим затратам на реализацию. Детерминированность вычислительного процесса систем реального времени (заранее известен набор выполняемых задач, их требования к памяти, а иногда и моменты запуска) компенсирует недостаточную гибкость данного способа управления памятью.

Динамические разделы

В следующем способе управления памятью память машины не делится заранее на разделы. Сначала вся память, отводимая для приложений, свободна. Каждому вновь поступающему на выполнение приложению на этапе создания процесса выделяется вся необходимая ему память (если достаточный объем памяти отсутствует, то приложение не принимается на выполнение и процесс для нее не создается). После завершения процесса память освобождается, и на это место может быть загружен другой процесс.

Таким образом, в произвольный момент времени оперативная память представляет собой случайную последовательность занятых и свободных участков (разделов) произвольного размера. На рис. 5.9 показано состояние памяти в различные моменты времени при динамическом распределении. Так, в момент t_0 в памяти находится только ОС, а к моменту t_1 память оказывается разделенной между 5 процессами, причем процесс П4, завершаясь, покидает память. На освободившееся от процесса П4 место загружается процесс П6, поступивший в момент t_3.

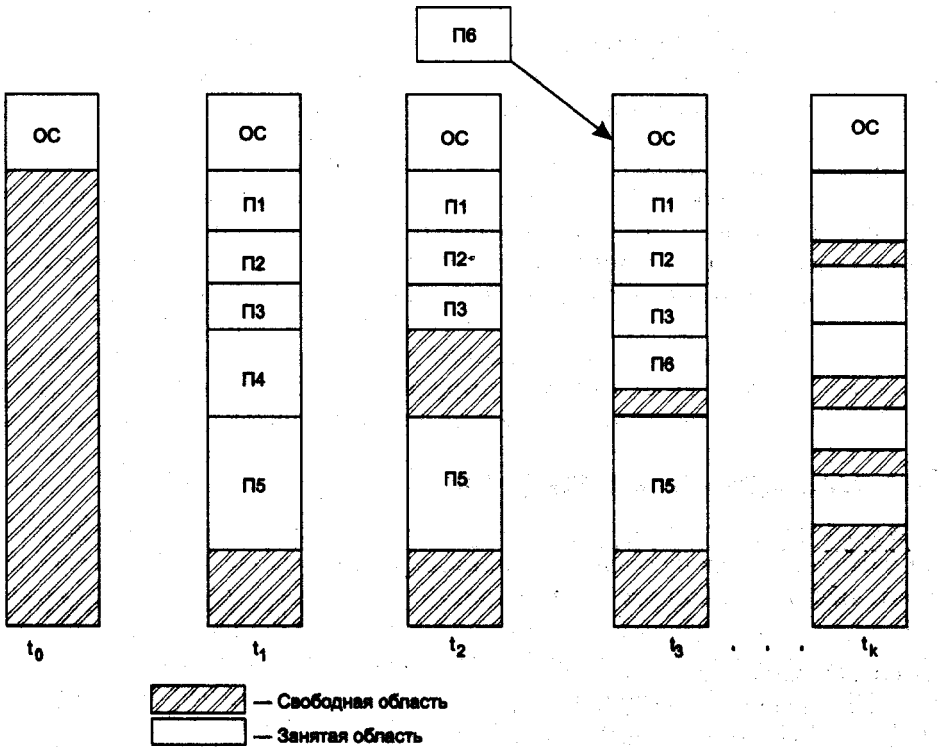


Рис. 5.9. Распределение памяти динамическими разделами

Распределение памяти **динамическими разделами** сводится к выполнению операционной системой следующих функций.

- Ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти.
- При создании нового процесса — анализ требований к памяти, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения кодов и данных нового процесса.
- Загрузка программы в выделенный ей раздел и корректировка таблиц свободных и занятых областей.

- После завершения процесса — корректировка таблиц свободных и занятых областей.

Выбор раздела для загрузки кодов процесса при его создании может осуществляться по разным правилам, например: «первый попавшийся раздел достаточного размера», «раздел, имеющий наименьший достаточный размер» или «раздел, имеющий наибольший достаточный размер».

Поскольку данный способ предполагает, что программный код не перемещается во время выполнения, то настройка адресов может быть проведена один раз во время загрузки.

По сравнению с методом распределения памяти фиксированными разделами данный метод обладает существенно большей гибкостью, но ему присущ очень серьезный недостаток — фрагментация памяти.

> **Фрагментация памяти** — это процесс появления большого числа несмежных участков свободной памяти маленького размера (**фрагментов**), таких, что ни одна из вновь поступающих программ не может поместиться ни в одном из участков, хотя суммарный объем фрагментов может составить значительную величину, намного превышающую требуемый объем памяти.

Распределение памяти динамическими разделами лежит в основе подсистем управления памятью многих мультипрограммных операционных системах 60–70-х годов, в частности такой популярной операционной системы, как OS/360.

Перемещаемые разделы

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших или младших адресов, так чтобы вся свободная память образовала единую область (рис. 5.10).

В дополнение к функциям, которые выполняет ОС при распределении памяти динамическими разделами, в случае **перемещаемых разделов** она должна еще время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется **сжатием**.

Сжатие может выполняться либо при каждом завершении процесса, либо только тогда, когда для вновь создаваемого процесса нет свободного раздела достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц свободных и занятых областей, во втором реже выполняется процедура сжатия.

Так как программы в ходе своего выполнения перемещаются по оперативной памяти, в данном случае невозможно выполнить настройку адресов с помощью перемещающего загрузчика. Здесь более подходящим оказывается динамическое преобразование адресов.

Хотя процедура сжатия и приводит к более эффективному использованию памяти, она может требовать *значительного времени*, что часто перевешивает преимущества данного метода.

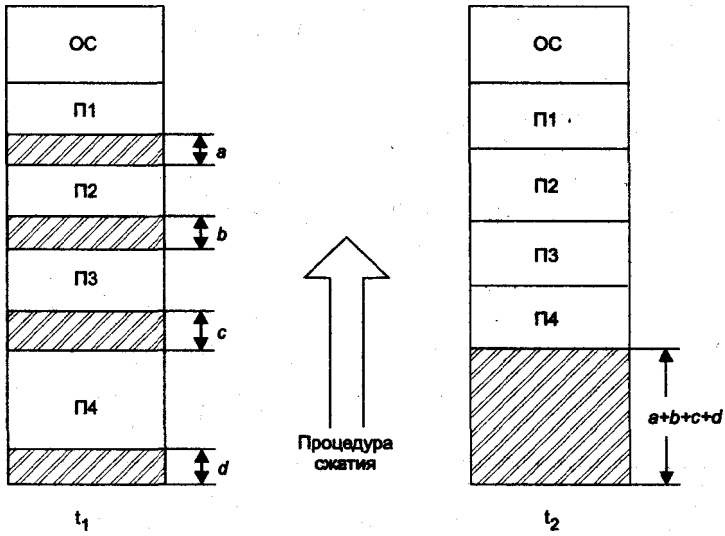


Рис. 5.10. Распределение памяти перемещаемыми разделами

Концепция сжатия применяется и при использовании других методов распределения памяти, когда отдельному процессу выделяется не одна сплошная область памяти, а несколько несмежных участков памяти произвольного размера (сегментов). Такой подход был характерен для ранних версий систем OS/2, в которых память распределялась сегментами, а возникавшая при этом фрагментация периодически устранялась путем перемещения сегментов.

Виртуальная память

Необходимым условием для того, чтобы программа могла выполняться, является ее нахождение в оперативной памяти. Только в этом случае процессор может извлекать команды из памяти и интерпретировать их, выполняя заданные действия. Объем оперативной памяти, который имеется в компьютере, существенно сказывается на характере протекания вычислительного процесса. Он ограничивает число одновременно выполняющихся программ и размеры их виртуальных адресных пространств. В некоторых случаях, когда все задачи мультипрограммной смеси являются вычислительными (то есть выполняют относительно мало операций ввода-вывода, разгружающих центральный процессор), для хорошей загрузки процессора может оказаться достаточным всего 3–5 задач. Однако если вычислительная система загружена выполнением интерактивных задач, то для эффективного использования процессора может потребоваться уже несколько десятков, а то и сотен задач. Эти рассуждения хорошо иллюстрирует рис. 5.11, на котором показан график зависимости коэффициента загрузки процессора от числа одновременно выполняемых процессов и доли времени, проводимого этими процессами в состоянии ожидания ввода-вывода.

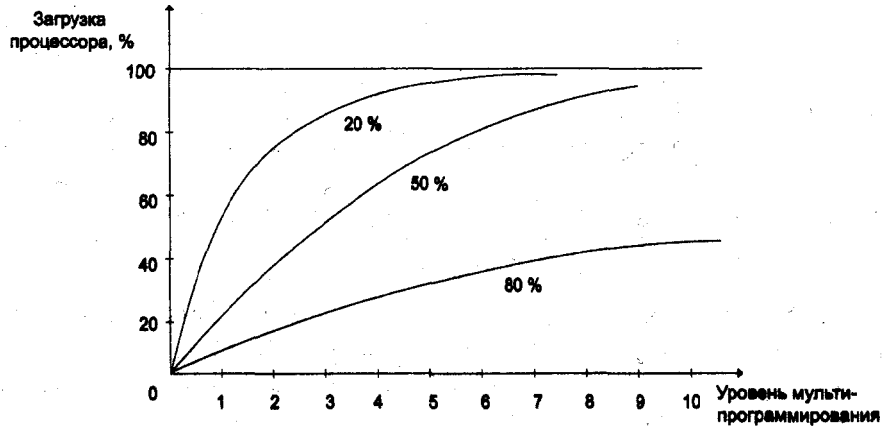


Рис. 5.11. Зависимость загрузки процессора от числа задач и интенсивности ввода-вывода

Большое количество задач, необходимое для высокой загрузки процессора, требует большого объема оперативной памяти. В условиях, когда для обеспечения приемлемого уровня мультипрограммирования имеющейся оперативной памяти недостаточно, был предложен метод организации вычислительного процесса, при котором образы некоторых процессов целиком или частично временно выгружаются на диск.

В мультипрограммном режиме помимо активного процесса, то есть процесса, коды которого в настоящий момент интерпретируются процессором, имеются приостановленные процессы, находящиеся в ожидании завершения ввода-вывода или освобождения ресурсов, а также процессы в состоянии готовности, стоящие в очереди к процессору. Образы таких неактивных процессов могут быть временно, до следующего цикла активности, выгружены на диск. Несмотря на то что коды и данные процесса отсутствуют в оперативной памяти, ОС «знает» о его существовании и в полной мере учитывает это при распределении процессорного времени и других системных ресурсов. К моменту, когда подходит очередь выполнения выгруженного процесса, его образ возвращается с диска в оперативную память. Если при этом обнаруживается, что свободного места в оперативной памяти не хватает, то на диск выгружается другой процесс.

Такая подмена (виртуализация) оперативной памяти дисковой памятью позволяет повысить уровень мультипрограммирования — объем оперативной памяти компьютера в этом случае не столь жестко ограничивает количество одновременно выполняемых процессов, поскольку суммарный объем памяти, занимаемой образами этих процессов, может существенно превосходить имеющийся объем оперативной памяти. Виртуальным называется ресурс, свойства которого с точки зрения пользователя или пользовательской программы отличаются от реальных свойств этого ресурса. В данном случае в распоряжение прикладного программиста предоставляется виртуальная оперативная память, размер которой намного превосходит всю имеющуюся в системе реальную оперативную память. Пользователь пишет программу, а транслятор, используя

виртуальные адреса, переводит ее в машинные коды так, как будто в распоряжении программы имеется однородная оперативная память большого объема. В действительности же все коды и данные, используемые программой, хранятся на дисках и только при необходимости загружаются в реальную оперативную память. Понятно, однако, что работа такой «оперативной памяти» происходит значительно медленнее.

Виртуализация оперативной памяти осуществляется совокупностью программных модулей ОС и аппаратных схем процессора и подразумевает решение следующих задач:

- размещение данных в запоминающих устройствах разного типа, например часть кодов программы — в оперативной памяти, а часть — на диске;
- выбор образов процессов или их частей для перемещения из оперативной памяти на диск и обратно;
- перемещение по мере необходимости данных между памятью и диском;
- преобразование виртуальных адресов в физические.

Очень важно то, что все действия по организации совместного использования диска и оперативной памяти — выделение места для перемещаемых фрагментов, настройка адресов, выбор кандидатов на загрузку и выгрузку — осуществляются операционной системой и аппаратурой процессора *автоматически*, без участия программиста и никак не сказываются на логике работы приложений.

ПРИМЕЧАНИЕ Уже достаточно давно пользователи столкнулись с проблемой размещения в памяти программы, размер которой превышает имеющуюся в наличии свободную память. Одним из первых решений было разбиение программы на части, называемые оверлеями. Когда первый оверлей заканчивал свое выполнение, он вызывал другой оверлей. Все оверлеи хранились на диске и перемещались между памятью и диском средствами операционной системы на основании явных директив программиста, содержащихся в программе. Этот способ, несмотря на внешнее сходство, имеет принципиальное отличие от виртуальной памяти, заключающееся в том, что разбиение программы на части и планирование их загрузки в оперативную память должны были выполняться заранее программистом во время написания программы.

Различают два достаточно близких подхода к виртуализации памяти:

- **свопинг** (swapping) — образы процессов выгружаются на диск и возвращаются в оперативную память *целиком*;
- **виртуальная память** (virtual memory) — между оперативной памятью и диском перемещаются *части* (сегменты, страницы и т. п.) образов процессов.

Свопинг представляет собой частный случай виртуальной памяти и, следовательно, — это более простой в реализации способ совместного использования оперативной памяти и диска. Однако ему свойственна избыточность: когда ОС решает активизировать процесс, для его выполнения, как правило, не требуется

загружать в оперативную память все его сегменты полностью — достаточно загрузить небольшую часть кодового сегмента с подлежащей выполнению инструкцией и частью сегментов данных, с которыми работает эта инструкция, а также отвести место под сегмент стека. Аналогично, при освобождении памяти для загрузки нового процесса очень часто вовсе не требуется выгружать другой процесс на диск целиком, достаточно вытеснить на диск только часть его образа. Перемещение избыточной информации замедляет работу системы, а также приводит к неэффективному использованию памяти. Кроме того, системы, поддерживающие свопинг, имеют еще один очень существенный недостаток: они не способны загрузить для выполнения процесс, виртуальное адресное пространство которого превышает имеющуюся в наличии свободную память.

Именно из-за указанных недостатков свопинг как основной механизм управления памятью почти не используется в современных ОС[1]. На смену ему пришел более совершенный механизм *виртуальной памяти*, который, как уже было сказано, заключается в том, что при нехватке места в оперативной памяти на диск выгружаются только части образов процессов.

Ключевой проблемой виртуальной памяти, возникающей в результате многократного изменения местоположения в оперативной памяти образов процессов или их частей, является преобразование виртуальных адресов в физические. Решение этой проблемы, в свою очередь, зависит от того, какой способ структуризации виртуального адресного пространства принят в данной системе управления памятью.

Все множество реализаций виртуальной памяти может быть представлено тремя классами

- В случае **страничной виртуальной памяти** перемещение данных между памятью и диском организуется страницами — частями виртуального адресного пространства фиксированного и сравнительно небольшого размера.
- **Сегментная виртуальная память** предусматривает перемещение данных сегментами — частями виртуального адресного пространства произвольного размера, полученными с учетом смыслового значения данных.
- В **сегментно-страничной виртуальной памяти** используется двухуровневое деление: виртуальное адресное пространство делится на сегменты, а затем сегменты делятся на страницы. Единицей перемещения данных здесь является страница. Этот способ управления памятью объединяет в себе элементы обоих предыдущих подходов.

Для временного хранения сегментов и страниц на диске отводится либо специальная область, либо специальный файл, которые во многих ОС по традиции продолжают называть областью, или файлом, свопинга, хотя перемещение информации между оперативной памятью и диском осуществляется уже не в фор-

[1] В некоторых современных ОС, например версиях Unix, основанных на коде SVR4, механизм свопинга используется как дополнительный к виртуальной памяти и включается только при серьезных перегрузках системы.

ме полного замещения одного процесса другим, а частями. Другое популярное название этой области — страничный файл (page file, или paging file). Текущий размер страничного файла является важным параметром, оказывающим влияние на возможности операционной системы: чем больше страничный файл, тем больше приложений может одновременно выполнять ОС (при фиксированном размере оперативной памяти). Однако необходимо понимать, что увеличение числа одновременно работающих приложений за счет увеличения размера страничного файла замедляет их работу, так как значительная часть времени при этом тратится на перекачку кодов и данных из оперативной памяти на диск и обратно. Размер страничного файла в современных ОС является настраиваемым параметром, который выбирается администратором системы для достижения компромисса между уровнем мультипрограммирования и быстродействием системы.

Страничное распределение

На рис. 5.12 показана схема страничного распределения памяти.

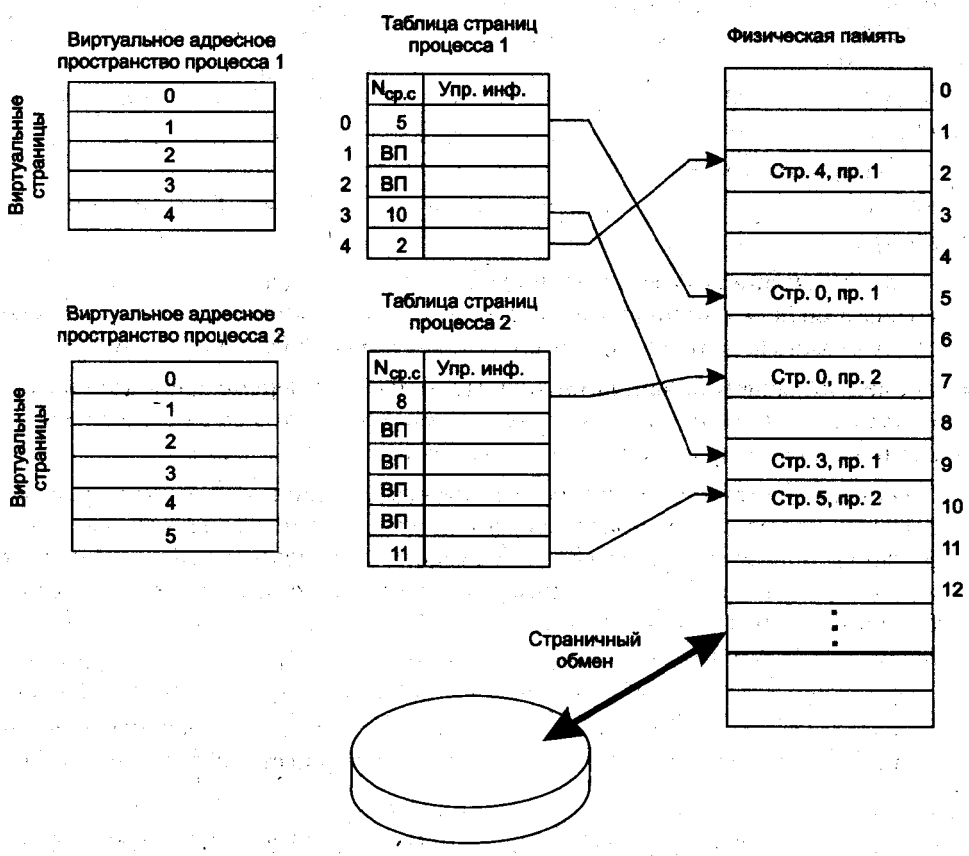


Рис. 5.12. Страничное распределение памяти

При страничном распределении памяти виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые **виртуальными страницами** (virtual pages).

В общем случае размер виртуального адресного пространства процесса не кратен размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

Вся оперативная память машины также делится на части такого же размера, называемые **физическими страницами** (или блоками, или кадрами).

Размер страницы выбирается равным степени двойки: 512, 1024, 4096 байт и т. д. Это позволяет упростить механизм преобразования адресов.

При создании процесса ОС загружает в оперативную память несколько его виртуальных страниц (начальные страницы кодового сегмента и сегмента данных). Копия всего виртуального адресного пространства процесса находится на диске. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах.

Для каждого процесса операционная система создает **таблицу страниц** — информационную структуру, содержащую записи обо всех виртуальных страницах процесса.

Дескриптор страницы — это отдельная запись таблицы. Она включает следующую информацию:

- номер физической страницы, в которую загружена данная виртуальная страница;
- признак присутствия, устанавливаемый в единицу, если виртуальная страница находится в оперативной памяти;
- признак модификации страницы, который устанавливается в единицу всякий раз, когда производится запись по адресу, относящемуся к данной странице;
- признак обращения к странице, называемый также **битом доступа**, который устанавливается в единицу при каждом обращении по адресу, относящемуся к данной странице.

Признаки присутствия, модификации и обращения в большинстве моделей современных процессоров устанавливаются схемами процессора аппаратно при выполнении операции с памятью. Информация из таблиц страниц используется для решения вопроса о необходимости перемещения той или иной страницы между памятью и диском, а также для преобразования виртуального адреса в физический. Сами таблицы страниц, так же как и описываемые ими страницы, размещаются в оперативной памяти. Адрес таблицы страниц включается в контекст соответствующего процесса. При активизации очередного процесса операционная система загружает адрес его таблицы страниц в специальный регистр процессора.

При каждом обращении к памяти выполняется поиск номера виртуальной страницы, содержащей требуемый адрес, затем по этому номеру определяется нужный элемент таблицы страниц и из него извлекается описывающая страницу

информация[1]. Далее анализируется признак присутствия, и если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический, то есть виртуальный адрес заменяется указанным в записи таблицы физическим адресом. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое **страничное прерывание**. Выполняющийся процесс переводится в состояние ожидания и активизируется другой процесс из очереди процессов, находящихся в состоянии готовности. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу (для этого операционная система должна помнить положение вытесненной страницы в страничном файле диска) и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то на основании принятой в данной системе стратегии замещения страниц решается вопрос о том, какую страницу следует выгрузить из оперативной памяти.

После того как выбрана страница, которая должна покинуть оперативную память, обнуляется ее бит присутствия и анализируется ее признак модификации. Если выталкиваемая страница за время последнего пребывания в оперативной памяти была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то принимается во внимание, что на диске уже имеется предыдущая копия этой виртуальной страницы, и никакой записи на диск не производится. Физическая страница объявляется свободной. Из соображений безопасности в некоторых системах освобождаемая страница обнуляется, с тем чтобы невозможно было использовать содержимое выгруженной страницы.

Для хранения информации о положении вытесненной страницы в страничном файле ОС может использовать поля таблицы страниц или же другую системную структуру данных (например, дескриптор сегмента при сегментно-страничной организации виртуальной памяти).

Виртуальный адрес при страничном распределении может быть представлен в виде пары (p, s_v), где p — порядковый номер виртуальной страницы процесса (нумерация страниц начинается с 0), а s_v — смещение в пределах виртуальной страницы. Физический адрес также может быть представлен в виде пары (n, s_f), где n — номер физической страницы, а s_f — смещение в пределах физической страницы. Задача подсистемы виртуальной памяти состоит в отображении пары (p, s_v) на (n, s_f).

Прежде чем приступить к рассмотрению схемы преобразования виртуального адреса в физический, остановимся на двух базисных свойствах страничной организации.

Первое из них состоит в том, что объем страницы выбирается равным степени двойки — 2^k. Из этого следует, что смещение s может быть получено простым отделением k младших разрядов в двоичной записи адреса, а оставшиеся

[1] Здесь не учитывается возможность кэширования записей из таблицы страниц, которая рассматривается несколько позже.

старшие разряды адреса представляют собой двоичную запись номера страницы (при этом неважно, является страница виртуальной или физической). Например, если размер страницы составляет 1 Кбайт (2^{10}), то из двоичной записи адреса $5071_8 = 101\ 000\ 111\ 001_2$ можно определить, что он принадлежит странице, номер которой в двоичном выражении равен 10_2 и смещен относительно ее начала на $1\ 000\ 111\ 001_2$ байт (рис. 5.13).

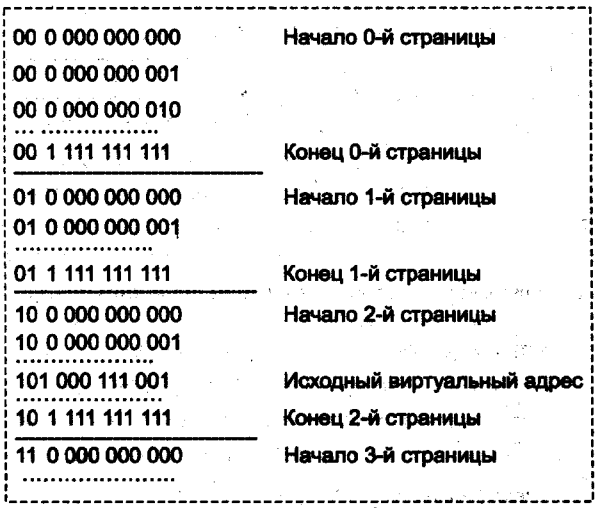


Рис. 5.13. Двоичное представление адресов

Из рисунка хорошо видно, что номер страницы и ее начальный адрес легко могут быть получены один из другого дополнением или отбрасыванием k нулей, соответствующих смещению. Именно по этой причине часто говорят, что таблица страниц содержит начальный физический адрес страницы в памяти (а не номер физической страницы), хотя на самом деле в таблице указаны только старшие разряды адреса. Начальный адрес страницы называется **базовым адресом**.

Второе свойство заключается в том, что в пределах страницы непрерывная последовательность виртуальных адресов однозначно отображается на непрерывную последовательность физических адресов, а, значит, смещения в виртуальном и физическом адресах s_v и s_f равны между собой (рис. 5.14.).

Отсюда следует простая схема преобразования виртуального адреса в физический (рис. 5.15). Младшие разряды физического адреса, соответствующие смещению, получаются переносом такого же количества младших разрядов из виртуального адреса. Старшие разряды физического адреса, соответствующие номеру физической страницы, определяются из таблицы страниц, в которой указывается соответствие виртуальных и физических страниц.

Итак, пусть произошло обращение к памяти по некоторому виртуальному адресу. Аппаратными схемами процессора выполняются следующие действия.

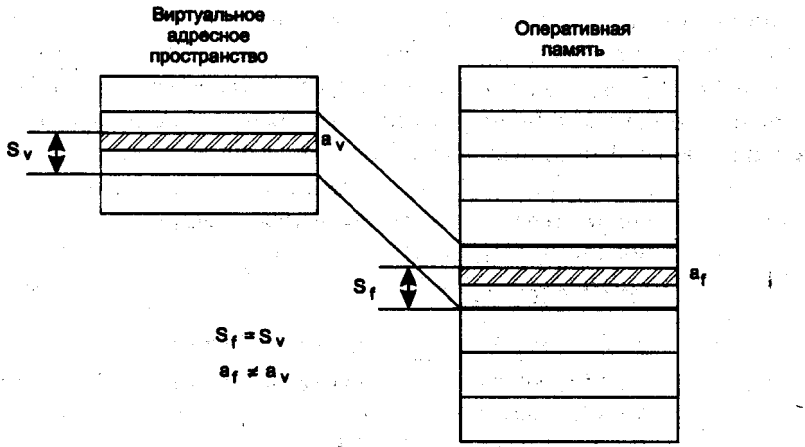


Рис. 5.14. При отображении виртуального адреса на физический смещение не меняется

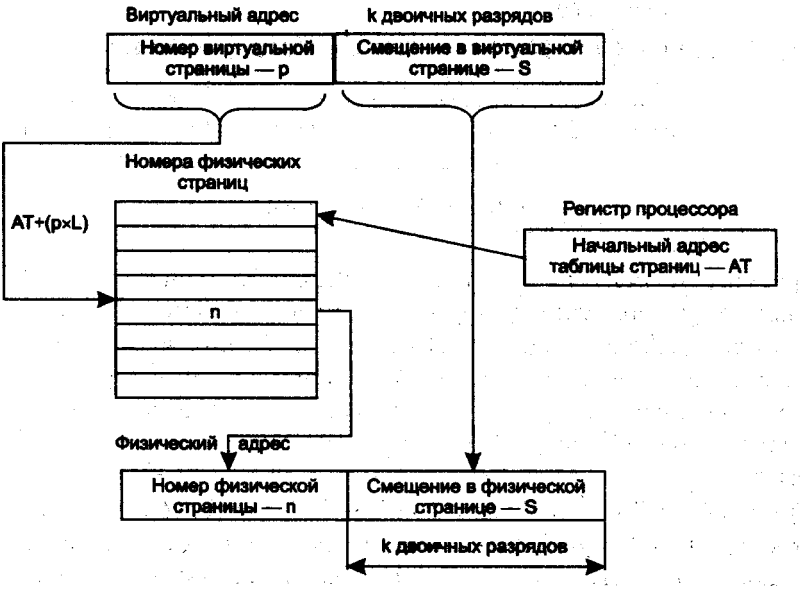


Рис. 5.15. Схема преобразования виртуального адреса в физический при страничной организации памяти

1. Из специального регистра процессора извлекается адрес AT таблицы страниц активного процесса. На основании начального адреса таблицы страниц, номера виртуальной страницы p (старшие разряды виртуального адреса) и длины отдельной записи в таблице страниц L (системная константа) определяется адрес нужного дескриптора в таблице страниц: $a = \text{AT} + (p \times L)$.
2. Из этого дескриптора извлекается номер соответствующей физической страницы — n.

3. К номеру физической страницы присоединяется смещение *s* (младшие разряды виртуального адреса).

Страничное распределение памяти может быть реализовано в упрощенном варианте, *без выгрузки страниц* на диск. В этом случае все виртуальные страницы всех процессов постоянно находятся в оперативной памяти. Такой вариант страничной организации, хотя и не предоставляет пользователю преимуществ работы с виртуальной памятью большого объема, сохраняет другое достоинство страничной организации — позволяет успешно бороться с фрагментацией физической памяти. Действительно, во-первых, программу можно разбить на части и загрузить в разрозненные участки свободной памяти, во-вторых, при загрузке виртуальных страниц никогда не образуется неиспользуемых остатков, так как размеры виртуальных и физических страниц совпадают. Такой режим работы системы управления памятью используется в некоторых специализированных ОС, когда требуется высокая реактивность системы и способность выполнять переменный набор приложений (пример — ОС семейства Novell NetWare 3.x и 4.x).

Оптимизация страничной виртуальной памяти

При выполнении типичной машинной команды происходит в среднем 3–4 обращения к памяти (выборка команды, извлечение операндов, запись результата). В системе с виртуальной страничной памятью при каждом из этих обращений необходимо выполнить либо преобразование виртуального адреса в физический, либо обработку страничного прерывания. Время выполнения этих операций в значительной степени влияет на общую производительность вычислительной системы, поэтому столь большое внимание разработчиков уделяется оптимизации виртуальной памяти.

Накладные расходы системы на поддержание страничной виртуальной памяти могут быть уменьшены за счет:

- *аппаратного способа* преобразования виртуального адреса в физический;
- *снижения частоты страничных прерываний* путем оптимизации размера страницы, а также выбора страниц на загрузку/выгрузку.

Эффективность аппаратного получения физического адреса по виртуальному может быть повышена, если размер страницы выбрать равным степени двойки. В этом случае двоичная запись адреса легко разделяется на номер страницы и смещение, а значит, в процедуре преобразования адресов более длительную операцию сложения можно заменить операцией присоединения (конкатенации). Применяются и другие способы ускорения преобразования, такие, например, как кэширование таблицы страниц — хранение наиболее активно используемых записей в быстродействующих запоминающих устройствах, в частности в регистрах процессора.

Другим важным фактором, влияющим на производительность системы, является частота страничных прерываний, на которую, в свою очередь, влияют размер страницы и принятые в данной системе правила выбора страниц для вы-

грузки и загрузки. При неправильно выбранной стратегии замещения страниц могут возникать ситуации, когда система тратит большую часть времени впустую на подкачку страниц из оперативной памяти на диск и обратно.

При выборе страницы на выгрузку могут быть использованы различные критерии, смысл которых сводится к одному: на диск выталкивается страница, к которой в будущем, начиная с данного момента, дольше всего не будет обращений. Поскольку точно предсказать ход вычислительного процесса невозможно, то невозможно точно определить страницу, подлежащую выгрузке. В таких условиях решение принимается на основе неких эмпирических критериев, часто основывающихся на предположении об инерционности вычислительного процесса. Так, например, из того, что страница не использовалась долгое время, делается вывод о том, что она, скорее всего, не будет использоваться и в ближайшее время. Однако привлечение критериев такого рода не исключает ситуаций, когда сразу после выгрузки страницы к ней происходит обращение и ее снова приходится загружать в память. Вероятность таких «напрасных» перемещений настолько велика, что в некоторых реализациях виртуальной памяти вообще отказываются от количественных критериев и предпочитают случайный выбор, при котором на диск выгружается первая попавшаяся страница. Возникающее при этом некоторое увеличение интенсивности страничного обмена компенсируется снижением вычислительных затрат на поддержание и анализ критерия выборки страниц на выгрузку.

Наиболее популярным критерием выбора страницы на выгрузку является число обращений к ней за последний период времени. Вычисление этого критерия происходит следующим образом. Операционная система ведет для каждой страницы программный счетчик. Значения счетчиков определяются значениями признаков доступа. Всякий раз, когда происходит обращение к какой-либо странице, процессор устанавливает в единицу признак доступа в относящейся к данной странице записи таблицы страниц. ОС периодически просматривает признаки доступа всех страниц во всех существующих в данный момент записях таблицы страниц. Если какой-либо признак оказывается равным 1 (было обращение), то система сбрасывает его в 0, увеличивая при этом на единицу значение связанного с этой страницей счетчика обращений. Когда возникает необходимость удалить какую-либо страницу из памяти, ОС находит страницу, счетчик обращений которой имеет наименьшее значение. Для того чтобы критерий учитывал интенсивность обращений за последний период, ОС с соответствующей периодичностью обнуляет все счетчики.

Интенсивность страничного обмена может быть также снижена в результате так называемой **упреждающей загрузки**, в соответствие с которой при возникновении страничного прерывания в память загружается не одна страница, содержащая адрес обращения, а сразу несколько прилегающих к ней страниц. Здесь используется эмпирическое правило: *если обращение произошло по некоторому адресу, то велика вероятность того, что следующие обращения произойдут по соседним адресам.*

Резервом повышения производительности системы является также правильный выбор размера страницы. Каким же должен быть оптимальный размер страницы? С одной стороны, чтобы уменьшить частоту страничных прерываний, следовало бы увеличивать размер страницы. С другой стороны, если страница велика, то велика и фиктивная область в последней виртуальной странице каждого процесса. Если учесть, что в среднем в каждом процессе фиктивная область составляет половину страницы, то в сумме при большом объеме страницы потери могут составить существенную величину. Из приведенных соображений следует, что выбор размера страницы является сложной оптимизационной задачей, требующей учета многих факторов. На практике же разработчики ОС и процессоров ограничиваются неким рациональным решением, пригодным для широкого класса вычислительных систем. Типичный размер страницы составляет несколько килобайтов, например, наиболее распространенные процессоры Pentium компании Intel, а также операционные системы, устанавливаемые на компьютерах с этими процессорами, поддерживают страницы размером 4096 байт (4 Кбайт)[1].

Размер страницы влияет также на количество записей в таблицах страниц. Чем меньше страница, тем более объемными являются таблицы страниц процессов и тем больше места они занимают в памяти. Учитывая, что в современных процессорах максимальный объем виртуального адресного пространства процесса, как правило, не меньше 4 Гбайт (2^{32}), то при размере страницы 4 Кбайт (2^{12}) и длине записи 4 байт для хранения таблицы страниц может потребоваться 4 Мбайт памяти!

Выходом в такой ситуации является хранение в памяти только той части таблицы страниц, которая активно используется в данный период времени — так как сама таблица страниц хранится в таких же страницах физической памяти, что и описываемые ею страницы, то принципиально возможно временно вытеснять часть таблицы страниц из оперативной памяти. Именно такой результат может быть достигнут путем более сложной структуризации виртуального адресного пространства, который рассматривается в следующем разделе.

Двухуровневое страничное распределение памяти

При двухуровневом страничном распределении памяти множество виртуальных адресов процесса делится на разделы, а разделы делятся на страницы (рис. 5.16). Все страницы имеют одинаковый размер, а разделы содержат одинаковое количество страниц. Если размер страницы и количество страниц в разделе выбрать равными степени двойки (2^k и 2^n соответственно), то принадлежность виртуального адреса к разделу и странице, а также смещение внутри страницы можно определить очень просто: младшие k двоичных разрядов дают

[1] Процессор Pentium позволяет использовать также страницы размером до 4 Мбайт одновременно со страницами объемом 4 Кбайта.

смещение, следующие n разрядов представляют собой номер виртуальной страницы, а оставшиеся старшие разряды (обозначим их количество через m) содержат номер раздела.

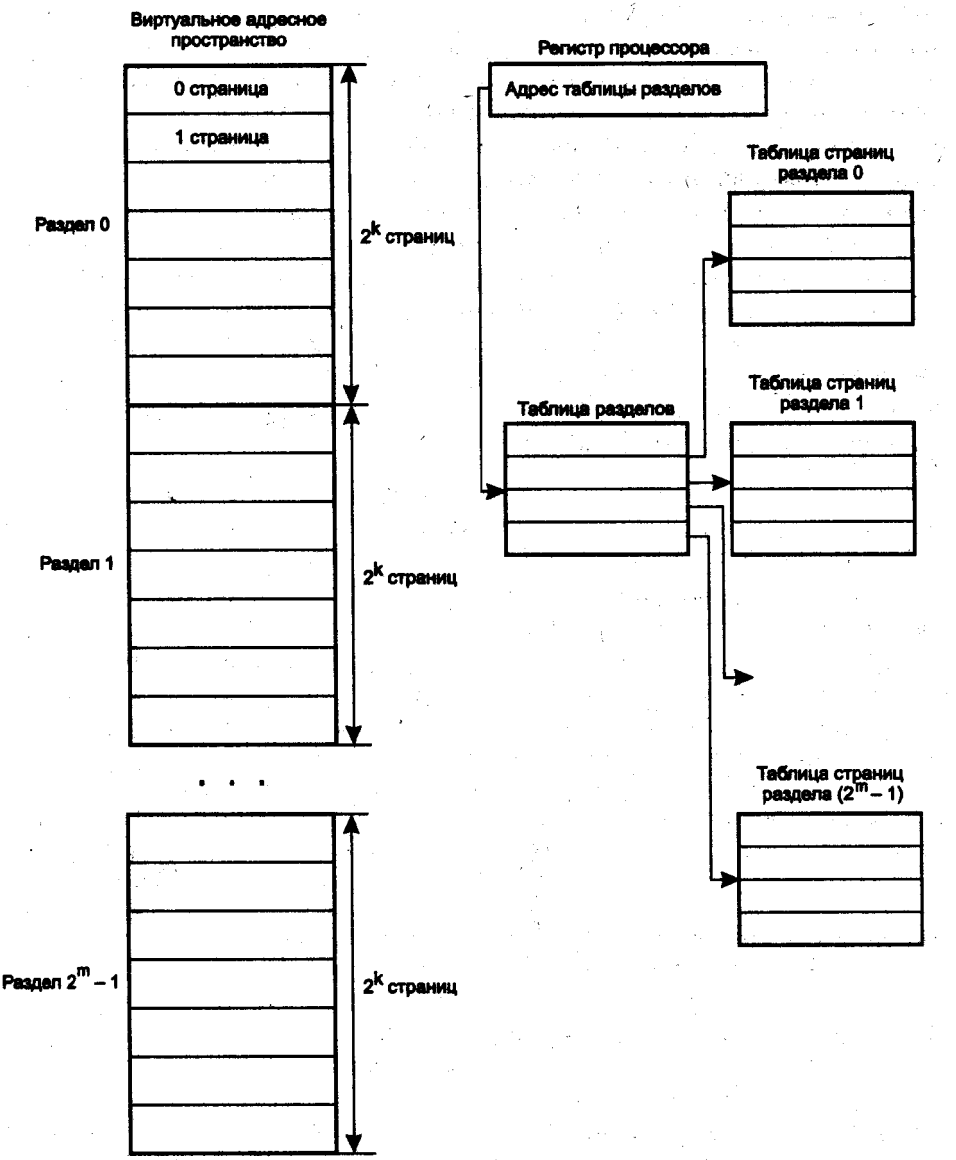


Рис. 5.16. Структура виртуального адресного пространства с разделами

Для каждого раздела строится собственная таблица страниц. Количество дескрипторов в таблице и их размер подбираются такими, чтобы объем таблицы оказался равным объему страницы. Например, в процессоре Pentium при раз-

мере страницы 4 Кбайт длина дескриптора страницы составляет 4 байта, и количество записей в таблице страниц, помещающейся на странице, равняется соответственно 1024. Каждая таблица страниц описывается дескриптором, структура которого полностью совпадает со структурой дескриптора обычной страницы. Эти дескрипторы сведены в таблицу разделов, называемую также каталогом страниц. Физический адрес таблицы разделов активного процесса содержится в специальном регистре процессора и поэтому всегда известен операционной системе. Страница, содержащая таблицу разделов, *никогда не выгружается из памяти*, в противном случае работа виртуальной памяти была бы невозможна.

Выгрузка страниц с таблицами страниц позволяет сэкономить память, но при этом приводит к дополнительным временным затратам при получении физического адреса. Действительно, может случиться так, что та таблица страниц, которая содержит нужный дескриптор, в данный момент выгружена на диск, тогда процесс преобразования адреса приостанавливается до тех пор, пока требуемая страница не будет снова загружена в память. Для снижения вероятности отсутствия страницы в памяти используются различные приемы, основным из которых является кэширование.

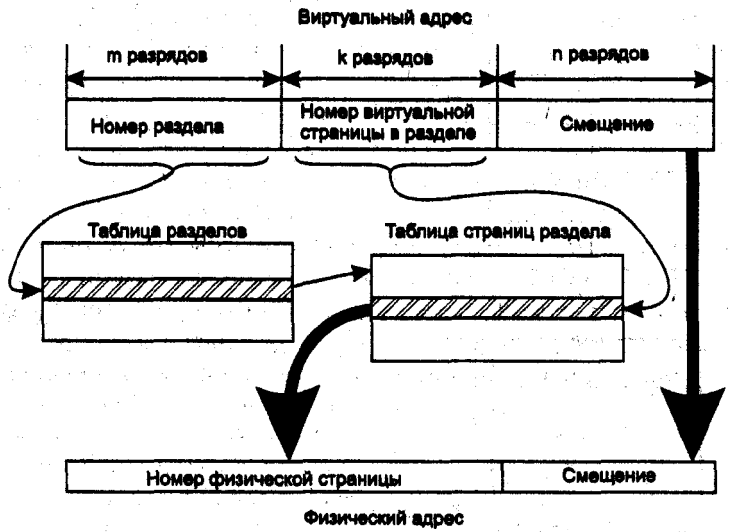


Рис. 5.17. Схема преобразования виртуального адреса при двухуровневой структуризации адресного пространства

Проследим более подробно схему преобразования адресов для случая двухуровневой структуризации виртуального адресного пространства (рис. 5.17).

1. Путем отбрасывания $k + n$ младших разрядов в виртуальном адресе определяется номер раздела, которому принадлежит данный виртуальный адрес.
2. По этому номеру из таблицы разделов извлекается дескриптор соответствующей таблицы страниц. Проверяется, находится ли данная таблица стра-

ниц в памяти. Если нет, происходит страничное прерывание, и система загружает нужную страницу с диска.

3. Далее из этой таблицы страниц извлекается дескриптор виртуальной страницы, номер которой содержится в средних *n* разрядах преобразуемого виртуального адреса. Снова выполняется проверка наличия данной страницы в памяти и при необходимости ее загрузка.
4. Из дескриптора определяется номер (базовый адрес) физической страницы, в которую загружена данная виртуальная страница. К номеру физической страницы пристыковывается смещение, взятое из *k* младших разрядов виртуального адреса. В результате получается искомый физический адрес.

Сегментное распределение

При страничной организации виртуальное адресное пространство процесса делится на равные части *механически*, без учета смыслового значения данных. В одной странице могут оказаться и коды команд, и инициализируемые переменные, и массив исходных данных программы. Такой подход не позволяет обеспечить дифференцированный доступ к разным частям программы, а это свойство могло бы быть очень полезным во многих случаях. Например, можно было бы запретить обращаться с операциями записи в сегмент программы, содержащий коды команд, разрешив эту операцию для сегментов данных.

Кроме того, разбиение виртуального адресного пространства на *«осмысленные»* части делает принципиально возможным совместное использование фрагментов программ разными процессами. Пусть, например, двум процессам требуется одна и та же подпрограмма, которая к тому же обладает свойством реентерабельности[1]. Тогда коды этой подпрограммы могут быть оформлены в виде отдельного сегмента и включены в виртуальные адресные пространства обоих процессов. При отображении на физическую память сегменты, содержащие коды подпрограммы из обоих виртуальных пространств, проецируются на одну и ту же область физической памяти. Таким образом, оба процесса получают доступ к одной и той же копии подпрограммы (рис. 5.18).

> При **сегментной организации памяти** виртуальное адресное пространство процесса делится на части — сегменты, размер которых определяется с учетом смыслового значения содержащейся в них информации.

Отдельный сегмент может представлять собой подпрограмму, массив данных и т. п. Деление виртуального адресного пространства на сегменты осуществляется компилятором на основе указаний программиста или по умолчанию, в соответствии с принятыми в системе соглашениями. Максимальный размер сегмента определяется разрядностью виртуального адреса, например, при 32-разряд-

[1] Реентерабельность (reentrantable) — свойство повторной входимости кода, которое позволяет одновременно использовать его несколькими процессами. При выполнении реентерабельного кода процессы не изменяют его, поэтому в память достаточно загрузить только одну копию кода.

ной организации процессора он равен 4 Гбайт. При этом максимально возможное виртуальное адресное пространство процесса представляет собой набор из N виртуальных сегментов, каждый размером по 4 Гбайт. В каждом сегменте виртуальные адреса находятся в диапазоне от 00000000_{16} до $FFFFFFFF_{16}$. Сегменты не упорядочиваются друг относительно друга, так что общего для сегментов линейного виртуального адреса не существует, виртуальный адрес задается парой чисел: номером сегмента и линейным виртуальным адресом внутри сегмента.

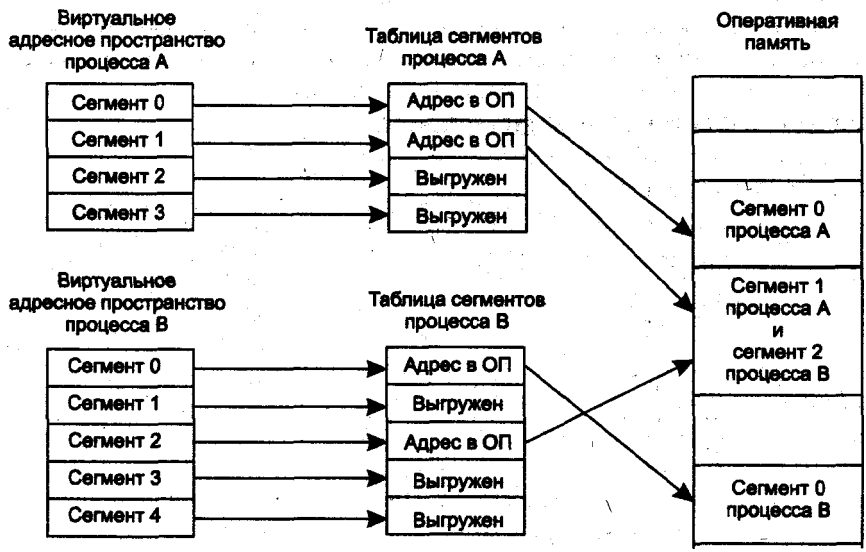


Рис. 5.18. Распределение памяти сегментами

При загрузке процесса в оперативную память помещается только часть его сегментов, полная копия виртуального адресного пространства находится в дисковой памяти. Для каждого загружаемого сегмента операционная система подыскивает непрерывный участок свободной памяти достаточного размера. Смежные в виртуальной памяти сегменты одного процесса могут занимать в оперативной памяти несмежные участки. Если во время выполнения процесса происходит обращение по виртуальному адресу, относящемуся к сегменту, который в данный момент отсутствует в памяти, то происходит прерывание. ОС приостанавливает активный процесс, запускает на выполнение следующий процесс из очереди, а параллельно организует загрузку нужного сегмента с диска. При отсутствии в памяти места, необходимого для загрузки сегмента, операционная система выбирает сегмент на выгрузку, при этом она использует критерии, аналогичные рассмотренным ранее критериям выбора страниц при страничном способе управления памятью.

На этапе создания процесса во время загрузки его образа в оперативную память система создает таблицу сегментов процесса (аналогичную таблице страниц).

Каждая запись в **таблице сегментов** содержит следующие характеристики соответствующего сегмента:

- базовый физический адрес сегмента в оперативной памяти;
- размер сегмента;
- правила доступа к сегменту;
- признаки модификации, присутствия и обращения к данному сегменту, а также некоторая другая информация.

Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

Как видно, сегментное распределение памяти имеет очень много общего со страничным распределением.

Механизмы преобразования адресов этих двух способов управления памятью тоже весьма схожи, однако в них имеются и существенные отличия, которые являются следствием того, что сегменты, в отличие от страниц, имеют произвольный размер. Виртуальный адрес при сегментной организации памяти может быть представлен парой (g, s), где g — номер сегмента, а s — смещение в сегменте. Физический адрес получается путем *сложения* базового адреса сегмента, который определяется по номеру сегмента g из таблицы сегментов и смещения s (рис. 5.19).

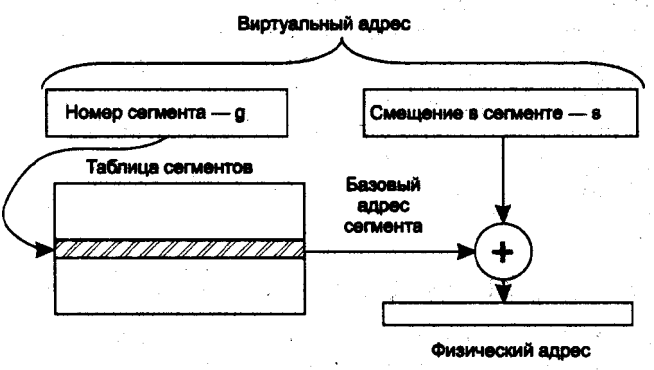


Рис. 5.19. Преобразование виртуального адреса при сегментной организации памяти

В данном случае нельзя обойтись операцией конкатенации, как это делается при страничной организации памяти. Действительно, поскольку размер страницы равен степени двойки, то в двоичном виде он выражается числом с несколькими нулями в младших разрядах. Страницы имеют одинаковый размер, а, значит, их начальные адреса кратны размеру страниц и выражаются также числами с нулями в младших разрядах. Именно поэтому ОС заносит в таблицы страниц не полные адреса, а номера физических страниц, которые совпадают со старшими разрядами базовых адресов. Сегмент же может в общем случае располагаться

в физической памяти, начиная с любого адреса, следовательно, для определения местоположения в памяти необходимо задать его полный начальный физический адрес. Использование операции сложения вместо конкатенации *замедляет* процедуру преобразования виртуального адреса в физический по сравнению со страничной организацией.

Другим недостатком сегментного распределения является *избыточность*. При сегментной организации единицей перемещения между памятью и диском является сегмент, имеющий в общем случае объем больший, чем страница. Однако во многих случаях для работы программы вовсе не требуется загружать весь сегмент целиком, достаточно было бы одной или двух страниц. Аналогично, при отсутствии свободного места в памяти не стоит выгружать целый сегмент, когда можно обойтись выгрузкой нескольких страниц.

Однако главный недостаток сегментного распределения — это *фрагментация*, которая возникает из-за непредсказуемости размеров сегментов. В процессе работы системы в памяти образуются небольшие участки свободной памяти, в которые не может быть загружен ни один сегмент. Суммарный объем, занимаемый фрагментами, может составить существенную часть общей памяти системы, приводя к ее неэффективному использованию.

Система с сегментной организацией функционирует аналогично системе со страничной организацией: при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический, время от времени происходят прерывания, связанные с отсутствием нужных сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются.

Одним из существенных отличий сегментной организации памяти от страничной является *возможность задания дифференцированных прав доступа* процесса к его сегментам. Например, один сегмент данных, содержащий исходную информацию для приложения, может иметь права доступа «только чтение», а сегмент данных, представляющий результаты, — «чтение и запись». Это свойство означает принципиальное преимущество сегментной модели памяти над страничной.

Сегментно-страничное распределение

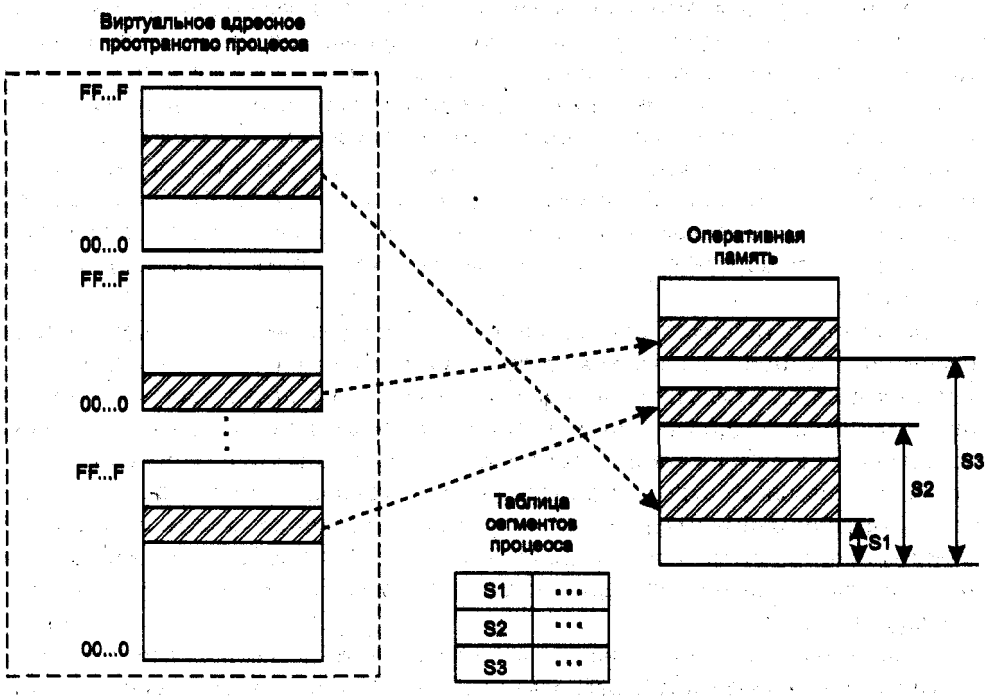
Сегментно-страничное распределение представляет собой комбинацию страничного и сегментного механизмов управления памятью и направлен на реализацию достоинств обоих подходов.

Так же как и при сегментной организации памяти, виртуальное адресное пространство процесса делится на сегменты. Это позволяет определять разные права доступа к разным частям кодов и данных программы.

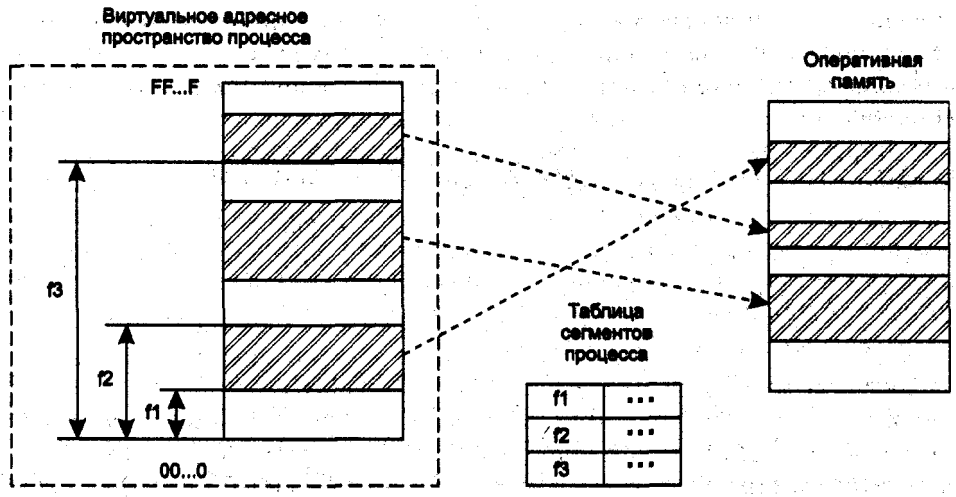
Перемещение данных между памятью и диском осуществляется не сегментами, а страницами. Для этого каждый виртуальный сегмент и физическая память делятся на страницы равного размера, что позволяет более эффективно использовать память, сократив до минимума фрагментацию.

В большинстве современных реализаций сегментно-страничной организации памяти, в отличие от набора виртуальных диапазонов адресов при сегмент-

ной организации памяти (рис. 5.20, *а*), все виртуальные сегменты образуют одно непрерывное линейное виртуальное адресное пространство (рис. 5.20, *б*).



а



б

Рис. 5.20. Два варианта сегментации

Координаты байта в виртуальном адресном пространстве при сегментно-страничной организации можно задать двумя способами. Во-первых, линейным виртуальным адресом, который равен сдвигу данного байта относительно границы общего линейного виртуального пространства, во-вторых, — парой чисел, одно из которых является номером сегмента, а другое — смещением относительно начала сегмента. При этом в отличие от сегментной модели для однозначного задания виртуального адреса вторым способом необходимо каким-то образом указать также начальный виртуальный адрес сегмента с данным номером. Системы виртуальной памяти ОС с сегментно-страничной организацией используют второй способ, так как он позволяет непосредственно определить принадлежность адреса некоторому сегменту и проверить права доступа процесса к нему.

Для каждого процесса операционная система создает отдельную таблицу сегментов, в которой содержатся описатели (дескрипторы) всех сегментов процесса. Описание сегмента включает назначенные ему права доступа и другие характеристики, подобные тем, которые содержатся в дескрипторах сегментов при сегментной организации памяти. Однако имеется и принципиальное отличие. В поле базового адреса указывается не начальный физический адрес сегмента, отведенный ему в результате загрузки в оперативную память, а начальный линейный виртуальный адрес сегмента в пространстве виртуальных адресов (на рис. 5.20 базовые физические адреса обозначены *S*1, *S*2, *S*3, а базовые виртуальные адреса — *f*1, *f*2, *f*3).

Наличие базового виртуального адреса сегмента в дескрипторе позволяет однозначно преобразовать адрес, заданный в виде пары (номер сегмента, смещение в сегменте), в линейный виртуальный адрес байта, который затем преобразуется в физический адрес страничным механизмом.

Деление общего линейного виртуального адресного пространства процесса и физической памяти на страницы осуществляется так же, как это делается при страничной организации памяти. Размер страниц выбирается равным степени двойки, что упрощает механизм преобразования виртуальных адресов в физические. Виртуальные страницы нумеруются в пределах виртуального адресного пространства каждого процесса, а физические страницы — в пределах оперативной памяти. При создании процесса в память загружается только часть страниц, остальные загружаются по мере необходимости. Время от времени система выгружает уже ненужные страницы, освобождая память для новых страниц. ОС ведет для каждого процесса таблицу страниц, в которой указывается соответствие виртуальных страниц физическим.

Базовые адреса таблиц сегментов и страниц процесса являются частью его контекста. При активизации процесса эти адреса загружаются в специальные регистры процессора и используются механизмом преобразования адресов.

Преобразование виртуального адреса в физический происходит в два этапа (рис. 5.21).

1. На первом этапе работает *механизм сегментации*. Исходный виртуальный адрес, заданный в виде пары (номер сегмента, смещение), преобразуется

в линейный виртуальный адрес. Для этого на основании базового адреса таблицы сегментов и номера сегмента вычисляется адрес дескриптора сегмента. Анализируются поля дескриптора и выполняется проверка возможности выполнения заданной операции. Если доступ к сегменту разрешен, то вычисляется линейный виртуальный адрес путем сложения базового адреса сегмента, извлеченного из дескриптора, и смещения, заданного в исходном виртуальном адресе.

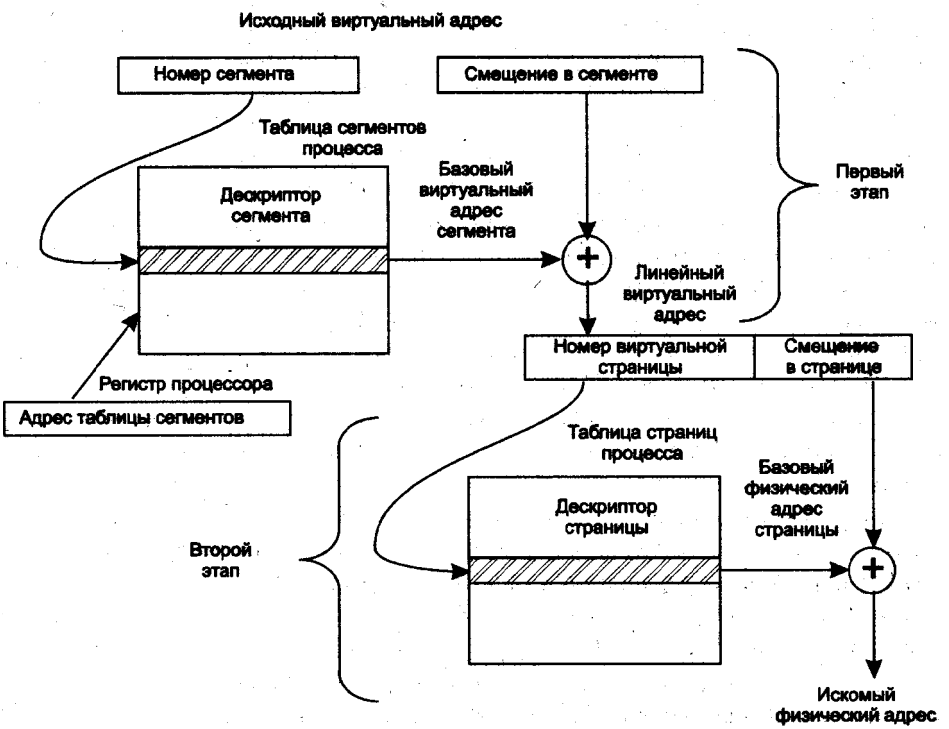


Рис. 5.21. Преобразование виртуального адреса в физический при сегментно-страничной организации памяти

2. На втором этапе работает *страничный механизм*. Полученный линейный виртуальный адрес преобразуется в искомый физический адрес. В результате преобразования линейный виртуальный адрес представляется в том виде, в котором он используется при страничной организации памяти, а именно в виде пары (номер страницы, смещение в странице). Благодаря тому, что размер страницы выбран равным степени двойки, эта задача решается простым отделением некоторого количества младших двоичных разрядов. В результате в старших разрядах содержится номер виртуальной страницы, а в младших — смещение искомого элемента относительно начала страницы. Так, если размер страницы равен 2^k, то смещением является содержимое младших k разрядов, а остальные, старшие, разряды содержат номер виртуальной страницы,

которой принадлежит искомый адрес. Далее преобразование адреса происходит так же, как при страничной организации: старшие разряды линейного виртуального адреса, содержащие номер виртуальной страницы, заменяются номером физической страницы, взятым из таблицы страниц, а младшие разряды виртуального адреса, содержащие смещение, остаются без изменения.

Как видно, механизм сегментации и страничный механизм действуют достаточно независимо друг от друга. Поэтому нетрудно представить себе реализацию сегментно-страничного управления памятью, в которой механизм сегментации работает по вышеописанной схеме, а страничный механизм изменен в соответствии с двухуровневой схемой, когда виртуальное адресное пространство делится сначала на разделы, а уж потом на страницы. В таком случае преобразование виртуального адреса в физический происходит в несколько этапов. Сначала механизм сегментации обычным образом, используя таблицу сегментов, вычисляет линейный виртуальный адрес. Затем из данного виртуального адреса вычленяются номера раздела и страницы, а также смещение. И далее по номеру раздела из таблицы разделов определяется адрес таблицы страниц, а затем по номеру виртуальной страницы из таблицы страниц определяется номер физической страницы, к которому пристыковывается смещение. Именно такой подход реализован компанией Intel в процессорах Pentium.

Рассмотрим еще одну возможную схему управления памятью, основанную на *комбинировании сегментного и страничного механизмов*. Так же как и в предыдущих случаях, виртуальное пространство процесса делится на сегменты, а каждый сегмент, в свою очередь, делится на виртуальные страницы. Первое отличие состоит в том, что виртуальные страницы нумеруются не в пределах всего адресного пространства процесса, а в пределах сегмента. Виртуальный адрес в этом случае выражается тройкой (номер сегмента, номер страницы, смещение в странице).

Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть — на диске. Для каждого процесса создается собственная таблица сегментов, а для каждого сегмента — своя таблица страниц. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс.

Таблица страниц содержит дескрипторы страниц, содержимое которых полностью аналогично содержимому описанных ранее дескрипторов страниц. А вот таблица сегментов состоит из дескрипторов сегментов, которые вместо информации о расположении сегментов в виртуальном адресном пространстве содержат описание расположения таблиц страниц в физической памяти. Это является вторым существенным отличием данного подхода от ранее рассмотренной схемы сегментно-страничной организации.

На рис. 5.22 показана схема преобразования виртуального адреса в физический для данного метода.

1. По номеру сегмента, заданному в виртуальном адресе, из таблицы сегментов извлекается физический адрес соответствующей таблицы страниц.

2. По номеру виртуальной страницы, заданному в виртуальном адресе, из таблицы страниц извлекается дескриптор, в котором указан номер физической страницы.
3. К номеру физической страницы пристыковывается младшая часть виртуального адреса — смещение.

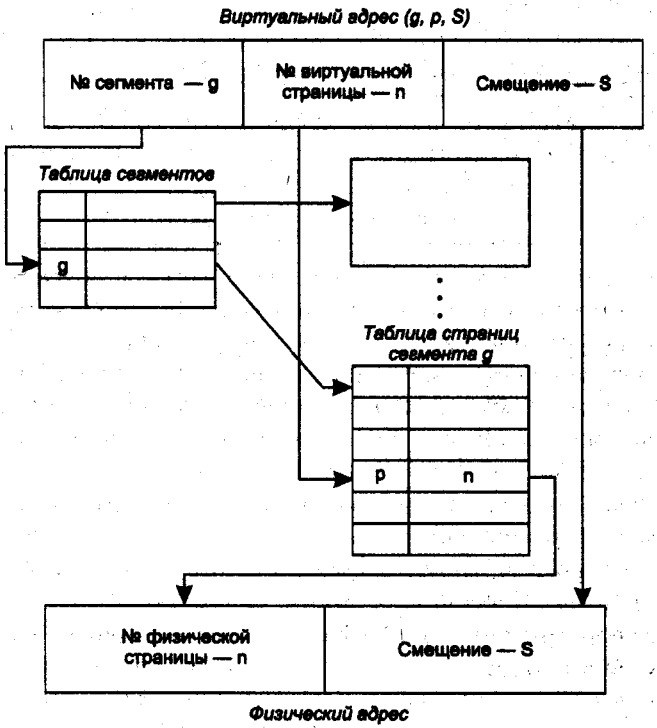


Рис. 5.22. Еще одна схема преобразования виртуального адреса в физический при сегментно-страничной организации памяти

Разделяемые сегменты памяти

Хотя основной задачей операционной системы при управлении памятью является защита областей оперативной памяти, принадлежащей одному из процессов, от доступа к ней остальных процессов, в некоторых случаях оказывается полезным организовать контролируемый совместный доступ нескольких процессов к определенной области памяти.

Например, в том случае, когда несколько пользователей одновременно работают с некоторым текстовым редактором, нецелесообразно многократно загружать его код в оперативную память. Гораздо экономичней загрузить всего одну копию кода, которая обслуживала бы всех пользователей, работающих в данное время с этим редактором (для этого код редактора должен быть реентерабель-

ным). Очевидно, что сегмент данных редактора не может присутствовать в памяти в единственном разделяемом экземпляре — для каждого пользователя должна быть создана своя копия этого сегмента, в которой помещаются редактируемый текст и значения других переменных редактора, например его конфигурация, индивидуальная для каждого пользователя, и т. п.

Другим примером применения разделяемой области памяти может быть использование ее в качестве буфера при межпроцессном обмене данными. В этом случае один процесс пишет в разделяемую область, а другой — читает.

> Подсистема виртуальной памяти представляет собой удобный механизм для решения задачи совместного доступа нескольких процессов к одному и тому же сегменту памяти, который в этом случае становится разделяемым, а сама память называется **разделяемой** (shared memory).

Для организации разделяемого сегмента при наличии подсистемы виртуальной памяти достаточно поместить его в виртуальное адресное пространство каждого процесса, которому нужен доступ к данному сегменту, а затем настроить параметры отображения этих виртуальных сегментов так, чтобы они соответствовали одной и той же области оперативной памяти. Детали такой настройки зависят от типа используемой в ОС модели виртуальной памяти: сегментной или сегментно-страничной (чисто страничная организация не поддерживает понятие «сегмент», что делает невозможным решение рассматриваемой задачи). Например, при сегментной организации необходимо в дескрипторах виртуального сегмента каждого процесса указать один и тот же базовый физический адрес. При сегментно-страничной организации отображение на одну и ту же область памяти достигается за счет соответствующей настройки таблицы страниц каждого процесса.

В приведенном описании подразумевается, что разделяемый сегмент помещается в индивидуальную часть виртуального адресного пространства каждого процесса (рис. 5.23, *а*) и описывается в каждом процессе индивидуальным дескриптором сегмента (и индивидуальными дескрипторами страниц, если используется сегментно-страничный механизм). «Попадание» же этих виртуальных сегментов в общую часть оперативной памяти достигается за счет согласованной настройки операционной системой многочисленных дескрипторов для множества процессов.

Есть и более экономичное для ОС решение этой задачи — помещение единственного разделяемого виртуального сегмента в общую часть виртуального адресного пространства процессов, то есть в ту часть, которая обычно используется для модулей ОС (рис. 5.23, *б*). В этом случае настройка дескриптора сегмента (и дескрипторов страниц) выполняется только один раз, а все процессы на основе такой настройки совместно используют часть оперативной памяти.

При работе с разделяемыми сегментами памяти ОС должна выполнять некоторые функции, общие для любых разделяемых между процессами ресурсов — файлов, семафоров и т. п. Эти функции состоят в поддержке схемы именования ресурсов, проверке прав доступа определенного процесса к ресурсу,

а также в отслеживании количества процессов, пользующихся данным ресурсом (чтобы удалить его в случае ненадобности). Для того чтобы отличать разделяемые сегменты памяти от индивидуальных, дескриптор сегмента должен содержать поле, имеющее два значения: разделяемый (shared) и индивидуальный (private).

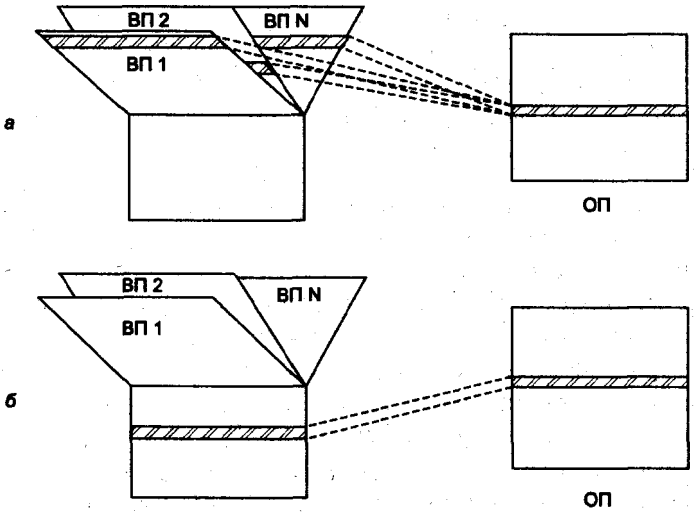


Рис. 5.23. Два способа создания разделяемого сегмента памяти

Операционная система может создавать разделяемые сегменты как по явному запросу, так и по умолчанию. В первом случае прикладной процесс должен выполнить соответствующий системный вызов, по которому операционная система создает новый сегмент в соответствии с указанными в вызове параметрами: размером сегмента, разрешенными над ним операциями (чтение/запись) и идентификатором. Все процессы, выполнившие подобные вызовы с одним и тем же идентификатором, получают доступ к этому сегменту и используют его по своему усмотрению, например, в качестве буфера для обмена данными.

Во втором случае операционная система сама в определенных ситуациях принимает решение о том, что нужно создать разделяемый сегмент. Наиболее типичным примером такого рода является поступление нескольких запросов на выполнение одного и того же приложения. Если кодовый сегмент приложения помечен в исполняемом файле как реентерабельный и разделяемый, то ОС не создает при поступлении нового запроса новую индивидуальную для процесса копию кодового сегмента этого приложения, а отображает уже существующий разделяемый сегмент на виртуальное адресное пространство процесса. При закрытии приложения каким-либо процессом ОС проверяет, существуют ли другие процессы, пользующиеся данным приложением, и если их нет, то удаляет данный разделяемый сегмент.

Разделяемые сегменты выгружаются на диск системой виртуальной памяти по тем же алгоритмам и с помощью тех же механизмов, что и индивидуальные.

Кэширование данных

Универсальная концепция

Кэширование данных — это универсальный метод ускорения доступа к данным, основанный на комбинации двух типов памяти, отличающихся временем доступа, объемом и стоимостью хранения данных. Наиболее часто используемая в данный период информация динамически копируется из «медленной, но большой» памяти в «быструю, но маленькую» **кэш-память**, или **кэш** (cash).

Например, если кэширование применяется для уменьшения среднего времени доступа к оперативной памяти, то в качестве кэша выступает быстродействующая статическая память процессора или даже его регистры. А если система ввода-вывода кэширует данные для ускорения доступа к информации, хранящейся на диске, то в этом случае роль кэша исполняют буферы в оперативной памяти, в которых оседают наиболее активно используемые данные.

Кэширование широко используется и в Интернете, в частности при разрешении символьных DNS-имен, в протоколе ARP, в работе браузера. Страницы информации с удаленных веб-сайтов («медленная, но большая» память), к которым с помощью браузера обращается пользователь, кэшируются на диске его компьютера («быстрая, но маленькая» память). При чтении новой страницы, если кэш на локальном диске к этому моменту заполнен, происходит вытеснение более старой (наименее используемой в последнее время) информации. Такой способ работы не только сокращает время доступа к данным — считывание с локального диска существенно быстрее получения данных из сети, но и уменьшает загрузку сети, так как исключает избыточные передачи страниц.

Кэширование позволяет не только сократить среднее время доступа к данным, но и экономить более дорогую быстродействующую память. Виртуальную память можно считать одним из вариантов реализации принципа кэширования данных, при котором основной целью является экономия памяти. Действительно, оперативная память выступает здесь в роли кэша по отношению к внешней памяти — жесткому диску. Кэширование задействуется здесь не для того, чтобы сократить время доступа к данным (команды в принципе не могут выполняться, находясь на диске), а для того, чтобы заставить диск частично подменить оперативную память за счет перемещения временно неиспользуемых кода и данных на диск с целью освобождения места для активных процессов. В результате наиболее интенсивно применяемые данные «оседают» в оперативной памяти, остальная же информация хранится в более объемной и менее дорогостоящей внешней памяти.

Неотъемлемым свойством кэш-памяти является ее *прозрачность* для программ и пользователей. Система, реализующая кэширование, не требует никакой внешней информации об интенсивности обращения к данным; ни пользо-

ватели, ни программы не принимают никакого участия в перемещении данных из одного типа памяти в память другого типа. Все это делается автоматически системными средствами.

Иерархия памяти

Память вычислительной машины представляет собой иерархию запоминающих устройств (ЗУ), отличающихся средним временем доступа к данным, объемом и стоимостью хранения одного бита (рис. 5.24). Фундаментом этой пирамиды запоминающих устройств служит внешняя память, как правило, представляемая жестким диском. Она имеет большой объем (десятки и сотни гигабайтов), но скорость доступа к данным является невысокой. Время доступа к диску измеряется миллисекундами.

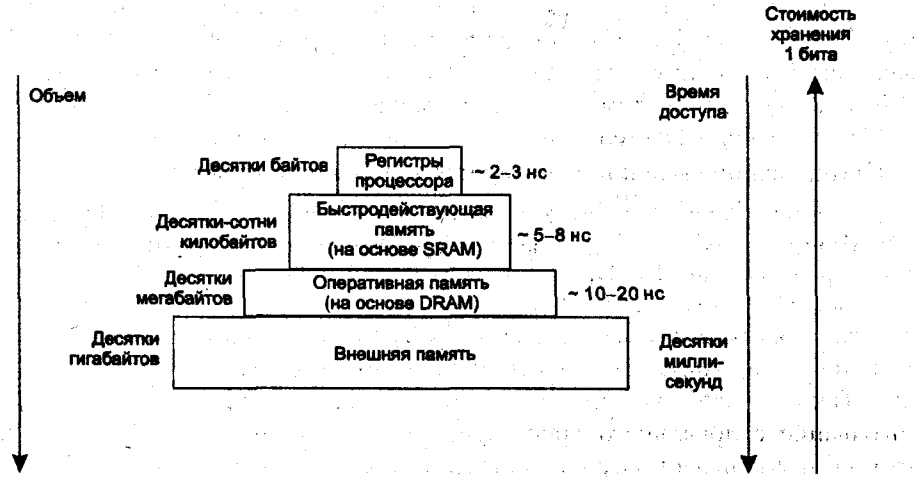


Рис. 5.24. Иерархия запоминающих устройств

На следующем уровне располагается более быстродействующая (время доступа[1] равно примерно 10–20 наносекундам) и менее объемная (от десятков мегабайтов до нескольких гигабайтов) оперативная память, реализуемая на относительно медленной динамической памяти DRAM.

Для хранения данных, к которым необходимо обеспечить быстрый доступ, используются компактные быстродействующие запоминающие устройства на основе статической памяти SRAM, объем которых составляет от нескольких десятков до нескольких сотен килобайтов, а время доступа к данным обычно не превышает 8 нс.

И наконец, верхушку в этой пирамиде составляют внутренние регистры процессора, которые также могут быть использованы для промежуточного хра-

[1] Все перечисленные характеристики ЗУ быстро изменяются по мере совершенствования вычислительной аппаратуры. В данном случае важны не абсолютные значения времени доступа или объема памяти, а их соотношение для разных типов запоминающих устройств.

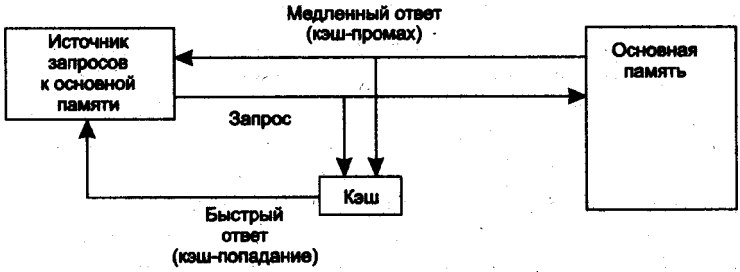
нения данных. Общий объем регистров составляет несколько десятков байтов, а время доступа определяется быстродействием процессора и равно в настоящее время примерно 2–3 нс.

Таким образом, можно констатировать печальную закономерность — чем больше объем устройства, тем менее быстродействующим оно является. Более того, стоимость хранения данных в расчете на один бит также увеличивается с ростом быстродействия устройств. В то же время пользователю хотелось бы иметь и недорогую, и быструю память. Кэш-память представляет некоторое компромиссное решение этой проблемы.

Принцип действия кэш-памяти

Рассмотрим одну из возможных схем кэширования (рис. 5.25). Содержимое кэш-памяти представляет собой совокупность *записей* обо всех загруженных в нее элементах данных из основной памяти. Каждая запись об элементе данных включает в себя:

- значение элемента данных;
- адрес, который этот элемент данных имеет в основной памяти;
- дополнительную информацию, которая используется для реализации алгоритма замещения данных в кэше и обычно включает признак модификации и признак действительности данных.



Структура кэш-памяти

Адрес данных в основной памяти	Данные	Управляющая информация

Рис. 5.25. Схема функционирования кэш-памяти

При каждом обращении к основной памяти по физическому адресу просматривается содержимое кэш-памяти с целью определения, не находятся ли там нужные данные. Кэш-память не является адресуемой, поэтому поиск нужных данных осуществляется по содержимому — по взятому из запроса значению

поля адреса в оперативной памяти. Далее возможен один из двух вариантов развития событий:

- если данные обнаруживаются в кэш-памяти, то есть происходит **кэш-попадание** (cache-hit), они считываются из нее, и результат передается источнику запроса;
- если нужные данные отсутствуют в кэш-памяти, то есть происходит **кэш-промах** (cache-miss), они считываются из основной памяти, передаются источнику запроса и одновременно с этим копируются в кэш-память.

Интуитивно понятно, что эффективность кэширования зависит от вероятности кэш-попадания. Покажем это путем нахождения зависимости среднего времени доступа к основной памяти от вероятности кэш-попаданий. Пусть имеется основное запоминающее устройство со средним временем доступа к данным t_1 и кэш-память, имеющая время доступа t_2, очевидно, что $t_2 < t_1$. Пусть t — среднее время доступа к данным в системе с кэш-памятью, а p — вероятность кэш-попадания. По формуле полной вероятности имеем:

$$t = t_1(1 - p) + t_2 p = (t_2 - t_1)p + t_1.$$

Среднее время доступа к данным в системе с кэш-памятью линейно зависит от вероятности кэш-попадания и изменяется от среднего времени доступа в основное запоминающее устройство t_1 при $p = 0$ до среднего времени доступа непосредственно в кэш-память t_2 при $p = 1$. Отсюда видно, что кэширование имеет смысл только при высокой вероятности кэш-попадания.

Вероятность обнаружения данных в кэше зависит от разных факторов, таких, например, как объем кэша, объем кэшируемой памяти, алгоритм замещения данных в кэше, особенности выполняемой программы, время ее работы, уровень мультипрограммирования и других особенностей вычислительного процесса. Тем не менее в большинстве реализаций кэш-памяти процент кэш-попаданий оказывается весьма высоким — свыше 90 %. Такое высокое значение вероятности нахождения данных в кэш-памяти объясняется наличием у данных объективных свойств: пространственной и временной локальности.

- **Временная локальность.** Если произошло обращение по некоторому адресу, то следующее обращение по тому же адресу с большой вероятностью произойдет в ближайшее время.
- **Пространственная локальность.** Если произошло обращение по некоторому адресу, то с большой вероятностью в ближайшее время произойдет обращение к соседним адресам.

Именно основываясь на свойстве временной локальности, данные, только что считанные из основной памяти, размещают в запоминающем устройстве быстрого доступа, предполагая, что скоро они опять понадобятся. Вначале работы системы, когда кэш-память еще пуста, почти каждый запрос к основной памяти выполняется «по полной программе»: просмотр кэша, констатация кэш-промаха, чтение данных из основной памяти, передача результата источнику запроса и копирование данных в кэш. Затем, по мере заполнения кэша, в полном

соответствии со свойством временной локальности возрастает вероятность обращения к данным, которые уже были использованы на предыдущем этапе работы системы, то есть к данным, которые содержатся в кэше и могут быть считаны значительно быстрее, чем из основной памяти.

Свойство пространственной локальности также используется для повышения вероятности кэш-попадания: как правило, в кэш-память считывается не один информационный элемент, к которому произошло обращение, а целый блок данных, расположенных в основной памяти в непосредственной близости с данным элементом. Поскольку при выполнении программы очень высока вероятность, что команды выбираются из памяти последовательно одна за другой из соседних ячеек, то имеет смысл загружать в кэш-память целый фрагмент программы. Аналогично, если программа ведет обработку некоторого массива данных, то ее работу можно ускорить, загрузив в кэш часть или даже весь массив данных. При этом учитывается высокая вероятность того, что значительное число обращений к памяти будет выполняться к адресам массива данных.

Проблема согласования данных

В процессе работы содержимое кэш-памяти постоянно обновляется, а значит, время от времени данные из нее должны вытесняться. Вытеснение означает либо простое объявление свободной соответствующей области кэш-памяти (сброс бита действительности), если вытесняемые данные за время нахождения в кэше не были изменены, либо в дополнение к этому копирование данных в основную память, если они были модифицированы. Алгоритм замены данных в кэш-памяти существенно влияет на ее эффективность. В идеале такой алгоритм должен, во-первых, быть максимально быстрым, чтобы не замедлять работу кэш-памяти, во-вторых, обеспечивать максимально возможную вероятность кэш-попаданий. Поскольку из-за непредсказуемости вычислительного процесса ни один алгоритм замещения данных в кэш-памяти не может гарантировать *оптимальный* результат, разработчики ограничиваются *рациональными* решениями, которые, по крайней мере, не слишком замедляют работу кэша — запоминающего устройства, изначально призванного быть быстрым.

Наличие в компьютере двух копий данных — в основной памяти и в кэше — порождает проблему согласования данных. Если происходит запись в основную память по некоторому адресу, а содержимое этой ячейки находится в кэше, то в результате соответствующая запись в кэше становится недостоверной. Рассмотрим два подхода к решению этой проблемы.

- **Сквозная запись** (write through). При каждом запросе к основной памяти, в том числе и при записи, просматривается кэш. Если данные по запрашиваемому адресу отсутствуют, то запись выполняется только в основную память. Если же данные, к которым выполняется обращение, находятся в кэше, то запись выполняется одновременно в кэш и основную память.
- **Обратная запись** (write back). Аналогично, при возникновении запроса к памяти выполняется просмотр кэша, и если запрашиваемых данных там нет, то запись выполняется только в основную память. В противном же случае за-

пись производится *только в кэш-память*, при этом в описателе данных делается специальная отметка (признак модификации), которая указывает на то, что при вытеснении этих данных из кэша необходимо переписать их в основную память, чтобы актуализировать устаревшее содержимое основной памяти.

В некоторых алгоритмах замещения предусматривается первоочередная выгрузка модифицированных или, как еще говорят, «грязных» данных. Модифицированные данные могут выгружаться не только при освобождении места в кэш-памяти для новых данных, но и в «фоновом режиме», когда система не очень загружена.

Отображение основной памяти на кэш

Алгоритм поиска и алгоритм замещения данных в кэше непосредственно зависят от того, каким образом основная память отображается на кэш-память. Принцип прозрачности требует, чтобы правило отображения основной памяти на кэш-память не зависело от работы программ и пользователей.

При кэшировании данных из оперативной памяти широко используются две основных схемы отображения: **случайное** и **детерминированное**.

При *случайном* отображении элемент оперативной памяти в общем случае может быть размещен в произвольном месте кэш-памяти. Для того чтобы в дальнейшем можно было найти нужные данные в кэше, они помещаются туда вместе со своим адресом, то есть тем адресом, который данные имеют в оперативной памяти. При каждом запросе к оперативной памяти выполняется поиск в кэше, причем критерием поиска выступает адрес оперативной памяти из запроса. Очевидная схема простого перебора для поиска нужных данных в случае кэша оказывается непригодной из-за недопустимо больших временных затрат.

Для кэшей со случайным отображением используется так называемый **ассоциативный поиск**, при котором сравнение выполняется не последовательно с каждой записью кэша, а параллельно со всеми его записями (рис. 5.26). Признак, по которому выполняется сравнение, называется **тегом** (tag). В данном случае тегом является адрес данных в оперативной памяти. Электронная реализация такой схемы приводит к удорожанию памяти, причем стоимость существенно возрастает с увеличением объема запоминающего устройства. Поэтому ассоциативная кэш-память используется в тех случаях, когда для обеспечения высокого процента кэш-попаданий достаточно небольшого объема памяти.

В кэшах, построенных на основе случайного отображения, вытеснение старых данных происходит только в том случае, когда вся кэш-память заполнена и нет свободного места. Выбор данных на выгрузку осуществляется среди всех записей кэша. Обычно этот выбор основывается на тех же приемах, что и в алгоритмах замещения страниц, например, выгрузка данных, к которым дольше всего не было обращений, или данных, к которым было меньше всего обращений.

Второй, *детерминированный*, способ отображения предполагает, что любой элемент основной памяти всегда отображается на одно и то же место в кэш-

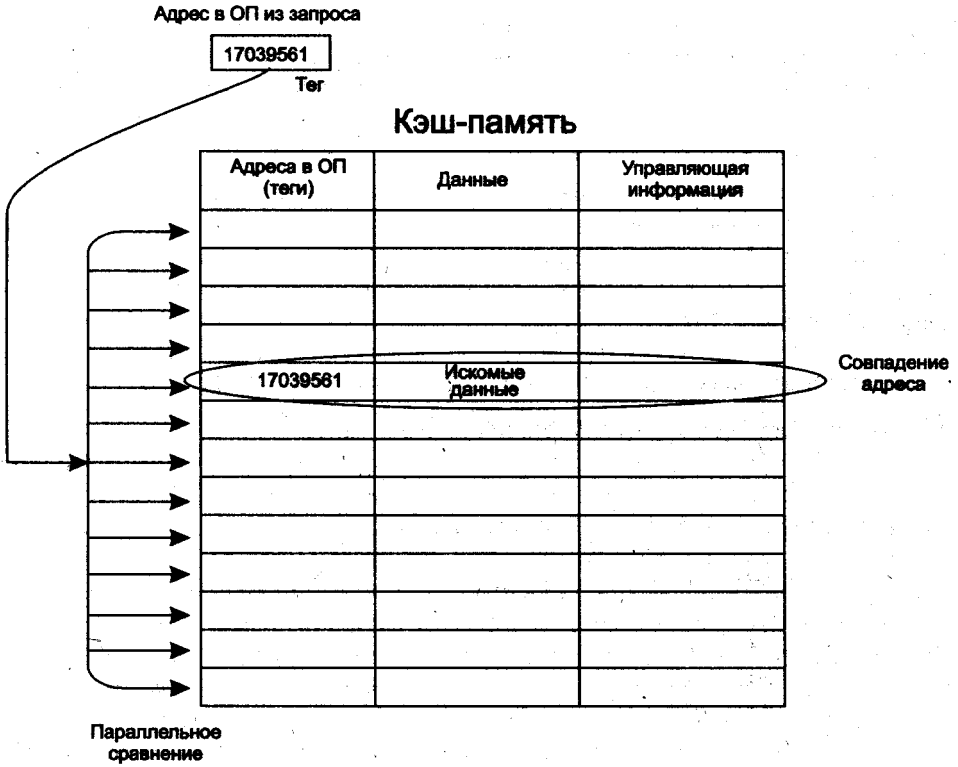


Рис. 5.26. Ассоциативный поиск в кэше со случайным отображением

памяти. В этом случае кэш-память разделена на строки, каждая из которых предназначена для хранения одной записи об одном элементе данных[1] и имеет свой номер. Между номерами строк кэш-памяти и адресами оперативной памяти устанавливается соответствие «один ко многим»: одному номеру строки соответствует несколько (обычно достаточно много) адресов оперативной памяти.

ПРИМЕЧАНИЕ Описанный здесь способ отображения является частным случаем хэширования, а соответствующий алгоритм — хэш-функцией. Хэш-функции требуются для решения самых разных задач, например, они широко используются в криптографии. Приближенным аналогом хэширования является размещение слов в справочнике по алфавиту. Хэш-функцией здесь является отделение первой буквы, которая определяет раздел справочника. Далее можно вести поиск только в пределах этого раздела.

В качестве отображающей функции может использоваться простое выделение нескольких разрядов из адреса оперативной памяти, которые интерпрети-

[1] В действительности запись в кэше обычно содержит несколько элементов данных.

руются как номер строки кэш-памяти (такое отображение называется прямым). Например, пусть в кэш-памяти может храниться 1024 записи, то есть кэш имеет 1024 строки, пронумерованных от 0 до 1023. Тогда любой адрес оперативной памяти может быть отображен на адрес кэш-памяти путем использования лишь 10 младших двоичных разрядов (рис. 5.27).

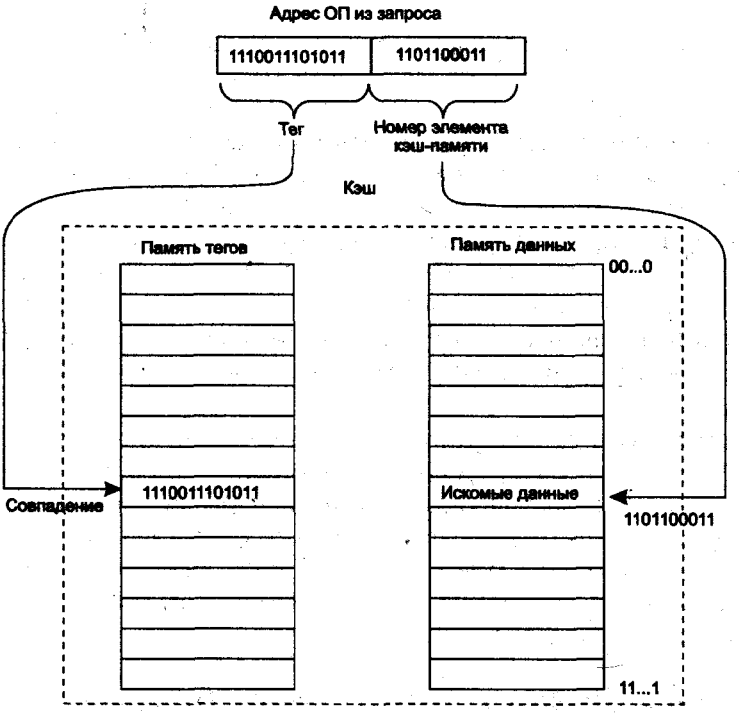


Рис. 5.27. Прямое отображение

Хэширование дает возможность выполнить быстрый поиск данных в кэше при поступлении запроса. Адрес оперативной памяти, указанный в запросе, обрабатывается с использованием той же хэш-функции, которой в данном случае является отбрасывание всех старших двоичных разрядов. Младшие 10 разрядов интерпретируются как номер строки в кэш-памяти, и, таким образом, исчезает необходимость просматривать всю кэш-память.

Однако поскольку в найденной строке могут находиться данные из любой ячейки оперативной памяти, младшие разряды адреса которой совпадают с номером строки, необходимо выполнить дополнительную проверку. Для этих целей каждая строка кэш-памяти дополняется тегом, содержащим старшую часть адреса данных в оперативной памяти. При совпадении тега с соответствующей частью адреса из запроса констатируется кэш-попадание.

Если же происходит кэш-промах, данные считываются из оперативной памяти и копируются в кэш. Если строка кэш-памяти, в которую должен быть скопирован элемент данных из оперативной памяти, содержит другие данные,

то последние вытесняются из кэша. Заметим, что процесс замещения данных в кэш-памяти на основе прямого отображения существенно отличается от процесса замещения данных в кэш-памяти со случайным отображением. Во-первых, вытеснение данных происходит не только в случае отсутствия свободного места в кэше, во-вторых, никакого выбора данных для замещения не происходит.

Во многих современных процессорах кэш-память строится на основе сочетания этих двух подходов, что позволяет найти компромисс между сравнительно низкой стоимостью кэша с прямым отображением и интеллектуальностью алгоритмов замещения в кэше со случайным отображением. При смешанном подходе произвольный адрес оперативной памяти отображается не на один адрес кэш-памяти (что характерно для прямого отображения) и не на любой адрес кэш-памяти (как это делается при случайном отображении), а на некоторую группу адресов. Все группы пронумерованы. Поиск в кэше осуществляется вначале по номеру группы, полученному из адреса оперативной памяти в запросе, а затем в пределах группы путем ассоциативного просмотра всех записей в группе на предмет совпадения старших частей адресов оперативной памяти (рис. 5.28).

При промахе данные копируются по любому свободному адресу из однозначно заданной группы. Если свободных адресов в группе нет, то выполняется вытеснение данных. Поскольку кандидатов на выгрузку несколько (все записи данной группы), алгоритм замещения может учесть интенсивность обращений к данным и тем самым повысить вероятность попаданий в будущем. Таким образом, в данном способе комбинируется прямое отображение на группу и случайное отображение в пределах группы.

Схемы выполнения запросов в системах с кэш-памятью

На рис. 5.29 приведена обобщенная схема работы кэш-памяти. Большая часть ветвей этой схемы уже была подробно рассмотрена ранее, поэтому остановимся только на некоторых особых случаях.

Из схемы видно, что когда выполняется запись, кэш просматривается только с целью согласования содержимого кэша и основной памяти. Если происходит промах, то запросы на запись не вызывают никаких изменений содержимого кэша. В некоторых же реализациях кэш-памяти при отсутствии данных в кэше они копируются туда из основной памяти независимо от того, выполняется запрос на чтение или на запись.

В соответствии с описанной логикой работы кэш-памяти следует, что при возникновении запроса сначала просматривается кэш, а затем, если имеет место промах, выполняется обращение к основной памяти. Однако часто реализуется и другая схема работы кэша: поиск в кэше и в основной памяти начинается одновременно, а затем, в зависимости от результата просмотра кэша, операция в основной памяти либо продолжается, либо прерывается.

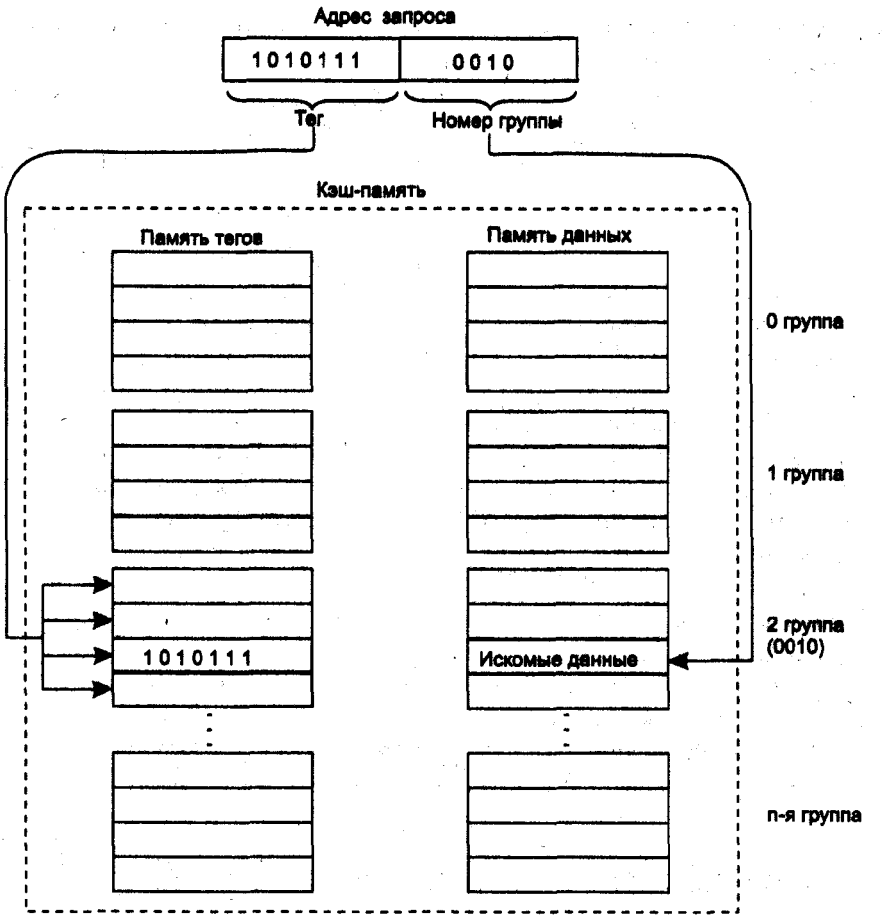


Рис. 5.28. Комбинирование прямого и случайного отображения

При выполнении запросов к оперативной памяти во многих вычислительных системах происходит двухуровневое кэширование (рис. 5.30). Кэш первого уровня имеет меньший объем и более высокое быстродействие, чем кэш второго уровня. Кэш второго уровня играет роль основной памяти по отношению к кэшу первого уровня.

На рис. 5.31 показана схема выполнения запроса на чтение в системе с двухуровневым кэшем. Сначала делается попытка обнаружить данные в кэше первого уровня. Если произошел промах, поиск продолжается в кэше второго уровня. Если же нужные данные отсутствуют и здесь, тогда происходит считывание данных из основной памяти. Понятно, что время доступа к данным оказывается минимальным, когда кэш-попадание происходит уже на первом уровне, несколько большим при обнаружении данных на втором уровне и обычным временем доступа к оперативной памяти, если нужных данных нет ни в том, ни в другом кэше. При считывании данных из оперативной памяти происходит их

копирование в кэш второго уровня, а если данные считываются из кэша второго уровня, то они копируются в кэш первого уровня.

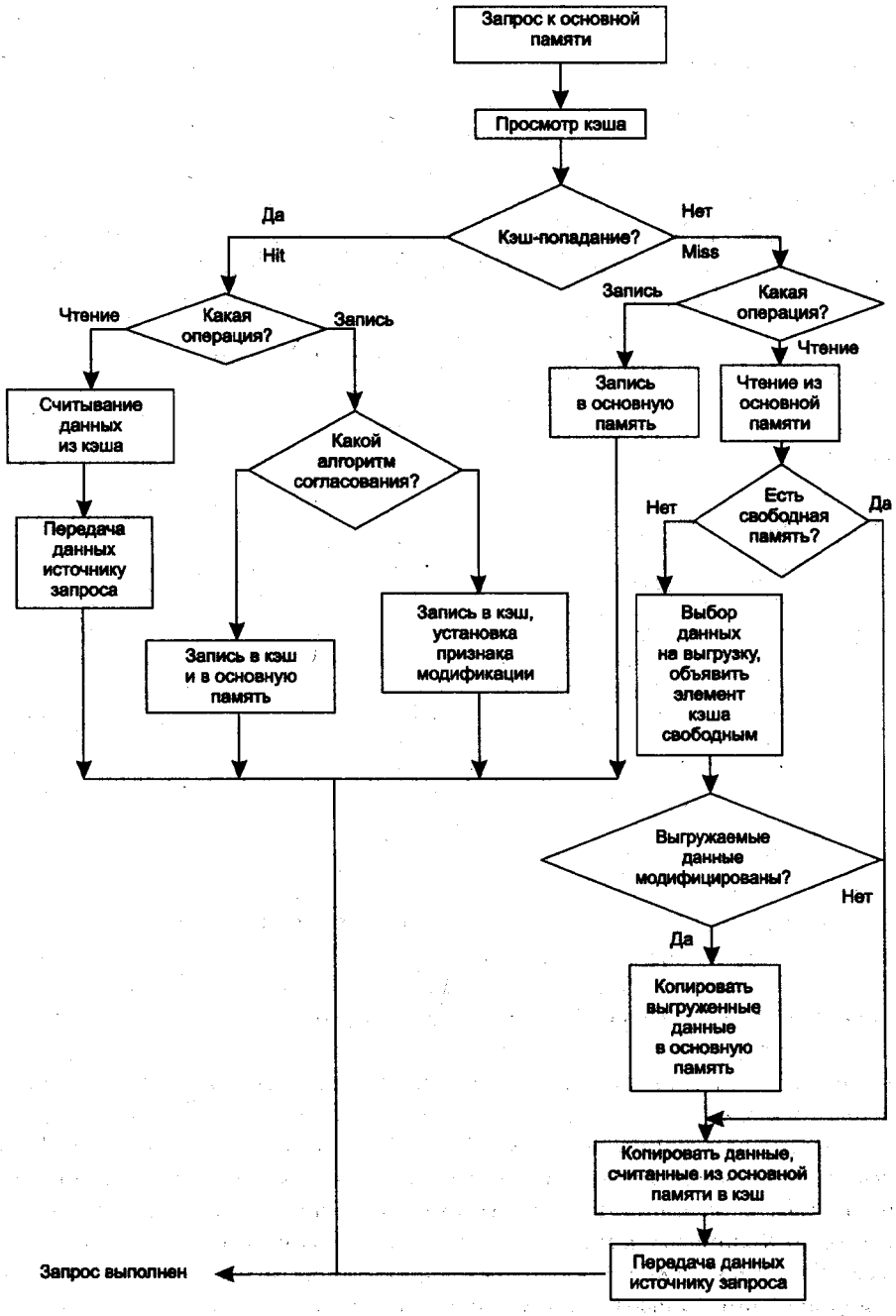


Рис. 5.29. Схема выполнения запроса к памяти в системе с кэшированием

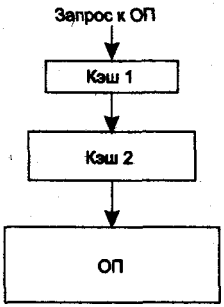


Рис. 5.30. Двухуровневое кэширование

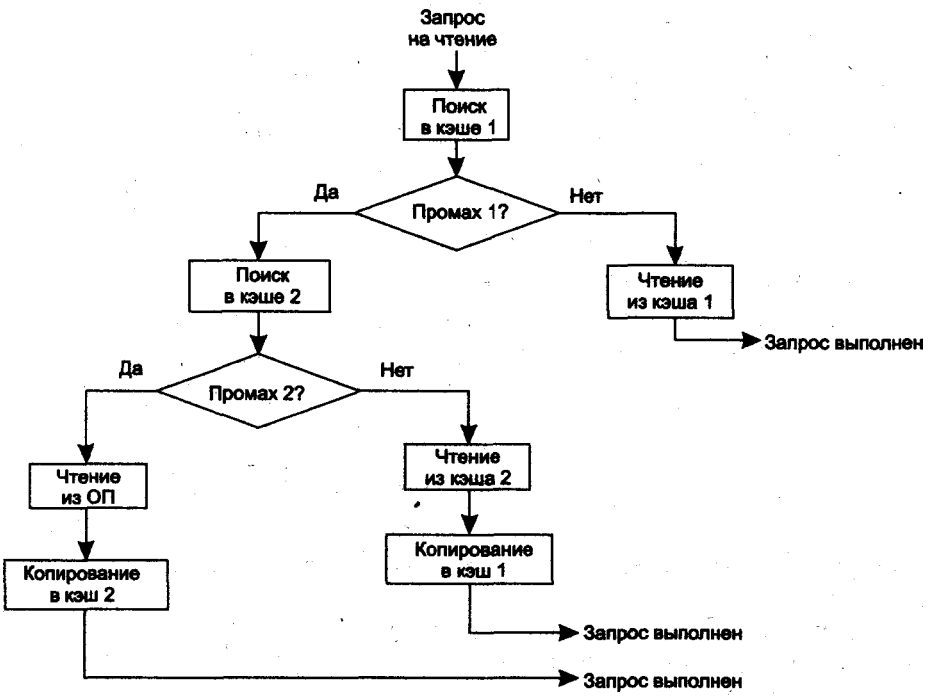


Рис. 5.31. Схема выполнения запроса на чтение в системе с двухуровневым кэшем

При работе такой иерархически организованной памяти необходимо обеспечить непротиворечивость данных на всех уровнях. Кэши разных уровней могут согласовывать данные разными способами. Пусть, например, кэш первого уровня использует сквозную запись, а кэш второго уровня — обратную запись. (Именно такая комбинация алгоритмов согласования применена в процессоре Pentium при одном из возможных вариантов его работы.)

На рис. 5.32 приведена схема выполнения запроса на запись в такой системе. При модификации данных необходимо убедиться, что они отсутствуют в кэшах. В этом случае выполняется запись только в оперативную память.

Если данные обнаруживаются в кэше первого уровня, то вступает в силу алгоритм сквозной записи: выполняется запись в кэш первого уровня и передается запрос на запись в кэш второго уровня, играющему в данном случае роль основной памяти. Запись в кэш второго уровня в соответствии с алгоритмом обратной записи, принятом на данном уровне, сопровождается установкой признака модификации, при этом никакой записи в оперативную память не производится.

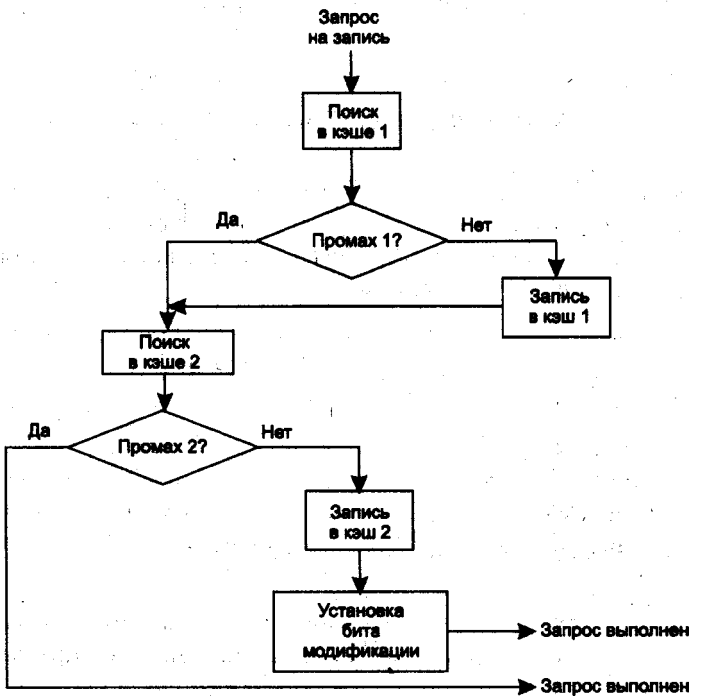


Рис. 5.32. Схема выполнения запроса на запись в системе с двухуровневым кэшем

Если данные находятся в кэше второго уровня, то, так же как и в предыдущем случае, выполняется запись в этот кэш и устанавливается признак модификации.

Рассмотренные в данном разделе проблемы кэширования охватывают только такой класс систем организации памяти, в котором на каждом уровне имеется одно кэширующее устройство. Существует и другой класс систем памяти, главной отличительной особенностью которого является наличие нескольких кэшей одного уровня. Этот вариант характерен для распределенных систем обработки информации — мультипроцессорных компьютеров и компьютерных сетей.

Выводы

- Оперативная память является важнейшим ресурсом вычислительной системы, требующим тщательного управления со стороны мультипрограммной

операционной системы. Особая роль памяти объясняется тем, что процессор может выполнять инструкции программы только в том случае, если они находятся в памяти.

- Память распределяется как между модулями прикладных программ, так и между модулями самой операционной системы.
- Функциями ОС по управлению памятью в мультипрограммной системе являются:
  - отслеживание наличия свободной и занятой памяти;
  - выделение памяти процессам и освобождение памяти при завершении процессов;
  - вытеснение кодов и данных процессов из оперативной памяти на диск (полное или частичное), когда размеры основной памяти недостаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место;
  - настройка адресов программы на конкретную область физической памяти;
  - защита памяти процессов от взаимного вмешательства.
- На разных этапах жизненного цикла программы для представления переменных и кодов требуется три типа адресов: символьные (имена, используемые программистом), виртуальные (условные числа, вырабатываемые компилятором) и физические (адреса фактического размещения в оперативной памяти).
- Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. Диапазон возможных адресов виртуального пространства у всех процессов является одним и тем же.
- Виртуальное адресное пространство может быть плоским (линейным) или структурированным.
- Необходимо различать максимально возможное виртуальное адресное пространство процесса, которое определяется только разрядностью виртуального адреса и архитектурой компьютера, и назначенное (выделенное) процессу виртуальное адресное пространство, состоящее из набора виртуальных адресов, действительно нужных процессу для работы.
- Виртуальное адресное пространство процесса делится на две непрерывные части: системную и пользовательскую. Системная часть является общей для всех процессов, в ней размещаются коды и данные операционной системы.
- Наиболее эффективным методом управления памятью является использование виртуальной памяти. Этот подход вытеснил в современных ОС методы распределения памяти фиксированными, динамическими или перемещаемыми разделами.

- Механизм виртуальной памяти использует дисковую память для временного хранения не помещающихся в оперативную память данных и кодов выполняемых процессов ОС.
- В настоящее время все множество реализаций виртуальной памяти может быть представлено тремя классами:
  - страничная виртуальная память организует перемещение данных между памятью и диском страницами — частями виртуального адресного пространства фиксированного и сравнительно небольшого размера (достоинства — высокая скорость обмена, низкий уровень фрагментации; недостатки — сложно организовать защиту данных, разделенных на части механически);
  - сегментная виртуальная память предусматривает перемещение данных сегментами — частями виртуального адресного пространства произвольного размера, полученными с учетом смыслового значения данных (достоинства — «осмысленность» сегментов упрощает их защиту; недостатки — медленное преобразование адреса, высокий уровень фрагментации);
  - сегментно-страничная виртуальная память сочетает достоинства обоих предыдущих подходов.
- Сегменты виртуальной памяти могут быть разделяемыми между несколькими процессами. Разделяемые сегменты применяют либо для экономии физической памяти, когда несколько пользователей работают с одним кодовым сегментом приложения, либо в качестве средства обмена данными между процессами.
- Кэширование данных — это универсальный метод ускорения доступа к данным, основанный на комбинации двух типов памяти, отличающихся временем доступа, объемом и стоимостью хранения данных. Наиболее часто используемая в данный период информация динамически копируется из «медленной, но большой» памяти в «быструю, но маленькую» память.
- Универсальность концепции кэширования подтверждают примеры ее использования в самых различных технологиях и продуктах: в процессорах для сокращения среднего времени доступа к оперативной памяти, в подсистеме ввода-вывода для ускорения доступа к информации, хранящейся на диске, в Интернете при разрешении символьных DNS-имен, в протоколе ARP, в работе браузера.
- Неотъемлемым свойством кэш-памяти является ее прозрачность для программ и пользователей.
- При кэшировании данных из оперативной памяти широко используются две основных схемы отображения: случайное и детерминированное. В первом случае элемент оперативной памяти вместе со своим адресом размещается в произвольном месте кэш-памяти, во втором элемент оперативной памяти всегда отображается на одно и то же место — в строке, которая имеет свой номер и предназначена для хранения записи об одном элементе данных.

Задачи и упражнения

1. Чем ограничивается максимальный размер физической памяти, которую можно установить в компьютере определенной модели?
2. Чем ограничивается максимальный размер виртуального адресного пространства, доступного приложению?
3. Может ли прикладной процесс использовать системную часть виртуальной памяти?
4. Какое из этих двух утверждений верно:
   1) все виртуальные адреса заменяются физическими во время загрузки программы в оперативную память;
   2) виртуальные адреса заменяются физическими во время выполнения программы в момент обращения по данному виртуальному адресу.
5. В каких случаях транслятор создает объектный код программы не в виртуальных, а в физических адресах?
6. Что такое виртуальная память? Какой из следующих методов распределения памяти может рассматриваться как частный случай использования виртуальной памяти:
   1) распределение фиксированными разделами;
   2) распределение динамическими разделами;
   3) страничное распределение;
   4) сегментное распределение;
   5) сегментно-страничное распределение.
7. Распределение памяти перемещаемыми разделами основано на применении процедуры сжатия. Имеет ли смысл использовать данную процедуру при страничном распределении? А при сегментном?
8. Поясните разные значения термина «свопинг».
9. Как величина файла свопинга влияет на производительность системы?
10. Почему размер страницы выбирается равным степени двойки? Можно ли принять такое же ограничение для сегмента?
11. На что влияет размер страницы? Каковы преимущества и недостатки большого размера страницы?
12. Пусть в некоторой программе, работающей в системе со страничной организацией памяти, произошло обращение по виртуальному адресу 012356_8. Преобразуйте этот адрес в физический, учитывая, что размер страницы равен 2^{14} байт и что таблица страниц данного процесса содержит следующий фрагмент.

Номер виртуальной страницы	Номер физической страницы
0000	0101
0001	0010

продолжение

Номер виртуальной страницы	Номер физической страницы
0010	0011
0011	0000

13. Где хранятся таблицы страниц и таблицы сегментов?
14. Чем определяется количество таблиц сегментов, имеющихся в операционной системе в произвольный момент времени?
15. Какие характеристики содержит таблица сегментов и таблица страниц при сегментно-страничной организации памяти?
16. Пусть ОС реализует выгрузку страниц на основе критерия «выгружается страница, которая не использовалась дольше остальных». Предложите алгоритм вычисления данного критерия, основанный на аппаратно устанавливаемых битах доступа.
17. Пусть в кэше хранятся данные, которые наиболее активно используются в последнее время. Каким образом система определяет, какие данные должны быть загружены в кэш?
18. Пусть программа циклически обрабатывает данные, то есть в некотором диапазоне адресов идет последовательное обращение к данным, а затем следует возврат в начало и т. д. В системе имеется кэш, объем которого меньше объема обрабатываемых программой данных. Какой алгоритм вытеснения данных из кэша в данном случае будет эффективнее:
    1) выгружаются данные, которые не использовались дольше остальных;
    2) выгружаются данные, выбранные случайным образом.
19. Почему загрузка и выгрузка данных из кэш-памяти производится блоками?
20. Как обеспечивается согласование данных в кэше с помощью методов обратной и сквозной записи?
21. Известно, что с помощью программных конвейеров данными могут обмениваться только процессы-родственники. В то же время все процессы в Unix являются родственниками, так как все они — потомки специального процесса, инициализирующего систему. Почему же механизм программных конвейеров не работает для двух произвольных процессов?

Глава 6

Аппаратная поддержка мультипрограммирования на примере процессора Pentium

Аппаратные средства поддержки мультипрограммирования имеются во всех современных процессорах. Несмотря на различия в реализации, для большинства типов процессоров эти средства имеют общие черты.

Это в полной мере относится и к рассматриваемому далее популярному семейству процессоров Intel: 80386, 80486, Pentium, Pentium Pro, Pentium II, Celeron и Pentium III, Pentium 4, Pentium D и Core 2. Большая часть моделей этих процессоров была 32-разрядной, а начиная с процессоров Pentium 4, поддерживается и 64-разрядная архитектура, к тому же последние 64-разрядные модели процессоров позволяют выполнять команды 32-разрядной архитектуры, то есть обладают обратной совместимостью.

Средства поддержки операционной системы во всех этих процессорах построены почти идентично, а отличия сводятся практически только к количеству разрядов команды и/или наличию нескольких дополнительных команд, не связанных напрямую с организацией вычислительного процесса. В данной главе средства аппаратной поддержки мультипрограммирования рассматриваются на примере 32-разрядного режима этих процессоров, которые здесь мы будем условно называть «процессоры Pentium».

Регистры процессора

В организации вычислительного процесса важную роль играют регистры процессора, которые в процессорах Pentium делятся на следующие группы:

- регистры общего назначения;
- регистры сегментов;

- указатель инструкций;
- регистр флагов;
- управляющие регистры;
- регистры системных адресов;
- регистры отладки и тестирования;
- регистры математического сопроцессора, выполняющего операции с плавающей точкой.

В процессоре Pentium имеется восемь 32-разрядных **регистров общего назначения**. Четыре из них, которые можно условно назвать A, B, C и D, используются для временного хранения операндов арифметических, логических и других команд. Программист может обращаться к этим регистрам как к единому целому, используя обозначения EAX, EBX, ECX, EDX, а также к некоторым их частям, как это показано на рис. 6.1. Здесь обозначение AL (L — Low) относится к первому, самому младшему байту регистра EAX, AH (H — High) — к следующему по старшинству байту, а AX означает оба младших байта регистра. Приставка E в обозначении этих регистров (а также некоторых других) образована от слова Extended (расширенный), указывая на то, что в прежних моделях процессоров Intel эти регистры были 16-разрядными, а затем их разрядность была увеличена до 32.

Рис. 6.1. Основные регистры процессора Pentium

Остальные четыре регистра общего назначения — ESI, EDI, EBP и ESP — предназначены для задания смещения адреса относительно начала некоторого сегмента данных. Эти регистры используются совместно с регистрами сегмен-

тов в системе адресации процессора Pentium для задания виртуального адреса, который затем с помощью таблиц страниц отображается на физический адрес.

Регистры сегментов CS, SS, DS, ES, FS и GS в защищенном режиме ссылаются на дескрипторы сегментов памяти — описатели, в которых содержатся такие параметры сегментов, как базовый адрес, размер сегмента, атрибуты защиты и некоторые другие. Регистры сегментов хранят 16-разрядное число, называемое селектором, в котором 12 старших разрядов представляет собой индекс в таблице дескрипторов сегментов, 1 разряд указывает, в какой из двух таблиц, GDT или LDT, находится дескриптор, а 3 разряда поля RPL хранят значение уровня привилегий запроса к данному сегменту. Регистр CS (Code Segment) предназначен для хранения индекса дескриптора кодового сегмента, регистр SS (Stack Segment) — дескриптора сегмента стека, а остальные регистры используются для указания на дескрипторы сегментов данных. Все регистры сегментов, кроме CS, программно доступны, то есть в них можно загрузить новое значение селектора соответствующей командой (например, `LDS`). Значение регистра CS изменяется при выполнении команд межсегментных вызовов `CALL` и переходов `JMP`, а также при переключении задач[1].

Указатель инструкций EIP содержит смещение адреса текущей инструкции, которое используется совместно с регистром CS для получения соответствующего виртуального адреса.

Регистр флагов EFLAGS содержит признаки, характеризующие результат выполнения операции, например флаги знака, нуля, переполнения, паритета, переноса и некоторые другие. Кроме того, здесь хранятся некоторые признаки, устанавливаемые и анализируемые механизмом прерываний, в частности флаг разрешения аппаратных прерываний IF.

В процессоре Pentium имеется пять **управляющих регистров** — CR0, CR1, CR2, CR3 и CR4, которые хранят признаки и данные, характеризующие общее состояния процессора (рис. 6.2).

Регистр CR0 содержит все основные признаки, существенно влияющие на работу процессора, такие как: признаки реального/защищенного режима работы, включения/выключения страничного механизма системы виртуальной памяти, а также признаки, влияющие на работу кэша и выполнение команд с плавающей точкой. Младшие два байта регистра CR0 имеют название «слово состояния машины» (Mashine State Word, MSW). Это название использовалось в процессоре 80286 для обозначения управляющего регистра, имевшего аналогичное назначение.

Регистр CR1 в настоящее время не используется (зарезервирован).

Регистр CR2 содержит **линейный виртуальный адрес**, который вызвал так называемый **страничный отказ** (отсутствие страницы в оперативной памяти или отказ из-за нарушения прав доступа).

[1] В этом разделе термин «задача» часто будет употребляться вместо равнозначного (и более распространенного) термина «процесс» в связи с тем, что именно этот термин выбрали в свое время разработчики процессоров Intel x86, и он фигурирует в названиях регистров и структур данных.

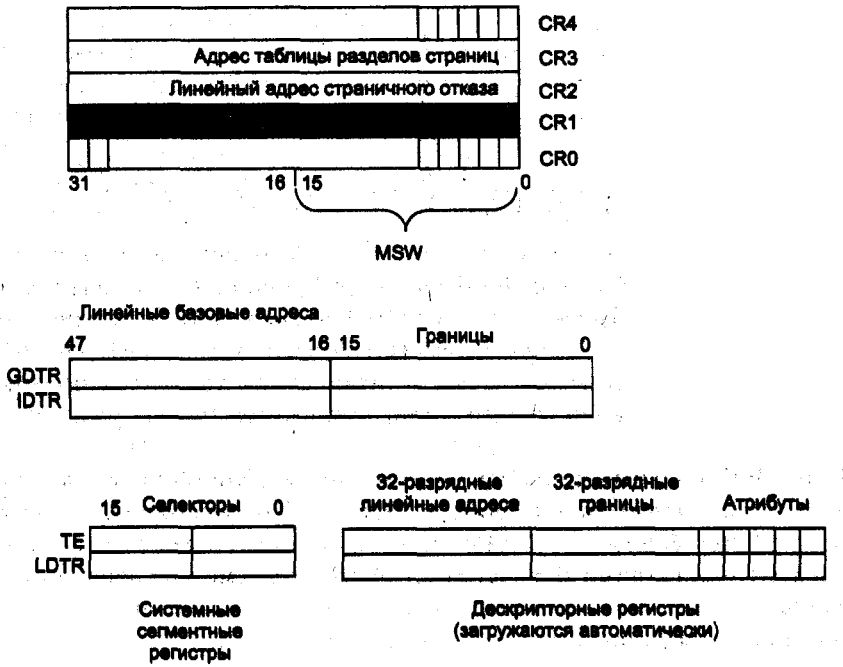


Рис. 6.2. Управляющие и системные регистры процессора Pentium

Регистр CR3 содержит физический адрес таблицы разделов, используемой страничным механизмом процессора.

Регистр CR4 хранит признаки, разрешающие работу так называемых архитектурных расширений, например, признак возможности использования страниц размером 4 Мбайта и т. п.

Регистры системных адресов содержат адреса важных системных таблиц и структур, используемых при управлении процессами и памятью. *Регистр GDTR* (Global Descriptor Table Register) содержит физический 32-разрядный адрес *глобальной* таблицы дескрипторов (Global Descriptor Table, GDT) сегментов памяти, образующих общую часть виртуального адресного пространства всех процессов. *Регистр IDTR* (Interrupt Descriptor Table Register) хранит физический 32-разрядный адрес таблицы дескрипторов прерываний (Interrupt Descriptor Table, IDT), используемой для вызова процедур обработки прерываний в защищенном режиме работы процессора. Помимо этих адресов, в регистрах GDTR и IDTR хранятся 16-разрядные лимиты, задающие ограничения на размер соответствующих таблиц.

Два 16-разрядных регистра хранят не физические адреса системных структур, а значения индексов дескрипторов этих структур в таблице GDT, что позволяет косвенно получить соответствующие физические адреса. *Регистр TR* (Task Register) содержит индекс дескриптора сегмента состояния задачи TSS. *Регистр LDTR* (Local Descriptor Table Register) содержит индекс дескриптора

сегмента локальной таблицы дескрипторов LDT сегментов памяти, образующих индивидуальную часть виртуального адресного пространства процесса.

Регистры отладки хранят значения точек останова, а **регистры тестирования** позволяют проверить корректность работы внутренних блоков процессора.

Привилегированные команды

Операционная система должна обладать исключительными полномочиями для того, чтобы играть роль арбитра в споре приложений за ресурсы компьютера, а также для того, чтобы защитить себя от некорректного поведения пользовательских процессов и обеспечить взаимную защиту ресурсов этих процессов друг от друга. Приложения ставятся в подчиненное положение по отношению к ОС за счет ограничений в выполнении некоторых команд, называемых привилегированными.

Привилегированные команды — это команды, которые могут быть выполнены только при определенном уровне привилегий текущего кода (Current Privilege Level, CPL). В процессорах Pentium поддерживается четыре уровня привилегий, от самого привилегированного нулевого до наименее привилегированного третьего[1].

К привилегированным командам относятся:

- команды для работы с управляющими регистрами CRn, а также для загрузки регистров системных адресов `GDTR`, `LDTR`, `IDTR` и `TR`;
- команда останова процессора `HALT`;
- команды запрета/разрешения маскируемых аппаратных прерываний `CLI`/`SLI`;
- команды ввода-вывода `IN`, `INS`, `OUT`, `OUTS`.

Первые две группы команд могут выполняться только при самом высшем уровне привилегий кода, то есть при CPL = 0. Для двух последних групп условия выполнения не требуют высшего уровня привилегий кода, а связаны с соотношением уровня привилегий ввода-вывода IOPL и уровня привилегий кода CPL — выполнение этих команд разрешено в том случае, если CPL ≤ IOPL. Поэтому две последние группы команд иногда называют *чувствительными*.

Средства поддержки сегментации памяти

Средства поддержки механизма виртуальной памяти в процессорах Pentium позволяют отображать виртуальное адресное пространство на физическую память, максимальный размер которой определяется разрядностью адресов при

[1] Механизм задания привилегий более детально рассмотрен далее в разделах, посвященных средствам управления памятью, так как уровни привилегий используются для определения не только допустимости выполнения той или иной команды, но и возможности доступа исполняемого кода к сегментам данных и кодов, включенным в виртуальное адресное пространство процесса.

работе с оперативной памятью. Поскольку здесь рассматривается 32-разрядная архитектура Pentium и, следовательно, 32-разрядные адреса оперативной памяти, то максимальный размер поддерживаемой физической памяти составляет 4 Гбайт.

Процессор может поддерживать две модели распределения памяти:

- сегментную модель,
- сегментно-страничную модель.

Средства сегментации образуют верхний уровень средств управления виртуальной памятью процессора Pentium, а средства страничной организации — нижний уровень. Это означает, что сегментные средства работают всегда, а средства страничной организации могут как включаться, так и выключаться путем установки *одноразрядного признака PE* (Paging Enable) в регистре CR0 процессора. В зависимости от того, включены ли средства страничной организации, изменяется смысл процедуры преобразования адресов, которая выполняется средствами сегментации. Сначала рассмотрим случай работы средств сегментации при отключенном механизме управления страницами.

Виртуальное адресное пространство

При работе процессора Pentium в сегментном режиме в распоряжении программиста имеется виртуальное адресное пространство, представляемое совокупностью сегментов.

Каждый сегмент виртуальной памяти процесса имеет описание, называемое **дескриптором сегмента**. Размер дескриптора сегмента равен 8 байт, и он содержит все характеристики сегмента, необходимые для проверки правильности доступа к нему и нахождения его в физическом адресном пространстве (рис. 6.3).

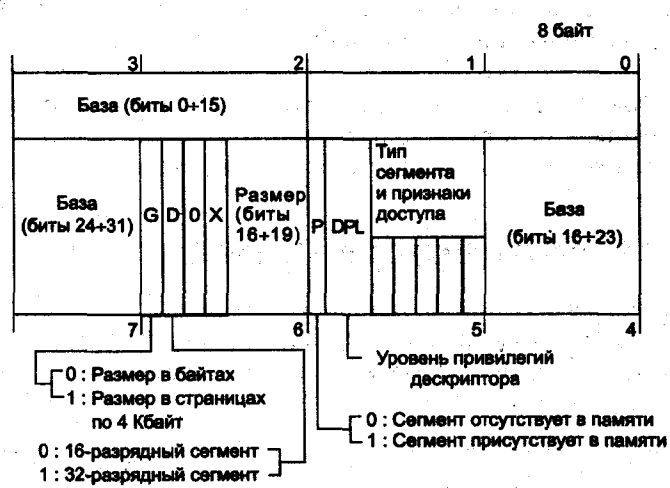


Рис. 6.3. Формат дескриптора сегмента данных или кода

Структура дескриптора, которая поддерживается в процессоре Pentium, сложилась исторически. Многое в ней связано с обеспечением совместимости с предыдущими процессорами семейства x86. Именно этим объясняется то, что базовый адрес сегмента представлен в дескрипторе в виде трех частей, а размер сегмента занимает два поля.

Ниже перечислены основные поля дескриптора.

- **База** — базовый адрес сегмента (32 бита).
- **Размер** — размер сегмента (24 бита).
- **G** (Granularity) — единица измерения размера сегмента, один бит. Если G = = 0, то размер задан в байтах, и тогда сегмент не может быть больше 64 Кбайт, если G = 1, то размер сегмента измеряется в страницах по 4 Кбайт.
- **Байт доступа** (пятый байт дескриптора) содержит информацию, которая используется для принятия решения о возможности или невозможности обращения к данному сегменту. *Бит P* (Present) определяет, находится ли соответствующий сегмент в данный момент в памяти (P = 1) или он выгружен на диск (P = 0). *Поле DPL* (Descriptor Privilege Level) содержит данные об уровне привилегий, необходимом для доступа к сегменту. Остальные 5 бит байта доступа зависят от типа сегмента и определяют способ, которым можно использовать данный сегмент (то есть читать, писать, выполнять). Различается три основных типа сегментов:
  - сегмент данных;
  - кодовый сегмент;
  - системный сегмент (GDT, TSS и т. п.).

Дескрипторы сегментов объединяются в таблицы. Процессор Pentium для управления памятью поддерживает два типа таблиц дескрипторов сегментов[1]:

- **глобальная таблица дескрипторов** (Global Descriptor Table, GDT) предназначена для описания сегментов операционной системы и общих сегментов для всех прикладных процессов, например сегментов межпроцессного взаимодействия;
- **локальная таблица дескрипторов** (Local Descriptor Table, LDT) содержит дескрипторы сегментов отдельного пользовательского процесса.

Таблица GDT одна, а таблиц LDT столько, сколько в системе выполняется задач (процессов). При этом в каждый момент времени операционной системой и аппаратными средствами процессора используется только одна из таблиц LDT, а именно та, которая соответствует выполняемому в данный момент пользовательскому процессу. Таблица GDT описывает общую часть виртуального адресного пространства процессов, а LDT — индивидуальную часть для каждого процесса. Таблицы GDT и LDT размещены в оперативной памяти в виде отдельных сегментов. Сегменты LDT и GDT содержат системные данные, поэтому

[1] Процессор Pentium поддерживает еще один тип таблицы дескрипторов — таблицу дескрипторов прерываний (Interrupt Descriptor Table, IDT). Эта таблица используется системой прерываний.

их дескрипторы хранятся в таблице GDT. Таким образом, таблица GDT наряду с записями о других сегментах содержит запись о самой себе, а также обо всех таблицах LDT.

В каждый момент времени в специальных регистрах GDTR и LDTR хранится информация о местоположении и размерах глобальной таблицы GDT и активной таблицы LDT соответственно. Регистр GDTR содержит 32-разрядный физический адрес начала сегмента GDT в памяти, а также 16-разрядный размер этого сегмента (рис. 6.4). Регистр LDTR указывает на расположение сегмента LDT в оперативной памяти косвенно — он содержит индекс дескриптора в таблице GDT, в котором содержится адрес таблицы LDT и ее размер.

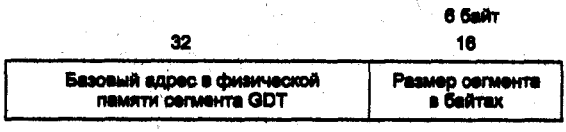


Рис. 6.4. Формат регистра GDTR

Процесс обращается к физической памяти по виртуальному адресу, представляющему собой пару (селектор, смещение). **Селектор** однозначно определяет виртуальный сегмент, к которому относится искомый адрес, то есть он может интерпретироваться как номер сегмента, а **смещение**, как это и следует из его названия, фиксирует положение искомого адреса относительно начала сегмента. Смещение задается в машинной инструкции, а селектор помещается в один из сегментных регистров процессора. Под смещение отводится 32 бита, что обеспечивает максимальный размер сегмента 4 Гбайт (2^{32}).

Селектор извлекается из одного из шести 16-разрядных сегментных регистров процессора (CS, SS, DS, ES, FS или GS) в зависимости от типа команды и стадии ее выполнения — выборка кода команды или данных. Например, при обращении к памяти во время выборки следующей команды используется селектор из сегментного регистра кода CS, а при записи результатов в сегмент данных процесса селекторы извлекаются из сегментных регистров данных DS или ES, если же данные записываются в стек по команде `PUSH`, то механизм виртуальной памяти извлекает селектор из сегментного регистра стека SS и т. п.

Селектор состоит из трех полей (рис. 6.5):

- **индекс** — задает последовательный номер дескриптора в таблице GDT или LDT (13 бит);
- **указатель типа** используемой таблицы дескрипторов: GDT или LDT (1 бит);
- **требуемый уровень привилегий** — RPL (2 бита), это поле используется механизмом защиты данных, который рассматривается далее.

Виртуальное адресное пространство процесса складывается из всех сегментов, описанных в общей для всех процессов таблице GDT, и сегментов, описанных в его собственной таблице LDT. Разрядность поля индекса определяет максимальное число глобальных и локальных сегментов процесса — по 8 Кбайт

(2^{13}) сегментов каждого типа, всего 16 Кбайт сегментов. Учтем также, что максимальный размер сегмента составляет 4 Гбайт. Отсюда следует, что:

> Каждый процесс при сегментной организации виртуальной памяти (без включения страничного механизма) может работать в виртуальном адресном пространстве размером *64 Тбайт*.

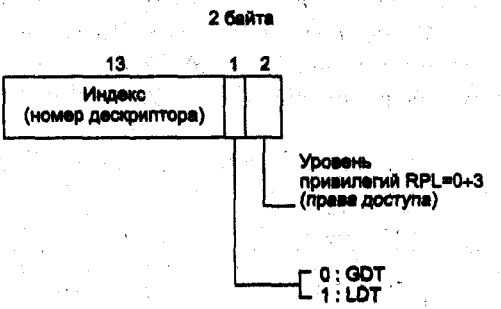


Рис. 6.5. Формат селектора

Каждый дескриптор в таблицах GDT и LDT имеет размер 8 байт, поэтому максимальный размер каждой из этих таблиц составляет 64 Кбайт (8 байт × × 8 Кбайт дескрипторов).

Из приведенного описания видно, что процессор Pentium обеспечивает поддержку работы ОС в двух отношениях:

- поддерживает работу подсистемы виртуальной памяти за счет быстрого аппаратного преобразования виртуального адреса в физический;
- обеспечивает взаимную защиту данных и кодов различных прикладных процессов.

Преобразование адресов

Теперь проследим, каким образом виртуальное пространство в 64 Тбайт отображается на физическое пространство размером в 4 Гбайт. Механизм отображения преобразует виртуальный адрес, который представлен селектором, находящимся в одном из сегментных регистров, и смещением, извлеченным из соответствующего поля машинной инструкции, в линейный физический адрес.

Рассмотрим сначала случай, когда виртуальный адрес относится к одному из сегментов, дескрипторы которых содержатся в таблице GDT (рис. 6.6).

1. Значение селектора указывает механизму преобразования адресов, что виртуальный адрес относится к сегменту, описываемому в таблице GDT. Местонахождение таблицы GDT система определяет из регистра GDTR, в котором хранится полный 32-разрядный базовый физический адрес таблицы. Процессор складывает базовый адрес таблицы, взятый из регистра GDTR, со сдвинутым на 3 разряда влево (умножение на 8 в соответствии с числом байтов в одном дескрипторе сегмента) значением поля индекса из селектора.

Результатом является физический адрес дескриптора сегмента, к которому относится заданный виртуальный адрес.

2. По вычисленному адресу процессор извлекает из памяти дескриптор нужного сегмента.
3. Выполняется проверка возможности выполнения заданной операции доступа по заданному виртуальному адресу:
   - сначала процессор определяет правильность адреса, сравнивая смещение, заданное в виртуальном адресе, с размером сегмента, извлеченным из регистра LDTR (в случае выхода адреса за границы сегмента происходит прерывание);
   - затем процессор проверяет права доступа задачи к данному сегменту памяти;
   - далее проверяется наличие сегмента в физической памяти (если бит P дескриптора равен 0, то есть сегмент отсутствует в физической памяти, то происходит прерывание).
4. Если все три условия выполнены, то доступ по заданному виртуальному адресу разрешен. Выполняется преобразование виртуального адреса в физический путем сложения базового адреса сегмента, извлеченного из дескриптора, и смещения, заданного в инструкции. Выполняется заданная операция над элементом физической памяти по этому адресу.

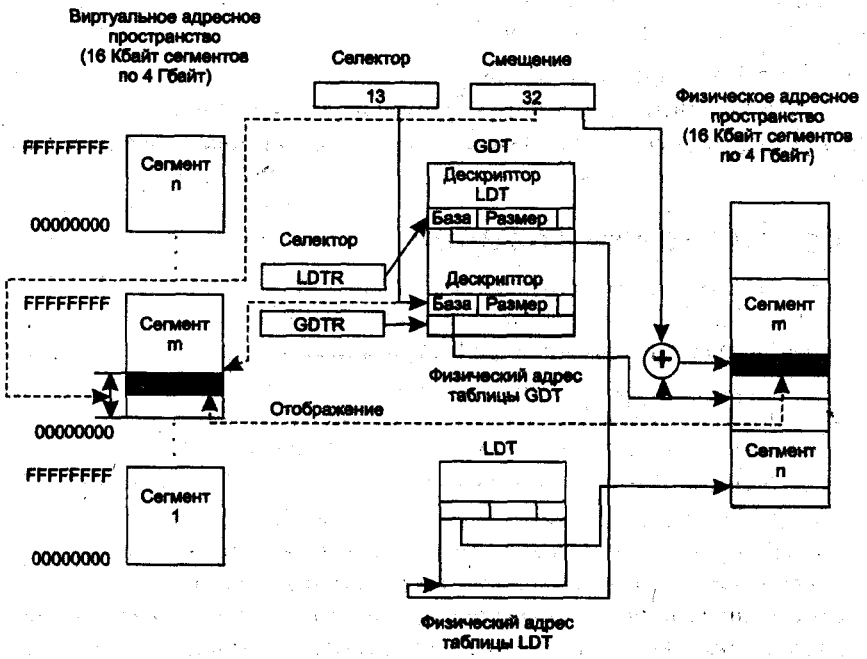


Рис. 6.6. Механизм преобразования виртуального адреса в физический при работе процессора в сегментном режиме

В случае когда селектор в виртуальном адресе указывает не на таблицу GDT, а на таблицу LDT, процедура вычисления физического адреса несколько усложняется. Это связано с тем, что регистр LDTR, в отличие от GDTR, указывает на размещение таблицы сегментов не прямо, а косвенно. В LDTR содержится индекс дескриптора сегмента LDT. Поэтому в процедуре преобразования адресов появляется дополнительный этап — определение базового адреса таблицы LDT. На основании базового адреса таблицы GDT, взятого из регистра GDTR, и селектора, взятого из регистра LDTR, вычисляется смещение в таблице GDT, которое и является адресом дескриптора сегмента LDT.

Из дескриптора извлекается базовый адрес таблицы LDT, и с этого момента работа механизма отображения полностью аналогична описанному ранее преобразованию виртуального адреса с помощью таблицы GDT: на основании базового адреса таблицы LDT и селектора задачи, заданного в одном из сегментных регистров, вычисляется смещение в таблице LDT и определяется базовый адрес дескриптора искомого сегмента. Из этого дескриптора извлекается базовый адрес сегмента, который складывается со смещением из виртуального адреса, что и дает в результате искомый физический адрес.

Таким образом, для использования сегментного механизма процессора Pentium операционной системе необходимо выполнить следующие действия:

1. Сформировать таблицы GDT и LDT.
2. Загрузить их в память (для начала достаточно загрузить только таблицу GDT).
3. Загрузить указатели на эти таблицы в регистры GDTR и LDTR.
4. Выключить страничную поддержку.

Операционная система может отказаться от использования средств сегментации процессора Pentium, в таком случае ей достаточно назначить каждому процессу только по одному сегменту и занести в соответствующие таблицы LDT по одному дескриптору. Виртуальное адресное пространство задачи будет состоять из одного сегмента длиной максимум в 4 Гбайт. Поскольку выгрузка процессов на диск будет осуществляться целиком, механизм виртуальной памяти вырождается в механизм свопинга.

Защита данных при сегментной организации памяти

Процессор Pentium при работе в сегментном режиме предоставляет операционной системе следующие средства, направленные на обеспечение защиты процессов друг от друга.

- *Для каждого процесса поддерживается отдельная таблица дескрипторов сегментов LDT.* Эта таблица формируется операционной системой на этапе создания процесса. После активизации процесса в регистр LDTR заносится адрес (селектор) его таблицы LDT. Тем самым система делает недоступным для процесса локальные сегменты других процессов, описанные в их таблицах LDT.

- Вместе с тем, таблица GDT, в которой хранятся дескрипторы сегментов операционной системы, а также сегменты, используемые несколькими процессами совместно, доступна всем процессам, а, следовательно, не защищена от их несанкционированного вмешательства. Для решения этой проблемы в процессоре Pentium предусмотрена *поддержка системы безопасности на основе привилегий*. Каждый сегмент памяти наделяется атрибутами безопасности, которые характеризуют степень защищенности данного сегмента. Процессы также наделяются атрибутами безопасности, которые в этом случае характеризуют степень привилегированности процесса. Уровень привилегий процесса определяется уровнем привилегий его кодового сегмента. Доступ к сегменту регулируется в зависимости от его защищенности и уровня привилегированности запроса. Это позволяет защитить от несанкционированного доступа сегменты GDT, а также обеспечить дифференцированный доступ к локальным сегментам процесса.
- Само по себе наличие отдельных таблиц дескрипторов для каждого процесса еще не обеспечивает надежной защиты процессов друг от друга. Действительно, что мешает одному пользовательскому процессу изменить содержимое регистра LDTR так, чтобы он указывал на таблицу другого процесса? Аналогично, никакие значения атрибутов безопасности не смогут защитить от несанкционированного доступа сегменты, описанные в таблице GDT, если процесс имеет возможность по своей инициативе устанавливать желаемый уровень привилегий для себя. Поэтому необходимым элементом защиты является наличие аппаратных *ограничений в наборе инструкций*, разрешенных для выполнения процессу. Некоторые инструкции, имеющие критически важное значение для функционирования системы, такие, например, как останов процессора, загрузка регистра GDTR, загрузка регистра LDTR, являются привилегированными как раз по этой причине — процесс может выполнять их, только обладая наивысшим уровнем привилегий.
- Еще одним механизмом защиты, поддерживаемым в процессоре Pentium, является *ограничение на способ использования сегмента*. В зависимости от того, к какому типу относится сегмент — сегмент данных, кодовый сегмент или системный сегмент, — некоторые действия по отношению к нему могут быть запрещены. Например, в кодовый сегмент нельзя записывать какие-либо данные, а сегменту данных нельзя передать управление, даже если загрузить его селектор в регистр CS.

В процессоре Pentium используется **мандатный** способ определения прав доступа к сегментам памяти (называемый также механизмом колец защиты), при котором имеется несколько уровней привилегий, и объекты каждого уровня имеют доступ ко всем объектам равного уровня или более низких уровней, но не имеет доступа к объектам более высоких уровней. В процессоре Pentium существуют 4 уровня привилегий, от нулевого, который является самым высоким, до третьего — самого низкого (рис. 6.7). Очевидно, что операционная система может использовать механизм колец защиты по своему усмотрению. Однако предполагается, что нулевой уровень доступен для ядра операционной

системы, третий уровень — для прикладных программ, а промежуточные уровни — для утилит и подсистем операционной системы, менее привилегированных, чем ядро.

Рис. 6.7. Кольца защиты

Система защиты манипулирует несколькими переменными, характеризующими **уровень привилегий**:

- **DPL** (Descriptor Privilege Level) — уровень привилегий дескриптора, задается полем DPL в дескрипторе сегмента;
- **RPL** (Requested Privilege Level) — запрашиваемый уровень привилегий, задается полем RPL селектора сегмента;
- **CPL** (Current Privilege Level) — текущий уровень привилегий выполняемого кода, задается полем RPL селектора кодового сегмента;
- **EPL** (Effective Privilege Level) — эффективный уровень привилегий запроса.

Под запросом здесь понимается любое обращение к памяти независимо от того, произошло оно при выполнении кода, оформленного в виде процесса, или вне рамок процесса, как это бывает при выполнении кодов операционной системы.

Поле уровня привилегий является частью двух информационных структур — дескриптора и селектора. В том и другом случаях оно задается двумя битами: 00, 01, 10 и 11, характеризуя четыре степени привилегированности, от самой низкой 11 до самой высокой 00.

Уровни привилегий назначаются дескрипторам и селекторам. Во время загрузки операционной системы в память, а также при создании новых процессов операционная система назначает процессу сегменты кода и данных, генерирует дескрипторы этих сегментов и помещает их в таблицу GDT или LDT. Конкретные значения уровней привилегий DPL и RPL задаются операционной системой и транслятором либо по умолчанию, либо на основании указаний программиста. Значения DPL и RPL определяют возможности создаваемого процесса.

Уровень привилегий дескриптора DPL является в некотором смысле первичной характеристикой, которая «переносится» на соответствующие сегменты и запросы. Сегмент обладает тем уровнем привилегий, который записан в поле DPL его дескриптора. DPL определяет степень защищенности сегмента. Уровень привилегий сегмента данных учитывается системой защиты, когда она принимает решение о возможности выполнения для этого сегмента чтения или записи. Уровень привилегий кодового сегмента используется системой защиты при проверке возможности чтения или выполнения кода для данного сегмента.

Уровень привилегий кодового сегмента определяет не только степень защищенности этого сегмента, но и текущий уровень привилегий CPL всех запросов к памяти (на чтение, запись или выполнение), которые возникнут при выполнении этого кодового сегмента. Другими словами, уровень привилегий кодового сегмента DPL характеризует текущий уровень привилегий CPL выполняемого кода[1]. При запуске кода на выполнение значение DPL из дескриптора копируется в поле RPL селектора кодового сегмента, загружаемого в регистр сегмента команд CS. Значение поля RPL кодового сегмента собственно и является текущим уровнем привилегий выполняемого кода, то есть уровнем CPL.

От того, какой уровень привилегий имеет выполняемый код, зависят не только возможности его доступа к сегментам и дескрипторам, но и разрешенный ему набор инструкций.

Во время приостановки процесса его текущий уровень привилегий сохраняется в контексте, роль которого в процессоре Pentium играет системный сегмент состояния задачи (Task State Segment, TSS). Если какой-либо процесс имеет несколько кодовых сегментов с разными уровнями привилегий, то поле RPL регистра CS позволяет узнать значение текущего уровня привилегий процесса. Заметим, что пользовательский процесс не может изменить значение поля привилегий в дескрипторе, так как необходимые для этого инструкции ему недоступны. В дальнейшем уровень привилегий процесса может измениться только в случае передачи управления другому кодовому сегменту путем использования особого дескриптора — шлюза.

Контроль доступа процесса к сегментам данных осуществляется на основе сопоставления эффективного уровня привилегий EPL запроса и уровня привилегий DPL дескриптора сегмента данных. Эффективный уровень привилегий учитывает не только значение CPL, но и значение запрашиваемого уровня привилегий для конкретного сегмента, к которому выполняется обращение. Перед тем как обратиться к сегменту данных, выполняемый код загружает селектор, указывающий на этот сегмент, в один из регистров сегментов данных DS, ES, FS или GS. Значение поля RPL данного селектора задает уровень запрашивае-

[1] В литературе, посвященной этому вопросу, уровни привилегий CPL и EPL часто называют уровнями привилегий задачи (Task Privilege). Но поскольку понятие «задача» имеет совершенно конкретный смысл, являясь синонимом понятия «процесс», использование его в данном контексте сужает область действия механизма защиты. Как уже было сказано, контроль доступа осуществляется не только для кодов, оформленных в виде процесса, но и для кодов, выполняющихся вне рамок процесса, например процедур обработки прерываний.

мых привилегий. Эффективный уровень привилегий выполняемого кода EPL определяется как максимальное (то есть худшее) из значений текущего и запрашиваемого уровней привилегий:

$$EPL = \max\{CPL,RPL\}$$

Выполняемый код может получить доступ к сегменту данных для операций чтения или записи, если его эффективный уровень привилегий не ниже (а в арифметическом смысле «не больше») уровня привилегий дескриптора этого сегмента:

$$\max\{CPL,RPL\} \le DPL$$

Уровень привилегий дескриптора DPL определяет степень защищенности сегмента. Значение DPL говорит о том, каким эффективным уровнем привилегий должен обладать запрос, чтобы получить доступ к данному сегменту. Например, если дескриптор имеет уровень DPL равный, 2, то к нему разрешено обращаться всем процессам, имеющим уровень EPL, равный 0, 1 или 2, а для процессов с уровнем EPL, равным 3, доступ запрещен.

ПРИМЕЧАНИЕ Существует принципиальное различие в использовании полей RPL в селекторах, указывающих на кодовые сегменты и сегменты данных. Содержимое поля RPL из селектора, загруженного в регистр CS кодового сегмента, характеризует текущий уровень привилегий CPL для кода, выполняемого из этого сегмента; значение его никак не связано с сегментами данных, к которым может происходить обращение из этого кодового сегмента. Содержимое поля RPL из селектора, загруженного в один из регистров DS, ES, FS или GS сегментов данных, определяет некую «поправку» к текущему уровню привилегий выполняемого кода, вносимую при доступе к сегменту данных, на который указывает этот селектор.

Правило вычисления эффективного уровня привилегий показывает, что он не может быть выше уровня привилегий кодового сегмента. Поэтому загрузка в регистр DS (или любой другой сегментный регистр данных) селектора с высоким уровнем привилегий (например, RPL = 0) при низком значении уровня CPL (например, 3) ничего нового не даст — если сегмент данных, на который указывает DS, имеет высокий уровень привилегий (например, 1), то в доступе к нему будет отказано (так как уровень EPL будет равен max {3,0}, то есть 3). По этой причине команды загрузки сегментных регистров не являются привилегированными.

Если запрос обращен к сегменту данных, обладающему более высоким, чем EPL, уровнем привилегий, то происходит прерывание.

Задавая в поле RPL селекторов, ссылающихся на сегменты данных, различные значения, программист может управлять доступом выполняемого кода к этим сегментам. Так, при RPL = 0 эффективный уровень привилегий будет всецело определяться только текущим уровнем привилегий CPL. А если поместить в селектор значение RPL = 3, то это будет означать снижение эффективного

уровня привилегий до самого низкого значения, при котором процессу разрешен доступ только к наименее защищенным сегментам с DPL = 3.

Контроль доступа процесса к сегменту стека позволяет предотвратить обращение низкоуровневого кода к данным, выработанным высокоуровневым кодом и помещенным в стек, например, к локальным переменным процедуры. Доступ к сегменту стека разрешается только в том случае, когда уровень EPL кода совпадает с уровнем DPL сегмента стека, то есть коду разрешается работать только со стеком своего уровня привилегий. Использование одного и того же стека для процедур разного уровня привилегий может привести к тому, что низкоуровневая процедура, получив управление после возврата из вызванной ею высокоуровневой процедуры (обратная последовательность вызова процедур в процессоре Pentium запрещена, о чем рассказывается далее), может прочитать из стекового сегмента записываемые туда во время работы высокоуровневой процедуры данные. Так как в ходе выполнения процесса уровень привилегий его кода может измениться, то для каждого уровня привилегий используется отдельный сегмент стека.

Контроль доступа процесса к кодовому сегменту производится путем сопоставления уровня привилегий дескриптора этого кодового сегмента DPL с текущим уровнем привилегий выполняемого кода CPL. В зависимости от того, какой способ обращения к кодовому сегменту используется, выполняется внутрисегментная или межсегментная передача управления и вызывается *подчиненный* или *неподчиненный* сегмент, по-разному формулируются правила контроля доступа. Эти вопросы рассмотрены в разделе «Средства вызова процедур и задач».

Осуществляя контроль доступа к сегменту, аппаратура процессора учитывает не только уровень привилегий, но и «легитимность» способа использования данного сегмента.

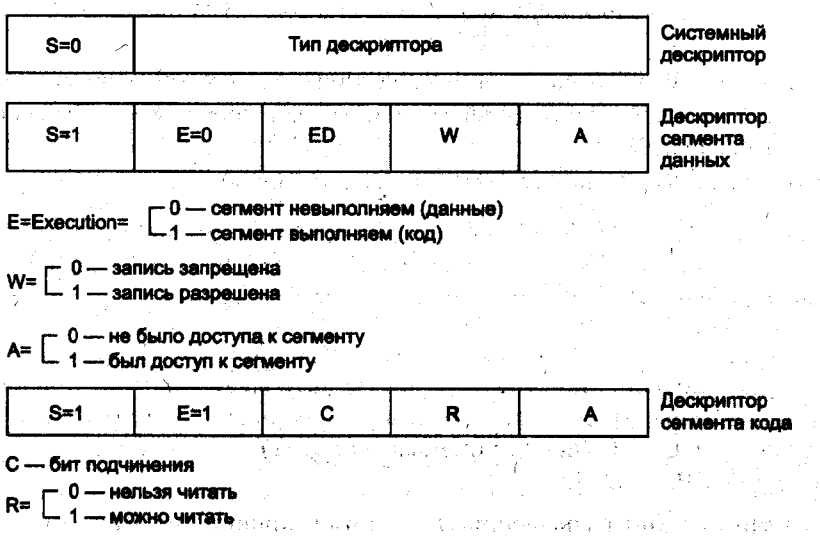


Рис. 6.8. Признаки, задающие тип сегмента и права доступа

Способ, с помощью которого разрешено осуществлять доступ к сегменту того или иного типа, определяется несколькими битами байта доступа дескриптора сегмента. Как уже было сказано, старший (седьмой) бит байта доступа является признаком присутствия сегмента в памяти (P), а шестой и пятый биты отведены для хранения уровня привилегий DPL. На рис. 6.8 показаны оставшиеся 5 бит байта доступа, смысл которых зависит от типа сегмента.

Поле S занимает 1 бит и определяет, является ли сегмент системным (S = 0) или сегментом кода или данных (S = 1).

Системные сегменты предназначены для хранения служебной информации операционной системы. Примерами системных сегментов являются сегменты, хранящие таблицы LDT, или рассматриваемые далее сегменты состояния задачи TSS. Конкретный тип системного сегмента указывается в четырех младших битах байта доступа, например, если это поле содержит значение 2, это означает, что данный дескриптор описывает сегмент LDT.

Признак E позволяет отличить сегмент данных (E = 0) от сегмента кода (E = 1).

Для сегмента данных определяются следующие поля:

- ED (Expand Down) — направления распространения сегмента (ED = 0 для обычного сегмента данных, распространяющегося в сторону увеличения адресов, ED = 1 для стекового сегмента данных, распространяющегося в сторону уменьшения адресов);
- W (Writeable) — бит разрешения записи в сегмент (при W = 1 запись разрешена, при W = 0 — запрещена);
- A (Accessed) — признак обращения к сегменту (1 означает, что после очистки этого поля к сегменту было обращение по чтению или записи, это поле может использоваться операционной системой для реализации ее стратегии замены сегментов в оперативной памяти).

Если признак W запрещает запись в сегмент данных, то разрешается только чтение сегмента. Выполнение сегмента данных запрещено всегда независимо от значения признака W.

Для сегмента кода используются следующие признаки:

- C (Conforming) — бит подчинения, определяет возможность вызова кода на выполнение из других сегментов; при C = 1 (**подчиненный сегмент**) сегмент может выполняться в том случае, если текущий уровень привилегий вызывающего процесса CPL не выше уровня привилегий DPL данного кодового сегмента, то есть в арифметическом смысле CPL ≥ DPL; после передачи управления новый кодовый сегмент начинает выполняться с уровнем привилегий вызвавшего сегмента, то есть с более низкими или теми же привилегиями; при C = 0 (**неподчиненный сегмент**) код может быть выполнен только при CPL = DPL;
- R (Readable) — бит разрешения (R = 1) или запрета (R = 0) чтения из кодового сегмента;

- A (Accessed) — признак обращения, имеет смысл, аналогичный полю A сегмента данных.

Если признак R запрещает чтение кодового сегмента, то разрешается только его выполнение. Запись в кодовый сегмент запрещена всегда независимо от значения признака R.

Выполнение многих операций в процессоре Pentium сопровождается проверкой их допустимости для данного типа сегмента.

Первый этап проверки выполняется при загрузке селекторов в сегментные регистры. Так, диагностируется ошибка, если в сегментный регистр CS загружается селектор, ссылающийся на дескриптор сегмента данных, и наоборот, если в регистр сегмента данных DS загружается селектор, указывающий на дескриптор кодового сегмента. Ошибка возникает также, если в регистр стекового сегмента загружается селектор, ссылающийся на дескриптор сегмента данных, который допускает только чтение.

Вторым этапом является проверка ссылок операндов во время выполнения команд записи или чтения. Прерывания происходят в следующих случаях:

- делается попытка прочитать данные из кодового сегмента, который допускает только выполнение;
- делается попытка записать данные по адресу, принадлежащему кодовому сегменту;
- делается попытка записать данные в сегмент данных, который допускает только чтение.

Все средства защиты, используемые при работе процессора Pentium в сегментном режиме, полностью применимы и при включении страничного механизма. В этом случае они дополняются возможностями защиты страниц, которые рассматриваются позже.

Сегментно-страничный механизм

Включение страничного механизма происходит, если в регистре управления CR0 самый старший бит PG установлен в единицу. При включенной системе управления страницами параллельно продолжает работать и описанный ранее сегментный механизм, однако, как будет показано далее, смысл его работы меняется.

> Виртуальное адресное пространство процесса при сегментно-страничном режиме работы процессора ограничивается размером *4 Гбайт*.

В этом пространстве определены виртуальные сегменты процесса (рис. 6.9). Так как теперь все виртуальные сегменты разделяют одно виртуальное адресное пространство, то возможно их наложение, поскольку процессор не контролирует такие ситуации, оставляя эту проблему операционной системе.

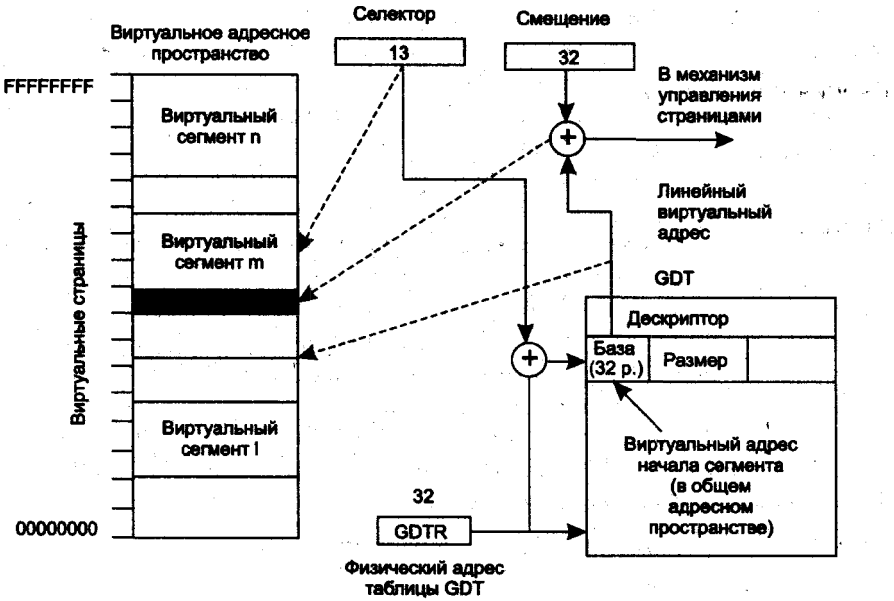


Рис. 6.9. Работа сегментного механизма в сегментно-страничном режиме

Для реализации механизма управления страницами как физическое, так и виртуальное адресные пространства разбиваются на страницы размером 4 Кбайт (начиная с модели Pentium, в процессорах Intel существует возможность использования страниц и по 4 Мбайт, но дальнейшее изложение ориентируется на традиционный размер страницы в 4 Кбайт). Всего в виртуальном адресном пространстве в сегментно-страничном режиме насчитывается 1 Мбайт (2^{20}) страниц. Несмотря на наличие нескольких виртуальных сегментов, все виртуальное адресное пространство задачи имеет общее разбиение на страницы, так что нумерация виртуальных страниц сквозная.

Виртуальный адрес по-прежнему представляет собой пару: **селектор**, который определяет номер виртуального сегмента, и **смещение** внутри этого сегмента. Преобразование виртуального адреса выполняется в два этапа: сначала работает сегментный механизм, а затем результат его работы поступает на вход страничного механизма, который и вычисляет искомый физический адрес.

Работа сегментного механизма в данном случае во многом повторяет его работу при отключенном страничном механизме. На основании значения индекса в селекторе выбирается нужный дескриптор в таблице GDT или LDT. Из дескриптора извлекается базовый адрес сегмента и складывается со смещением. Дескрипторы и таблицы имеют ту же структуру. Однако имеется и принципиальное отличие, оно состоит в интерпретации содержимого поля базового адреса в дескрипторах сегментов. Если раньше дескриптор сегмента содержал базовый

адрес сегмента в физической памяти и при сложении этого адреса со смещением из виртуального адреса получался *физический адрес*, то теперь дескриптор содержит базовый адрес сегмента в виртуальном адресном пространстве, и в результате его сложения со смещением получается *линейный виртуальный адрес*.

Результирующий линейный 32-разрядный виртуальный адрес передается страничному механизму для дальнейшего преобразования. Исходя из того, что размер страницы равен 4 Кбайт (2^{12}), в адресе можно легко выделить номер виртуальной страницы (старшие 20 разрядов) и смещение в странице (младшие 12 разрядов). Как известно, для отображения виртуальной страницы на физическую достаточно построить таблицу страниц, каждый элемент которой — дескриптор виртуальной страницы — содержал бы номер соответствующей ей физической страницы и ее атрибуты. В процессоре Pentium так и сделано, и структура дескриптора страницы показана на рис. 6.10. Двадцать разрядов номера страницы могут интерпретироваться и как базовый адрес страницы в памяти, который необходимо дополнить 12 нулями, так как младшие 12 разрядов базового адреса страницы всегда равны нулю. Помимо номера страницы, дескриптор страницы содержит также следующие поля, близкие по смыслу соответствующим полям дескриптора сегмента:

- P — бит присутствия страницы в физической памяти;
- W — бит разрешения записи в страницу;
- U — бит пользователь/супервизор;
- A — признак имевшего место доступа к странице;
- D — признак модификации содержимого страницы;
- PWT и PCD — управляют механизмом кэширования страниц (введены, начиная с процессора i486);
- AVL — резерв для нужд операционной системы (AVaiLable for use).

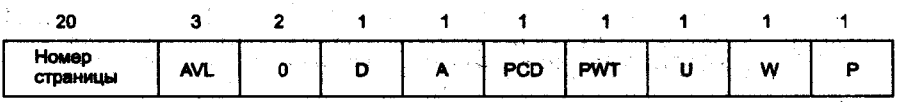


Рис. 6.10. Формат дескриптора страницы

При небольшом размере страницы процессора Pentium относительно размеров адресных пространств таблица страниц должна занимать в памяти весьма значительное место:

$$4 \text{ байт} \times 1 \text{ Мбайт} = 4 \text{ Мбайт}$$

Это слишком много для нынешних моделей персональных компьютеров, поэтому в процессоре Pentium вся таблица страниц делится на части — разделы по 1024 дескриптора. Размер раздела выбран так, чтобы один раздел занимал одну физическую страницу (1024 × 4 байт = 4 Кбайт). Таким образом, таблица страниц делится на 1024 раздела.

Чтобы постоянно не хранить в памяти все разделы, создается **таблица разделов (каталог страниц)**, состоящая из дескрипторов разделов, которые имеют такую же структуру, что и дескрипторы страниц. Максимальный размер таблицы разделов составляет 4 Кбайт, то есть одна страница. Виртуальные страницы, содержащие разделы, как и все остальные страницы, могут выгружаться на диск. Виртуальная страница, хранящая таблицу разделов, всегда находится в физической памяти, и номер ее физической страницы указан в специальном управляющем регистре CR3 процессора.

Преобразование линейного виртуального адреса в физический происходит следующим образом (рис. 6.11).

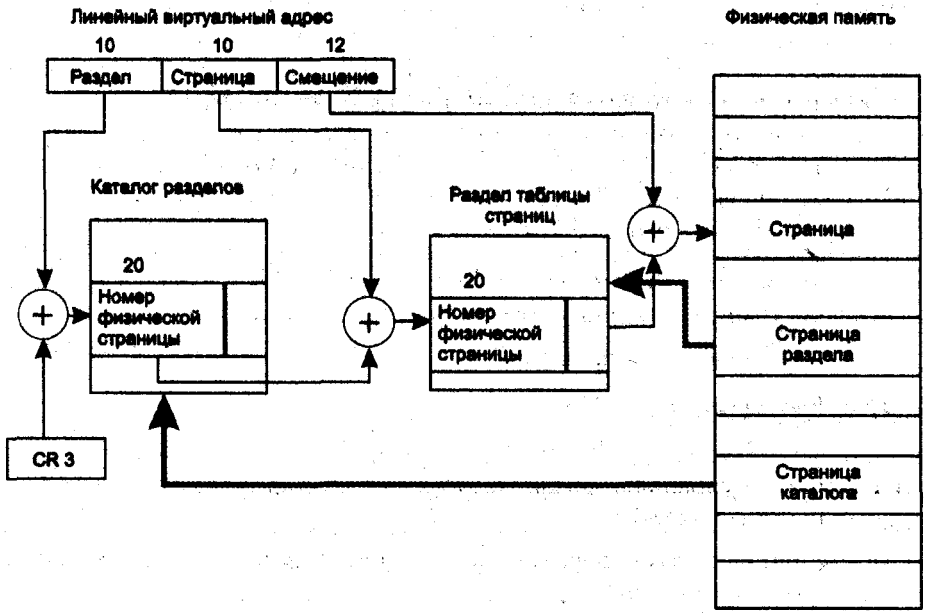


Рис. 6.11. Преобразование линейного виртуального адреса в физический адрес

Поле номера виртуальной страницы (старшие 20 разрядов) делится на две равные части по 10 разрядов — поле номера раздела и поле номера страницы в разделе. На основании заданного в регистре CR3 номера физической страницы, хранящей таблицу разделов, и смещения в этой странице, задаваемого полем номера раздела, процессор находит дескриптор виртуальной страницы раздела. В соответствии с атрибутами этого дескриптора определяются права доступа к странице, а также наличие ее в физической памяти. Если страницы нет в оперативной памяти, то происходит прерывание, в результате которого операционная система должна выполнить загрузку требуемой страницы в память. После того как страница (содержащая нужный раздел) загружена, из нее извлекается дескриптор страницы данных, номер которой указан в линейном виртуальном адресе. И наконец, на основании базового адреса страницы, полученного

из дескриптора, и смещения, заданного в линейном виртуальном адресе, вычисляется искомый физический адрес.

Таким образом, при доступе к странице в процессоре используется двухуровневая схема адресации страниц, которая хотя и замедляет преобразование, но позволяет задействовать страничный механизм для таблицы страниц, что существенно уменьшает объем физической памяти, требуемой для ее хранения. Для ускорения преобразования адресов в блоке управления страницами применяется ассоциативная память, в которой кэшируются 32 дескриптора активно используемых страниц, что позволяет по номеру виртуальной страницы быстро извлекать номер физической страницы без обращения к таблицам разделов и страниц.

Средства вызова процедур и задач

Операционная система, как однозадачная, так и многозадачная, должна предоставлять задачам средства вызова процедур операционной системы (библиотечных процедур). Она должна также иметь средства для запуска задач, а при многозадачной работе — средства быстрого переключения с задачи на задачу. Вызов процедуры отличается от запуска задачи тем, что в первом случае виртуальное адресное пространство задачи остается тем же (таблица LDT остается прежней), а при вызове задачи это адресное пространство полностью меняется.

Вызов процедур

Вызов процедуры без смены кодового сегмента в защищенном режиме процессора Pentium производится обычным образом с помощью команд `JMP` и `CALL`.

Для вызова процедуры, код которой находится в другом сегменте (этот сегмент может принадлежать другому программному модулю приложения, библиотеке, другой задаче или операционной системе), процессор Pentium предоставляет несколько способов, причем во всех используется защита, основанная на уровнях привилегий.

Прямой вызов процедуры из неподчиненного сегмента состоит в непосредственном указании в поле команды `JMP` или `CALL` селектора, который указывает на дескриптор нового кодового сегмента. Этот сегмент содержит код вызываемой процедуры. Базовый адрес сегмента, содержащийся в дескрипторе, и смещение, задаваемое в команде `JMP` или `CALL`, определяют начальный адрес вызываемой процедуры.

Схема такого вызова приведена на рис. 6.12. Здесь и далее показан только этап получения линейного адреса в виртуальном пространстве, а следующий этап (подразумевается, что механизм поддержки страниц включен) преобразования этого адреса в номер физической страницы опущен, так как он не содержит ничего нового по сравнению с рассмотренным ранее доступом к сегменту данных. Разрешение вызова происходит в зависимости от значения поля C в дескрипторе сегмента вызываемого кода.

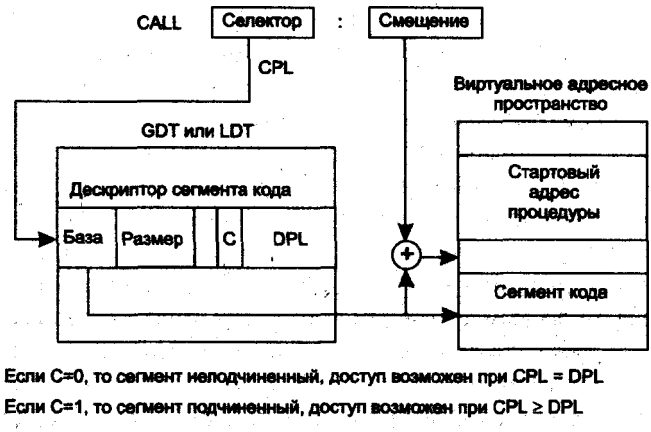


Рис. 6.12. Прямой вызов процедуры

При C = 0 вызываемый сегмент не является подчиненным, и вызов разрешается, *только если* уровень привилегий вызывающего кода совпадает с уровнем привилегий вызываемого сегмента (CPL = DPL). Случаи, когда вызываемый код имеет более низкий или более высокий уровень привилегий, являются запрещенными.

В первом случае запрет является естественным средством защиты ОС от вызова произвольных привилегированных процедур из пользовательских программ. Очевидно, что система защиты ОС не должна быть абсолютной, так как приложения должны иметь возможность обращаться к определенным процедурам ОС, реализующим системные вызовы. Так как с помощью неподчиненных кодовых сегментов этого сделать нельзя, то для решения проблемы применяют другие способы, используя подчиненные сегменты и шлюзы вызовов, рассматриваемые далее.

ПРИМЕЧАНИЕ При обсуждении вопросов доступа к сегментам данных с уровнем привилегий сегмента DPL сравнивалось значение эффективного уровня привилегий EPL. При доступе же к кодовому сегменту с его DPL сравнивается значение CPL. Внимательный читатель мог заметить, что здесь нет никакого противоречия или особого случая, потому что при доступе к кодовому сегменту значение EPL всегда равно CPL: EPL = = max{RPL,CPL}. А так как CPL — это и есть RPL кодового сегмента, то EPL = max{CPL,CPL} = CPL.

Во втором случае запрет звучит так: код не может вызвать другой код, если у последнего привилегии ниже. Это на первый взгляд кажется странным. Действительно, в соответствии с этим правилом операционная система не может вызывать код приложения из неподчиненного сегмента, хотя ее уровень привилегий выше, чем у кода приложения. Однако, по сути, этот запрет является проявлением общего иерархического принципа построения системы защиты процессоров Pentium — привилегированный код не может пользоваться ненадежными

в общем случае процедурами с более низким уровнем привилегий, и этот принцип соблюдается для всех способов вызова процедур, а не только для данного.

Рассмотрим теперь *прямой вызов процедуры из подчиненного сегмента*. Процессор должен поддерживать способ безопасного вызова модулей ОС, чтобы пользовательские программы могли получать доступ к службам ОС, например, выполнять ввод-вывод с помощью соответствующих системных вызовов. Для реализации этой возможности существует несколько способов и одним из них является размещение процедур ОС в подчиненном сегменте (C = 1). Подчиненный сегмент можно вызывать путем указания его селектора в командах `CALL` или `JMP` из кода программ с равным или более низким уровнем привилегий (CPL ≥ DPL). Но нужно иметь в виду, что вызываемый код будет в этом случае выполняться с привилегиями вызывающей программы. Например, если код ОС, хранящийся в сегменте с уровнем привилегий 0, будет вызван из пользовательского приложения с уровнем привилегий 3, то процедура ОС унаследует привилегии пользовательской программы и возможности этой процедуры по доступу к системным данным будут весьма ограничены. Тем не менее выполнить действия над пользовательскими данными вызванная таким способом процедура ОС сможет.

Остается рассмотреть *косвенный вызов процедуры через шлюз*. Очевидно, что оба рассмотренных способа вызова процедур не подходят для реализации системных вызовов. Первый способ в принципе не позволяет вызвать из пользовательской программы с третьим уровнем привилегий процедуру операционной системы, находящуюся в неподчиненном сегменте и имеющую более высокий уровень привилегий. С помощью второго способа могут быть вызваны процедуры ОС, находящиеся в подчиненном сегменте, однако они будут выполняться с пользовательским уровнем привилегий и не смогут обрабатывать системные данные, что нужно для большинства системных вызовов. Поэтому процессор Pentium предоставляет еще один способ вызова подпрограмм — через **шлюз (вентиль)**, позволяющий пользовательскому коду вызывать привилегированные процедуры, которые будут работать со своим высоким уровнем привилегий. Шлюзы вызова обладают еще одним преимуществом — появляется возможность контроля точек входа в вызываемые процедуры. В обоих рассмотренных ранее способах адрес точки входа в вызываемую процедуру определяется смещением, заданным в команде `CALL` вызывающей процедуры, то есть существует возможность задания некорректного значения смещения, в результате чего может произойти передача управления не на нужную команду или вообще в середину команды. Шлюзы вызова свободны от данного недостатка.

Набор точек входа в привилегированные кодовые сегменты определяется заранее, и эти точки входа описываются с помощью специальных дескрипторов шлюзов вызова процедур. Дескрипторы этого типа принадлежат к системным, и хотя их структура отличается от структуры дескрипторов сегментов кода и данных (рис. 6.13), они также включены в таблицы LDT и GDT.

Селектор 0÷15	Смещение 0÷15				
Смещение 16÷32	P	DPL	S=0	C (386) или 4 (286)	0÷4 Счетчик слов (WC)

Рис. 6.13. Формат дескриптора шлюза вызова подпрограммы

Схема вызова процедуры через шлюз приведена на рис. 6.14. Селектор из поля команды CALL указывает на дескриптор шлюза в таблицах GDT или LDT. Для того чтобы получить доступ к процедуре через шлюз, описываемый данным дескриптором, вызывающий код должен иметь не меньший уровень прав, чем дескриптор шлюза (то есть CPL ≤ DPL). При этом вызываемый код может иметь любой уровень привилегий (в том числе и более высокий, чем у шлюза), который сохраняется при его выполнении. Это позволяет из пользовательской программы вызывать процедуры ОС, работающие с высоким уровнем привилегий. При определении адреса входа в вызываемом сегменте смещение из поля команды CALL не используется, а используется смещение из дескриптора шлюза, что не дает возможности задаче самой определять точку входа в защищенный кодовый сегмент.

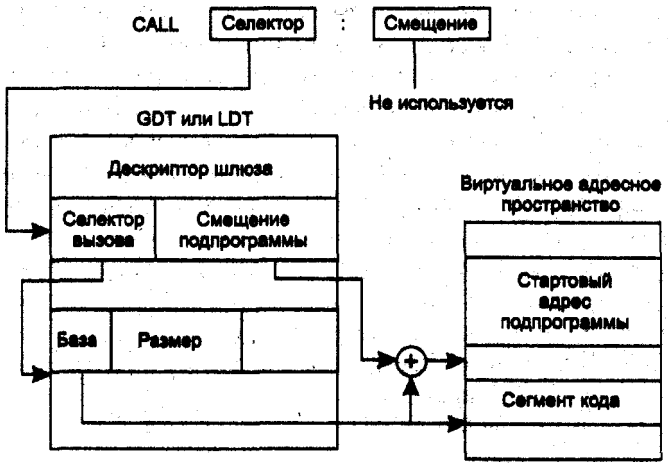


Рис. 6.14. Вызов подпрограммы через шлюз вызова

При вызове кодов, обладающих различными уровнями привилегий, возникает проблема передачи параметров между вызывающей и вызываемой процедурами. Для ее решения в процессоре предусмотрено существование стеков разных уровней, по одному стеку на каждый уровень привилегий. Используемый кодовым сегментом стек всегда соответствует текущему уровню привилегий кодового сегмента, то есть значению CPL. В сегменте контекста задачи TSS (более детально он описан далее) хранятся значения селекторов стека SS для уровней привилегий 0, 1 и 2. Если вызывается процедура, имеющая уровень

привилегий, отличный от текущего, то при выполнении команды CALL создается новый стек. Для этого из сегмента TSS извлекается новое значение селектора стека, соответствующее новому уровню привилегий, которое загружается в регистр SS, а из текущего стека в новый стек копируется столько 32-разрядных слов, сколько указано в поле счетчика слов дескриптора шлюза. В новом стеке также запоминается селектор старого стека, который используется при возврате в вызывающую процедуру.

Вызов задач

Механизм вызова при переключении между задачами отличается от механизма вызова процедур. В этом случае селектор команды CALL должен указывать на дескриптор системного сегмента TSS. Сегмент TSS хранит контекст задачи, то есть информацию, которая нужна для восстановления выполнения прерванной в произвольный момент времени задачи. Контекст задачи включает значения регистров процессора, указатели на открытые файлы и некоторые другие переменные, зависящие от операционной системы. Скорость переключения контекста в значительной степени влияет на производительность многозадачной операционной системы.

Процессор Pentium производит аппаратное переключение контекстов задач, используя для этого сегменты специального типа TSS. Структура сегмента TSS задачи приведена на рис. 6.15. Как видно из рисунка, сегмент TSS имеет фиксированные поля, отведенные для содержимого регистров процессора, как универсальных, так и некоторых управляющих (например, LDTR и CR3). Для описания возможностей доступа задачи к портам ввода-вывода процессор использует в защищенном режиме поле IOPL (Input/Output Privilege Level) в своем регистре EFLAGS и карту битовых полей доступа к портам в сегменте TSS. Для получения возможности безусловно выполнять команды ввода-вывода текущий код должен иметь уровень прав CPL не ниже, чем *уровень привилегий операций ввода-вывода*, задаваемый значением поля IOPL в регистре EFLAGS. Если же это условие не соблюдается, то возможность доступа к порту с конкретным адресом определяется значением соответствующего бита в карте ввода-вывода сегмента TSS (карта состоит из 64 Кбит для описания доступа к 65 536 портам) — значение 0 разрешает операцию ввода-ввода с данным номером порта.

Кроме этого, сегмент TSS может включать дополнительную информацию, необходимую для выполнения задачи и зависящую от конкретной операционной системы (например, указатели открытых файлов или указатели на именованные конвейеры сетевого обмена).

Информация сегмента TSS автоматически заменяется процессором при выполнении команды CALL, селектор которой указывает на дескриптор сегмента TSS в таблице GDT (дескрипторы этого типа могут быть расположены только в этой таблице). Формат дескриптора сегмента TSS аналогичен формату дескриптора сегмента данных (за исключением, естественно, поля типа сегмента, в котором указывается, что это дескриптор сегмента TSS).

Битовая карта ввода-вывода (БКВВ)			8 Кбайт
Дополнительная информация ОС			
Относительный адрес БКВВ	0 . . . 0	T	64
0 . . . 0	Селектор LDT		60
0 . . . 0	GS		5C
0 . . . 0	FS		58
0 . . . 0	DS		54
0 . . . 0	SS		50
0 . . . 0	CS		4C
0 . . . 0	ES		48
EDI			44
ESI			40
EBP			3C
ESP			38
EBX			34
EDX			30
ECD			2C
EAX			28
EFLAGS			24
EIP			20
CR3			1C
0 . . . 0	SS уровня 2		18
ESP2			14
0 . . . 0	SS уровня 1		10
ESP1			C
0 . . . 0	SS уровня 0		8
ESP0			4
0 . . . 0	Селектор TSS возврата		0

Рис. 6.15. Структура сегмента TSS

Как и в случае вызова процедуры, имеется два способа вызова задачи: непосредственный вызов путем указания селектора дескриптора сегмента TSS нужной задачи в поле команды `CALL` и косвенный вызов через шлюз вызова задачи.

Однако условие, разрешающее непосредственный вызов задачи, отличается от условия непосредственного вызова процедуры: вызов возможен только в случае, если вызывающий код обладает уровнем привилегий, не меньшим, чем вызываемая задача (CPL ≤ DPL). Здесь применяется то же правило, что и при доступе к данным. Действительно, операционная система, работающая с высоким уровнем привилегий, должна иметь возможность запускать на выполнение пользовательские задачи, работающие с низким уровнем привилегий. В этом случае ОС не поручает ненадежному низкоуровневому коду выполнять некоторые свои функции, как это происходило бы при вызове низкоуровневых процедур, а просто выполняет переключение между пользовательскими процессами.

При вызове через шлюз (который может располагаться и в таблице LDT) вызывающему коду достаточно иметь права доступа к шлюзу, а шлюз может указывать на дескриптор TSS в таблице GDT с равным или более высоким уровнем привилегий. Поэтому через шлюз вызова задачи можно выполнить переключение на более привилегированную задачу.

Непосредственный вызов задачи иллюстрирует рис. 6.16. При переключении задач процессор выполняет следующие действия.

1. Выполняется команда CALL, селектор которой указывает на дескриптор сегмента типа TSS. Происходит проверка прав доступа, успешная при CPL ≤ ≤ DPL.
2. В TSS текущей задачи сохраняются значения регистров процессора. На текущий сегмент TSS указывает регистр процессора TR, содержащий селектор сегмента.
3. В TR загружается селектор сегмента TSS задачи, на которую переключается процессор.
4. Из нового сегмента TSS в регистр LDTR переносится значение селектора таблицы LDT в таблице GDT задачи.
5. Восстанавливаются значения регистров процессора (из соответствующих полей нового сегмента TSS).
6. В поле селектора возврата нового сегмента TSS заносится селектор сегмента TSS снимаемой с выполнения задачи для организации возврата к ней в будущем.

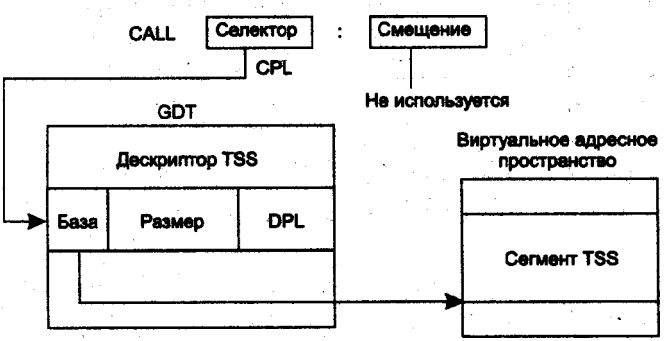


Рис. 6.16. Непосредственный вызов задачи

Вызов задачи через шлюз происходит аналогично, добавляется только этап поиска дескриптора сегмента TSS по значению селектора дескриптора шлюза вызова.

Использование всех возможностей, предоставляемых процессорами Intel 80386, 80486 и Pentium, позволяет организовать операционной системе высоконадежную многозадачную среду.

Механизм прерываний

Процессор Pentium поддерживает векторную схему прерываний, с помощью которой может быть вызвано 256 процедур обработки прерываний (вектор имеет длину 1 байт). Соответственно, таблица процедур обработки прерываний имеет 256 элементов, которые в реальном режиме работы процессора состоят из дальних адресов (CS:IP) этих процедур, а в защищенном режиме — из дескрипторов.

Прерывания, которые обрабатывает Pentium, делятся на следующие классы:

- **аппаратные** (внешние) **прерывания** — источником таких прерываний является сигнал на входе процессора;
- **исключения** (exceptions) — внутренние прерывания процессора;
- **программные прерывания** происходят по команде `INT`.

Аппаратные прерывания бывают маскируемыми и немаскируемыми.

Маскируемые прерывания вызываются сигналом INTR на одном из входов микросхемы процессора. При его возникновении процессор завершает выполнение очередной инструкции, сохраняет в стеке значение регистра признаков программы EFLAGS и адреса возврата, а затем считывает с входов шины данных байт вектора прерываний и в соответствии с его значением передает управление одной из 256 процедур обработки прерываний.

Маскируемость прерываний управляется флагом разрешения прерываний IF (Interrupt Flag), находящимся в регистре EFLAGS процессора. При IF = 1 маскируемые прерывания разрешены, а при IF = 0 — запрещены. Для явного управления флагом IF в процессоре имеются чувствительные к уровню привилегий инструкции разрешения маскируемых прерываний `STI` (SeT Interrupt flag) и запрета маскируемых прерываний `CLI` (CLear Interrupt flag). Эти инструкции разрешается выполнять при CPL ≤ IOPL. Кроме того, состояние флага изменяется неявным образом в некоторых ситуациях, например, он сбрасывается процессором при распознавании сигнала INTR, чтобы процессор не входил во вложенные циклы процедуры обработки одного и того же прерывания. Процедура обработки прерывания завершается инструкцией `IRET`, по которой происходит извлечение из стека признаков EFLAGS, адреса возврата, установка флага разрешения прерываний IF и передача управления по адресу возврата. Для маскируемых прерываний в процессоре отведены процедуры обработки прерываний с номерами 32–255. Соответствие между сигналом запроса прерывания на шине ввода-вывода (например, сигналом IRQ*n* на шине PCI) и значением вектора задается внешним по отношению к процессору блоком компьютера контроллером прерываний.

Немаскируемое аппаратное прерывание происходит при появлении сигнала NMI (Non Maskable Interrupt) на входе процессора. Этот сигнал всегда прерывает работу процессора вне зависимости от значения флага IF. При обработке немаскируемого прерывания вектор не считывается, а управление всегда передается процедуре с номером 2, описываемой третьим элементом таблицы процедур обработки прерываний (нумерация в этой таблице начинается с нуля). Немаскируемые прерывания предназначаются для реакции на «сверхважные» для компьютерной системы события, например сбой по питанию. В ходе процедуры обслуживания немаскируемого прерывания процессор не реагирует на другие запросы немаскируемых и маскируемых прерываний до тех пор, пока не будет выполнена команда `IRET`. Если при обработке немаскируемого прерывания возникает новый сигнал NMI, то он фиксируется и обрабатывается после

завершения обработки текущего прерывания, то есть после выполнения команды IRET.

Исключения делятся в процессоре Pentium на отказы, ловушки и аварийные завершения.

Отказы (faults) соответствуют некорректным ситуациям, которые выявляются до выполнения инструкции, например, при обращении по адресу, находящемуся в отсутствующей в оперативной памяти странице (страничный отказ). После обработки исключения-отказа процессор повторяет выполнения команды, которую он не смог выполнить из-за отказа.

Ловушки (traps) обрабатываются процессором после выполнения инструкции, например, при возникновении переполнения. После обработки процессор выполняет инструкцию, следующую за той, которая вызвала исключение.

Аварийные завершения (aborts) соответствуют ситуациям, когда невозможно точно определить команду, вызвавшую прерывание. Чаще всего это происходит во время серьезных отказов, связанных со сбоями в работе аппаратуры компьютера. Для обработки исключений в таблице прерываний отводятся номера 0–31.

Программные прерывания в процессоре Pentium происходят при выполнении инструкции INT с однобайтовым аргументом, в котором указывается вектор прерывания. Общая длина инструкции INT — два байта, исключение составляет инструкция INT 3, которая целиком помещается в один байт — это удобно при отладке программ, когда инструкция INT заменяет первый байт любой команды, вызывая переход на процедуру отладки. Программные прерывания подобно ловушкам обрабатываются после выполнения соответствующей инструкции INT, а возврат происходит в следующую инструкцию. Программное прерывание может вызвать любую из 256 процедур обработки прерываний, указанных в таблице прерываний.

При одновременном возникновении запросов прерываний различных типов процессор Pentium разрешает коллизию с помощью *приоритетов*. Немаскируемые прерывания имеют более высокий приоритет, чем маскируемые. Приоритетность внутри маскируемых прерываний устанавливается не процессором, а контроллером прерываний (процессор не может этого сделать, так как для него все маскируемые запросы представлены одним сигналом INTR). Проверка некорректных ситуаций, порождающих исключения (в том числе и при выполнении одной команды), выполняется в процессоре в соответствии с определенной последовательностью.

Таблица прерываний в реальном режиме состоит из 256 элементов, каждый из которых имеет длину в 4 байта и представляет собой дальний адрес (CS:IP) процедуры обработки прерываний. Таблица прерываний реального режима всегда находится в фиксированном месте физической памяти — с начального адреса 00000 по адрес 003FF. В защищенном режиме таблица прерываний носит название IDT (Interrupt Descriptor Table) и может располагаться в любом месте физической памяти. Ее начало (32-разрядный физический адрес) и размер (16 бит) можно найти в регистре системных адресов IDTR. Каждый из 256 эле-

ментов таблицы прерываний представляет собой 8-байтный дескриптор. В таблице прерываний могут находиться только дескрипторы определенного типа — дескрипторы шлюзов прерываний, ловушек и задач.

Шлюзы задач уже рассматривались, они используется всегда для переключения с задачи на задачу.

Шлюзы прерываний и ловушек специально вводятся для вызова процедур обработки прерываний. Если для вызова процедуры обработки прерывания используется шлюз задач, то происходит смена процесса, а при завершении обработки — возврат к прерванному процессу. Обычно обслуживание прерываний со сменой процесса (и запоминанием его контекста) применяется для внешних прерываний, которые не связаны с текущим процессом, например, когда принтер с помощью прерывания требует загрузить в его буфер новую порцию распечатываемых данных приостановленного процесса.

Шлюзы прерываний и ловушек не вызывают смены контекста задачи, следовательно, процедуры обработки прерываний в этом случае вызываются быстрее, чем при использовании шлюза задачи. Формат дескриптора шлюза прерывания и ловушки аналогичен формату дескриптора шлюза вызова, и обработка процессором этих шлюзов во многом аналогична вызову процедуры через шлюз вызова. Отличие состоит в том, что при вызове процедуры через шлюз прерываний сбрасывается флаг IF и тем самым запрещаются вложенные прерывания. При использовании шлюза ловушки сброса флага IF не происходит, но в стек при некоторых видах исключений дополнительно помещается код ошибки, вызвавшей исключение.

Кэширование в процессоре Pentium

В процессоре Pentium кэширование используется в следующих случаях.

- *Кэширование дескрипторов сегментов в скрытых регистрах.* Для каждого сегментного регистра в процессоре имеется так называемый *скрытый* регистр дескриптора. В скрытый регистр при загрузке сегментного регистра помещается информация из дескриптора, на который указывает данный сегментный регистр. Информация из дескриптора сегмента используется для преобразования виртуального адреса в физический при чисто сегментной организации памяти либо для получения линейного виртуального адреса при страничном механизме. Доступ к скрытому регистру выполняется быстрее, чем поиск и извлечение информации из таблицы страниц, находящейся в оперативной памяти. Поэтому если очередное обращение будет относиться к одному из сегментов, дескриптор которого еще хранится в скрытом регистре (а вероятность этого велика), то преобразование адресов будет выполнено быстрее. Тем самым скрытые регистры играют роль кэша таблицы дескрипторов и ускоряют работу процессора.
- *Кэширование пар номеров виртуальных и физических страниц в буфере ассоциативной трансляции* (Translation Lookaside Buffer, TLB) позволяет ускорять преобразование виртуальных адресов в физические при сегментно-

страничной организации памяти. TLB представляет собой ассоциативную память небольшого объема, предназначенную для хранения интенсивно используемых дескрипторов страниц. В процессоре Pentium имеются отдельные буферы TLB для инструкций и данных.

- *Кэширование данных и инструкций в кэш-памяти первого уровня.* Эта память, называемая также *внутренней* кэш-памятью, поскольку она размещена непосредственно на кристалле микропроцессора, имеет объем 16/32 Кбайт. В процессоре Pentium кэш первого уровня разделен на память для хранения данных и память для хранения инструкций. Согласование данных выполняется только методом сквозной записи.
- *Кэширование данных и инструкций в кэш-памяти второго уровня.* Эта память называется также *внешней* кэш-памятью, поскольку она устанавливается в виде отдельной микросхемы на системной плате. Кэш-память второго уровня является общей для данных и инструкций и имеет объем 256/512 Кбайт. Поиск в кэше второго уровня выполняется в случае, когда констатируется промах в кэше первого уровня. Для согласования данных в кэше второго уровня может использоваться как сквозная, так и обратная запись.

Рассмотрим более подробно принципы работы буфера ассоциативной трансляции и кэша первого уровня.

Буфер ассоциативной трансляции

В буфере TLB кэшируются дескрипторы страниц из таблицы страниц (рис. 6.17). Для хранения дескриптора в кэше отводится одна *строка*. Каждая строка дополнена *тегом*, в котором содержится номер соответствующей виртуальной страницы. Строки объединены по четыре в группы, называемые *наборами*. Таблица TLB, используемая для преобразования адресов инструкций, имеет 32 строки и соответственно 8 наборов. Номер набора называют *индексом* (index). Таким образом, путем кэширования может быть получен физический адрес для доступа к 32 страницам памяти, содержащим инструкции.

После того как механизмом сегментации получен линейный адрес, он должен быть преобразован в физический адрес. Для этого, прежде всего, необходимо найти дескриптор страницы, которой принадлежит данный адрес, и извлечь из него номер физической страницы. Обычная процедура предусматривает обращение к таблице разделов, а затем — к таблице страниц. Однако физический адрес может быть получен гораздо быстрее благодаря тому, что в буфере TLB хранятся копии дескрипторов наиболее интенсивно используемых страниц. Поэтому перед тем, как начать сравнительно длительную процедуру преобразования адресов, делается попытка обнаружить нужный дескриптор страницы в быстрой ассоциативной памяти TLB. Затем на основании номера физической страницы, полученного из TLB, вычисляется физический адрес.

При поиске данных в TLB используется линейный виртуальный адрес. Разряды 12–14 используются как индекс набора. Далее проверяются **биты действительности** всех строк выбранного набора. В начале работы кэш-памяти биты действительности всех строк сбрасываются в нуль. В результате работы алго-

ритма замещения бит действительности принимает значение 1, когда в соответствующей строке содержится достоверная информация, и сбрасывается в нуль, когда строка объявляется свободной. Для всех действительных строк выполняется ассоциативная процедура сравнения тегов со старшими разрядами (15–31 разряд) линейного виртуального адреса. Если произошло кэш-попадание, то номер физической страницы быстро поступает в схему формирования физического адреса.

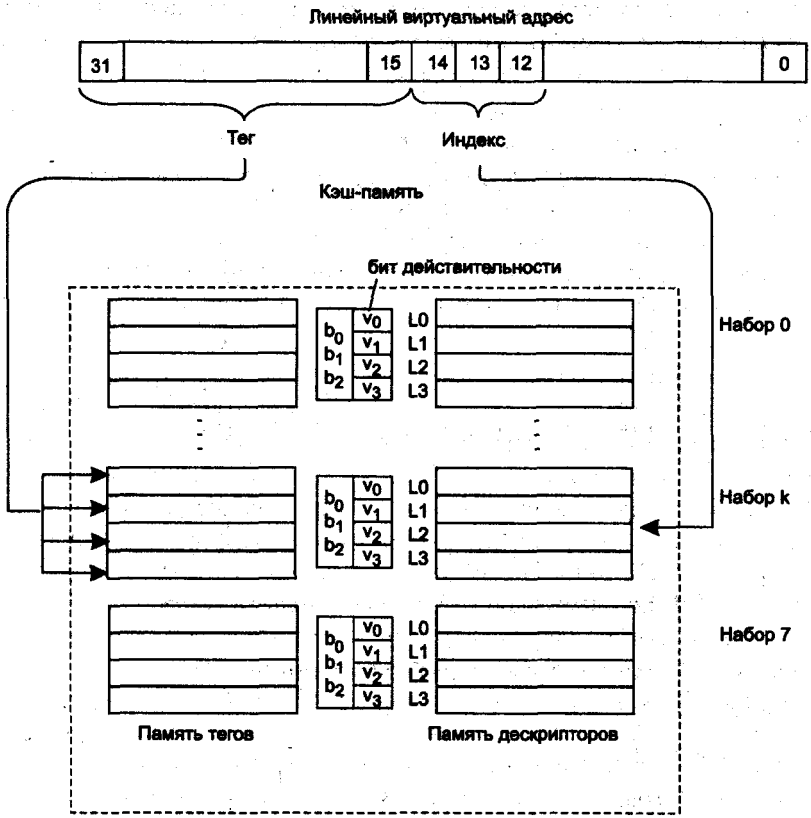


Рис. 6.17. Буфер ассоциативной трансляции

Если произошел промах и нужного дескриптора в TLB нет, то запускается многоэтапная процедура преобразования адреса, включающая обращения к таблицам разделов и страниц. Когда нужный дескриптор отыскивается в таблице страниц, он копируется в TLB. Номер набора, в который записывается кэшируемый дескриптор, определяется тремя младшими разрядами номера виртуальной страницы (разряды 12–14 линейного виртуального адреса).

Однако поскольку в наборе имеется четыре строки, необходимо определить, в какую именно надо поместить кэшируемые данные. Дескриптор записывается либо в первую попавшуюся свободную строку, либо, если все строки заняты,

в строку, к которой дольше всего не обращались. Признаком занятости строки служит бит действительности v, имеющийся у каждой строки. Если v = 0, значит, строка свободна для записи в нее нового содержимого. Для определения строки, которая не использовалась дольше всех других в данном наборе, применяется упрощенный вариант алгоритма PseudoLRU (Pseudo Least Recently Used). Этот алгоритм основан на анализе трех битов b0, b1, b3, называемых **битами обращения**. Биты обращения приписываются набору и устанавливаются в соответствии с алгоритмом, приведенном на рис. 6.18. Здесь символы L0, L1, L2, L3 обозначают последовательные строки набора.

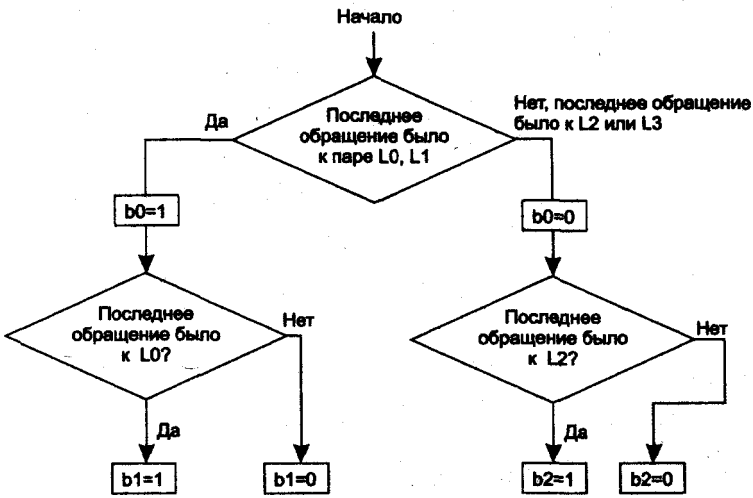


Рис. 6.18. Алгоритм установки битов обращения

На замену выбирается одна из следующих строк:

- L0, если b0 = 0 и b1 = 0;
- L1, если b0 = 0 и b1 = 1;
- L2, если b0 = 1 и b2 = 0;
- L3, если b0 = 1 и b2 = 1.

Можно легко показать, что данная процедура не всегда приводит к выбору действительно дольше всех не вызывавшейся строки. Пусть, например, обращения к строкам выполнялись в следующей хронологической последовательности: L0, L2, L3, L1, то есть ближайшее по времени обращение было к строке L1, дольше же всего не было обращений к строке L0. Биты обращения в данном случае примут следующие значения. Поскольку последнее по времени обращение было к строке из пары (L0, L1), значит, b0 = 1. А в паре (L2, L3) последнее обращение было к L3, следовательно, b2 = 0. Отсюда, по правилу, приведенному ранее, на замену выбирается строка L2, вместо строки L0, к которой на самом деле дольше всего не было обращений.

Однако в большинстве случаев этот алгоритм дает результат, совпадающий с оптимальным. Например, для последовательности L0, L3, L1, L2 биты обращения имеют значения b0 = 0, b1 = 0, отсюда точное решение — L0. Даже в случае ошибки (вероятность которой составляет 33 %) решения, найденные по алгоритму PseudoLRU, близки к оптимальным. Так, в первом примере вместо строки L0, являющейся правильным решением, алгоритм дал ближайшую к ней по времени обращения строку L2.

Несмотря на то что алгоритм PseudoLRU дает в общем случае приближенные решения, он широко применяется при кэшировании, так как является быстрым и экономичным, что чрезвычайно важно для кэш-памяти.

Таким образом, в буфере TLB процессора Pentium используется комбинированный способ отображения кэшируемых данных на кэш-память: *прямое отображение* дескрипторов на наборы и *случайное отображение* на строки в пределах набора.

Наличие TLB позволяет в подавляющем числе случаев заменить сравнительно долгую процедуру преобразования адресов, связанную с несколькими обращениями к оперативной памяти, быстрым поиском в ассоциативной памяти.

Кэш первого уровня

Кэш первого уровня используется на этапе обработки запроса к основной памяти по физическому адресу.

Работа кэш-памяти первого уровня имеет много общего с работой буфера TLB. В TLB единицей хранения является дескриптор, а в кэше первого уровня — байт данных. Обновление данных в кэше происходит *блоками* по 16 байт. Таким образом, младшие 4 бита физического адреса байта могут интерпретироваться как смещение в блоке, а старшие разряды — как номер блока.

Для хранения блоков данных в кэше отводятся строки, также имеющие объем 16 байт. Строки объединены в наборы по четыре. При объеме кэша 16 Кбайт в него входят 256 (2^8) наборов.

При копировании данных в кэш номера блоков основной памяти прямо отображаются на номера наборов. Для этого в адресе основной памяти, относящемся к одному из байтов, входящих в блок, значение 8 бит перед битами смещения интерпретируется как номер набора в кэш-памяти (рис. 6.19). Остальные старшие биты адреса в дальнейшем используются в качестве тега.

Так же как в TLB, выбор строки в наборе осуществляется на основе анализа битов действительности и битов обращения по алгоритму PseudoLRU. Блок данных заносится в строку кэш-памяти вместе со своим тегом — старшими разрядами адреса основной памяти. Бит действительности строки устанавливается в 1.

При запросе на чтение из основной памяти вначале делается попытка найти данные в кэше (либо поиск в кэше совмещается с выполнением запроса к основной памяти). По индексу, извлеченному из адреса запроса, определяется набор, в котором могут находиться искомые данные. Затем для строк данного набора, содержимое которых действительно (установлены биты действительности),

выполняется ассоциативный поиск: старшие разряды адреса из запроса сравниваются с тегами всех строк набора. Если для какой-нибудь строки фиксируется совпадение, это означает, что произошло кэш-попадание, и из соответствующей строки извлекается байт, смещение которого относительно начала строки определяется четырьмя младшими разрядами из адреса запроса.

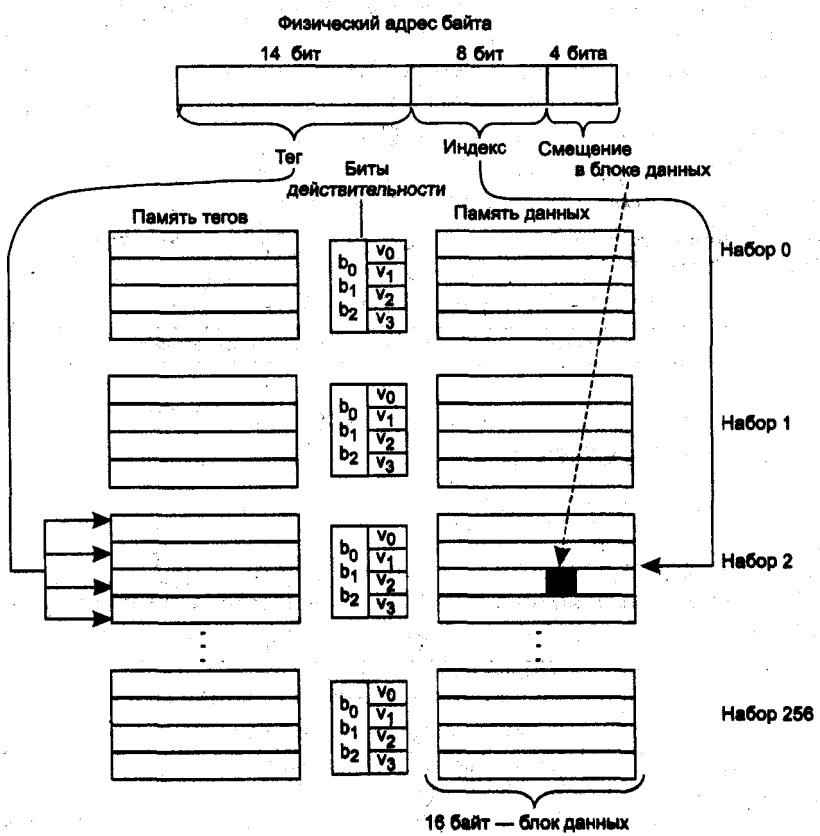


Рис. 6.19. Кэш первого уровня процессора Pentium

Для согласования данных в кэше первого уровня используется метод сквозной записи, то есть при возникновении запроса на запись обновляется не только содержимое соответствующей ячейки основной памяти, но и его копия в кэш-памяти. Заметим также, что запрос на запись при промахе не вызывает обновления кэша.

Совместная работа кэшей разного уровня

Разные виды кэш-памяти вступают «в игру» на разных этапах обработки запроса к основной памяти. В зависимости от того, насколько удачно для запроса сложилась ситуация с попаданиями в кэш-память разного типа, время его вы-

полнения может измениться в десятки раз. На рис. 6.20 показана схема выполнения запроса к памяти с сегментно-страничной организацией.

Рассмотрим операцию считывания операнда из оперативной памяти по его виртуальному адресу — номеру виртуального сегмента и смещения в этом сегменте.

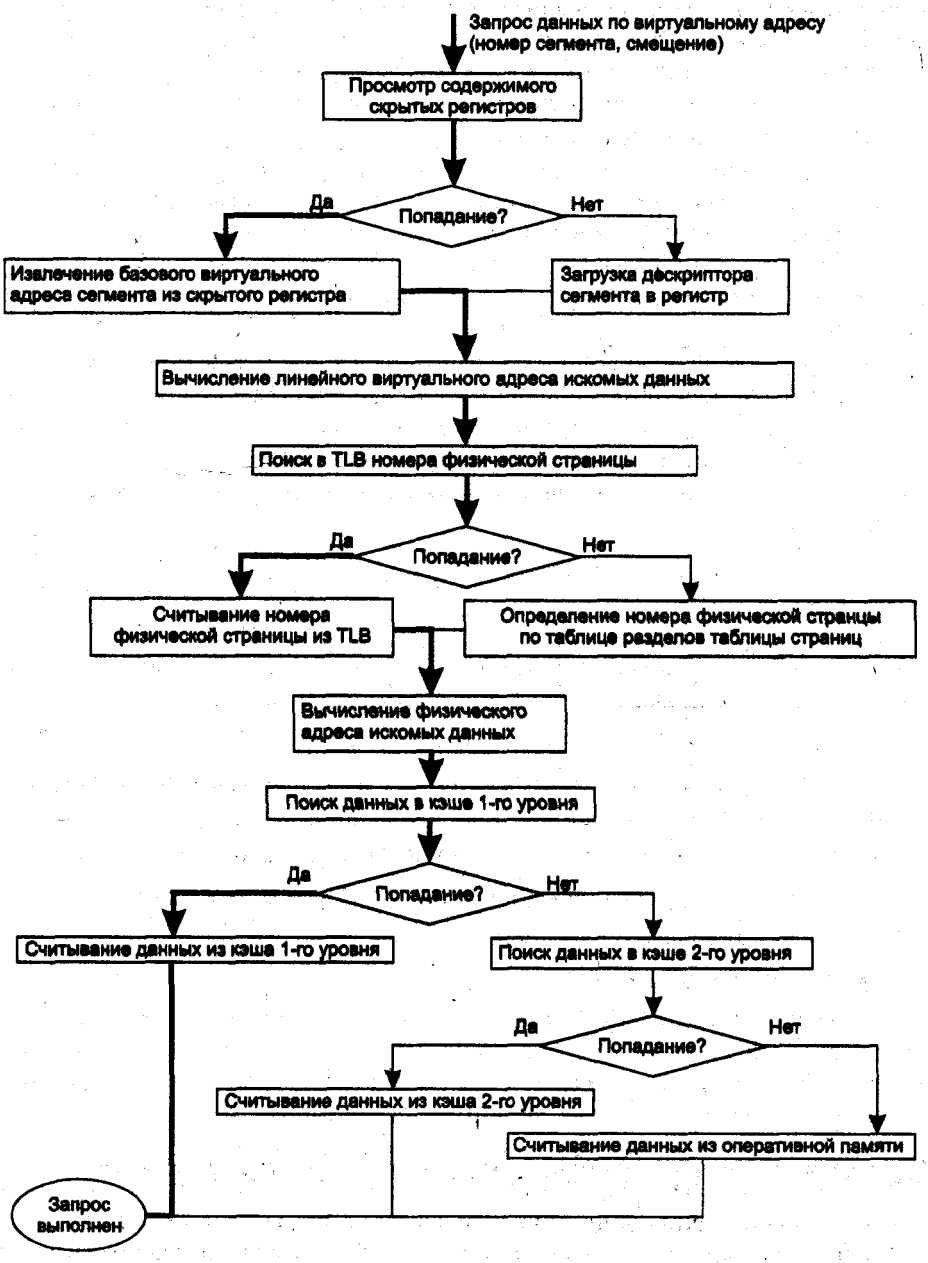


Рис. 6.20. Использование кэширования на разных этапах обработки запроса

Первое обращение к кэш-пямяти происходит на этапе работы сегментного механизма, когда необходимо вычислить линейный виртуальный адрес, используя информацию дескриптора сегмента. Все дескрипторы сегментов, входящих в виртуальное адресное пространство процесса, хранятся в оперативной памяти в таблицах GDT и LDT. Однако реального обращения к оперативной памяти может и не быть, если нужный сегмент является одним из активных сегментов процесса — в этом случае его дескриптор находится в соответствующем скрытом регистре. Кэширование дескрипторов сегментов предоставляет первую возможность сокращения времени доступа к оперативной памяти.

Следующую возможность предоставляет буфер ассоциативной трансляции TLB, в котором кэшируются дескрипторы страниц, что позволяет сэкономить время при вычислении физического адреса. Вероятность кэш-попадания в данном случае очень велика — в среднем она составляет 98 %, и только 2 % обращений требуют действительного чтения таблиц разделов и страниц из оперативной памяти.

При известном физическом адресе и определенном везении искомый операнд может быть обнаружен в кэше первого уровня. Если же повезет немного меньше, то операнд найдется в кэше второго уровня.

Таким образом, наличие разнообразных кэшей в процессоре Pentium позволяет во многих случаях существенно сократить время обработки запроса к оперативной памяти.

Выводы

- Процессоры семейства Pentium обладают развитыми механизмами, необходимыми для организации мультипрограммного режима:
  - набором привилегированных команд;
  - средствами защиты сегментов кодов и данных, обеспечивающими четыре уровня привилегий;
  - сегментным и сегментно-страничным механизмами виртуальной памяти;
  - механизмом быстрого переключения процессов с сохранением контекста;
  - встроенным кэшем оперативной памяти;
  - векторной системой прерываний.
- Процессор Pentium при управлении памятью поддерживает два типа таблиц дескрипторов сегментов: глобальную таблицу дескрипторов GDT, описывающую сегменты операционной системы и разделяемые сегменты прикладных процессов, и локальные таблицы дескрипторов LDT, которые содержат дескрипторы сегментов отдельных пользовательских процессов.
- При страничном режиме работы виртуальное адресное пространство 32-разрядных процессоров семейства Pentium состоит из 16 Кбайт сегментов по 4 Гбайт каждый — всего 64 Тбайт, а при сегментно-страничном режиме работы все сегменты отображаются в общий диапазон адресов 4 Гбайт.

- Каждый сегмент виртуального адресного пространства описывается дескриптором, который содержит базовый адрес, размер сегмента, а также ряд признаков, в том числе уровень привилегий сегмента DPL, определяющий права доступа к нему.
- В процессоре Pentium поддерживается несколько способов вызова процедур, а также специальные средства вызова задач, позволяющие автоматически сохранять и восстанавливать наиболее значимую часть контекста задачи.
- Процессор Pentium поддерживает векторную схему прерываний, с помощью которой может быть вызвано 256 процедур обработки прерываний. Прерывания могут быть инициированы внешним сигналом (аппаратные прерывания), некорректным выполнением инструкции (исключения), а также специальной инструкцией `INT` (программные прерывания).
- В процессоре Pentium активно применяется кэширование:
  - кэширование дескрипторов сегментов в скрытых регистрах процессора;
  - кэширование дескрипторов страниц в буфере ассоциативной трансляции (TLB);
  - кэширование данных и инструкций в кэш-памяти первого уровня;
  - кэширование данных и инструкций в кэш-памяти второго уровня.

Задачи и упражнения

1. Существует ли защищенный режим в большинстве современных процессоров или это специфический режим процессоров Pentium?
2. Значения каких системных регистров процессора должен использовать программный модуль ОС, чтобы произвести обращение к индивидуальной части памяти текущего процесса?
3. Представьте, что для задач всех уровней привилегий используется один общий стек. К каким последствиям это может привести?
4. Почему в сегменте состояния задачи TSS хранятся значения селекторов стека для уровней привилегий 0, 1 и 2, но нет значения для селектора уровня 3?
5. В какой памяти — физической или виртуальной — базовый адрес, хранимый в дескрипторе сегмента, определяет положение сегмента при выключенном страничном механизме?
6. Зачем нужны шлюзы вызовов процедур и задач, если существует возможность непосредственного вызова?
7. Заполните следующую таблицу, в которой укажите возможность или невозможность непосредственного вызова процедуры со сменой кодового сегмента для различных сочетаний уровней привилегий вызывающего и вызываемого сегментов и типов сегментов.

Соотношение уровней	Тип сегмента	Возможность доступа
CPL > DPL	C = 1	
CPL < DPL	C = 1	
CPL = DPL	C = 1	
CPL > DPL	C = 0	
CPL < DPL	C = 0	
CPL = DPL	C = 0	

8. Можно ли на базе процессора Pentium реализовать систему управления памятью с фиксированными разделами?
9. Можно ли выгружать страницы, которые хранят разделы таблицы страниц?
10. По каким соображениям в процессорах Pentium запрещено вызывать менее привилегированные процедуры, но разрешено вызывать менее привилегированные задачи?
11. В чем принципиальное отличие использования шлюза прерываний от использования шлюза задачи?
12. Поддерживает ли процессор Pentium приоритезацию запросов прерывания между несколькими внешними устройствами?

Глава 7

Ввод-вывод и файловая система

Одной из главных задач ОС является обеспечение обмена данными между приложениями и периферийными устройствами компьютера. Собственно, ради выполнения этой задачи и были разработаны первые системные программы, послужившие прототипами операционных систем. В современной ОС функции обмена данными с периферийными устройствами выполняет подсистема ввода-вывода. Клиентами этой подсистемы являются не только пользователи и приложения, но и некоторые компоненты самой ОС, которым требуется получение системных данных или их вывод, например, подсистеме управления процессами при смене активного процесса необходимо записать на диск контекст приостанавливаемого процесса и считать с диска контекст активизируемого процесса.

Основными компонентами подсистемы ввода-вывода являются драйверы, управляющие внешними устройствами, и файловая система. К подсистеме ввода-вывода можно также с некоторой долей условности отнести рассмотренный ранее диспетчер прерываний. Условность заключается в том, что диспетчер прерываний обслуживает не только модули подсистемы ввода-вывода, но и другие модули ОС, в частности такой важный модуль, как планировщик/диспетчер потоков. Но из-за того, что планирование работ подсистемы ввода-вывода составляет основную долю нагрузки диспетчера прерываний, его вполне логично рассматривать как ее составную часть (к тому же первопричиной появления в компьютерах системы прерываний были в свое время именно операции с устройствами ввода-вывода).

Файловая система ввиду ее сложности, специфичности и важности как основного хранилища всей информации вычислительной системы заслуживает изучения в отдельной главе. Тем не менее в этой книге файловая система рассматривается совместно с другими компонентами подсистемы ввода-вывода по двум причинам. Во-первых, файловая система активно задействует остальные компоненты подсистемы ввода-вывода, во-вторых, модель файла лежит

в основе большинства механизмов доступа к устройствам, используемых в современной подсистеме ввода-вывода.

Задачи ОС по управлению файлами и устройствами

Подсистема ввода-вывода (Input-Output Subsystem) мультипрограммной ОС при обмене данными с внешними устройствами компьютера должна решать ряд общих задач, из которых наиболее важными являются следующие:

- организация параллельной работы устройств ввода-вывода и процессора;
- согласование скоростей обмена и кэширование данных;
- разделение устройств и данных между процессами;
- предоставление удобного программного интерфейса к устройствам;
- поддержка широкого спектра драйверов с возможностью простого включения в систему нового драйвера;
- динамическая загрузка и выгрузка драйверов;
- поддержка нескольких файловых систем;
- поддержка синхронных и асинхронных операций ввода-вывода.

В следующих разделах все эти задачи рассматриваются подробно.

Параллельная работа устройства ввода-вывода и процессора

Параллельная работа устройства ввода-вывода и процессора организуется путем взаимодействия контроллера и драйвера, относящихся к данному устройству.

> **Контроллер** — это специализированный блок управления, которым снабжается каждое устройство ввода-вывода вычислительной системы, включая диск, принтер, терминал и т. п.
>
> **Драйвер** — программный модуль ОС, предназначенный для управления устройством ввода-вывода.

Контроллер периодически принимает от драйвера выводимую на устройство информацию, а также команды управления, которые говорят о том, что с этой информацией нужно сделать (например, вывести в виде текста в определенную область экрана или записать в определенный сектор диска).

Под управлением контроллера устройство может некоторое время выполнять свои операции *автономно*, не требуя внимания со стороны центрального процессора, а следовательно, и со стороны ОС. Это время зависит от многих факторов — объема выводимой информации, степени интеллектуальности управляющего устройством контроллера, быстродействия устройства и т. п. Даже самый примитивный контроллер, выполняющий простые функции, обычно тратит довольно много времени на самостоятельную реализацию подобной функции

после получения очередной команды от процессора. Это же справедливо и для сложных контроллеров, так как скорость работы любого устройства ввода-вывода, даже самого быстродействующего, обычно существенно ниже скорости работы процессора.

Итак, в периоды между выдачами команд от процессора контроллеры выполняют свои действия независимо от ОС. От подсистемы ввода-вывода ОС требуется «лишь» осуществлять запуск и приостановку драйверов всех устройств ввода-вывода, входящих в состав компьютера, обеспечивая приемлемое время реакции каждого драйвера на сигналы прерываний, поступающие от соответствующих контроллеров. При этом время, которое тратит процессор на обеспечение ввода-вывода, должно быть минимизировано в пользу пользовательских процессов.

Данная задача является классической задачей планирования/диспетчеризации процессов в системах реального времени и обычно решается на основе многоуровневой приоритетной схемы обслуживания по прерываниям.

Для обеспечения приемлемого уровня реакции все драйверы (или части драйверов) распределяются по нескольким приоритетным уровням, в соответствии с временем реакции, допустимым для данного типа устройства, и временем, затрачиваемым процессором на выполнение соответствующего драйвера. Для реализации приоритетной схемы обычно задействуется общий диспетчер прерываний ОС.

Согласование скоростей обмена и кэширование данных

При обмене данными всегда возникает задача согласования скорости. Например, если один пользовательский процесс вырабатывает некоторые данные и передает их другому пользовательскому процессу через оперативную память, то в общем случае скорости генерации данных и их чтения не совпадают. Согласование скорости при обмене данными между *прикладными процессами* обычно достигается за счет буферизации данных в оперативной памяти и синхронизации доступа процессов к буферу.

Аналогичным образом решается проблема согласования скоростей обмена в *подсистеме ввода-вывода*. Здесь также широко используется *буферизация данных в оперативной памяти*. В тех специализированных операционных системах, в которых обеспечение высокой скорости ввода-вывода является первоочередной задачей (управление в реальном времени, услуги сетевой файловой службы и т. п.), большая часть оперативной памяти отводится не под коды прикладных программ, а под буферы данных.

Однако буферизация только на основе оперативной памяти в подсистеме ввода-вывода оказывается недостаточной — разница между скоростью обмена с оперативной памятью, куда процессы помещают данные для обработки, и скоростью работы внешнего устройства часто становится слишком значительной, чтобы в качестве временного буфера можно было бы использовать оперативную память — ее объема может просто не хватить.

В системах ввода-вывода при необходимости обмена большими объемами данных используется **спулинг**, при котором данные *буферизуются на диске* в особой области, называемой **дисковым файлом**, или **спул-файлом**[1].

Типичный пример применения спул-файла дает организация вывода данных на принтер. Для печатаемых документов объем в несколько десятков мегабайтов — не редкость, поэтому для их временного хранения (а печать каждого документа занимает от нескольких минут до десятков минут) объема оперативной памяти явно недостаточно.

Другим решением этой проблемы является использование большой *буферной памяти в контроллерах* внешних устройств. Такой подход особенно полезен в тех случаях, когда помещение данных на диск слишком замедляет обмен (или когда данные выводятся на сам диск). Например, в контроллерах графических дисплеев применяется буферная память, соизмеримая по объему с оперативной, и это существенно ускоряет вывод графики на экран.

Буферизация данных при вводе-выводе позволяет не только согласовать скорости работы процессора и внешнего устройства, но и решить другую задачу — сократить количество реальных операций ввода-вывода за счет *кэширования данных*. Дисковый кэш является непременным атрибутом подсистем ввода-вывода практически всех операционных систем, значительно сокращая время доступа к хранимым данным.

Разделение устройств и данных

Устройства ввода-вывода могут предоставляться процессам как в **монопольное**, так и в **совместное (разделяемое)** использование.

Какой тип доступа к устройству разрешен тому или иному процессу, определяет операционная система. Контроль доступа осуществляется теми же способами, что и при доступе процессов к другим ресурсам вычислительной системы — путем проверки прав пользователя или группы пользователей, от имени которых действует процесс, на выполнение той или иной операции с устройством. Например, определенной группе пользователей может быть разрешено захватывать последовательный порт в монопольное владение, в то время как другим пользователям — запрещено.

Одно и то же устройство в разные периоды времени может использоваться как в разделяемом, так и в монопольном режимах. Тем не менее существуют устройства, для которых обычно характерен один из этих режимов, например, последовательные порты и алфавитно-цифровые терминалы чаще используются в монопольном режиме, а диски — в режиме совместного доступа. Операционная система должна предоставлять эти устройства в обоих режимах: в монопольном режиме отслеживая процедуры захвата и освобождения устройств, а в режиме разделения оптимизируя последовательность операций ввода-вывода для различных процессов в целях повышения общей производительности, если это возможно. Например, при обмене данными нескольких процессов

[1] От английского spool — шпулька (тоже буфер; только для ниток).

с диском можно так упорядочить последовательность операций, что непроизводительные затраты времени на перемещение головок существенно сократятся (при этом для отдельных процессов возможно некоторое замедление операций ввода-вывода).

Операционная система может контролировать доступ не только к устройству в целом, но и к отдельным *порциям данных*, хранимых или отображаемых этим устройством.

Диск является типичным примером устройства, для которого важно контролировать доступ не к устройству в целом, а к отдельным каталогам и файлам. При выводе информации на графический дисплей отдельные окна на экране также представляют собой ресурсы, к которым необходимо обеспечить тот или иной вид доступа для протекающих в системе процессов. При этом для каждой порции данных или части устройства могут быть заданы свои права доступа, не связанные напрямую с правами доступа к устройству в целом. Так, в файловой системе для каждого каталога и файла обычно можно задать индивидуальные права доступа. Очевидно, что для организации совместного доступа к частям устройства или данных, хранящихся на нем, непременным условием является задание режима совместного использования устройства в целом.

При разделении устройства несколькими процессами может возникнуть необходимость в *разграничении* порций данных, относящихся к каждому из этих процессов. Такая потребность возникает при совместном использовании так называемых последовательных устройств, данные в которых, в отличие от устройств прямого доступа, не адресуются. Типичным примером такого рода устройства является принтер, который, с одной стороны, не выделяется в монопольное владение процессам, так как это устройство достаточно дорогое, а с другой — его совместное использование может вызвать проблему. Например, пусть две программы, работая в мультизадачном режиме, выводят на печать документы. Каждая программа выдает данные на печать порциями в своем темпе. Без принятия специальных мер на выходе принтера была бы получена мешанина строк, относящихся к двум разным документам, созданным каждой из этих двух программ. Поэтому подсистема ввода-вывода для подобных устройств организует очередь так называемых *заданий на вывод*, при этом с каждым заданием связана порция данных, которую нельзя разрывать. Для хранения очереди заданий используется спул-файл, который одновременно согласует скорости работы принтера и оперативной памяти и позволяет организовать разбиение данных на логические порции. Так как спул-файл находится на разделяемом устройстве прямого доступа, то процессы могут одновременно выполнять вывод на принтер, помещая данные в свой раздел спул-файла.

Программный интерфейс к устройствам

Для того чтобы облегчить разработку прикладных программ, включающих процедуры ввода-вывода, ОС должна поддерживать *экранирующий логический интерфейс* между периферийными устройствами и приложениями, который

давал бы возможность использовать общий набор базовых операций ввода-вывода для любых устройств независимо от их специфики.

В качестве основы такого интерфейса практически все современные операционные системы поддерживают *файловую модель устройств ввода-вывода*. Здесь мы еще раз сталкиваемся с плодотворной концепцией виртуализации.

Все разнообразие типов реальных устройств ввода-вывода операционная система подменяет одним виртуальным типом устройства. Все виртуальные устройства работают единым образом и представляются в виде файлов, называемых также **специальными**, или **виртуальными, файлами**[1].

Каждому устройству ввода-вывода ставится в соответствие отдельный специальный файл. Он представляет это устройство для прикладных процессов и остальной части операционной системы в виде неструктурированного набора байтов. В результате вместо написания сложных процедур ввода-вывода для реальных устройств программист может теперь просто использовать операции чтения из специальных файлов и записи в специальные файлы, ассоциированные с этими устройствами.

Привлекательность модели файла-устройства состоит в ее простоте и универсальности для устройств любого типа. Однако во многих случаях, как, например, вывода графической информации на дисплей или принтер, программирования операций сетевого обмена и др., эта модель является слишком примитивной и не освобождает прикладного программиста от рутинных задач программирования ввода-вывода для этих устройств. Поэтому для некоторых типов устройств ОС поддерживает более развитый интерфейс, отражающий их специфику.

Поддержка широкого спектра драйверов

Достоинством подсистемы ввода-вывода любой универсальной ОС является наличие разнообразного набора драйверов для наиболее популярных периферийных устройств. Прекрасно спланированная и реализованная операционная система может потерпеть неудачу на рынке только из-за того, что в ее состав не включен достаточный набор драйверов, в результате администраторы и пользователи вынуждены будут искать нужный им драйвер для имеющегося у них внешнего устройства у производителей оборудования или, что еще хуже, заниматься его разработкой. Именно в такой ситуации оказались пользователи первых версий OS/2, и, возможно, данное обстоятельство послужило в свое время не последней причиной того, что эта неплохая операционная система сдала позиции богатой на драйверы ОС Windows 3.x.

Чтобы операционная система не испытывала недостатка в драйверах, необходимо наличие логически ясного, удобного и открытого интерфейса между драйверами и другими компонентами ОС. Такой интерфейс нужен для того, чтобы драйверы писали не только непосредственные разработчики данной операционной системы, но и большая армия программистов по всему миру, в пер-

[1] Подробнее о специальных файлах читайте в главе 8.

вую очередь — тех предприятий, которые выпускают внешние устройства для компьютеров. Открытость интерфейса драйверов, то есть доступность его описания для независимых разработчиков программного обеспечения (а, возможно, также и разработка его на основе согласительных процедур между ведущими коллективами разработчиков), является необходимым условием успешного развития операционной системы.

Драйвер должен поддерживать два типа интерфейсов:

- **интерфейс «драйвер-ядро»** (Driver Kernel Interface, **DKI**) с модулями ядра ОС (модулями подсистемы ввода-вывода, системных вызовов, подсистем управления процессами и памятью и т. д.);
- **интерфейс «драйвер-устройство»** (Driver Device Interface, **DDI**) с контроллерами внешних устройств.

Интерфейс «драйвер-ядро» должен быть стандартизован в любом случае, а интерфейс «драйвер-устройство» имеет смысл стандартизировать тогда, когда подсистема ввода-вывода не разрешает драйверу непосредственно взаимодействовать с аппаратурой контроллера, а выполняет эти операции самостоятельно. Экранирование драйвера от аппаратуры является весьма полезной функцией, так как драйвер в этом случае становится независимым от аппаратной платформы. Подсистема ввода-вывода может поддерживать несколько различных типов интерфейсов DKI/DDI, предоставляя специфический интерфейс для устройств определенного класса. Так, в ОС семейства Windows NT для драйверов сетевых адаптеров предусмотрен интерфейс стандарта NDIS (Network Driver Interface Specification), в то время как драйверы сетевых транспортных протоколов взаимодействуют с верхними слоями сетевого программного обеспечения по интерфейсу TDI (Transport Driver Interface).

Для поддержки процесса разработки драйверов операционной системы обычно выпускается так называемый пакет **DDK** (Driver Development Kit), представляющий собой набор соответствующих инструментальных средств — библиотек, компиляторов и отладчиков.

Динамическая загрузка и выгрузка драйверов

Набор потенциально поддерживаемых данной ОС периферийных устройств всегда существенно шире набора устройств, которыми ОС должна управлять при установке на конкретной машине. Поэтому ценным свойством ОС является способность *динамически* загружать в оперативную память требуемый драйвер (без останова ОС) и выгружать его после того, как потребность в поддержке устройства миновала, что может существенно сэкономить системную область памяти. Поддержка динамической загрузки драйверов является практически обязательным требованием для современных универсальных операционных систем.

Альтернативой динамической загрузке драйверов при изменении текущей конфигурации внешних устройств компьютера является *повторная компиляция кода ядра* с требуемым набором драйверов, что создает между всеми компонентами ядра статические связи вместо динамических. Например, таким образом

решалась данная проблема в ранних версиях операционной системы Unix. При статических связях между ядром и драйверами структура ОС упрощается, но этот подход требует наличия исходных кодов модулей операционной системы, доступность которых скорее является исключением (для некоммерческих версий Unix/Linux), а не правилом. Кроме того, в этом варианте предыдущую работающую версию операционной системы необходимо остановить и заменить новой, а перерывы в работе ОС в некоторых применениях могут и не допускаться.

Поддержка файловых систем

Диски представляют собой особый род периферийных устройств, так как именно на них хранится большая часть как пользовательских, так и системных данных. Данные на дисках организуются в файловые системы, и свойства файловой системы во многом определяют свойства самой ОС. Популярность файловой системы часто приводит к ее миграции из «родной» ОС в другие операционные системы, например, файловая система FAT появилась первоначально в MS-DOS, но затем была реализована в OS/2, семействе MS Windows и многих реализациях Unix. Ввиду этого, поддержка нескольких популярных файловых систем для подсистемы ввода-вывода так же важна, как и поддержка широкого спектра периферийных устройств. Важно также, чтобы архитектура подсистемы ввода-вывода позволяла достаточно просто включать в ее состав новые типы файловых систем без необходимости переписывания кода. Обычно в операционной системе имеется специальный слой программного обеспечения, отвечающий за решение данной задачи, например слой VFS (Virtual File System) в версиях Unix на основе кода System V Release 4.

Синхронный и асинхронный режимы

По отношению к программному модулю, запросившему операцию ввода-вывода, она может выполняться в синхронном или асинхронном режиме. Смысл этих режимов тот же, что и для рассмотренных ранее системных вызовов:

- в **синхронном режиме** программный модуль приостанавливает свою работу до тех пор, пока операция ввода-вывода не будет завершена (рис. 7.1, *а*);
- в **асинхронном режиме** программный модуль продолжает выполняться в мультипрограммной среде одновременно с операцией ввода-вывода (рис. 7.1, *б*).

Отличие же заключается в том, что операция ввода-вывода может быть инициирована не только пользовательским процессом (в этом случае операция выполняется в рамках системного вызова), но и кодом ядра, например кодом подсистемы виртуальной памяти для считывания отсутствующей в памяти страницы.

Подсистема ввода-вывода должна предоставлять своим клиентам (пользовательским процессам и ядру) возможность выполнять как синхронные, так и асинхронные операции ввода-вывода в зависимости от потребностей вызывающей стороны.

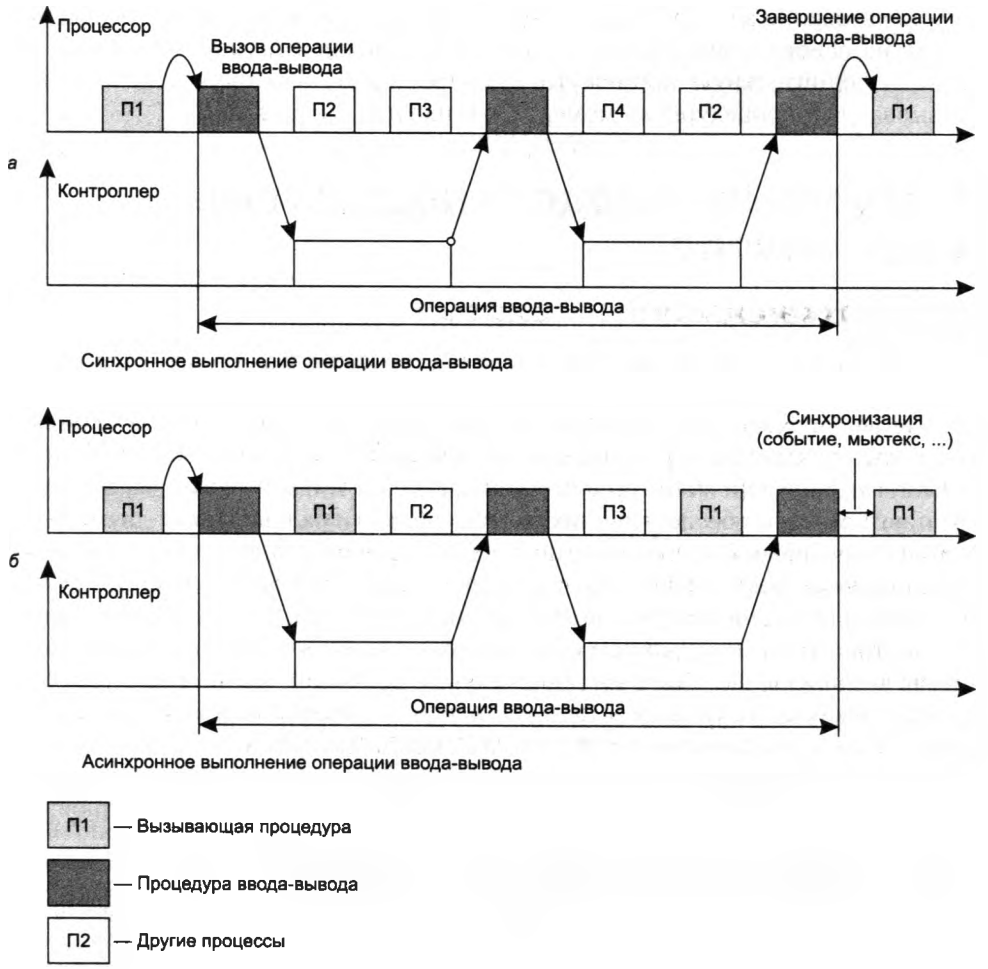


Рис. 7.1. Два режима выполнения операций ввода-вывода

Системные вызовы ввода-вывода чаще оформляются как синхронные процедуры в связи с тем, что такие операции длятся долго, и пользовательскому процессу или потоку все равно придется ждать получения результатов операции для того, чтобы продолжить свою работу.

Внутренние же вызовы операций ввода-вывода из модулей ядра обычно выполняются в виде асинхронных процедур, так как кодам ядра нужна свобода в выборе дальнейшего поведения после запроса операции ввода-вывода. Иногда и прикладному процессу может потребоваться выполнить асинхронную операцию ввода-вывода, например, если этот процесс является прикладным только по форме, а по сути он представляет собой часть микроядерной ОС. В таком случае ему необходима полная свобода действий и после того, как он вызвал операцию ввода-вывода.

Использование асинхронных процедур приводит к более гибким решениям, так как на основе асинхронного вызова всегда можно построить синхронный, создав дополнительную промежуточную процедуру, блокирующую выполнение вызвавшей процедуры до момента завершения ввода-вывода.

Многослойная модель подсистемы ввода-вывода

Общая схема

Многослойное построение программного обеспечения, характерное для операционных систем вообще, оказывается особенно естественным и полезным в случае подсистемы ввода-вывода. При большом разнообразии устройств ввода-вывода, обладающих существенно различными характеристиками (принтер и диски, графический монитор и сетевой адаптер и т. п.), иерархическая структура программного обеспечения позволяет соблюсти баланс между двумя весьма противоречивыми требованиями: с одной стороны, необходимо учесть все особенности каждого устройства, с другой стороны, обеспечить единое логическое представление и унифицированный интерфейс для устройств всех типов.

В соответствии с иерархическим подходом нижние слои подсистемы ввода-вывода должны включать индивидуальные драйверы, написанные для конкретных физических устройств, а верхние слои — обобщать процедуры управления этими устройствами, предоставляя общий интерфейс если не для всех устройств, то, по крайней мере, для групп устройств, обладающих некоторыми общими характеристиками, например, для принтеров определенного производителя или для всех матричных принтеров и т. п.

Многослойность структуры способствует решению большинства задач подсистемы ввода-вывода, делая, например более простым включение новых драйверов, поддержку нескольких файловых систем, динамическую загрузку-выгрузку драйверов и других.

Обобщенная структура подсистемы ввода-вывода представлена на рис. 7.2.

Из рисунка видно, что программное обеспечение ввода-вывода делится не только на горизонтальные слои, но и на вертикальные. Это объясняется тем, что для такого разнообразного мира, как внешние устройства, трудно обеспечить единообразие в разбиении функций управления на слои. Поэтому общий принцип многослойности остается справедливым, однако для устройств определенного типа он реализуется по-разному с разным количеством слоев и их функциями. В представленной структуре в качестве примера приведены три вертикальные подсистемы, управляющие дисками, графическими устройствами (такими как мониторы, принтеры и плоттеры) и сетевыми адаптерами. Естественно, к этому перечню можно добавить и другие, например, подсистему управления символьными терминалами или какими-либо специализированными устройствами, такими как аналого-цифровые и цифро-аналоговые преобразователи.

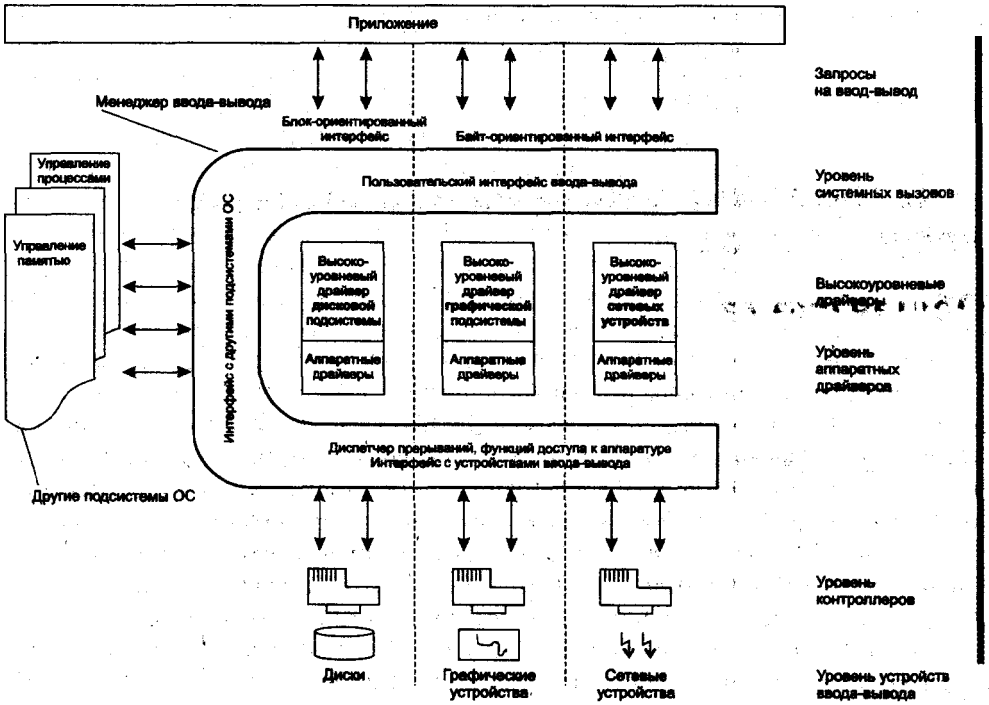


Рис. 7.2. Структура подсистемы ввода-вывода

В каждой вертикальной подсистеме существует несколько слоев модулей. Нижний слой образуют так называемые аппаратные драйверы устройств, название которых отражает тот факт, что они управляют аппаратурой внешних устройств, осуществляя обмен байтами и блоками байтов, и не имеют, как правило, дела с более высокоуровневыми вопросами логической организации данных, например с файлами или сложными графическими объектами. Функции вышележащих слоев в значительной степени зависят от типа вертикальной подсистемы.

Менеджер ввода-вывода

В подсистеме ввода-вывода наряду с модулями, отражающими специфику внешних устройств и образующими вертикальные подсистемы, существуют модули универсального назначения. Эти модули организуют согласованную работу всех остальных компонентов подсистемы ввода-вывода и взаимодействие с пользовательскими процессами и другими подсистемами ОС. Так же как функции управления устройствами, эти организующие функции распределены по всем уровням, образуя *оболочку*, называемую **менеджером ввода-вывода**.

Задачи менеджера ввода-вывода могут быть сведены к поддержке четырех интерфейсов:

- пользовательский интерфейс ввода-вывода;

- интерфейс с устройствами ввода-вывода;
- интерфейс с другими подсистемами ОС;
- внутренний интерфейс между компонентами подсистемы ввода-вывода.

Верхний слой менеджера составляют модули ОС, которые принимают от пользовательских процессов запросы на ввод-вывод, поступающие в виде системных вызовов, и переадресуют их отвечающим за определенный класс устройств модулям и драйверам, а также возвращают процессам результаты операций ввода-вывода. Таким образом, этот слой поддерживает *пользовательский интерфейс ввода-вывода*, создавая для прикладных программистов максимум удобств по манипулированию внешними устройствами и расположенными на них данными.

Нижний слой менеджера реализует непосредственное взаимодействие с контроллерами внешних устройств, экранируя драйверы от особенностей аппаратной платформы компьютера — шины ввода-вывода, системы прерываний и т. п. Этот слой принимает от драйверов запросы на обмен данными с регистрами контроллеров в некоторой обобщенной форме с использованием независимых от шины ввода-вывода адресации и формата, а затем преобразует эти запросы в зависящий от аппаратной платформы формат. Диспетчер прерываний, рассмотренный ранее, может входить в состав менеджера ввода-вывода или же представлять собой отдельный модуль ядра. В последнем случае менеджер ввода-вывода выполняет для диспетчера прерываний первичную обработку запросов прерываний, передавая диспетчеру обобщенные сведения об источнике запроса.

Еще одной функцией менеджера является организация взаимодействия модулей подсистемы ввода-вывода с модулями других подсистем ОС, таких как подсистемы управления процессами, виртуальной памяти и другие.

Важной функцией менеджера ввода-вывода является также создание некоторой удобной среды для взаимодействия компонентов подсистемы ввода-вывода. Для этого в менеджер ввода-вывода включают поддержку некоторого стандартного *внутреннего интерфейса* взаимодействия модулей ввода-вывода между собой, а также нагружают менеджер выполнением наиболее часто используемых при работе драйверов функций. Эти функции оформляются как системные процедуры, которые драйвер может вызывать для выполнения некоторых типовых действий. Примерами могут служить операции обмена с регистрами контроллера, ведение буферов для промежуточного хранения данных ввода-вывода, синхронизация работы нескольких драйверов, копирование данных из пользовательского пространства в пространство системы. Такой подход существенно облегчает разработку и включение новых драйверов и файловых систем в состав ОС.

Роль менеджера ввода-вывода выполняет среда STREAMS, существующая во многих версиях операционной системы Unix. Другим примером является менеджер ввода-вывода ОС семейства Windows NT. Он организует взаимодействие между модулями с помощью пакетов запросов ввода-вывода (I/O Request Packet, IRP). Получив запрос от процедуры системного вызова, менеджер фор-

мирует IRP и передает его нужному драйверу. Драйвер после выполнения запрошенной операции возвращает ответ в виде еще одного пакета IRP менеджеру, а тот, в свою очередь, может при необходимости передать этот пакет IRP другому драйверу. Менеджер позволяет драйверам задавать *взаимосвязи* (bindings) между собой, и на основании информации о взаимосвязях и происходит передача пакетов IRP. Кроме того, менеджер ОС семейства Windows NT поддерживает динамическую загрузку-выгрузку драйверов без останова системы.

Наличие стандартного внутреннего межмодульного интерфейса повышает устойчивость и улучшает расширяемость подсистемы ввода-вывода, хотя может несколько замедлить ее работу, так как любое разделение на слои и части приводит к дополнительным операциям взаимодействия по сравнению с монолитной организацией с прямыми передачами управления.

Многоуровневые драйверы

Первоначально термин «драйвер» применялся в более узком смысле, чем в настоящее время:

Драйвер — это программный модуль, который:

- входит в состав ядра операционной системы, работая в привилегированном режиме;
- непосредственно управляет внешним устройством, взаимодействуя с его контроллером с помощью команд ввода-вывода компьютера;
- обрабатывает прерывания от контроллера устройства;
- предоставляет прикладному программисту удобный логический интерфейс работы с устройством, экранируя от него низкоуровневые детали управления устройством и организации его данных;
- взаимодействует с другими модулями ядра ОС с помощью строго оговоренного интерфейса, описывающего формат передаваемых данных, структуру буферов, способы включения драйвера в состав ОС, способы вызова драйвера, набор общих процедур подсистемы ввода-вывода, которыми драйвер может пользоваться, и т. п.

Согласно этому определению драйвер вместе с контроллером устройства и прикладной программой воплощали идею многослойного подхода к организации программного обеспечения. Контроллер представлял нижний слой управления устройством, выполняющий операции в терминах блоков и агрегатов устройства (например, передвижение головки дисковода, побитная передача байта по двухпроводному кабелю). Драйвер выполнял более сложные операции, преобразуя, например, данные, адресуемые в терминах номеров цилиндров, головок и секторов диска, в линейную последовательность блоков или устанавливая логическое соединение между двумя модемами через телефонную сеть. В результате прикладная программа уже работала с данными, преобразованными в достаточно понятную для человека форму — файлами, таблицами баз данных, текстовыми окнами на мониторе и т. п., не вдаваясь в детали представления этих данных в устройствах ввода-вывода. Кроме того, помещение

драйвера в привилегированный режим и запрет для пользовательских процессов выполнять операции ввода-вывода защищали критически важные для работы самой ОС устройства ввода-вывода от ошибок прикладных программ, а также позволяли ОС надежно контролировать процесс разделения устройств и их данных между пользователями и процессами.

В описанной схеме драйверы не делились на слои. При этом они решали задачи разного уровня сложности, как самые примитивные, например, просто последовательно передавая контроллеру байты для дальнейшего использования, так и достаточно сложные, связанные с отработкой протокола взаимодействия между модемами или вычерчиванием на экране математических кривых.

Постепенно, по мере развития операционных систем и усложнения структуры подсистемы ввода-вывода, наряду с традиционными драйверами в операционных системах появились так называемые **высокоуровневые драйверы**, которые располагаются в общей модели подсистемы ввода-вывода над традиционными драйверами. Появление высокоуровневых драйверов можно считать дальнейшим развитием идеи многослойной организации подсистемы ввода-вывода. Вместо того чтобы концентрировать все функции по управлению устройством в одном программном модуле, во многих случаях гораздо эффективней распределить их между несколькими модулями в соседних слоях иерархии.

Традиционные драйверы, которые стали называть **аппаратными драйверами**, **низкоуровневыми драйверами** или **драйверами устройств**, подчеркивая их непосредственную связь с управляемым устройством, освобождаются от высокоуровневых функций. Выполняемые ими низкоуровневые операции составляют фундамент, на котором можно построить тот или иной набор операций в драйверах более высоких уровней.

При таком подходе повышается гибкость и расширяемость функций по управлению устройством — вместо жесткого набора функций, сосредоточенных в единственном драйвере, администратор ОС может выбрать требуемый набор функций, установив нужный высокоуровневый драйвер. Если различным приложениям необходимо взаимодействовать с различными логическими моделями одного и того же физического устройства, то для этого достаточно установить в системе несколько высокоуровневых драйверов, работающих поверх одного аппаратного драйвера.

Количество уровней драйверов в подсистеме ввода-вывода обычно не ограничивается каким-либо пределом, но на практике чаще всего используют от двух до пяти — слишком большое количество уровней может снизить скорость операций ввода-вывода.

Несколько драйверов, управляющих одним устройством, но на разных уровнях, можно рассматривать как набор отдельных драйверов или как один **многоуровневый драйвер**.

Высокоуровневые драйверы оформляются по тем же правилам и придерживаются тех же внутренних интерфейсов, что и аппаратные драйверы. Единственным отличием является то, что высокоуровневые драйверы, как правило, не вызываются по прерываниям, так как взаимодействуют с управляемым устрой-

ством через аппаратные драйверы. Менеджер ввода-вывода управляет драйверами однотипно независимо от того, к какому уровню он относится. При наличии большого количества драйверов разного уровня усложняются связи между ними, что, в свою очередь, усложняет их взаимодействие, и именно эта ситуация привела к стандартизации внутреннего интерфейса в подсистеме ввода-вывода и выделения специальной оболочки в виде менеджера ввода-вывода, выполняющего служебные функции по организации работы драйверов.

Рассмотрим, как общие принципы построения многоуровневых драйверов могут быть реализованы при управлении определенными типами внешних устройств.

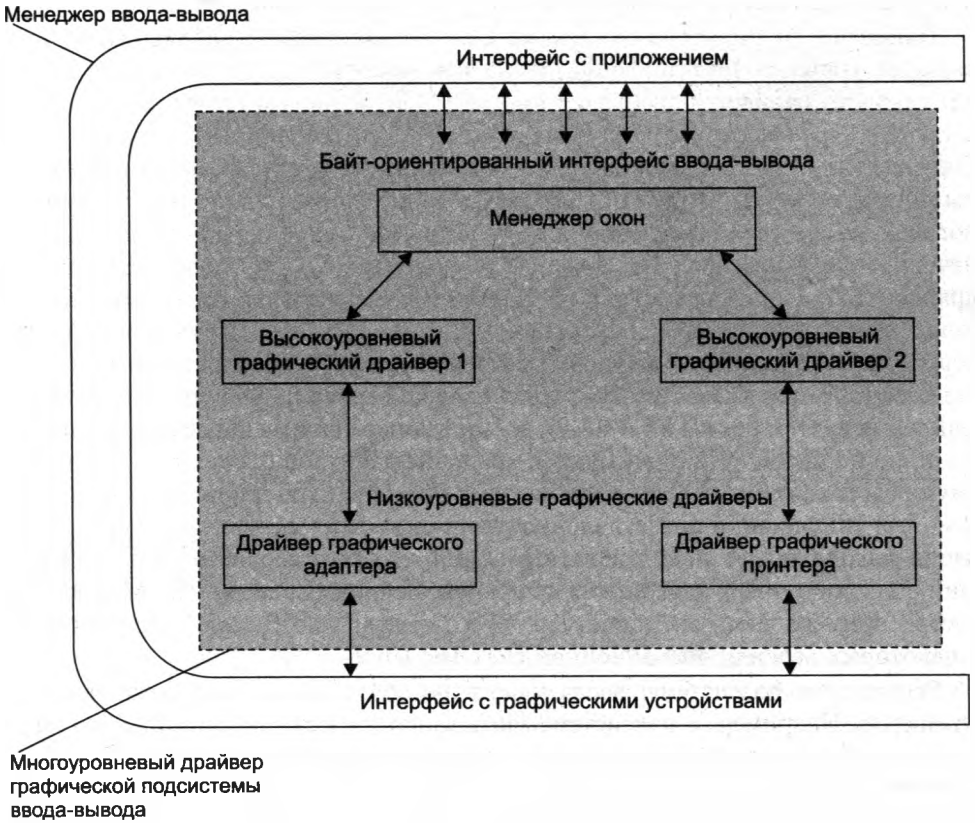


Рис. 7.3. Структура многоуровневого драйвера графической подсистемы

В подсистеме управления графическими устройствами, такими как графические мониторы и принтеры, существует несколько уровней драйверов (рис. 7.3). На нижнем уровне работают аппаратные драйверы, которые позволяют управлять конкретным графическим адаптером или принтером определенного типа, заставляя их выполнять некоторый набор примитивных графических операций: вывод точки, окружности, заполнение области цветом, вывод символов

и т. п. Высокоуровневые графические драйверы строят на базе этих операций более мощные операции, например масштабирование изображения, преобразование графического формата в соответствии с разрешающими возможностями устройства и т. п. Самый верхний уровень подсистемы составляет менеджер окон, создающий для каждого приложения виртуальный образ экрана в виде набора окон, в которые приложение может выводить свои графические данные. Менеджер управляет окнами, отображая их в определенной области физического экрана или делая их невидимыми, а также предоставляет к ним совместный доступ с контролем прав доступа. Менеджер окон уже не зависит от особенностей конкретного графического устройства, давая возможность высокоуровневым драйверам заниматься преобразованием форматов выводимых данных.

В подсистеме управления дисками аппаратные драйверы поддерживают для верхних уровней представление диска как последовательного набора блоков одинакового размера, преобразуя вместе с контроллером номер блока в более сложный адрес, состоящий из номеров цилиндра, головки и сектора (рис. 7.4). Однако такие понятия, как «файл» и «файловая система», аппаратные драйверы дисков не поддерживают — эти удобные для пользователя и программиста логические абстракции создаются на более высоком уровне программным обеспечением файловых систем, которое в современных ОС также оформляется как драйвер, только высокоуровневый. Наличие универсальной среды, создаваемой менеджером ввода-вывода, позволяет достаточно просто решить проблему поддержки в ОС нескольких файловых систем одновременно. Для этого в ОС устанавливается несколько высокоуровневых драйверов (на рисунке это драйверы файловых систем ufs, NTFS и FAT), работающих с общими аппаратными драйверами, но по-своему организующими хранение данных в блоках диска и по-своему представляющими файловую систему пользователю и прикладным процессам. Для унификации представления различных файловых систем в подсистеме ввода-вывода может использоваться общий драйвер верхнего уровня, играющий роль диспетчера нескольких драйверов файловых систем. На рисунке в качестве примера показан диспетчер VFS (Virtual File System), применяемый в некоторых версиях операционной системы Unix.

Все модули подсистемы ввода-вывода не обязательно оформляются в виде драйверов. Например, в подсистеме управления дисками обычно имеется такой модуль, как дисковый кэш, который служит для кэширования блоков дисковых файлов в оперативной памяти. Достаточно специфические функции кэша делают нецелесообразным оформление его в виде драйвера, взаимодействующего с другими модулями ОС только с помощью менеджера ввода-вывода. Другим примером модуля, который чаще всего не оформляется в виде драйвера, является диспетчер окон графического интерфейса. Иногда этот модуль вообще выносится из ядра ОС и реализуется в виде пользовательского процесса. Таким образом был реализован диспетчер окон (а также высокоуровневые графические драйверы) в Windows NT 3.5 и 3.51, но этот микроядерный подход заметно замедлял графические операции, поэтому в Windows NT 4.0 диспетчер окон и высокоуровневые графические драйверы, а также библиотека GDI (Graphic

Device Interface — интерфейс графического устройства) были перенесены в пространство ядра.

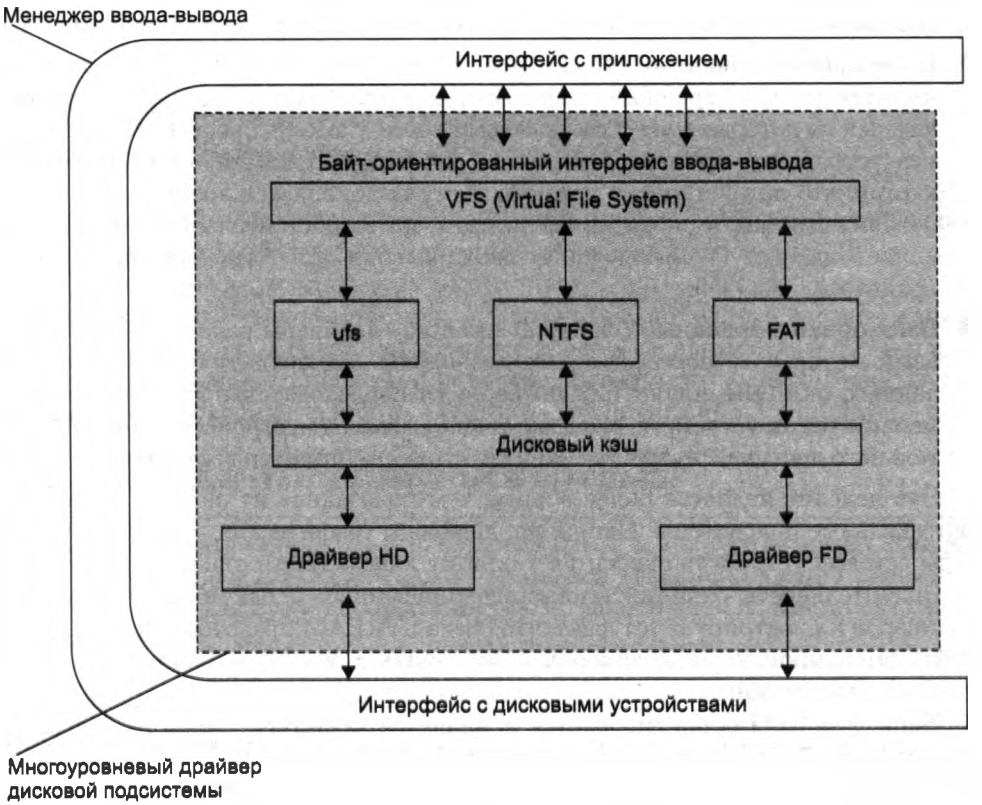


Рис. 7.4. Структура многоуровневого драйвера дисковой подсистемы

Аппаратные драйверы после запуска операции ввода-вывода должны своевременно реагировать на завершение контроллером заданного действия, и для решения этой задачи они *взаимодействуют с системой прерываний*. Драйверы более высоких уровней вызываются уже не по прерываниям, а по инициативе аппаратных драйверов или драйверов вышележащего уровня. Не все процедуры аппаратного драйвера нужно вызывать по прерываниям, поэтому драйвер обычно имеет определенную структуру, в которой выделяется **процедура обслуживания прерываний** (Interrupt Service Routine, ISR), — она и вызывается диспетчером прерываний при поступлении запроса от соответствующего устройства. Диспетчер прерываний можно считать частью подсистемы ввода-вывода, как это показано на рис. 7.2, а можно считать и независимым модулем ядра ОС, так как он служит не только для вызова процедур обслуживания прерываний драйверов, но и для диспетчеризации прерываний других типов.

Блок-ориентированные и байт-ориентированные драйверы

Драйверы могут быть разделены на два больших класса:

- **Блок-ориентированные** (block-oriented) драйверы управляют устройствами прямого доступа, хранящих информацию в блоках фиксированного размера, каждый из которых *имеет собственный адрес*. Самое распространенное внешнее устройство прямого доступа — диск. Адресуемость блоков приводит к тому, что для устройств прямого доступа появляется возможность кэширования данных в оперативной памяти, и это обстоятельство значительно влияет на общую организацию ввода-вывода для блок-ориентированных драйверов.
- **Байт-ориентированные** (character-oriented) драйверы работают с устройствами, которые *не адресуемы* и не позволяют производить операцию поиска данных, они генерируют или потребляют последовательности байтов. Примерами таких устройств, которые также называют устройствами последовательного доступа, служат терминалы, строчные принтеры, сетевые адаптеры.

Это деление является более общим, чем показанное на рис. 7.2 деление на вертикальные подсистемы. Например, драйверы графических устройств и драйверы сетевых устройств относятся к одному классу байт-ориентированных.

Значительность отличий блок-ориентированных и байт-ориентированных драйверов иллюстрирует тот факт, что среда STREAMS разработана только для байт-ориентированных драйверов и включить в нее блок-ориентированный драйвер невозможно.

Блок- или байт-ориентированность является характеристикой не только самого устройства, но и драйвера. Очевидно, что если устройство не поддерживает обмен адресуемыми блоками данных, а позволяет записывать или считывать последовательность байтов, то и устройство, и его драйвер можно назвать байт-ориентированными.

Для байт-ориентированного устройства невозможно разработать блок-ориентированный драйвер. Однако блок-ориентированным устройством можно управлять и с помощью байт-ориентированного драйвера.

Так, диск можно рассматривать не только как набор блоков, но и набор байтов, первый из которых начинает первый блок диска, а последний завершает последний блок. Физический обмен с контроллером устройства по-прежнему осуществляется блоками, но байт-ориентированный драйвер устройства будет преобразовывать блоки в последовательности байтов. Для устройств прямого доступа часто разрабатывают пару драйверов, чтобы к устройству можно было обращаться и по байт-ориентированному, и по блок-ориентированному интерфейсам, в зависимости от потребностей.

Деление всех драйверов на блок-ориентированные и байт-ориентированные оказывается полезным для структурирования подсистемы управления вводом-выводом. Тем не менее необходимо учитывать, что эта схема является упрощенной — имеются внешние устройства, драйверы которых не относятся ни

к одному из указанных классов. Таковым, например, является таймер, который, с одной стороны, не содержит адресуемой информации, а с другой стороны, не порождает потока байтов. Это устройство только выдает сигнал прерывания в некоторые моменты времени.

Логическая организация файловой системы

Одной из основных задач операционной системы является предоставление пользователю удобных средств работы с данными, хранящимися на дисках. Для этого ОС подменяет физическую структуру хранящихся данных некоторой удобной для пользователя логической моделью — системой иерархически организованных каталогов и файлов. Для пользователя это абстрактное представление материализуется в виде набора ярлыков или списка, выводимого на экран такими утилитами, как Проводник Windows (Windows Explorer).

Цели и задачи файловой системы

Файл — это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные, а также собственно хранимые в этой области данные и набор атрибутов, позволяющих ОС манипулировать этими данными.

Основные цели использования файла перечислены ниже.

- *Долговременное и надежное хранение информации.* Долговременность достигается за счет использования запоминающих устройств, не зависящих от питания[1], а высокая надежность определяется средствами защиты доступа к файлам и общей организацией программного кода ОС, при которой сбои аппаратуры чаще всего не разрушают информацию, хранящуюся в файлах.
- *Совместное использование информации.* Файлы обеспечивают естественный и простой способ разделения информации между приложениями и пользователями за счет наличия понятного человеку символьного имени и постоянства хранимой информации и расположения файла. Пользователь должен иметь удобные средства работы с файлами, включая каталоги-справочники, объединяющие файлы в группы, средства поиска файлов по признакам, набор команд для создания, модификации и удаления файлов. Файл может быть создан одним пользователем, а применяться совсем другим, при этом создатель файла или администратор могут определить права доступа к нему других пользователей. Эти цели реализуются в ОС файловой системой.

[1] Файлы хранятся в памяти, не зависящей от энергопитания, однако имеется исключение — так называемый электронный диск, представляющий собой созданную в оперативной памяти структуру, имитирующую файловую систему.

Файловая система (ФС) — это часть операционной системы, включающая:

- совокупность всех файлов на диске;
- наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске;
- комплекс системных программных средств, реализующих различные операции с файлами, такие как создание, уничтожение, чтение, запись, именование и поиск файлов.

Файловая система позволяет программам обходится набором достаточно простых операций для выполнения действий над некоторым абстрактным объектом, представляющим файл. При этом программистам не нужно иметь дело с деталями действительного расположения данных на диске, буферизацией данных и другими низкоуровневыми проблемами передачи данных с долговременного запоминающего устройства. Все эти функции файловая система берет на себя. Файловая система распределяет дисковую память, поддерживает именование файлов, отображает имена файлов на соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление файлов.

Таким образом, файловая система играет роль промежуточного слоя, экранирующего все сложности физической организации долговременного хранилища данных и создающего для программ более простую логическую модель этого хранилища, а также предоставляя им набор удобных в использовании команд для манипулирования файлами.

Задачи, решаемые ФС, зависят от способа организации вычислительного процесса в целом.

Самый простой тип — это ФС в однопользовательских и однопрограммных ОС, к числу которых относится, например, MS-DOS. Основные функции в *однопользовательских однопрограммных ФС* нацелены на решение следующих задач:

- именование файлов;
- программный интерфейс для приложений;
- отображения логической модели файловой системы на физическую организацию хранилища данных;
- устойчивость файловой системы к сбоям питания, ошибкам аппаратных и программных средств.

Задачи ФС усложняются в операционных *однопользовательских мультипрограммных* ОС, которые, хотя и предназначены для работы одного пользователя, но дают ему возможность запускать одновременно несколько процессов. Одной из первых ОС этого типа стала OS/2. К перечисленным задачам добавляется новая задача совместного доступа к файлу из нескольких процессов. Файл

в этом случае является разделяемым ресурсом, а, значит, файловая система должна решать весь комплекс проблем, связанных с такими ресурсами.

В частности, в однопользовательских мультипрограммных ФС должны быть предусмотрены средства блокировки файла и его частей, предотвращения гонок, исключение тупиков, согласование копий и т. п.

В *многопользовательских системах ФС* появляется еще одна задача — защита файлов одного пользователя от несанкционированного доступа другого пользователя.

Еще более сложными становятся функции ФС, которая работает в составе *сетевой* ОС. Эта тема рассматривается в последней главе книги, посвященной управлению сетевыми ресурсами.

Типы файлов

Файловые системы поддерживают несколько функционально различных типов файлов, в число которых, как правило, входят:

- обычные файлы;
- файлы-каталоги;
- специальные файлы;
- именованные конвейеры;
- файлы, отображаемые на память.

Обычные файлы, или просто **файлы**, содержат информацию произвольного характера, которую заносит в них пользователь или которая образуется в результате работы системных и пользовательских программ. Большинство современных операционных систем (например, Unix, Windows) никак не ограничивают и не контролируют содержимое и структуру обычного файла. Содержимое обычного файла определяется приложением, которое с ним работает. Например, текстовый редактор создает текстовые файлы, состоящие из строк символов, представленных в каком-либо коде. Это могут быть документы, исходные тексты программ и т. п. Текстовые файлы можно прочитать на экране и распечатать на принтере. Двоичные файлы не используют коды символов, они часто имеют сложную внутреннюю структуру, например исполняемый код программы или архивный файл. Все операционные системы должны уметь распознавать хотя бы один тип файлов — их собственные исполняемые файлы.

Каталоги — это особый тип файлов, которые содержат системную справочную информацию о наборе файлов, сгруппированных пользователями по какому-либо неформальному признаку (например, в одну группу объединяются файлы, содержащие документы одного договора, или файлы, составляющие один программный пакет). Во многих операционных системах в каталог могут входить файлы любых типов, в том числе другие каталоги, за счет чего образуется древовидная структура, удобная для поиска. Каталоги устанавливают соответствие между именами файлов и их характеристиками, используемыми файловой системой для управления файлами. В число таких характеристик входит, в частности, информация (или указатель на другую структуру, содер-

жащую эти данные) о типе файла и расположении его на диске, правах доступа к файлу и датах его создания и модификации. Во всех остальных отношениях каталоги рассматриваются файловой системой как обычные файлы.

Специальные файлы — это фиктивные файлы, ассоциированные с устройствами ввода-вывода, которые используются для унификации механизма доступа к файлам и внешним устройствам. Специальные файлы позволяют пользователю выполнять операции ввода-вывода посредством обычных команд записи в файл или чтения из файла. Эти команды обрабатываются сначала программами файловой системы, а затем на некотором этапе выполнения запроса преобразуются операционной системой в команды управления соответствующим устройством.

Современные файловые системы поддерживают и другие типы файлов, такие как символьные связи, именованные конвейеры, отображаемые на память файлы. Они будут рассмотрены позже.

Иерархическая структура файловой системы

Пользователи обращаются к файлам по символьным именам. Однако способности человеческой памяти ограничивают количество имен объектов, к которым пользователь может обращаться по имени. Иерархическая организация пространства имен позволяет значительно расширить эти границы. Именно поэтому большинство файловых систем имеет иерархическую структуру, в которой уровни создаются за счет того, что каталог более низкого уровня может входить в каталог более высокого уровня, при этом частным случаем иерархической структуры является одноуровневая организация, когда все файлы входят в один каталог (рис. 7.5, *а*). Граф, описывающий иерархию каталогов, может быть деревом (рис. 7.5, *б*) или сетью (рис. 7.5, *в*). Каталоги образуют **дерево**, если файлу разрешено входить только в один каталог, и **сеть**, если файл может входить сразу в несколько каталогов.

Например, в MS-DOS и Windows каталоги образуют древовидную структуру, а в Unix — сетевую. В древовидной структуре каждый файл является **листом**. Каталог самого верхнего уровня называется **корневым каталогом**, или **корнем** (root).

При такой организации пользователь освобожден от необходимости помнить имена всех файлов, ему достаточно примерно представлять, к какой группе может быть отнесен тот или иной файл, чтобы путем последовательного просмотра каталогов найти его. Иерархическая структура удобна для многопользовательской работы: каждый пользователь со своими файлами локализуется в своем каталоге или поддереве каталогов, и вместе с тем все файлы в системе логически связаны.

Имена файлов

Все типы файлов имеют символьные имена.

В иерархически организованных файловых системах обычно используются следующие типы имен файлов:

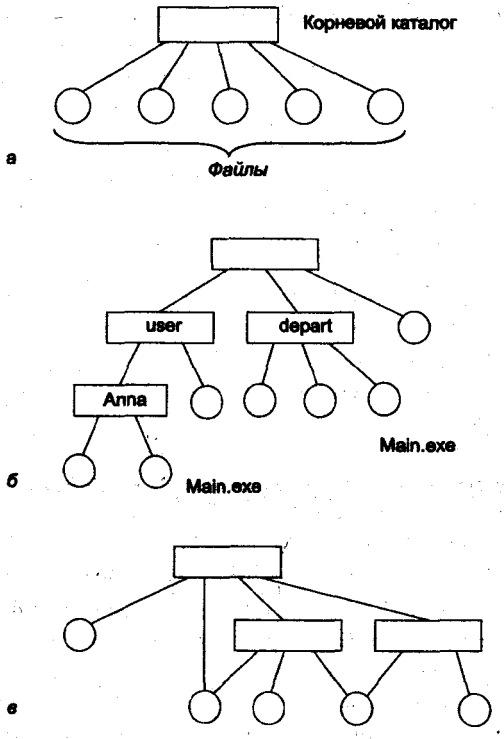


Рис. 7.5. Иерархия файловых систем

- простое (короткое) символьное имя;
- полное (составное) символьное имя;
- относительное символьное имя;
- уникальное имя (числовой идентификатор).

Простое, или **короткое**, **символьное имя** идентифицирует файл в пределах одного каталога. Простые имена присваивают файлам пользователи и программисты, при этом они должны учитывать ограничения ОС как на номенклатуру символов, так и на длину имени. До сравнительно недавнего времени эти границы были весьма узкими. Так, в известной файловой системе FAT MS-DOS длина имен ограничивалась схемой 8.3 (8 символов — собственно имя, 3 символа — расширение имени), а в файловой системе s5, поддерживаемой многими версиями ОС Unix, простое символьное имя не могло содержать более 14 символов. Однако пользователю гораздо удобнее работать с длинными именами, поскольку они позволяют давать файлам легко запоминающиеся названия, ясно говорящие о том, что содержится в том или ином файле. Поэтому современные файловые системы, а также усовершенствованные варианты уже существовавших файловых систем, как правило, поддерживают длинные простые символьные имена файлов. Например, в файловых системах NTFS и FAT32,

входящих в состав ОС семейства Windows, имя файла может содержать до 255 символов.

Примеры простых имен файлов и каталогов:

- quest_u1.doc;
- task-entran.exe;
- приложение к CD 254L на русском языке.doc;
- installable filesystem manager.doc.

В иерархических файловых системах разным файлам разрешено иметь одинаковые простые символьные имена при условии, что они принадлежат разным каталогам. То есть здесь работает схема *много файлов — одно простое имя*. Для однозначной идентификации файла в таких системах используется так называемое полное имя.

Полное имя представляет собой цепочку простых символьных имен всех каталогов, через которые проходит путь от корня до данного файла. Таким образом, полное имя является **составным именем**, в котором простые имена отделены друг от друга принятым в ОС разделителем. Часто в качестве разделителя используется прямой или обратный слэш, при этом принято не указывать имя корневого каталога. На рис. 7.5, *б* два файла имеют простое имя Main.exe, однако их составные имена /depart/Main.exe и /user/Anna/Main.exe различаются.

В древовидной файловой системе между файлом и его полным именем имеется взаимно однозначное соответствие *один файл — одно полное имя*. В файловых системах, имеющих сетевую структуру, файл может входить в несколько каталогов, а значит, иметь несколько полных имен, здесь справедливо соответствие *один файл — много полных имен*. В обоих случаях файл однозначно идентифицируется полным именем. Файл может быть идентифицирован также относительным именем.

Относительное имя файла определяется через понятие «текущий каталог». Для каждого пользователя в каждый момент времени один из каталогов файловой системы является текущим, причем этот каталог выбирается самим пользователем по команде ОС. Файловая система фиксирует имя текущего каталога, чтобы затем использовать его как дополнение к относительным именам для образования полного имени файла. При применении относительных имен пользователь идентифицирует файл цепочкой имен каталогов, через которые проходит маршрут от текущего каталога до данного файла. Например, если текущим каталогом является каталог /user, то относительное имя файла /user/Anna/Main.exe выглядит следующим образом: Anna/Main.exe.

В некоторых операционных системах разрешено присваивать одному и тому же файлу несколько простых имен, которые можно интерпретировать как псевдонимы. В этом случае, так же как в системе с сетевой структурой, устанавливается соответствие *один файл — много полных имен*, так как каждому простому имени файла соответствует, по крайней мере, одно полное имя.

Символьные имена удобны для пользователя, но не для операционной системы. Для своих внутренних целей ОС присваивает файлу **уникальное имя**, так

что справедливо соотношение *один файл — одно уникальное имя*. Уникальное имя существует наряду с одним или несколькими символьными именами, присваиваемыми файлу пользователями или приложениями. Уникальное имя представляет собой числовой идентификатор и предназначено только для использования операционной системой. Примером такого уникального имени файла является номер индексного дескриптора в системе Unix.

Монтирование

В общем случае вычислительная система может иметь несколько дисковых устройств. Даже типичный персональный компьютер обычно имеет один накопитель на жестком диске, один накопитель на гибких дисках и накопитель для компакт-дисков. Мощные же компьютеры, как правило, оснащены большим количеством дисковых накопителей, на которые устанавливаются пакеты дисков. Более того, даже одно физическое устройство с помощью средств операционной системы может быть представлено в виде нескольких логических устройств, в частности, путем разбиения дискового пространства на разделы. Возникает вопрос, каким образом организовать хранение файлов в системе, имеющей несколько устройств внешней памяти?

Первое решение состоит в том, что на каждом из устройств размещается автономная файловая система, то есть файлы, находящиеся на этом устройстве, описываются деревом каталогов, никак не связанным с деревьями каталогов на других устройствах. В таком случае для однозначной идентификации файла пользователь наряду с составным символьным именем файла должен указывать идентификатор логического устройства. Примером такого автономного существования файловых систем является операционная система MS-DOS, в которой полное имя файла включает буквенный идентификатор логического диска. Так, при обращении к файлу, расположенному на диске A, пользователь должен указать имя этого диска: `A:\privat\letter\uni\let1.doc`[1].

Другим вариантом является такая организация хранения файлов, при которой пользователю предоставляется возможность объединять файловые системы, находящиеся на разных устройствах, в единую файловую систему, описываемую единым деревом каталогов. Такая операция называется **монтированием**.

Рассмотрим, как осуществляется эта операция на примере ОС Unix.

Среди всех имеющихся в системе логических дисковых устройств операционная система выделяет одно устройство, называемое системным. Пусть имеются две файловые системы, расположенные на разных логических дисках, причем один из дисков является системным (рис. 7.6).

Файловая система, расположенная на системном диске, назначается **корневой**. Для связи иерархий файлов в корневой файловой системе выбирается некоторый существующий каталог, в данном примере — каталог `man`. После

[1] На практике чаще используется относительная форма именования, которая не включает имя диска и цепочку имен каталогов верхнего уровня, заданных по умолчанию.

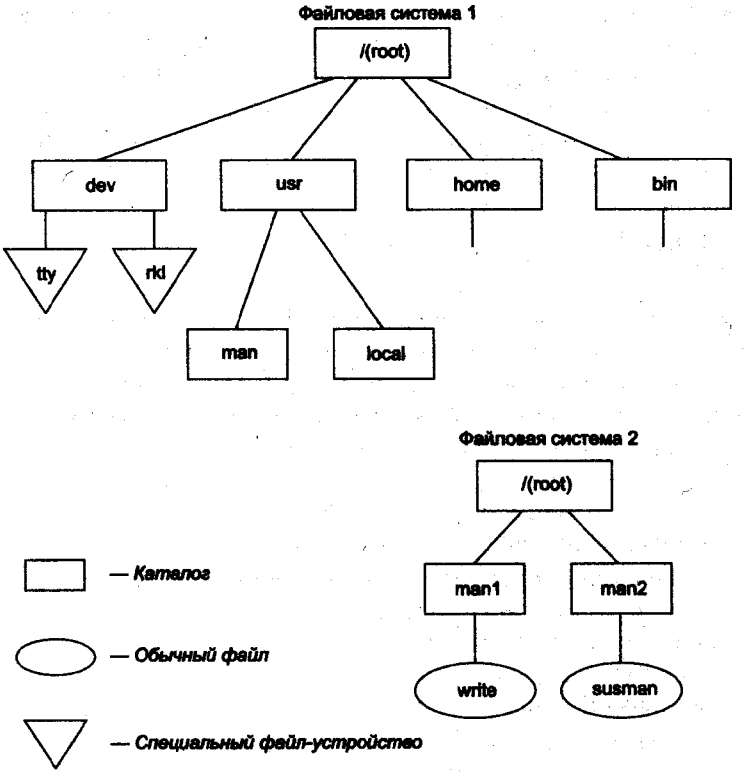


Рис. 7.6. Две файловые системы до монтирования

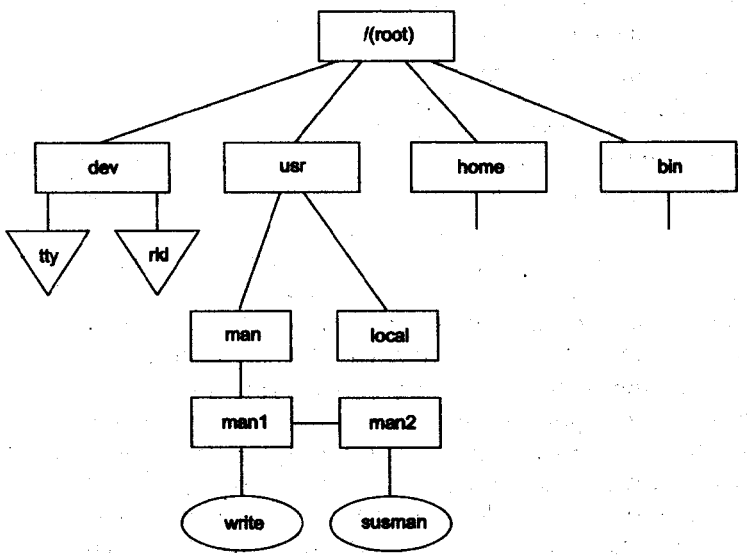


Рис. 7.7. Общая файловая система после монтирования

выполнения монтирования выбранный каталог man становится корневым каталогом второй файловой системы. Через этот каталог монтируемая файловая система подсоединяется как поддерево к общему дереву (рис. 7.7).

После монтирования общей файловой системы для пользователя нет логической разницы между корневой и смонтированной файловыми системами, в частности, именование файлов производится так же, как если бы система с самого начала была единой.

Атрибуты файлов

Понятие «файл» включает не только хранимые им данные и имя, но и атрибуты. **Атрибуты** — это информация, описывающая свойства файла. Примеры возможных атрибутов файла:

- тип файла (обычный файл, каталог, специальный файл и т. п.);
- владелец файла;
- создатель файла;
- пароль для доступа к файлу;
- информация о разрешенных операциях доступа к файлу;
- времена создания, последнего доступа и последнего изменения;
- текущий размер файла;
- максимальный размер файла;
- признак «только для чтения»;
- признак «скрытый файл»;
- признак «системный файл»;
- признак «архивный файл»;
- признак «двоичный/символьный»;
- признак «временный» (удалить после завершения процесса);
- признак блокировки;
- длина записи в файле;
- указатель на ключевое поле в записи;
- длина ключа.

Набор атрибутов файла определяется спецификой файловой системы: в файловых системах разного типа для характеристики файлов могут использоваться разные наборы атрибутов. Например, в файловых системах, поддерживающих неструктурированные файлы, нет необходимости в трех последних атрибутах из приведенного списка, поскольку они связанны со структуризацией файла. В однопользовательской ОС в наборе атрибутов будут отсутствовать атрибуты, имеющие отношение к пользователям и защите, такие как владелец файла, создатель файла, пароль для доступа к файлу, информация о разрешенном доступе к файлу.

Пользователь может получать доступ к атрибутам с помощью средств, предоставленных для этих целей файловой системой. Обычно разрешается читать значения любых атрибутов, а изменять — только некоторые. Например, пользователь может изменить права доступа к файлу (при условии, что он обладает необходимыми для этого полномочиями), но изменять дату создания или текущий размер файла ему не разрешается.

Значения атрибутов файлов могут непосредственно содержаться в каталогах, как это было сделано в файловой системе MS-DOS. На рисунке 7.8, *а* представлена структура записи в каталоге, содержащая простое символьное имя и атрибуты файла. Здесь буквами обозначены признаки файла: *R* — только для чтения, *A* — архивный, *H* — скрытый, *S* — системный.

Рис. 7.8. Структура каталогов: *а* — структура записи каталога MS-DOS (32 байта), *б* — структура записи каталога ОС Unix

Другим вариантом является размещение атрибутов в специальных таблицах, когда в каталогах содержатся только ссылки на эти таблицы. Такой подход реализован, например, в файловой системе ufs ОС Unix. В этой файловой системе структура каталога очень простая. Запись о каждом файле содержит короткое символьное имя файла и указатель на **индексный дескриптор файла** — так называется в ufs таблица, в которой сосредоточены значения атрибутов файла (рис. 7.8, *б*).

В том и другом вариантах каталоги обеспечивают связь между именами файлов и собственно файлами. Однако подход, когда имя файла отделено от его атрибутов, делает систему более гибкой. Например, файл может быть легко включен сразу в несколько каталогов. Записи об этом файле в разных каталогах могут содержать разные простые имена, но в поле ссылки будет указан один и тот же номер индексного дескриптора.

Логическая организация файла

В общем случае данные, содержащиеся в файле, имеют некую *логическую структуру*. Эта структура является базой при разработке программы, предназначенной для обработки этих данных. Например, чтобы текст мог быть правильно

выведен на экран, программа должна иметь возможность выделить отдельные слова, строки, абзацы и т. д. Признаками, отделяющими один структурный элемент от другого, могут служить определенные кодовые последовательности или просто известные программе значения смещений этих структурных элементов относительно начала файла. Поддержание структуры данных может быть либо целиком возложено на приложение, либо в той или иной степени эту работу может взять на себя файловая система.

В первом случае, когда все действия, связанные со структуризацией и интерпретацией содержимого файла, целиком относятся к ведению приложения, файл представляется ФС *неструктурированной* последовательностью данных. Приложение формулирует запросы к файловой системе на ввод-вывод, используя общие для всех приложений системные средства, например, указывая смещение от начала файла и количество байтов, которые необходимо считать или записать. Поступивший к приложению поток байтов интерпретируется в соответствии с заложенной в программе логикой. Например, компилятор генерирует, а редактор связей воспринимает вполне определенный формат объектного модуля программы. При этом формат файла, в котором хранится объектный модуль, известен только этим программам. Подчеркнем, что интерпретация данных никак не связана с действительным способом их хранения в файловой системе.

Модель файла, в соответствии с которой содержимое файла представляется неструктурированной последовательностью (потоком) байтов, стала популярной вместе с ОС Unix, а теперь она широко используется и в большинстве современных ОС, в том числе во всех версиях ОС Windows и NetWare. Неструктурированная модель файла позволяет легко организовать разделение файла между несколькими приложениями: разные приложения могут по-своему структурировать и интерпретировать данные, содержащиеся в файле.

Другая модель файла, которая применялась в ОС OS/360, DEC RSX и VMS, а в настоящее время используется достаточно редко, — это *структурированный* файл. В этом случае поддержание структуры файла поручается файловой системе. Файловая система видит файл как упорядоченную последовательность *логических записей*. Приложение может обращаться к ФС с запросами на ввод-вывод на уровне записей, например, «считать запись 25 из файла FILE.DOC». Файловая система должна обладать информацией о структуре файла, достаточной для того, чтобы выделить любую запись. ФС предоставляет приложению доступ к записи, а вся дальнейшая обработка данных, содержащихся в этой записи, выполняется приложением. Развитием этого подхода стали системы управления базами данных (СУБД), которые поддерживают не только сложную структуру данных, но и взаимосвязи между ними.

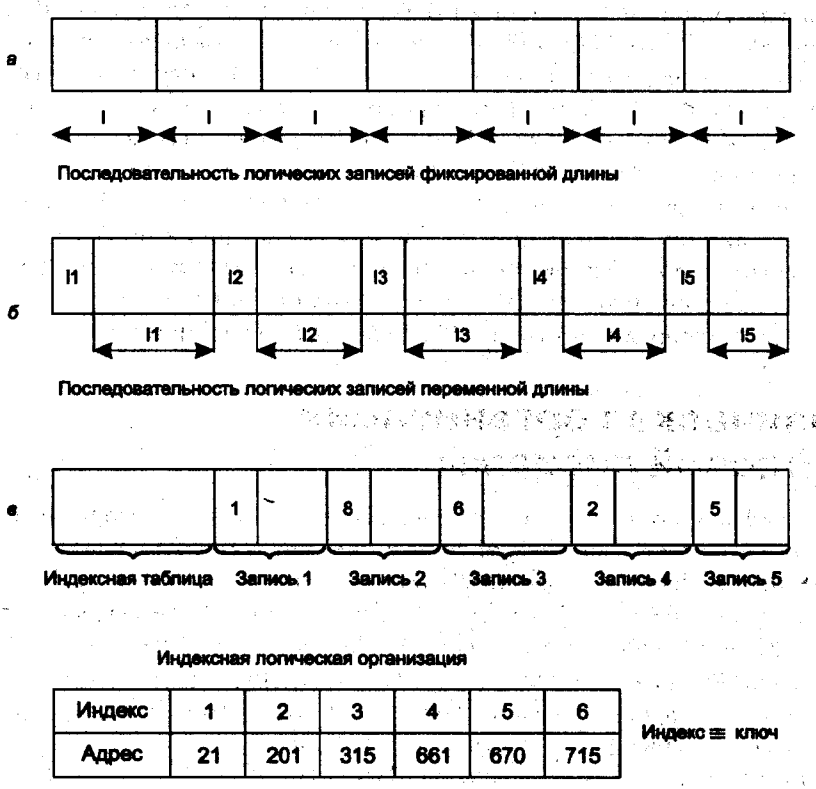
Логическая запись является наименьшим элементом данных, которым может оперировать программист при организации обмена с внешним устройством.

Даже если физический обмен с устройством осуществляется большими единицами, операционная система должна обеспечивать программисту доступ к отдельной логической записи.

Файловая система может использовать два способа доступа к логическим записям: читать или записывать логические записи последовательно (**последовательный доступ**) или позиционировать файл на запись с указанным номером (**прямой доступ**).

Очевидно, что ОС не может поддерживать все возможные способы структурирования данных в файле, поэтому в тех ОС, в которых вообще поддерживается логическая структуризация файлов, поддержка реализована для небольшого числа широко распространенных схем логической организации файла.

К числу таких способов структуризации относится представление данных в виде **записей фиксированной длины** (рис. 7.9, *а*). В этом случае доступ к n-й записи осуществляется либо путем последовательного чтения ($n - 1$) предшествующих записей, либо прямо по адресу, вычисленному по ее порядковому номеру. Например, если L — длина записи, то начальный адрес n-й записи равен $L \times n$. Заметим, что при такой логической организации размер записи фиксирован в пределах файла, а записи в различных файлах, принадлежащих одной и той же файловой системе, могут иметь различный размер.



Индексная логическая организация

Индекс	1	2	3	4	5	6
Адрес	21	201	315	661	670	715

Индекс ≡ ключ

Рис. 7.9. Способы логической организации файлов

Другой способ структуризации состоит в представлении данных в виде **последовательности записей переменного размера**. Если расположить значения

длин записей так, как это показано на рис. 7.9, *б*, то для поиска нужной записи система должна последовательно считать все предшествующие записи. Вычислить адрес нужной записи по ее номеру при такой логической организации файла невозможно, а следовательно, не может быть применен более эффективный метод прямого доступа.

Файлы, доступ к записям которых осуществляется последовательно, по номерам позиций, называются **неиндексированными**, или **последовательными, файлами**.

Другим типом файлов являются **индексированные файлы**; они допускают более быстрый прямой доступ к отдельной логической записи. В индексированном файле (рис. 7.9, *в*) записи имеют одно или более ключевых (индексных) полей и могут адресоваться путем указания значений этих полей. Для быстрого поиска данных в индексированном файле предусматривается специальная индексная таблица, в которой значениям ключевых полей ставится в соответствие адрес внешней памяти. Этот адрес может указывать либо непосредственно на искомую запись, либо на некоторую область внешней памяти, занимаемую несколькими записями, в число которых входит искомая запись. В последнем случае говорят, что файл имеет *индексно-последовательную* организацию, так как поиск включает два этапа: прямой доступ по индексу к указанной области диска, а затем последовательный просмотр записей в указанной области. Ведение индексных таблиц берет на себя файловая система. Понятно, что записи в индексированных файлах могут иметь произвольную длину.

Все сказанное в большей степени относится к обычным файлам, которые могут быть как структурированными, так и неструктурированными. Что же касается других типов файлов, таких как каталоги или файлы типа «символьная связь», то они обладают вполне определенной специфической для своего типа структурой, которая известна и «понятна» файловой системе.

Физическая организация файловой системы

Представление пользователя о файловой системе, как об иерархически организованном множестве информационных объектов, имеет мало общего с порядком хранения файлов на диске. Файл, имеющий образ цельного, непрерывающегося набора байтов, на самом деле очень часто разбросан «кусочками» по всему диску, причем это разбиение никак не связано с логической структурой файла, например, его отдельная логическая запись может быть расположена в несмежных секторах диска. Логически объединенные файлы из одного каталога совсем не обязаны соседствовать на диске.

Принципы размещения файлов, каталогов и системной информации на реальном устройстве описываются **физической организацией файловой системы**.

Очевидно, что разные файловые системы имеют разную физическую организацию.

Диски, разделы, секторы, кластеры

Основным типом устройства, которое используется в современных вычислительных системах для хранения файлов, является дисковый накопитель. Дисковые накопители предназначены для считывания и записи данных на жесткие и гибкие магнитные диски.

Жесткий диск состоит из одной или нескольких стеклянных или металлических **пластин**, каждая из которых покрыта с одной или двух сторон магнитным материалом. Таким образом, диск в общем случае состоит из **пакета** пластин (рис. 7.10).

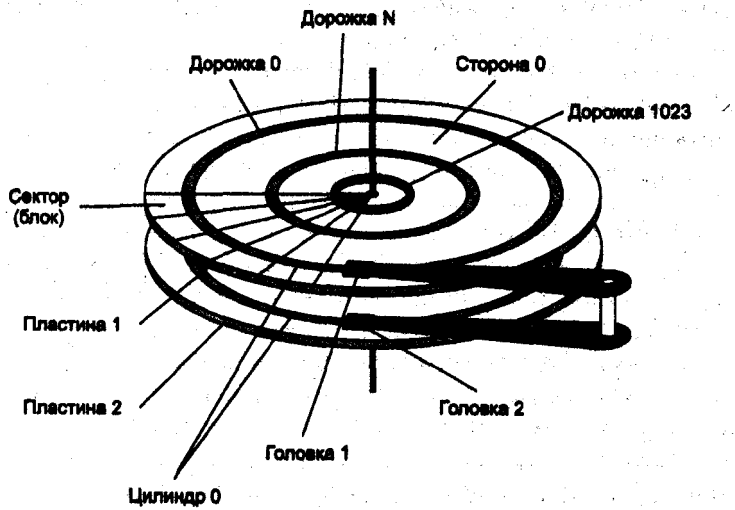


Рис. 7.10. Схема устройства жесткого диска

На каждой стороне каждой пластины размечены тонкие концентрические кольца — **дорожки** (traks), на которых хранятся данные. Количество дорожек зависит от типа диска. Нумерация дорожек начинается с 0 от внешнего края к центру диска. Когда диск вращается, элемент, называемый **головкой**, считывает двоичные данные с магнитной дорожки или записывает их на магнитную дорожку.

Головка может позиционироваться над заданной дорожкой. Головки перемещаются над поверхностью диска дискретными шагами, каждый шаг соответствует сдвигу на одну дорожку. Запись на диск осуществляется благодаря способности головки изменять магнитные свойства дорожки. В некоторых дисках вдоль каждой поверхности перемещается одна головка, а в других — имеется по головке на каждую дорожку. В первом случае для поиска информации головка должна перемещаться по радиусу диска. Обычно все головки закреплены на едином перемещающем механизме и двигаются синхронно. Поэтому, когда головка фиксируется на заданной дорожке одной поверхности, все остальные головки останавливаются над дорожками с такими же номерами. В тех же случаях,

когда на каждой дорожке имеется отдельная головка, никакого перемещения головок с одной дорожки на другую не требуется, за счет этого экономится время, затрачиваемое на поиск данных.

Совокупность дорожек одного радиуса на всех поверхностях всех пластин пакета называется **цилиндром** (cylinder).

Каждая дорожка разбивается на фрагменты, называемые **секторами** (sectors), или **блоками** (blocks), так что все дорожки имеют равное число секторов, в которые максимально можно записать одно и то же число байтов[1]. Сектор имеет фиксированный для конкретной системы размер, выражающийся степенью двойки. Чаще всего размер сектора составляет 512 байт. Учитывая, что дорожки разного радиуса имеют одинаковое число секторов, плотность записи становится тем выше, чем ближе дорожка к центру.

Для того чтобы контроллер мог найти на диске нужный сектор, необходимо задать ему все составляющие адреса сектора:

- номер цилиндра;
- номер поверхности;
- номер сектора на дорожке.

Сектор — наименьшая адресуемая единица обмена данными дискового устройства с оперативной памятью.

Это означает, что даже в тех случаях, когда программе требуется прочитать с диска только один байт, в действительности будет считан целый сектор и передан системе для выборки нужных данных.

Так как прикладная программа оперирует не секторами, а байтами, к тому же заданный объем требуемых данных не обязательно оказывается кратным размеру сектора, то типичный запрос включает чтение нескольких секторов, содержащих нужную информацию, и одного или двух секторов, содержащих наряду с требуемыми избыточные данные (рис. 7.11).

Операционная система при работе с диском использует собственную единицу дискового пространства, называемую **кластером** (cluster)[2]. При создании файла ОС запрашивает для него область на диске, размер которой кратен принятому в этой операционной системе размеру кластера. Например, если файл имеет размер 2560 байт, а размер кластера в файловой системе определен в 1024 байта, то файлу будет выделено на диске 3 кластера.

Кластер — наименьшая единица дискового пространства, которой оперирует файловая система при распределении памяти на диске.

---

[1] Внешняя дорожка может иметь несколько дополнительных секторов, используемых для замены поврежденных секторов в режиме горячего резервирования.

[2] Иногда кластер называют блоком (например, в ОС Unix), что может привести к терминологической путанице. Вообще, терминология, используемая при описании форматов дисков и файловых систем, зависит от аппаратной платформы (RISC, Wintel и т. п.) и операционной системы. Это нужно учитывать и трактовать термины в зависимости от контекста.

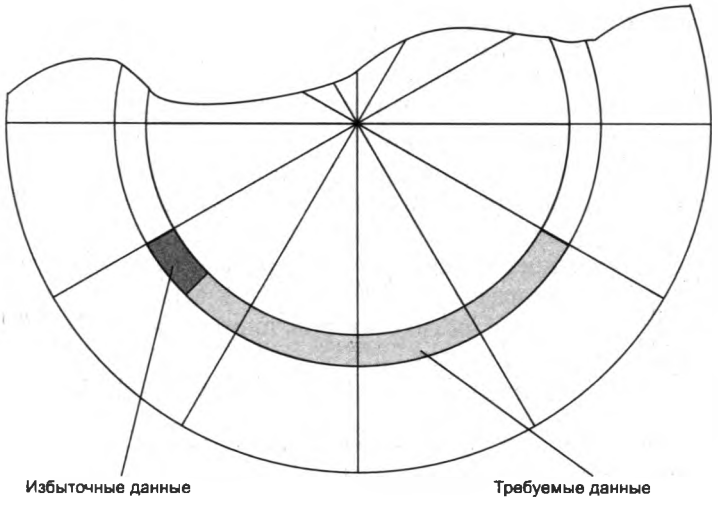


Рис. 7.11. Считывание избыточных данных при обмене с диском

Дорожки и секторы создаются в результате выполнения процедуры *физического*, или *низкоуровневого*, *форматирования диска*, предшествующей использованию диска. Для определения границ блоков на диск записывается идентификационная информация. Низкоуровневый формат диска не зависит от типа операционной системы, которая этот диск будет использовать.

Разметку диска под конкретный тип файловой системы выполняют процедуры *высокоуровневого*, или *логического*, *форматирования*. При высокоуровневом форматировании определяется размер кластера, и на диск записывается информация, необходимая для работы файловой системы, в том числе: информация о доступном и неиспользуемом пространстве, о границах областей, отведенных под файлы и каталоги, о поврежденных областях. Кроме того, на диск записывается загрузчик операционной системы — небольшая программа, которая начинает процесс инициализации операционной системы после включения питания или рестарта компьютера.

Прежде чем форматировать диск под определенную файловую систему, он может быть разбит на разделы.

Раздел — это непрерывная часть физического диска, которую операционная система представляет пользователю как логическое устройство[1]. В представлении прикладного программиста **логическое устройство** функционирует так, как если бы это был отдельный физический диск.

Именно с логическими устройствами работает пользователь, обращаясь к ним по символьным именам, например A, B, C, SYS и т. п. Операционные системы

[1] Используются также термины «логический диск», «логический раздел», «том» (volume). Для разных ОС толкование этих терминов имеет свои особенности.

разного типа используют единое для всех них представление о разделах, но создают на его основе логические устройства, специфические для каждого типа ОС. На каждом логическом устройстве может создаваться только одна файловая система.

В частном случае, когда все дисковое пространство охватывается одним разделом, логическое устройство представляет физическое устройство в целом. Если диск разбит на несколько разделов, то для каждого из этих разделов может быть создано отдельное логическое устройство. Логическое устройство может быть создано и на базе нескольких разделов, причем эти разделы не обязательно должны принадлежать одному физическому устройству. Объединение нескольких разделов в единое логическое устройство может выполняться разными способами и преследовать разные цели, основные из которых: увеличение общего объема логического раздела, повышение производительности и отказоустойчивости. Примерами организации совместной работы нескольких дисковых разделов являются так называемые RAID-массивы, подробнее о которых рассказано далее.

На разных логических устройствах одного и того же физического диска могут располагаться файловые системы как одного и того же, так и разных типов. На рис. 7.12 показан пример диска, разбитого на 3 раздела, в которых установлено две файловых системы NTFS (разделы `C` и `E`) и одна файловая система FAT (раздел `D`).

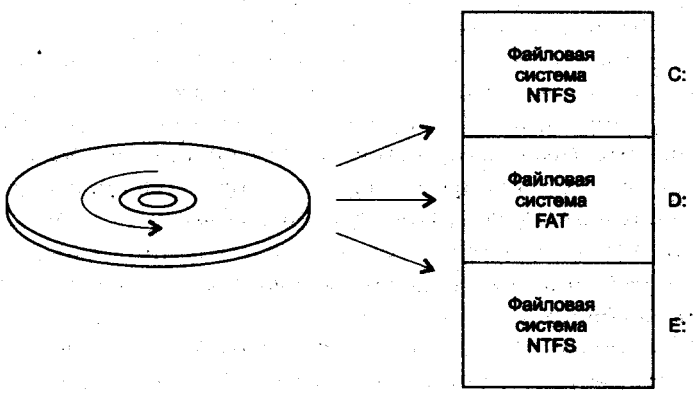


Рис. 7.12. Разбиение диска на разделы

ПРИМЕЧАНИЕ Все разделы одного диска имеют одинаковый размер блока, определенный для данного диска в результате низкоуровневого форматирования. Однако в результате высокоуровневого форматирования в разных разделах одного и того же диска, представленных разными логическими устройствами, могут быть установлены файловые системы, в которых определены кластеры отличающихся размеров.

Операционная система может поддерживать разные статусы разделов, особым образом отмечая разделы, которые могут быть использованы для загрузки

модулей операционной системы, и разделы, в которых можно устанавливать только приложения и хранить файлы данных. Один из разделов диска помечается как *загружаемый*, или *активный*. Именно из этого раздела считывается загрузчик операционной системы.

Физическая организация и адресация файла

Важным компонентом физической организации файловой системы является физическая организация файла, то есть способ размещения файла на диске.

Основными критериями эффективности физической организации файлов являются:

- скорость доступа к данным;
- объем адресной информации файла;
- степень фрагментированности дискового пространства;
- максимально возможный размер файла.

Непрерывное размещение — простейший вариант физической организации (рис. 7.13, *а*), при котором файлу предоставляется последовательность кластеров диска, образующих непрерывный участок дисковой памяти. Основным достоинством этого метода является высокая скорость доступа, так как затраты на поиск и считывание кластеров файла минимальны. Также минимален объем адресной информации — достаточно хранить только номер первого кластера и объем файла. Данная физическая организация максимально возможный размер файла не ограничивает. Однако этот вариант имеет существенные недостатки, которые затрудняют его применимость на практике, несмотря на всю его логическую простоту. При более пристальном рассмотрении оказывается, что реализовать эту схему не так уж просто. Действительно, какого размера должна быть непрерывная область, выделяемая файлу, если файл при каждой модификации может увеличить свой размер? Еще более серьезной проблемой является фрагментация. Спустя некоторое время после создания файловой системы, в результате выполнения многочисленных операций создания и удаления файлов пространство диска неминуемо превращается в «лоскутное одеяло», включающее большое число свободных областей небольшого размера. Как всегда бывает при фрагментации, суммарный объем свободной памяти может быть очень большим, а выбрать место для размещения файла целиком — невозможно. Поэтому на практике используются методы, в которых файл размещается в нескольких в общем случае несмежных областях диска.

Следующий способ физической организации файла — **размещение в виде связанного списка кластеров** дисковой памяти (рис. 7.13, *б*). При таком способе в начале каждого кластера содержится указатель на следующий кластер. В этом случае адресная информация минимальна: расположение файла может быть задано одним числом — номером первого кластера. В отличие от предыдущего способа, каждый кластер может быть присоединен к цепочке кластеров какого-либо файла, следовательно, фрагментация на уровне кластеров отсутствует. Файл может изменять свой размер во время своего существования,

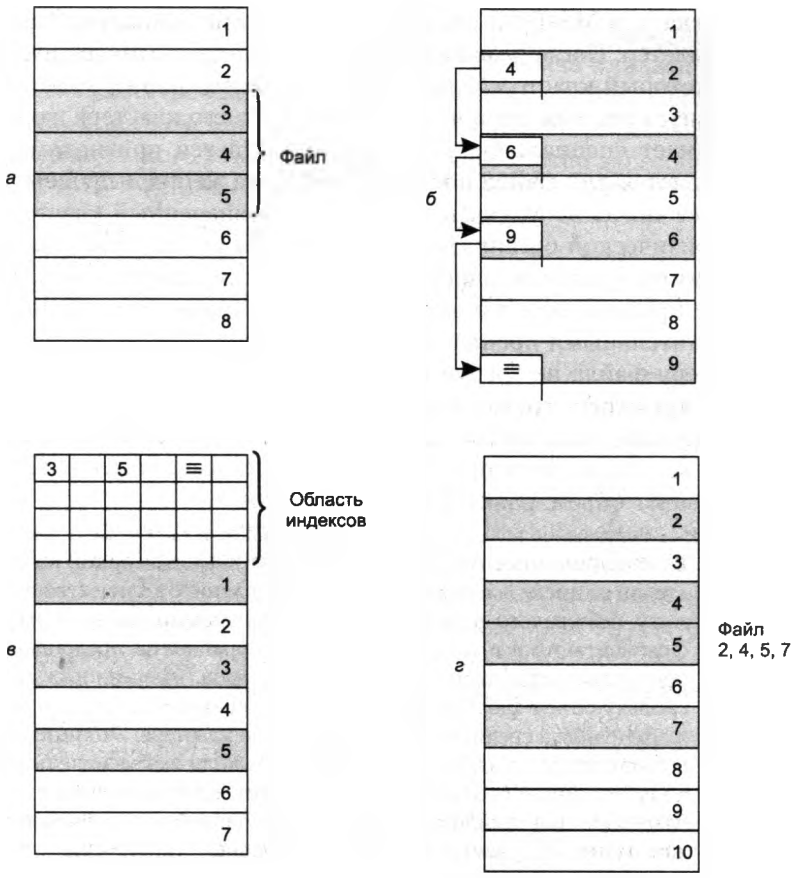


Рис. 7.13. Физическая организация файла: непрерывное размещение (*а*); связанный список кластеров (*б*); связанный список индексов (*в*); перечень номеров кластеров (*г*)

наращивая число кластеров. Недостатком является сложность реализации доступа к произвольно заданному месту файла — чтобы прочитать пятый по порядку кластер файла, необходимо последовательно прочитать четыре первых кластера, прослеживая цепочку номеров кластеров. Кроме того, при этом способе количество данных файла, содержащихся в одном кластере, не равно степени двойки (одно слово расходуется на номер следующего кластера), а многие программы читают данные кластерами, размер которых равен степени двойки.

Популярным способом, применяемым, например, в файловой системе FAT, является **размещение в виде связанного списка индексов** (рис. 7.13, *в*). Этот способ является некоторой модификацией предыдущего. Файлу также выделяется память в виде связанного списка кластеров. Номер первого кластера запоминается в записи каталога, где хранятся характеристики этого файла. Остальная адресная информация отделена от кластеров файла. С каждым кластером диска связывается некоторый элемент — индекс. Индексы располагаются в отдель-

ной области диска — в MS-DOS это *таблица FAT* (File Allocation Table), занимающая один кластер. Когда память свободна, все индексы имеют нулевое значение. Если некоторый кластер *N* назначен некоторому файлу, то индекс этого кластера становится равным либо номеру *M* следующего кластера данного файла, либо принимает специальное значение, являющееся признаком того, что этот кластер является для файла последним. Индекс же предыдущего кластера файла принимает значение *N*, указывая на вновь назначенный кластер.

При такой физической организации сохраняются все достоинства предыдущего способа: минимальность адресной информации, отсутствие фрагментации, отсутствие проблем при изменении размера. Кроме того, данный способ обладает дополнительными преимуществами. Во-первых, для доступа к произвольному кластеру файла не требуется последовательно считывать его кластеры, достаточно прочитать только секторы диска, содержащие таблицу индексов, отсчитать нужное количество кластеров файла по цепочке и определить номер нужного кластера. Во-вторых, данные файла заполняют кластер целиком, а значит, имеют объем, равный степени двойки.

ПРИМЕЧАНИЕ Необходимо отметить, что при отсутствии фрагментации на уровне кластеров на диске все равно имеется определенное количество областей памяти небольшого размера, которые невозможно задействовать, то есть фрагментация все же существует. Эти фрагменты представляют собой неиспользуемые части последних кластеров, назначенных файлам, поскольку объем файла в общем случае не кратен размеру кластера. На каждом файле в среднем теряется половина кластера. Это потери особенно велики, когда на диске имеется большое количество маленьких файлов, а кластер имеет большой размер. Размеры кластеров зависят от размера раздела и типа файловой системы. Примерный диапазон, в котором может меняться размер кластера, составляет от 512 байт до десятков килобайт.

Еще один способ задания физического расположения файла заключается в простом перечислении номеров кластеров, занимаемых этим файлом (рис. 7.13, *г*). Этот **набор номеров** и служит адресом файла. Недостаток данного способа очевиден: длина адреса зависит от размера файла и для большого файла может составить значительную величину. Достоинством же является высокая скорость доступа к произвольному кластеру файла, так как здесь применяется прямая адресация, которая исключает просмотр цепочки указателей при поиске адреса произвольного кластера файла. Фрагментация на уровне кластеров в этом способе также отсутствует.

Последний подход с некоторыми модификациями используется в традиционных файловых системах s5 и ufs ОС Unix[1]. В файловой системе ufs для сокращения объема адресной информации используется **комбинированная схема адресации** кластеров, позволяющая дополнять прямой способ адресации кос-

[1] Современные версии Unix поддерживают и другие типы файловых систем, в том числе и пришедшие из других ОС, как, например, FAT.

венным. Для хранения адреса файла выделено 15 полей, каждое из которых состоит из 4 байт (рис. 7.14). Если размер файла меньше или равен 12 кластерам, то номера этих кластеров непосредственно перечисляются в первых двенадцати полях адреса. Если кластер имеет размер 8 Кбайт (максимальный размер кластера, поддерживаемого в ufs), то таким образом можно адресовать файл размером до 8192 × 12 = 98 304 байт.

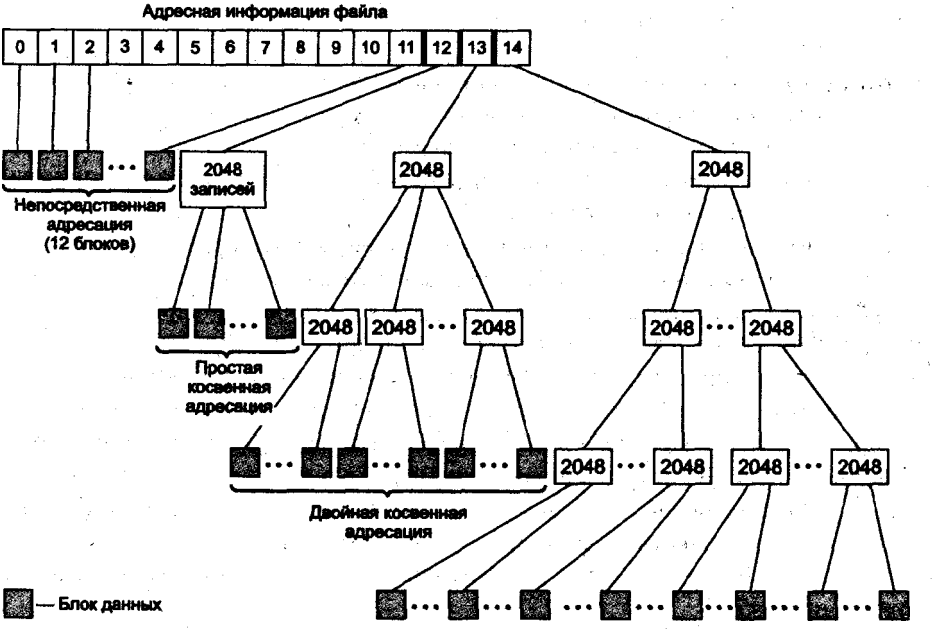


Рис. 7.14. Схема адресации файловой системы ufs

Если размер файла превышает 12 кластеров, то следующее 13-е поле содержит не номер следующего кластера, а номер кластера, в котором могут быть расположены номера следующих кластеров файла. Таким образом, 13-й элемент адреса используется для *косвенной адресации*. При размере в 8 Кбайт кластер, на который указывает 13-й элемент, может содержать 2048 номеров следующих кластеров данных файла, и размер файла может возрасти до 8192 × × (12 + 2048) = 16 875 520 байт.

Если размер файла превышает 12 + 2048 = 2060 кластеров, то используется 14-е поле. В нем находится номер кластера, содержащего 2048 номеров кластеров, каждый из которых хранит 2048 номеров кластеров данных файла. Здесь применяется уже *двойная косвенная адресация*. С ее помощью можно адресовать кластеры в файлах, содержащих до $8192 \times (12 + 2048 + 2048^2) = 3{,}43766 \times 10^{10}$ байт.

И наконец, если файл включает более $12 + 2048 + 2048^2 = 4\,196\,364$ кластеров, то используется последнее 15-е поле для *тройной косвенной адресации*, что позволяет задать адрес файла, имеющего следующий максимальный размер:

$$8192 \times (12 + 2048 + 2048^2 + 2048^3) = 7{,}0403 \times 10^{13} \text{ байт.}$$

Таким образом, файловая система ufs при размере кластера в 8 Кбайт поддерживает файлы, состоящие максимум из 70 триллионов байтов данных, хранящихся в 8 миллиардах кластеров. Как видно на рис. 7.14, для задания адресной информации о максимально большом файле требуется 15 элементов по 4 байта (60 байт) в центральной части адреса плюс 1 + (1 + 2048) + (1 + 2048 + + 2048^2) = 4 198 403 кластера в косвенной части адреса. Несмотря на огромную величину, это число составляет всего около 0,05 % от объема адресуемых данных.

Файловая система ufs поддерживает дисковые кластеры и меньших размеров, при этом максимальный размер файла будет другим. Используемая в более ранних версиях Unix, файловая система s5 имеет аналогичную схему адресации, но она рассчитана на файлы меньших размеров, поэтому в ней применяется 13 адресных элементов вместо 15.

Метод перечисления адресов кластеров файла характерен и для файловой системы NTFS, используемой в ОС семейства Windows NT. Здесь он дополнен достаточно естественным приемом, сокращающим объем адресной информации: адресуются не кластеры файла, а непрерывные области, состоящие из смежных кластеров диска. Каждая такая область, называемая **отрезком** (run), или **экстентом** (extent), описывается с помощью двух чисел: начального номера кластера и количества кластеров в отрезке. Так как для сокращения времени операции обмена ОС старается разместить файл в последовательных кластерах диска, то в большинстве случаев количество последовательных областей файла будет меньше количества кластеров файла, и объем служебной адресной информации в NTFS сокращается по сравнению со схемой адресации в файловых системах ufs и s5.

Для того чтобы корректно принимать решение о выделении файлу набора кластеров, файловая система должна отслеживать информацию о состоянии всех кластеров диска: свободен/занят. Эта информация может храниться как отдельно от адресной информации файлов, так и вместе с ней.

Физическая организация FAT

Ниже перечислены области, из которых состоит логический раздел, отформатированный под файловую систему FAT (рис. 7.15).

- *Загрузочный сектор* содержит программу начальной загрузки операционной системы. Вид этой программы зависит от типа операционной системы, которая будет загружаться из этого раздела.
- *Основная копия таблицы FAT* содержит информацию о размещении файлов и каталогов на диске.
- *Резервная копия таблицы FAT.*
- *Корневой каталог* занимает фиксированную область размером в 32 сектора (16 Кбайт), что позволяет хранить 512 записей о файлах и каталогах, так как каждая запись каталога состоит из 32 байт.

- *Область данных* предназначена для размещения всех файлов и всех каталогов, кроме корневого каталога.

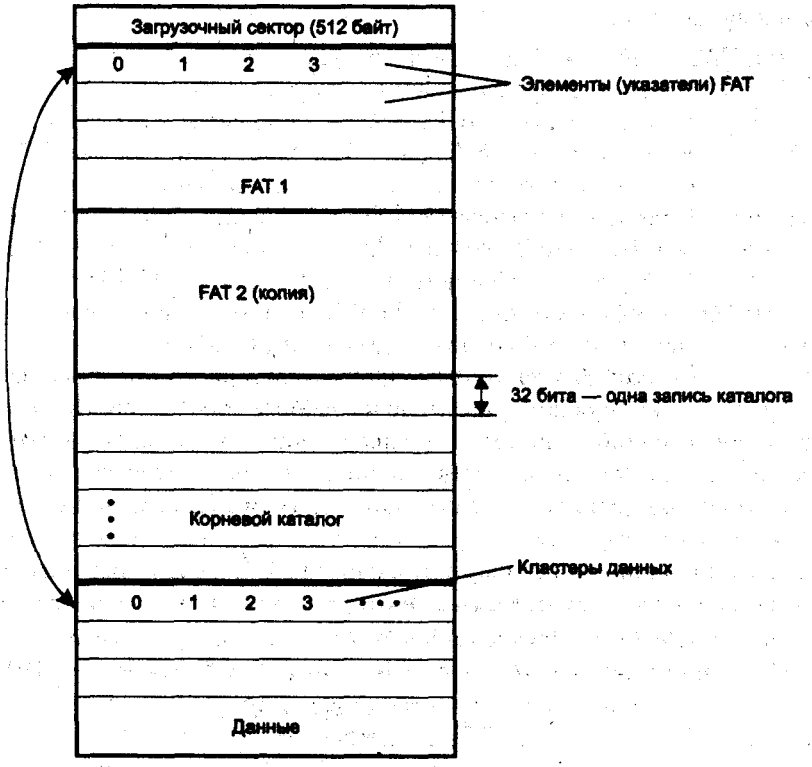


Рис. 7.15. Физическая структура файловой системы FAT

Файловая система FAT поддерживает всего два типа файлов: *обычный файл* и *каталог*. Файловая система распределяет память только из области данных, причем использует в качестве минимальной единицы дискового пространства кластер.

Таблица FAT (как основная копия, так и резервная) состоит из массива индексных указателей, количество которых равно количеству кластеров области данных. Между кластерами и индексными указателями имеется взаимно однозначное соответствие — нулевой указатель соответствует нулевому кластеру и т. д.

Индексный указатель может принимать следующие значения, характеризующие состояние связанного с ним кластера:

- кластер свободен (не используется);
- кластер используется файлом и не является последним кластером файла — в этом случае индексный указатель содержит номер следующего кластера файла;

- последний кластер файла;
- дефектный кластер;
- резервный кластер.

Таблица FAT является общей для всех файлов раздела. В исходном состоянии (после форматирования) все кластеры раздела свободны, и все индексные указатели (кроме тех, которые соответствуют резервным и дефектным блокам) принимают значение «кластер свободен». При размещении файла ОС просматривает FAT, начиная с начала, и ищет первый свободный индексный указатель. После его обнаружения номер этого указателя фиксируется в поле номера первого кластера записи каталога (см. рис. 7.8, *а*). В кластер с этим номером записываются данные файла, он становится первым кластером файла. Если файл умещается в одном кластере, то в указатель, соответствующий данному кластеру, заносится специальное значение, идентифицирующее последний кластер файла. Если же размер файла больше одного кластера, то ОС продолжает просмотр FAT и ищет следующий указатель на свободный кластер. После его обнаружения в предыдущий указатель заносится номер этого кластера, который теперь становится следующим кластером файла. Процесс повторяется до тех пор, пока не будут размещены все данные файла. Таким образом создается *связный список всех кластеров файла*.

В начальный период после форматирования файлы будут размещаться в последовательных кластерах области данных, однако после определенного количества удалений файлов кластеры одного файла окажутся в произвольных местах области данных, чередуясь с кластерами других файлов (рис. 7.16).

Имя	№ индексного дескриптора
Prog 1	23
firelights	126
doc_23.txt	51
glazing.txt	17
lambda_good	

Рис. 7.16. Списки указателей файлов в FAT

Размер таблицы FAT и разрядность используемых в ней индексных указателей определяется количеством кластеров в области данных. Для уменьшения потерь из-за фрагментации желательно кластеры делать небольшими, а для сокращения объема адресной информации и повышения скорости обмена наоборот — чем больше, тем лучше. При форматировании диска под файловую систему FAT обычно выбирается компромиссное решение, и размеры кластеров выбираются из диапазона от 1 до 128 секторов, или от 512 байт до 64 Кбайт.

Очевидно, что разрядность индексного указателя должна быть такой, чтобы в нем можно было задать максимальный номер кластера для диска определенного объема. Существует несколько разновидностей FAT, отличающихся разрядностью индексных указателей, которая и используется в качестве условного обозначения: FAT12, FAT16 и FAT32. В файловой системе FAT12 задействованы 12-разрядные указатели, что позволяет поддерживать до 4096 кластеров в области данных диска[1], в FAT16 — 16-разрядные указатели для 65 536 кластеров и в FAT32 — 32-разрядные для более чем 4 миллиардов кластеров.

Файловая система FAT12 обычно характерна только для небольших дисков объемом не более 16 Мбайт, что позволяет не использовать кластеры более 4 Кбайт. По этой же причине считается, что FAT16 целесообразнее для дисков с объемом не более 512 Мбайт, а для больших дисков лучше подходит система FAT32, которая способна использовать кластеры 4 Кбайт при работе с дисками объемом до 8 Гбайт, и только для дисков большего объема задействует 8, 16 и 32 Кбайт. Максимальный размер раздела FAT16 ограничен значением 4 Гбайт, такой объем дает 65 536 кластеров по 64 Кбайт каждый, а максимальный размер раздела FAT32 практически не ограничен — 2^{32} кластеров по 32 Кбайт.

Таблица FAT при фиксированной разрядности индексных указателей имеет переменный размер, зависящий от объема области данных диска.

При удалении файла из файловой системы FAT в первый байт соответствующей записи каталога заносится специальный признак, свидетельствующий о том, что эта запись свободна, а во все индексные указатели файла заносится признак «кластер свободен». Остальные данные в записи каталога, в том числе номер первого кластера файла, остаются нетронутыми, что оставляет шансы для восстановления ошибочно удаленного файла. Существует большое количество утилит для восстановления удаленных файлов FAT, выводящих пользователю список имен удаленных файлов с отсутствующим первым символом имени, затертым после освобождения записи. Очевидно, что надежно можно восстановить только файлы, которые были расположены в последовательных кластерах диска, так как при отсутствии связного списка выявить принадлежность произвольно расположенного кластера удаленному файлу невозможно (без анализа содержимого кластеров, выполняемого пользователем «вручную»).

Резервная копия FAT всегда синхронизируется с основной копией при любых операциях с файлами, поэтому резервную копию нельзя задействовать для отмены ошибочных действий пользователя, выглядевших с точки зрения системы вполне корректными. Резервная копия может быть полезна только в том случае, когда секторы основной памяти оказываются физически поврежденными и не читаются.

Используемый в FAT метод хранения адресной информации о файлах не отличается большой надежностью — при разрыве списка индексных указателей

[1] Реально это число немного меньше, так как несколько значений индексного указателя расходуется для идентификации специальных кластеров, таких как последний, неиспользуемый, дефектный и резервный.

в одном месте, например, из-за сбоя в работе программного кода ОС по причине внешних электромагнитных помех, теряется информация обо всех последующих кластерах файла.

Файловые системы FAT12 и FAT16 оперировали с именами файлов, состоящими из 12 символов по схеме «8.3». В версии FAT16 операционной системы Windows NT был введен новый тип записи каталога — «длинное имя», что позволяет использовать имена длиной до 255 символов, причем каждый символ длинного имени хранится в двухбайтном формате Unicode. Имя по схеме «8.3», названное теперь коротким (не нужно путать его с простым именем файла, также называемого иногда коротким), по-прежнему хранится в 12-байтовом поле имени файла в записи каталога, а длинное имя помещается порциями по 13 символов в одну или несколько записей, следующих непосредственно за основной записью каталога. Каждый символ в формате Unicode кодируется двумя байтами, поэтому 13 символов занимают 26 байт, а оставшиеся 6 отведены под служебную информацию. Таким образом, у файла появляется два имени — короткое, для совместимости со старыми приложениями, не понимающими длинных имен в Unicode, и длинное, удобное в использовании имя. Файловая система FAT32 также поддерживает короткие и длинные имена.

Файловые системы FAT12 и FAT16 получили большое распространение благодаря их применению в операционных системах MS-DOS и Windows 3.x — самых массовых операционных системах первого десятилетия эры персональных компьютеров. Однако из-за постоянно растущих объемов жестких дисков, а также возрастающих требований к надежности эти файловые системы быстро вытесняются как системой FAT32, так и файловыми системами других типов.

Физическая организация s5 и ufs

Файловые системы s5 (получившие название от System V, родового имени нескольких версий ОС Unix, разработанных в Bell Labs компании AT&T) и ufs (Unix File System) используют очень близкую физическую модель. Это не удивительно, так как система ufs является развитием системы s5. Файловая система ufs расширяет возможности s5 по поддержке больших дисков и файлов, а также повышает ее надежность.

ПРИМЕЧАНИЕ В этом разделе вместо термина «кластер» используется термин «блок», как это принято в файловых системах Unix.

Расположение файловой системы s5 на диске иллюстрирует рис. 7.17. Раздел диска, где размещается файловая система, делится на четыре области:

- *загрузочный блок*;
- *суперблок* (superblock), содержащий самую общую информацию о файловой системе, включая размер файловой системы, размер области индексных дескрипторов, число индексных дескрипторов, списки свободных блоков и свободных индексных дескрипторов, а также другую административную информацию;

- *область индексных дескрипторов* (inode list), порядок расположения индексных дескрипторов в которой соответствует их номерам;
- *область данных*, в которой расположены как обычные файлы, так и файлы-каталоги, в том числе корневой каталог; специальные файлы представлены в файловой системе только записями в соответствующих каталогах и индексными дескрипторами специального формата, но места в области данных не занимают.

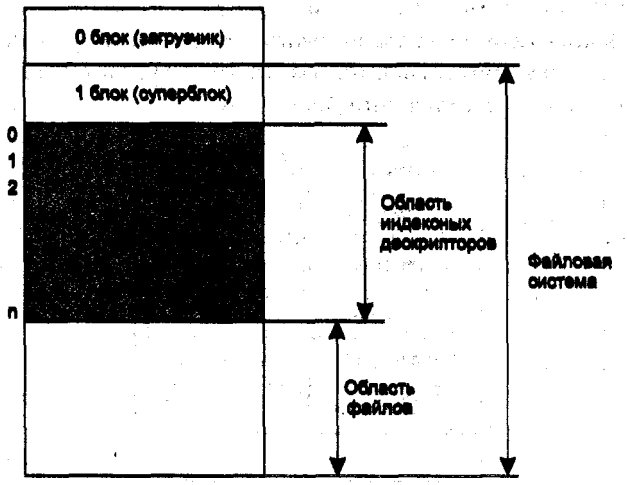


Рис. 7.17. Расположение файловой системы s5 на диске

Основной особенностью физической организации файловой системы s5 является отделение имени файла от его характеристик, хранящихся в отдельной структуре, называемой индексным дескриптором.

Индексный дескриптор (inode) в s5 имеет размер 64 байта и содержит данные о типе файла, адресную информацию, привилегии доступа к файлу и некоторую другую информацию:

- идентификатор владельца файла;
- тип файла (файл может быть файлом обычного типа, каталогом, специальным файлом, также конвейером или символьной связью);
- права доступа к файлу;
- временные характеристики (время последней модификации файла, время последнего обращения к файлу, время последней модификации индексного дескриптора);
- число ссылок на данный индексный дескриптор, равное количеству псевдонимов файла;
- адресная информация (структура адреса рассмотрена ранее в разделе «Физическая организация и адресация файла»);
- размер файла в байтах.

Каждый индексный дескриптор имеет номер, который одновременно является уникальным именем файла. Индексные дескрипторы расположены в особой области диска в строгом соответствии со своими номерами. Соответствие между полными символьными именами файлов и их уникальными именами устанавливается с помощью иерархии каталогов. Система ведет список номеров свободных индексных дескрипторов. При создании файла ему выделяется номер из этого списка, а при уничтожении файла номер его индексного дескриптора возвращается в список.

Запись о файле в каталоге состоит всего из двух полей: символьного имени файла и номера индексного дескриптора. Например, на рис. 7.18 показана информация, содержащаяся в каталоге /user.

Имя	№ индексного дескриптора
Prog 1	23
firelights	126
doc_23.txt	51
glazing.txt	17
lambda_good	

Рис. 7.18. Структура каталога в файловой системе s5

Файловая система не накладывает особых ограничений на размер корневого каталога, так как он расположен в области данных и может увеличиваться как обычный файл.

Доступ к файлу осуществляется путем последовательного просмотра всей цепочки каталогов, входящих в полное имя файла, и соответствующих им индексных дескрипторов. Поиск завершается после получения всех характеристик из индексного дескриптора заданного файла.

Рассмотрим эту процедуру на примере файла /bin/my_shell/print, входящего в состав файловой системы, изображенной на рис. 7.19. Определение физического адреса этого файла включает следующие этапы.

1. Прежде всего, просматривается корневой каталог с целью поиска первой составляющей символьного имени — bin. Определяется номер (в данном примере — 6) индексного дескриптора каталога, входящего в корневой каталог. Адрес корневого каталога известен системе.
2. Из области индексных дескрипторов считывается дескриптор с номером 6. Начальный адрес дескриптора определяется на основании известных системе номера начального сектора области индексных дескрипторов и размера

индексного дескриптора. Из индексного дескриптора 6 определяется физический адрес каталога /bin.

3. Просматривается каталог /bin с целью поиска второй составляющей символьного имени my_shell. Определяется номер индексного дескриптора каталога /bin/my_shell (в данном случае — 25).
4. Считывается индексный дескриптор 25, определяется физический адрес /bin/my_shell.
5. Просматривается каталог /bin/my_shell, определяется номер индексного дескриптора файла print (в данном случае — 131).
6. Из индексного дескриптора 131 определяются номера блоков данных, а также другие характеристики файла /bin/my_shell/print.

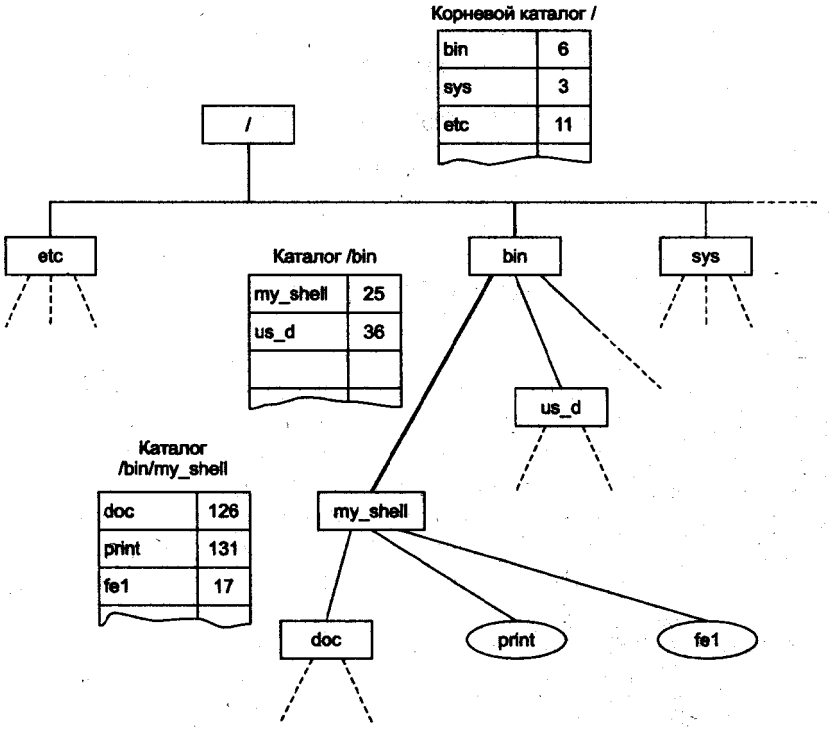


Рис. 7.19. Поиск адреса файла по его символьному имени

Эта процедура требует в общем случае нескольких обращений к диску, пропорционально числу составляющих в полном имени файла. Для уменьшения среднего времени доступа к файлу его дескриптор копируется в специальную системную область оперативной памяти. Копирование индексного дескриптора входит в процедуру открытия файла.

Физическая организация файловой системы ufs отличается от описанной физической организации файловой системы s5 тем, что раздел состоит

из повторяющейся несколько раз последовательности областей «загрузочный блок — суперблок — блок группы цилиндров — область индексных дескрипторов» (рис. 7.20).

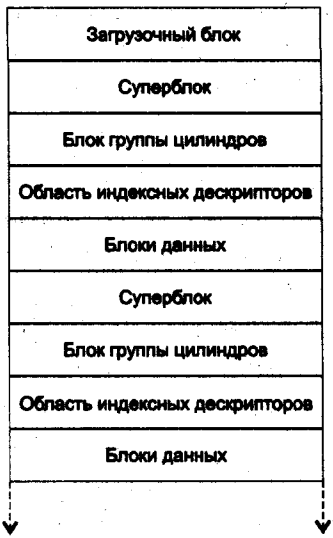


Рис. 7.20. Физическая организация файловой системы ufs

В этих повторяющихся последовательностях областей суперблок является резервной копией основной первой копии суперблока. При повреждении основной копии суперблока может быть использована резервная копия суперблока. Области же блока группы цилиндров и индексных дескрипторов содержат индивидуальные для каждой последовательности значения. Блок группы цилиндров описывает количество индексных дескрипторов и блоков данных, расположенных на данной группе цилиндров диска. Такая группировка делается для ускорения доступа, чтобы просмотр индексных дескрипторов и данных файлов, описываемых этими дескрипторами, не приводил к слишком большим перемещениям головок диска.

Кроме того, в ufs имена файлов могут иметь длину до 255 символов (кодировка ASCII, по одному байту на символ), в то время как в s5 длина имени не может превышать 14 символов.

Физическая организация NTFS

Файловая система **NTFS** (*NT File System*) была разработана в качестве основной файловой системы для ОС Windows NT в начале 90-х годов с учетом опыта разработки файловых систем FAT и HPFS (основная файловая система для OS/2), а также других существовавших в то время файловых систем. Сейчас NTFS поддерживается всеми ОС семейства Windows NT, которое включает операционные системы Windows NT 3.1, Windows NT 4.0, Windows 2000, Win-

dows XP, Windows Server 2003 и Windows Vista, а также многими другими операционными системами.

Основными отличительными свойствами NTFS являются:

- поддержка больших файлов и больших дисков объемом до 2^{64} байт;
- восстанавливаемость после сбоев и отказов программ и аппаратуры управления дисками;
- высокая скорость операций, в том числе для больших дисков;
- низкий уровень фрагментации, в том числе для больших дисков;
- гибкая структура, допускающая развитие за счет добавления новых типов записей и атрибутов файлов с сохранением совместимости с предыдущими версиями ФС;
- устойчивость к отказам дисковых накопителей;
- поддержка длинных символьных имен;
- контроль доступа к каталогам и отдельным файлам.

Структура тома NTFS

В отличие от разделов FAT и s5/ufs, все пространство тома[1] NTFS представляет собой либо файл, либо часть файла.

Основой структуры тома NTFS является **главная таблица файлов** (Master File Table, **MFT**), которая содержит, по крайней мере, одну запись для каждого файла тома, включая одну запись для самой себя. Каждая запись MFT имеет фиксированную длину, зависящую от объема диска — 1, 2 или 4 Кбайт. Для большинства дисков размер записи MFT равен 2 Кбайт, его мы далее будет считать размером записи, устанавливаемым по умолчанию.

Файлы на томе NTFS идентифицируются **номером файла**, который определяется позицией файла в MFT. Этот способ идентификации файла близок к способу, используемому в файловых системах s5 и ufs, где файл однозначно идентифицируется номером его записи в области индексных дескрипторов.

Весь том NTFS состоит из последовательности кластеров, что отличает эту файловую систему от рассмотренных ранее, где на кластеры делилась только область данных. Порядковый номер кластера в томе NTFS называется **логическим номером кластера** (Logical Cluster Number, **LCN**). Файл NTFS также состоит из последовательности кластеров, при этом порядковый номер кластера внутри файла называется **виртуальным номером кластера** (Virtual Cluster Number, **VCN**).

Базовая единица распределения дискового пространства для файловой системы NTFS — непрерывная область кластеров, называемая **отрезком**. В качестве адреса отрезка NTFS использует логический номер его первого кластера, а также количество кластеров в отрезке k, то есть пара (LCN, k). Таким образом,

[1] В Windows NT логический раздел принято называть томом.

часть файла, помещенная в отрезок и начинающаяся с виртуального кластера VCN, характеризуется адресом, состоящим из трех чисел: (VCN, LCN, k).

Для хранения номера кластера в NTFS используются 64-разрядные указатели, что дает возможность поддерживать тома и файлы размером до 2^{64} кластеров. При размере кластера в 4 Кбайт это позволяет использовать тома и файлы, состоящие из 64 миллиардов килобайт.

Структура тома NTFS показана на рис. 7.21. Загрузочный блок тома NTFS располагается в начале тома, а его копия — в середине тома. Загрузочный блок содержит стандартный блок параметров BIOS, количество блоков в томе, а также начальный логический номер кластера основной копии MFT и зеркальную копию MFT.

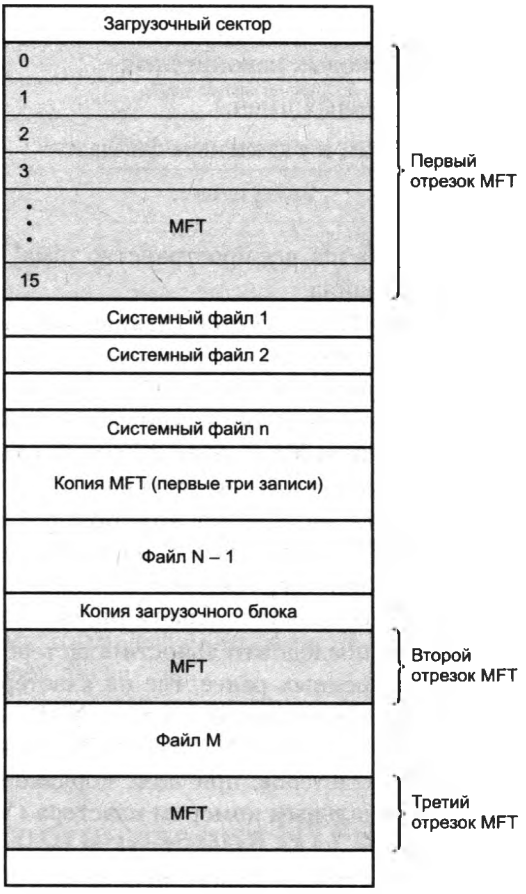


Рис. 7.21. Структура тома NTFS

Далее располагается первый отрезок MFT, содержащий 16 стандартных создаваемых при форматировании записей о системных файлах NTFS. Назначение этих записей иллюстрирует табл. 7.1

Таблица 7.1. Назначение 16 стандартных записей о системных файлах NTFS

Номер записи	Системный файл	Имя файла	Назначение файла
0	Главная таблица файлов	$Mft	Полный список файлов тома NTFS
1	Копия главной таблицы файлов	$MftMirr	Зеркальная копия первых трех записей MFT
2	Файл журнала	$LogFile	Список транзакций, который используется для восстановления файловой системы после сбоев
3	Том	$Volume	Имя тома, версия NTFS и другая информация о томе
4	Таблица определения атрибутов	$AttrDef	Таблица имен, номеров и описаний атрибутов
5	Индекс корневого каталога	$	Корневой каталог
6	Битовая карта кластеров	$Bitmap	Разметка использованных кластеров тома
7	Загрузочный сектор раздела	$Boot	Адрес загрузочного сектора раздела
8	Файл плохих кластеров	$BadClus	Файл со списком всех обнаруженных на томе плохих кластеров
9	Таблица квот	$Quota	Квоты пространства на диске для каждого пользователя
10	Таблица преобразования регистра символов	$Upcase	Используется для преобразования регистра символов в кодировке Unicode
11–15	Зарезервированы для будущего использования		

В NTFS файл целиком размещается в одной записи таблицы MFT, если это позволяет сделать его размер. В том же случае, когда размер файла больше размера записи MFT, то в запись помещаются только некоторые атрибуты файла, а остальная часть файла размещается в отдельном отрезке тома (или нескольких отрезках). Часть файла, размещаемая в записи MFT, называется **резидентной**, остальные части — **нерезидентными**. Адресная информация об отрезках, содержащих нерезидентные части файла, находится в атрибутах резидентной части.

Некоторые системные файлы являются полностью резидентными, а некоторые имеют и нерезидентные части, которые располагаются после первого отрезка MFT.

Нулевая запись MFT содержит описание самой таблицы MFT, в том числе и такой ее важный атрибут, как адреса всех ее отрезков. После форматирования MFT состоит из одного отрезка, но после создания первого же несистемного файла для хранения его атрибутов требуется еще один отрезок, так как непре-

рывная последовательность кластеров MFT изначально завершается системными файлами.

Из приведенного описания видно, что сама таблица MFT рассматривается как файл, к которому применим метод размещения в томе в виде набора произвольно расположенных нескольких отрезков.

Структура файлов NTFS

Каждый файл и каталог на томе NTFS состоит из набора **атрибутов**.

Важно отметить, что имя файла и его данные также рассматриваются как атрибуты файла, то есть в трактовке NTFS, помимо атрибутов, у файла нет никаких других компонентов.

Каждый атрибут файла NTFS состоит из полей: типа, длины, значения и, возможно, имени атрибута. Тип атрибута, длина и имя образуют **заголовок атрибута**.

Существует системный набор атрибутов, определяемых структурой тома NTFS. Системные атрибуты имеют фиксированные имена и коды их типа, а также определенный формат. Могут применяться также атрибуты, определяемые пользователями. Их имена, типы и форматы задаются исключительно пользователем. Атрибуты файлов упорядочены по убыванию кода атрибута, причем атрибут одного и того же типа может повторяться несколько раз. Существует два способа хранения атрибутов файла — резидентное хранение в записях таблицы MFT и нерезидентное хранение вне ее, во внешних отрезках. Таким образом, резидентная часть файла состоит из резидентных атрибутов, а нерезидентная — из нерезидентных атрибутов. Сортировка может осуществляться только по резидентным атрибутам.

Системный набор включает следующие атрибуты:

- список атрибутов (*Attribute List*), из которых состоит файл, содержит ссылки на номер записи MFT, где расположен каждый атрибут (этот редко используемый атрибут нужен только в том случае, если атрибуты файла не умещаются в основной записи и занимают дополнительные записи MFT);
- имя файла (*File Name*) — длинное имя файла в формате Unicode, а также номер входа в таблице MFT для родительского каталога; если этот файл содержится в нескольких каталогах, то у него будет несколько атрибутов типа *File Name*; этот атрибут всегда должен быть резидентным;
- имя MS-DOS (*MS-DOS Name*) — содержит имя файла в формате 8.3;
- версия (*Version*) — номер последней версии файла;
- дескриптор безопасности (*Security Descriptor*) — информация о защите файла, включая список прав доступа ACL (права доступа к файлу рассматриваются далее в разделе «Контроль доступа к файлам») и поле аудита, которое определяет, какого рода операции над этим файлом нужно регистрировать;
- версия тома (*Volume Version*) используется только в системных файлах тома;
- имя тома (*Volume Name*);

- данные (*Data*) — обычные данные файла;
- битовая карта MFT (*MFT bitmap*) — карта использования блоков на томе;
- корень индекса (*Index Root*) — корень B-дерева, используемого для поиска файлов в каталоге;
- размещение индекса (*Index Allocation*) — нерезидентные части индексного списка B-дерева;
- стандартная информация (*Standard Information*) — этот атрибут хранит всю остальную стандартную информацию о файле, которую трудно связать с каким-либо из других атрибутов файла, например время создания файла, время обновления и другие.

Файлы NTFS в зависимости от способа размещения делятся на небольшие, большие, очень большие и сверхбольшие.

Небольшие (small) **файлы.** Если файл имеет небольшой размер, то он может целиком располагаться внутри одной записи MFT, имеющей, например, размер 2 Кбайт. Небольшие файлы NTFS состоят, по крайней мере, из следующих атрибутов (рис. 7.22):

- стандартная информация (Standard Information, SI);
- имя файла (File Name, FN);
- данные (Data);
- дескриптор безопасности (Security Descriptor, SD).

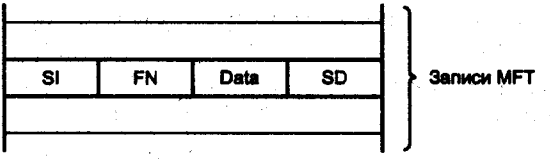


Рис. 7.22. Небольшой файл NTFS

Из-за того, что файл может иметь переменное количество атрибутов, а также из-за переменного размера атрибутов нельзя наверняка утверждать, что файл уместится внутри записи. Однако обычно файлы размером менее 1500 байт помещаются внутри записи MFT (размером 2 Кбайт).

Большие (large) **файлы.** Если данные файлы не помещаются в одну запись MFT, то этот факт отражается в заголовке атрибута *Data*, который содержит признак того, что этот атрибут является нерезидентным, то есть находится в отрезках вне таблицы MFT. В этом случае атрибут *Data* содержит адресную информацию (LCN, VCN, *k*) каждого отрезка данных (рис. 7.23).

Очень большие (huge) **файлы.** Если файл настолько велик, что его атрибут данных, хранящий адреса нерезидентных отрезков данных, не помещается в одной записи, то этот атрибут помещается в другую запись MFT, а ссылка на такой атрибут помещается в основную запись файла (рис. 7.24). Эта ссылка содержится в атрибуте *Attribute List*. Сам атрибут данных по-прежнему содержит адреса нерезидентных отрезков данных.

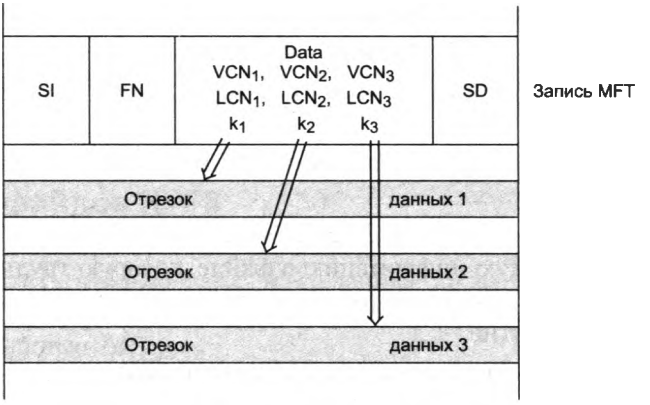


Рис. 7.23. Большой файл

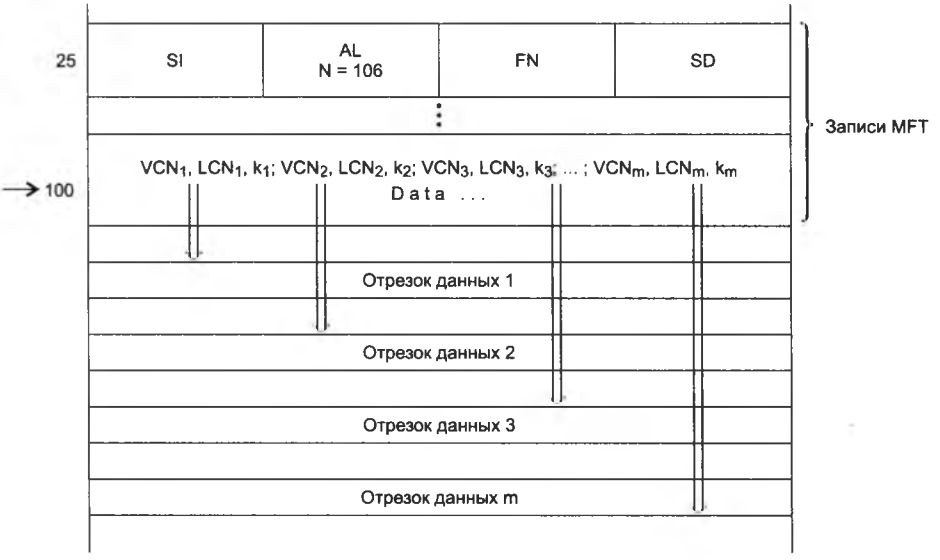


Рис. 7.24. Очень большой файл

Сверхбольшие (extremely huge) **файлы.** Для сверхбольших файлов в атрибуте *Attribute List* можно указать несколько атрибутов, расположенных в дополнительных записях MFT (рис. 7.25). Кроме того, можно использовать двойную косвенную адресацию — тогда нерезидентный атрибут будет ссылаться на другие нерезидентные атрибуты, в результате в NTFS не может быть атрибутов слишком большой для системы длины.

Каталоги NTFS

Каждый каталог NTFS представляет собой один вход в таблицу MFT, который содержит атрибут *Index Root*. Индекс содержит список файлов, входящих в каталог.

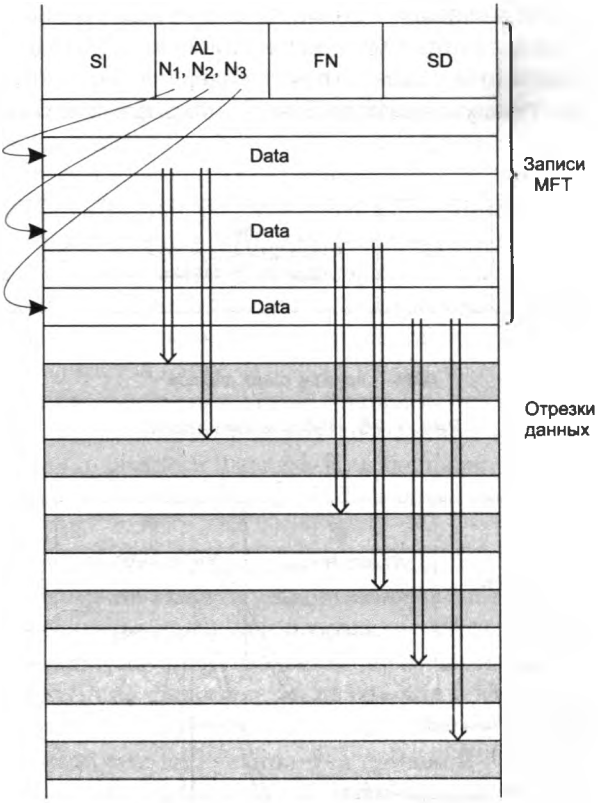


Рис. 7.25. Сверхбольшой файл

Индексы позволяют сортировать файлы для ускорения поиска, основанного на значении определенного атрибута. Обычно в файловых системах файлы сортируются по имени. NTFS позволяет использовать для сортировки любой атрибут, если он хранится в резидентной части.

Имеется две формы хранения списка файлов.

Небольшие каталоги (small indexes). Если количество файлов в каталоге невелико, то список файлов может быть резидентным в записи MFT, являющейся каталогом (рис. 7.26). Для резидентного хранения списка используется единственный атрибут — *Index Root*. Список файлов содержит значения атрибутов файла. По умолчанию — это имя файла, а также номер записи MTF, содержащей начальную запись файла.

Большие каталоги (large indexes). По мере роста каталога список файлов может потребовать нерезидентной формы хранения. Однако начальная часть списка всегда остается резидентной в корневой записи каталога таблицы MFT (рис. 7.27). Имена файлов резидентной части списка файлов являются узлами так называемого B-дерева (двоичного дерева). Остальные части списка файлов размещаются вне MFT. Для их поиска используется специальный атрибут

Index Allocation, представляющий собой адреса отрезков, хранящих остальные части списка файлов каталога. Одни части списков являются листьями дерева, другие — промежуточными узлами, то есть содержат наряду с именами файлов атрибут *Index Allocation*, указывающий на списки файлов более низких уровней.

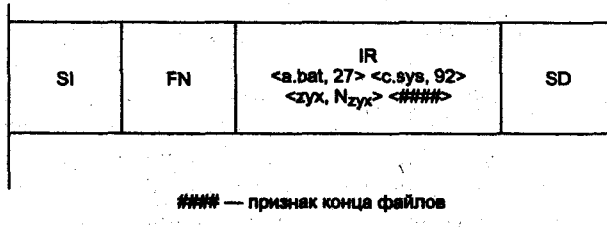


Рис. 7.26. Небольшой каталог

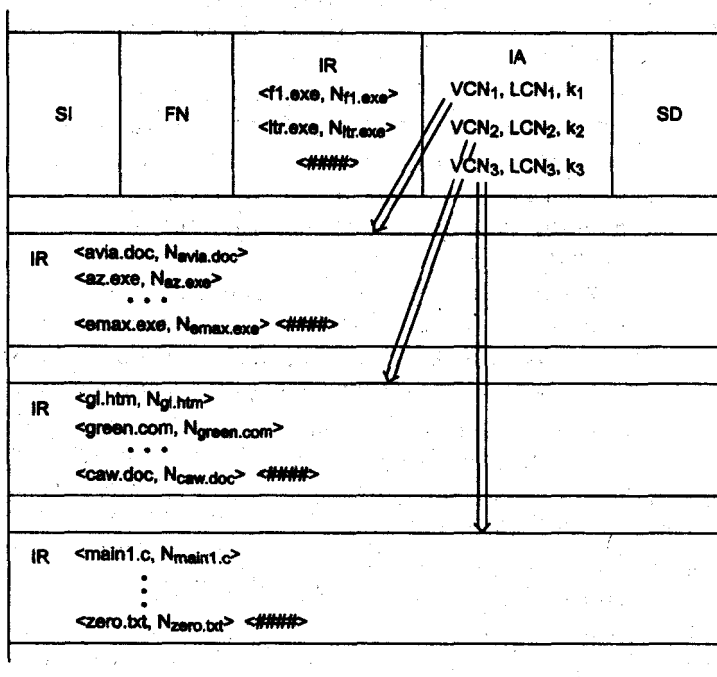


Рис. 7.27. Большой каталог

Узлы двоичного дерева делят весь список файлов на несколько групп. Имя каждого файла-узла является именем последнего файла в соответствующей группе. Считается, что имена файлов сравниваются лексикографически, то есть сначала принимаются во внимание коды первых символов двух сравниваемых имен, при этом имя считается меньшим, если код его первого символа имеет меньшее арифметическое значение, при равенстве кодов первых символов срав-

ниваются коды вторых символов имен и т. д. Например, файл f1.exe, являющийся первым узлом двоичного дерева, показанного на рис. 7.27, имеет имя, лексикографически большее имен avia.exe, az.exe, ..., emax.exe, образующих первую группу списка имен каталога. Соответственно, файл ltr.exe имеет наибольшее имя среди всех имен второй группы, а все файлы с именами, большими ltr.exe, образуют третью и последнюю группу.

Поиск в каталоге уникального имени файла, которым в NTFS является номер основной записи о файле в MFT, по его символьному имени происходит следующим образом. Сначала искомое символьное имя сравнивается с именем первого узла в резидентной части индекса. Если искомое имя меньше, это означает, что его нужно искать в первой нерезидентной группе, для чего из атрибута *Index Allocation* извлекается адрес отрезка (VCN_1, LCN_1, K_1), хранящего имена файлов первой группы. Среди имен этой группы поиск осуществляется их прямым перебором и сравнением до полного совпадения всех символов искомого имени с хранящимся в каталоге именем. При совпадении из каталога извлекается номер основной записи о файле в MFT, и остальные характеристики файла берутся уже оттуда.

Если же искомое имя больше имени первого узла резидентной части индекса, то его сравнивают с именем второго узла, и если искомое имя меньше, то описанная процедура применяется ко второй нерезидентной группе имен и т. д.

В результате вместо перебора большого количества имен (в худшем случае — всех имен каталога) выполняется сравнение с гораздо меньшим количеством имен узлов и имен в одной из групп каталога.

Если одна из групп каталога становится слишком большой, то ее также делят на группы, последние имена каждой новой группы оставляют в исходном нерезидентном атрибуте *Index Root*, а все остальные имена новых групп переносят в новые нерезидентные атрибуты типа *Index Root* (на рисунке этот случай не показан). К исходному нерезидентному атрибуту *Index Root* добавляется атрибут размещения индекса, указывающий на отрезки индекса новых групп. Если теперь при поиске имени в нерезидентной части индекса первого уровня какое-либо сравнение показывает, что искомое имя меньше, чем одно из хранящихся там имен, это говорит о том, что в данном атрибуте точного сравнения имени уже быть не может и нужно перейти к подгруппе имен следующего уровня дерева.

Файловые операции

ФС с запоминанием и без запоминания состояния операций

Файловая система ОС предоставляет пользователям набор операций для работы с файлами, оформленный в виде системных вызовов, таких, например, как `creat` (создать файл).

Чаще всего с одним и тем же файлом пользователь выполняет не одну операцию, а последовательность операций. Например, при работе текстового редактора с файлом, в котором содержится некоторый документ, пользователь обычно считывает несколько страниц текста, редактирует эти данные и записывает их на место считанных, а затем считывает страницы из другой области файла и т. п. После большого количества операций чтения и записи пользователь завершает работу с данным файлом и переходит к другому.

Какие бы операции над файлом ни выполнялись операционной системой, `read` или `write`, `creat` или `delete`, они непременно включают ряд *универсальных* действий, повторяемых в неизменном виде при выполнении любой операции.

1. По символьному имени файла ищутся его характеристики, которые хранятся в файловой системе на диске.
2. Характеристики файла копируются в оперативную память, так как только таким образом программный код может их использовать.
3. На основании характеристик файла проверяются права пользователя на выполнение запрошенной операции (чтение, запись, удаление, просмотр атрибутов файла).
4. Очищается область памяти, отведенная под временное хранение характеристик файла.

Помимо таких универсальных действий, каждая операция с файлом включает ряд *специфических* для этой операции действий. Например, считывание данных файла присуще только операции `read` (читать из файла), а запись новых данных в файл происходит только при выполнении операции `write` (записать в файл).

Пусть имеется некая последовательность операций с одним и тем же файлом `X`, например: `read(X)`, `write(X)`...

Операционная система может выполнить эту последовательность двумя способами.

- Первый подход состоит в последовательном выполнении для каждой операции всех относящихся к ней действий, как универсальных, так и специфических (рис. 7.28, *а*). То есть когда из прикладной программы поступает системный вызов на выполнение операции `read(X)`, ФС ее выполняет, отсылает ответ, а затем удаляет из своих внутренних таблиц всю информацию о выполненной операции и «с чистого листа» приступает к выполнению операции `write(X)`.
- В соответствии со вторым подходом универсальные действия выполняются лишь в начале и в конце последовательности операций, а для каждой промежуточной операции выполняются только специфические действия (рис. 7.28, *б*). То есть если первой была выполнена операция чтения `read(X)`, то при выполнении следующей операции `write(X)` вовсе не требуется снова производить поиск и перенос в оперативную память адреса файла и других его характеристик. Действительно, эта информация уже была най-

дена во время выполнения предыдущей операции `read(X)`, достаточно просто ее сохранить для повторного использования.

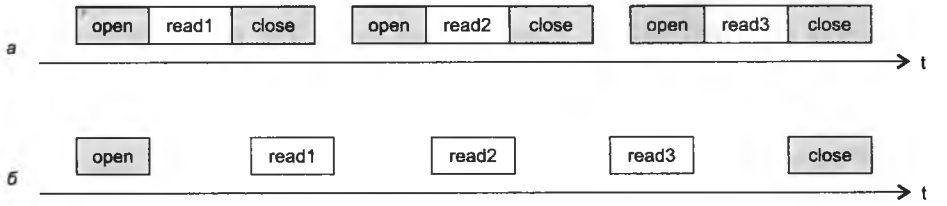


Рис. 7.28. Два способа выполнения файловых операций

В первом случае файловую систему относят к типу **ФС без запоминания состояния операций** (stateless). ФС такого типа не хранит никакой информации о предыдущих операциях, все операции выполняются абсолютно независимо друг от друга.

Во втором случае ФС запоминает информацию, относящуюся к универсальным действиям, полученную ею при выполнении самой первой операции с тем, чтобы использовать ее для выполнения всех последующих операций с файлом, вплоть до самой последней. Такие ФС относят к типу **ФС с запоминаниям состояния операций** (stateful).

Преимуществом первого способа является его большая устойчивость к сбоям в работе системы, так как каждая операция является самодостаточной и не зависит от результата предыдущей. Поэтому первый способ иногда применяется в распределенных сетевых файловых системах[1] (например, в Network File System, NFS компании Sun), когда сбои из-за потерь пакетов или отказов одного из сетевых узлов более вероятны, чем при локальном доступе к файлам.

Однако большинство файловых систем поддерживает второй способ организации, при котором файловая система сохраняет информацию о предыдущих операциях, как более экономичный и быстрый.

При втором способе в файловой системе вводится два специальных системных вызова: `open` (открытие файла) и `close` (закрытие файла).

Системный вызов открытия файла `open` выполняется перед началом любой последовательности операций с файлом, а вызов закрытия файла `close` — после окончания работы с файлом. Основной задачей вызова `open` является преобразование символьного имени файла в его уникальное числовое имя, копирование характеристик файла из дисковой области в буфер оперативной памяти и проверка прав пользователя на выполнение запрошенной операции. Вызов `close` освобождает буфер с характеристиками файла и делает невозможным продолжение операций с файлом без его повторного открытия.

Операции открытия и закрытия файла в той или иной форме утвердились в операционных системах очень давно. Даже в такой «старой» операционной

[1] Вы можете прочитать об этом в разделах «Файловые stateful- и stateless-серверы» и «Пример. Файловая система NFS» главы 11.

системе, как OS/360, существовала макрокоманда OPEN, по которой в специальном буфере, называемом DCB (Data Control Block), собирались из различных источников все нужные характеристики *набора данных* (понятие, близкое к современному понятию файла), используемые затем при выполнении операций чтения и записи.

Далее основные системные вызовы файловых операций рассматриваются более детально на примере их реализации в ОС Unix. В этой ОС они приобрели тот вид, который сегодня поддерживается практически всеми операционными системами.

Открытие файла

Системный вызов `open()` в ОС Unix работает с двумя аргументами:

- символьным именем открываемого файла;
- параметром, задающим режим открытия файла.

Режим открытия говорит системе, какие операции будут выполняться с файлом в последовательности операций до закрытия файла по системному вызову `close`, например, только чтение, только запись или чтение и запись.

При открытии файла ОС сначала выполняет преобразование первого аргумента системного вызова, то есть символьного имени файла, в его уникальное числовое имя, которым в традиционных файловых системах Unix является номер индексного дескриптора. Эта процедура была рассмотрена ранее при описании файловой системы s5.

По номеру индексного дескриптора `inode` файловая система находит нужную запись на диске и копирует из нее характеристики файла в оперативную память.

Для хранения копии индексного дескриптора используются буферные области системного виртуального пространства. Характеристики индексного дескриптора, перенесенные в оперативную память, помещаются в структуру так называемого **виртуального дескриптора** `vnode` (virtual node). Структура `vnode` включает поля индексного дескриптора файла `inode`, а также несколько перечисленных далее дополнительных полей, полезных при выполнении операций с файлом.

- Состояние индексного дескриптора в памяти, отражающее:
  - заблокирован ли файл;
  - ждет ли снятия блокировки с файла какой-либо процесс;
  - отличается ли представление характеристик файла в памяти от своей дисковой копии в результате изменения содержимого индексного дескриптора;
  - отличается ли представление файла в памяти от своей дисковой копии в результате изменения содержимого файла;
  - является ли файл точкой монтирования.
- Логический номер устройства файловой системы, содержащей файл.

- Номер индексного дескриптора. В дисковом индексном дескрипторе это поле отсутствует, так как номер определяется положением дескриптора относительно начала области индексных дескрипторов.
- Счетчик ссылок на данную структуру `vnode`.

С одним и тем же файлом в какой-то период времени могут работать различные процессы, но операционная система не создает для каждого процесса отдельную копию структуры `vnode`, а для каждого файла, с которым в данный момент работает хотя бы один процесс, хранит ровно одну копию виртуального дескриптора. При очередном открытии файла ОС проверяет, имеется ли в системной памяти структура `vnode` открываемого файла (по номеру логического устройства и номеру индексного дескриптора, которые определяются при преобразовании символьного имени), и если имеется, то счетчик ссылок на нее увеличивается на единицу. При очередном закрытии этого файла счетчик ссылок уменьшается на единицу, и если он становится равным 0, то буфер, хранящий данную структуру `vnode`, считается свободным.

Использование единственной копии характеристик файла и некоторых характеристик файловых операций (например, признака блокировки), общих для всех работающих с файлом процессов, экономит системную память. Тем не менее существуют характеристики, индивидуальные для каждого процесса, выполняющего некоторую последовательность операций с определенным файлом. Для их хранения в Unix используется структура типа `file`, которая так же, как и `vnode`, хранится в системной области памяти.

При каждом открытии процессом файла ОС проверяет права пользовательского процесса на выполнение запрошенной операции с файлом, и если проверка прошла успешно, создает в системной области памяти новую структуру `file`, описывающую как сам открытый файл, так и операции, которые процесс собирается производить с файлом (например, чтение).

Структура `file` содержит следующие поля:

- признак режима открытия (только для чтения, для чтения и записи и т. п.);
- указатель на структуру `vnode`;
- текущее смещение в файле (переменная `offset`) при операциях чтения/записи;
- счетчик ссылок на данную структуру;
- указатель на структуру, содержащую права процесса, открывшего файл (эта структура находится в дескрипторе процесса);
- указатели на предыдущую и последующую структуру `file`, связывающие все такие структуры в двойной список.

Переменная `offset`, хранящаяся в структуре `file`, позволяет ОС запомнить текущее положение условного указателя в последовательности байтов файла. При открытии файла эта переменная указывает на начальный или конечный байт файла в зависимости от заданного режима открытия. После выполнения операций чтения или записи указатель сдвигается на количество байтов, прочи-

танное или записанное в результате операции. Следующая операция застает указатель в том состоянии, в котором его оставила предыдущая операция. Прикладной программист может явно управлять положением указателя с помощью системного вызова `lseek` (см. далее).

При каждом новом открытии какого-либо файла ОС создает новую структуру `file` и помещает ее в дважды связанный список (рис. 7.29). Обычно под хранение структур `file` в системной области отводится ограниченная область, поэтому общее количество открытых файлов всеми процессами в любой момент ограничено.

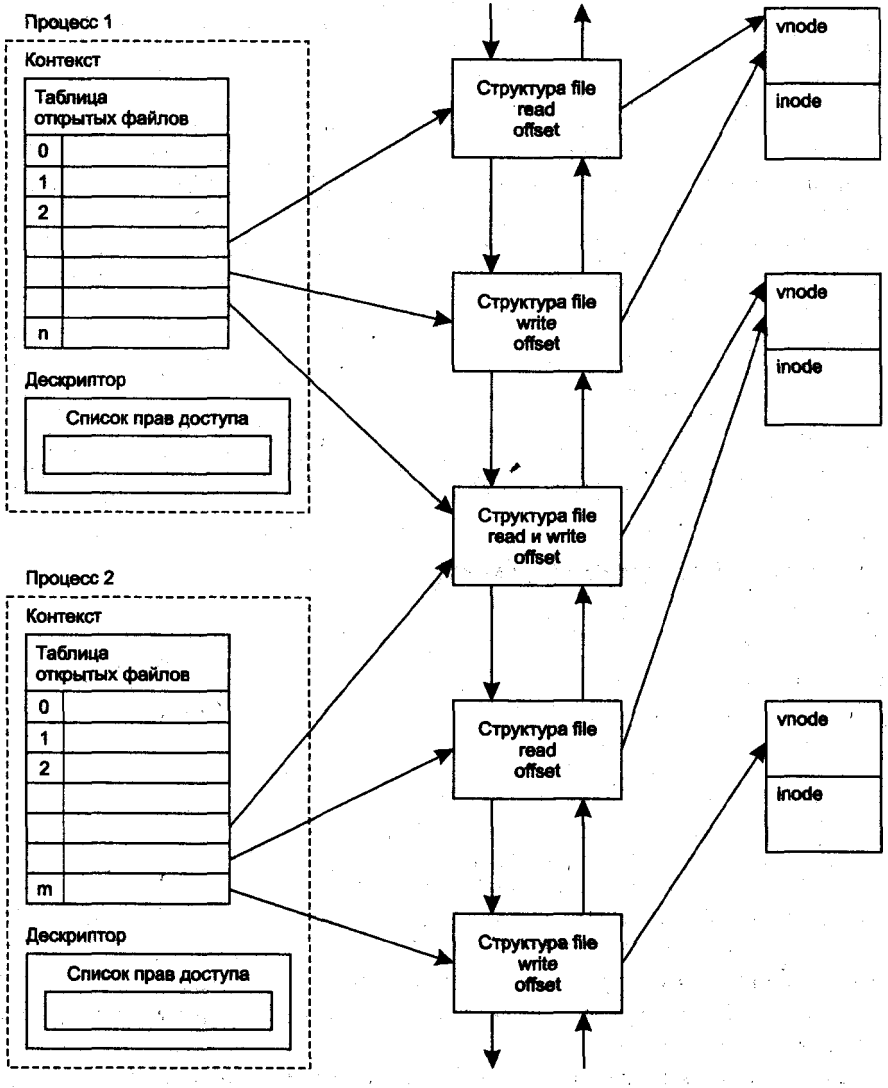


Рис. 7.29. Связь процесса с открытыми файлами

После создания структуры `file` операционная система помещает указатель на нее в таблицу открытых файлов процесса, которая находится в контексте процесса. Если процесс несколько раз открывает один и тот же файл, то структура `file` создается для каждой операции открытия. Так как контекст процесса-родителя в Unix наследуется процессом-потомком, потомок наследует и указатели на все открытые родителем файлы, получая возможность выполнять с ними операции.

Системный вызов `open` возвращает в пользовательский процесс **дескриптор файла**, который представляет собой номер записи в таблице открытых файлов процесса. Дескриптор файла имеет локальное значение только для того процесса, который открыл файл, для разных процессов одно и то же значение дескриптора указывает на разные операции в общем случае с разными файлами.

После открытия файла его дескриптор используется во всех дальнейших операциях с файлом вплоть до явного закрытия файла. Таким образом, дескриптор файла является временным уникальным именем, но не файла, а конкретной последовательности операций с этим файлом, ограниченной командами `open` и `close`.

Для открытия файла /bin/prog1.exe в режиме «только для чтения» прикладной программист может использовать следующее выражение на языке C:

```
fd = open("/bin/prog1.exe", O_RDONLY);
```

Здесь `fd` — это целочисленная переменная, сохраняющая значение дескриптора открытого файла. Ее значение должно использоваться в операциях обмена данными с файлом /bin/prog1.exe. При неудачной попытке открытия файла (нет прав для выполнения затребованной операции, неверное имя файла) переменной `fd` присваивается значение −1, которое является индикатором ошибки для всех системных вызовов Unix.

Обмен данными с файлом

Для обмена данными с предварительно открытым файлом в ОС Unix существуют системные вызовы `read` и `write`. В том случае, когда необходимо явным образом указать, с какого байта файла читать или записывать данные, используется также системный вызов `lseek`.

Системный вызов чтения данных из файла `read` имеет три аргумента:

```
read(fd, buffer, nbytes);
```

Первый аргумент `fd` является целочисленной переменной, имеющей значение дескриптора открытого файла. Второй аргумент `buffer` является указателем на область пользовательской памяти, в которую система должна поместить считанные данные. Количество байтов этой области памяти задается третьим целочисленным аргументом `nbytes`. Функция `read` возвращает действительное количество считанных байтов (оно может отличаться от заданного, если, например, была указана область чтения, выходящая за пределы файла) или же код ошибки −1.

Начало дисковой области, которую нужно прочитать с помощью вызова `read`, явно в этом системном вызове не указывается. Чтение начинается с того

байта, который задается смещением `offset` в структуре `file`. На это смещение указывает запись с номером `fd` в таблице открытых файлов процесса. После выполнения вызова `read` смещение `offset` наращивается на количество считанных байтов.

Системный вызов записи данных `write` аналогичен вызову `read`:

```
write(fd, buffer, nbytes);
```

Функция `write` записывает количество байтов `nbytes` из буфера оперативной памяти `buffer` в файл, описываемый дескриптором `fd`. Функция `write`, так же как и `read`, возвращает вызвавшей ее программе значение реально переданных ею байтов или код ошибки.

Рассмотрим пример, в котором прикладная программа работает с файлом, состоящем из записей фиксированной длины по 50 байт:

```
fd = open("/doc/qwery/base12.txt", O_RDWR);
read(fd, buffer1, 50);
read(fd, buffer2, 2500);
...
lseek(fd, 150, 0);
write (fd, output, 300);
close(fd);
```

В приведенном фрагменте программы после открытия файла /doc/query/base12.txt для чтения и записи выполняется чтение первой записи файла, а затем читается область файла, включающая еще 50 записей, со 2 по 51. После обработки считанных записей (эти инструкции опущены) указатель смещения в файле переводится на начало четвертой записи, и выполняется запись результатов в шесть последовательных записей, начиная с четвертой. Завершается фрагмент закрытием файла с помощью системного вызова `close`.

Все описанные системные вызовы являются синхронными, то есть пользовательский процесс переводится в состояние ожидания до тех пор, пока операция ввода-вывода не завершится.

Описанный набор системных вызовов, появившийся в ОС Unix еще в 70-х годах, стал стандартом де-факто для современных операционных систем. Эти традиционные системные вызовы часто в конкретных ОС дополняются оригинальными системными вызовами ввода-вывода, например операциями асинхронного типа. На основе системных вызовов ввода-вывода обычно строятся более мощные библиотечные функции ввода-вывода, составляющие прикладной интерфейс ОС.

Блокировки файлов

Блокировки файлов и отдельных записей в файлах являются средством синхронизации работающих в кооперации процессов, пытающихся использовать один и тот же файл одновременно.

Процессы могут иметь соответствующие права доступа к файлу, но одновременное применение этих прав (в особенности права записи) может привести к некорректным результатам. Примером такой ситуации является одновремен-

ное редактирование одного и того же документа несколькими пользователями. Если доступ к файлу не управляется блокировками, то каждый пользователь, который имеет право записи в файл, работает со своей копией данных файла. Результат такого редактирования непредсказуем — он зависит от того, в какой последовательности записывали изменения в файл применяемые пользователями приложения-редакторы.

Многопользовательские операционные системы обычно поддерживают специальный системный вызов, позволяющий программисту устанавливать и проверять блокировки на файл и его отдельные области. В Unix такой системный вызов называется `fcntl`. В его аргументах указываются дескриптор файла, для которого нужно установить или проверить блокировки, тип операции (блокирование или проверка, блокирование доступа для чтения или для записи), а также область блокирования — смещение от начала файла и размер в байтах.

При проверке наличия блокировок, установленных другими процессами, вызов `fcntl` немедленно возвращает управление с сообщением результата. При установке блокировки можно задать два режима работы системного вызова: с переходом процесса в состояние ожидания в том случае, если блокировку установить невозможно (синхронный системный вызов), и с немедленным возвратом в такой ситуации с сообщением отрицательного результата (асинхронный вызов).

Запрошенная блокировка записи не может быть установлена в том случае, если другой процесс уже установил свою блокировку записи на тот же файл. То есть блокировка записи является исключительной. Блокировки чтения не являются исключительными и могут устанавливаться на файл в том случае, если их области действия не перекрываются. Если на какую-то область файла установлена блокировка чтения, то на эту область нельзя установить блокировку записи.

В Unix существует два режима действия блокировок — *консультативный* (advisory) и *обязательный* (mandatory). Основным рекомендуемым для использования режимом является консультативный. При нем операционная система не занимается блокированием операций с файлом, а только устанавливает признаки блокирования областей в структурах `file`, поддерживающих операции с файлами. Кооперирующиеся процессы обязательно должны проверять наличие блокировок на файл, чтобы синхронизировать свою работу. Если же блокировки установлены, но процесс не проверяет их, то операционная система не запрещает доступ процесса к файлу, когда процесс делает системные вызовы `read` или `write`.

В обязательном режиме запрет на выполнение операции с заблокированным файлом поддерживает операционная система, поэтому процесс в любом случае не получит доступа к такому файлу. Однако при работе в этом режиме операционная система тратит много усилий и времени на его поддержание, поэтому обычно он не рекомендуется для использования.

Стандартные файлы ввода и вывода, перенаправление вывода

В ОС Unix были введены в свое время понятия **стандартных файлов**:

- стандартный файл ввода;
- стандартный файл вывода;
- стандартный файл ошибок.

Эти три уже открытых файла существуют у любого пользовательского процесса с момента его возникновения. Процесс в любое время может организовать ввод данных из стандартного файла ввода, выполнив следующий системный вызов:

```
read(stdio, bufer, nbytes);
```

Здесь `stdio` — предопределенное имя константы, обозначающей дескриптор стандартного файла ввода.

Аналогично, так как `stdout` — предопределенное имя дескриптора стандартного файла вывода, процесс может вывести данные в стандартный файл вывода, применив следующий системный вызов:

```
write(stout, buffer, nbytes);
```

За стандартным файлом ошибок закреплено имя `stderr`.

Фактически, при создании нового процесса ОС помещает в его таблицу открытых файлов три записи: с номером 0 — для стандартного файла ввода (следовательно, `stdin` всегда имеет значение 0), с номером 1 — для стандартного файла вывода (`stdout` = 1) и с номером 2 — для стандартного файла ошибок (`stderr` = 2). Соответственно, создается и три структуры типа `file`, на которые указывают первые три записи таблицы открытых файлов процесса.

В начальный период существования процесса эти три структуры `file` связываются операционной системой с одним файлом. В качестве этого файла выступает *специальный файл* — терминал, с которого вошел в систему пользователь. Такое назначение стандартных файлов достаточно естественно. Прикладные программы, запускаемые пользователем в ходе сеанса работы, чаще всего выводят результаты и сообщения об ошибках на экран терминала, за которым работает пользователь, и с клавиатуры этого же терминала считывают команды и другие исходные данные.

Модель стандартных файлов ввода-вывода рассчитана в основном на алфавитно-цифровые терминалы, управление которыми хорошо описывается потоком байтов, выводимых на экран в виде строк символов, а также потоком байтов, вводимых последовательными нажатиями клавиш.

Наиболее известной программой, широко использующей стандартные файлы ввода-вывода, является **интерпретатор команд**, называемый также **оболочкой** (shell) операционной системы. Интерпретатор постоянно читает вводимые пользователем с клавиатуры команды (из стандартного файла ввода) и либо

выполняет их самостоятельно с помощью своих внутренних функций (такие команды называются внутренними), либо интерпретирует как имена исполняемых файлов на диске, которые необходимо запускать на выполнение в качестве отдельных процессов (внешние команды). Сообщения интерпретатор выводит на экран терминала — стандартный файл вывода.

Стандартные файлы ввода и ввода широко используются не только интерпретатором команд, но и самими командами. Многие внутренние и внешние команды устроены так, что они либо читают свои исходные данные из стандартного файла ввода, либо выводят результаты в стандартный файл вывода. Если же команда делает то и другое, она называется **фильтром**.

Рассмотрим несколько примеров команд ОС Unix, работающих со стандартными файлами ввода и вывода:

- `ls dir2` — читает записи каталога `dir2` и выводит их в определенном символьном формате в стандартный файл вывода;
- `wc` — фильтр, который читает последовательность байтов из стандартного файла ввода, подсчитывает число слов, строк или символов в считанных данных и выводит результат в стандартный файл вывода;
- `who` — выводит в стандартный файл информацию о пользователях, работающих в системе.

Интерпретатор команд выполняет также такую важную функцию, как *перенаправление стандартного ввода и вывода*. Под этим понимается замена файла-терминала, используемого по умолчанию в качестве стандартных файлов ввода и вывода, произвольным файлом. Механизм перенаправления основан на том, что приложение не знает, какой именно файл является стандартным, а просто использует определенный дескриптор в качестве указателя на этот файл. Поэтому для перенаправления ввода-вывода достаточно создать процесс выполнения команды с нестандартной связью стандартной записи в таблице открытых файлов.

Перенаправление осуществляется с помощью специальных конструкций командного языка. Для указания интерпретатору на необходимость перенаправить стандартный ввод в файл `file` используется следующая конструкция:

```
< file
```

Для перенаправления стандартного вывода требуется следующая конструкция:

```
> file
```

Например, следующая командная строка запишет данные о содержимом каталога `dir2` в файл `a.txt`:

```
ls dir2 > a.txt
```

Механизм перенаправления ввода-вывода, введенный ОС Unix, получил широкое распространение в интерпретаторах команд многих операционных систем, включая MS-DOS, Windows, OS/2.

Контроль доступа к файлам

Файл как разделяемый ресурс

Файлы — это частный, хотя и самый популярный, вид разделяемых ресурсов, доступ к которым операционная система должна контролировать. Существуют и другие виды ресурсов, с которыми пользователи работают совместно. Прежде всего, это различные внешние устройства: принтеры, модемы, графопостроители и т. п. Область памяти, используемая для обмена данными между процессами, также является примером разделяемого ресурса. Да и сами процессы в некоторых случаях выступают в этой роли, например, когда пользователи ОС посылают процессам сигналы, на которые процесс должен реагировать.

Во всех этих случаях действует общая схема: пользователи пытаются выполнить с разделяемым ресурсом определенные операции, а ОС должна решать, имеют ли пользователи на это право. Пользователи являются *субъектами* доступа, а разделяемые ресурсы — *объектами*. Пользователь осуществляет доступ к объектам операционной системы не непосредственно, а с помощью прикладных процессов, которые запускаются от его имени. Для каждого типа объектов существует *набор операций*, которые с ними можно выполнять. Например, для файлов это операции чтения, записи, удаления, выполнения; для принтера — перезапуск, очистка очереди документов, приостановка печати документа и т. д. Система контроля доступа ОС должна предоставлять средства для задания прав пользователей по отношению к объектам дифференцированно по операциям, например, пользователю может быть разрешена операция чтения и выполнения файла, а операция удаления — запрещена.

Во многих операционных системах реализованы механизмы, которые позволяют с единых позиций управлять доступом к объектам различного типа. Так, представление устройств ввода-вывода в виде специальных файлов в операционных системах Unix является примером такого подхода: в этом случае при доступе к устройствам используются те же атрибуты безопасности и алгоритмы, что и при доступе к обычным файлам и каталогам.

Еще дальше продвинулись в этом направлении разработчики операционных систем семейства Windows NT (Windows NT 3.1, Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003, Windows Vista). В ОС этого семейства используется унифицированная структура — **объект безопасности**. Данная структура создается не только для файлов и внешних устройств, но и для любых разделяемых ресурсов: секций памяти, синхронизирующих примитивов типа семафоров и мьютексов и т. п. Это позволяет использовать для контроля доступа к ресурсам любого вида общий модуль ядра — **менеджер безопасности**.

В качестве субъектов доступа могут выступать как отдельные *пользователи*, так и *группы пользователей*. Определение индивидуальных прав доступа для каждого пользователя позволяет максимально гибко задать политику расходования разделяемых ресурсов в вычислительной системе. Однако этот способ приводит в больших системах к чрезмерной загрузке администратора рутинной

работой по повторению одних и тех же операций для пользователей с одинаковыми правами. Объединение таких пользователей в группу и задание прав доступа в целом для группы является одним из основных приемов администрирования в больших системах.

У каждого объекта доступа существует *владелец*. Владельцем может быть как отдельный пользователь, так и группа пользователей. Владелец объекта имеет право выполнять с ним любые допустимые для данного объекта операции. Во многих операционных системах существует особый пользователь (*superuser*, *root*, *administrator*), который имеет все права по отношению к любым объектам системы, не обязательно являясь их владельцем. Под таким именем работает администратор системы, которому необходим полный доступ ко всем файлам и устройствам для управления политикой доступа.

Различают два основных подхода к определению прав доступа:

- избирательный доступ;
- мандатный доступ.

Избирательный доступ имеет место, когда для каждого объекта сам владелец может определить допустимые операции с объектами. Этот подход называется также произвольным (от discretionary — предоставленный на собственное усмотрение) доступом, так как позволяет администратору и владельцам объектов определить права доступа произвольным образом, по их желанию. Между пользователями и группами пользователей в системах с избирательным доступом нет жестких иерархических взаимоотношений, то есть взаимоотношений, которые определены по умолчанию и которые нельзя изменить. Исключение делается только для администратора, по умолчанию наделяемого всеми правами.

Мандатный доступ (от mandatory — обязательный, принудительный) — это такой подход к определению прав доступа, при котором система наделяет пользователя определенными правами по отношению к каждому разделяемому ресурсу (в данном случае файлу) в зависимости от того, к какой группе пользователь отнесен. От имени системы выступает администратор, а владельцы объектов лишены возможности управлять доступом к ним по своему усмотрению. Все группы пользователей в такой системе образуют строгую иерархию, причем каждая группа пользуется всеми правами группы более низкого уровня иерархии, к которым добавляются права данного уровня. Членам какой-либо группы не разрешается предоставлять свои права членам групп более низких уровней иерархии. Мандатный способ доступа близок к схемам, применяемым для доступа к секретным документам: пользователь может входить в одну из групп, отличающихся правом на доступ к документам с соответствующим грифом секретности, например «для служебного пользования», «секретно», «совершенно секретно» и «государственная тайна». При этом пользователи группы «совершенно секретно» имеют право работать с документами «секретно» и «для служебного пользования», так как эти виды доступа разрешены для более низких в иерархии групп. Однако сами пользователи не распоряжаются правами доступа — этой возможностью наделен только особый чиновник учреждения.

Мандатные системы доступа считаются более надежными, но менее гибкими, обычно они применяются в специализированных вычислительных системах с повышенными требованиями к защите информации. В универсальных системах используются, как правило, избирательные методы доступа, о которых и идет речь далее.

Механизм контроля доступа

Каждый пользователь и каждая группа пользователей обычно имеют символьное имя, а также уникальный числовой идентификатор. При выполнении процедуры логического входа в систему пользователь сообщает свое символьное имя и пароль, а операционная система определяет соответствующие числовые идентификаторы пользователя и групп, в которые он входит. Все идентификационные данные, в том числе имена и идентификаторы пользователей и групп, пароли пользователей, а также сведения о вхождении пользователя в группы, хранятся либо в специальном файле; например файле /etc/passwd в Unix, либо специальной базе данных, как в ОС семейства Windows NT.

Вход пользователя в систему порождает процесс-оболочку, который поддерживает диалог с пользователем и запускает для него другие процессы. Процесс-оболочка получает от пользователя символьное имя и пароль и находит по ним числовые идентификаторы пользователя и его групп. Эти идентификаторы связываются с каждым процессом, запущенным оболочкой для данного пользователя. Говорят, что процесс выступает от имени данного пользователя и данных групп пользователей. В наиболее типичном случае любой порождаемый процесс наследует идентификаторы пользователя и групп от процесса родителя.

Определить права доступа к ресурсу значит определить для каждого пользователя набор операций, которые ему разрешено применять к данному ресурсу. В разных операционных системах для одних и тех же типов ресурсов может быть определен свой список дифференцируемых операций доступа.

ПРИМЕЧАНИЕ Для определенности далее мы будем рассматривать механизмы контроля доступа к таким объектам, как файлы и каталоги, но эти же механизмы могут использоваться в современных операционных системах для контроля доступа к объектам любого типа, отличия заключаются лишь в наборе операций, характерных для того или иного класса объектов.

Для файловых объектов этот список может включать следующие операции:

- создание файла;
- уничтожение файла;
- открытие файла;
- закрытие файла;
- чтение файла;
- запись в файл;

- дополнение файла;
- поиск в файле;
- получение атрибутов файла;
- установка новых значений атрибутов;
- переименование файла;
- выполнение файла;
- чтение каталога;
- смена владельца;
- изменение прав доступа.

Набор файловых операций ОС может состоять из большого количества элементарных операций, а может включать всего несколько укрупненных операций. Приведенный список является примером первого подхода, который позволяет весьма тонко управлять правами доступа пользователей, но создает значительную нагрузку на администратора. Пример укрупненного подхода демонстрируют операционные системы семейства Unix, в которых существуют всего три операции с файлами и каталогами: *читать* (read, r), *писать* (write, w) и *выполнить* (execute, x). Хотя в Unix для операций используется всего три названия, в действительности им соответствует гораздо больше операций. Например, содержание операции *выполнить* зависит от того, к какому объекту она применяется. Если операция *выполнить файл* интуитивно понятна, то операция *выполнить каталог* интерпретируется как поиск в каталоге определенной записи. Поэтому администратор Unix, по сути, располагает более обширным списком операций, чем это кажется на первый взгляд.

В ОС семейства Windows NT разработчики применили гибкий подход — они реализовали возможность выполнения операций с файлами на двух уровнях: по умолчанию администратор работает на укрупненном уровне (уровень стандартных операций), а при желании может перейти на элементарный уровень (уровень индивидуальных операций).

В самом общем случае права доступа могут быть описаны матрицей прав доступа, в которой столбцы соответствуют всем файлам системы, строки — всем пользователям, а на пересечении строк и столбцов указываются разрешенные операции (рис. 7.30).

Практически во всех операционных системах матрица прав доступа хранится «по частям», то есть для каждого файла или каталога создается так называемый **список управления доступом** (Access Control List, ACL), в котором описываются права на выполнение операций пользователей и групп пользователей по отношению к этому файлу или каталогу. Список управления доступа является частью характеристик файла или каталога и хранится на диске в соответствующей области, например в индексном дескрипторе файловой системы ufs. Не все файловые системы поддерживают списки управления доступом, например, его не поддерживает файловая система FAT, так как она разрабатывалась для однопользовательской однопрограммной операционной системы MS-DOS, для которой задача защиты от несанкционированного доступа не актуальна.

Имена пользователей	Имена файлов: modern.txt	win.exe	class.dbf	unix.ppt
kira	читать	выполнять	—	выполнять
genua	читать	выполнять	—	выполнять читать
nataly	читать	—	—	выполнять читать
victor	читать писать	—	создать	—

Рис. 7.30. Матрица прав доступа

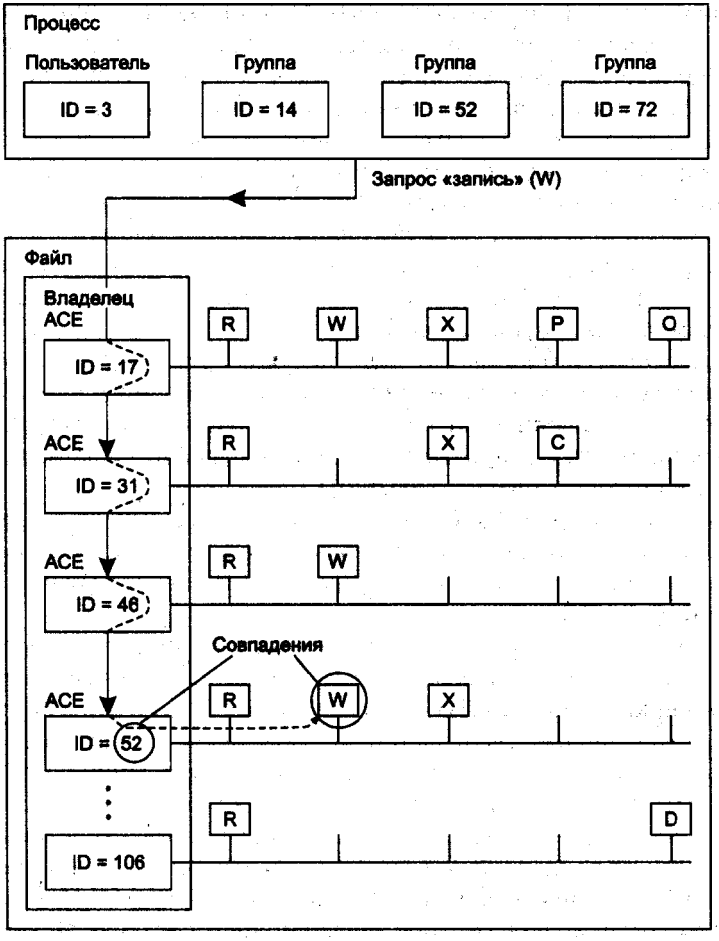


Рис. 7.31. Проверка прав доступа

Обобщенно формат списка управления доступом можно представить в виде набора идентификаторов пользователей и групп пользователей, в котором для каждого идентификатора указывается набор разрешенных операций с объектом (рис. 7.31). Говорят, что список ACL состоит из **элементов управления доступом** (Access Control Element, ACE), причем каждый элемент соответствует одному идентификатору. Список ACL с добавленным к нему идентификатором владельца называют характеристиками безопасности.

В приведенном на рисунке примере процесс, который выступает от имени пользователя с идентификатором 3 и групп с идентификаторами 14, 52 и 72, пытается выполнить операцию записи (W) в файл. Файлом владеет пользователь с идентификатором 17. Операционная система, получив запрос на запись, находит характеристики безопасности файла (на диске или в буферной системной области) и последовательно сравнивает все идентификаторы процесса с идентификатором владельца файла и идентификаторами пользователей и групп в элементах ACE. В данном примере один из идентификаторов группы, от имени которой выступает процесс, а именно 52, совпадает с идентификатором одного из элементов ACE. Так как пользователю с идентификатором 52 разрешена операция чтения (признак W имеется в наборе операций этого элемента), то ОС разрешает процессу выполнение операции.

Описанная обобщенная схема хранения информации о правах доступа и процедуры проверки имеет в каждой операционной системе свои особенности, которые рассматриваются далее на примере операционных систем Unix и семейства Windows NT.

Контроль доступа в ОС Unix

В ОС Unix права доступа к файлу или каталогу определяются для трех субъектов:

- владельца файла (идентификатор User ID, UID);
- членов группы, к которой принадлежит владелец (Group ID, GID);
- всех остальных пользователей системы.

С учетом того, что в Unix определены всего три операции над файлами и каталогами (*чтение*, *запись*, *выполнение*), характеристики безопасности файла включают девять признаков, задающих возможность выполнения каждой из трех операций для каждого из трех субъектов доступа. Например, если владелец файла разрешил себе выполнение всех трех операций, для членов группы — чтение и выполнение, а для всех остальных пользователей — только выполнение, то девять характеристик безопасности файла выглядят следующим образом:

```
rwx r-x r--
```

Здесь r, w и x обозначают операции чтения, записи и выполнения соответственно. Именно в таком виде выводит информацию о правах доступа к файлам команда просмотра содержимого каталога ls. Суперпользователю Unix все

виды доступа позволены всегда, поэтому его идентификатор (он имеет значение 0) не фигурирует в списках управления доступом.

С каждым процессом Unix связаны два идентификатора: пользователя, от имени которого был создан этот процесс, и группы, которой принадлежит данный пользователь. Эти идентификаторы носят название реального идентификатора пользователя (Real User ID, RUID) и реального идентификатора группы (Real Group ID, RGID).

Однако, как показано на рис. 7.32, при проверке прав доступа к файлу используются не эти идентификаторы, а так называемые эффективные идентификаторы пользователя (Effective User ID, EUID) и группы (Effective Group ID, EGID).

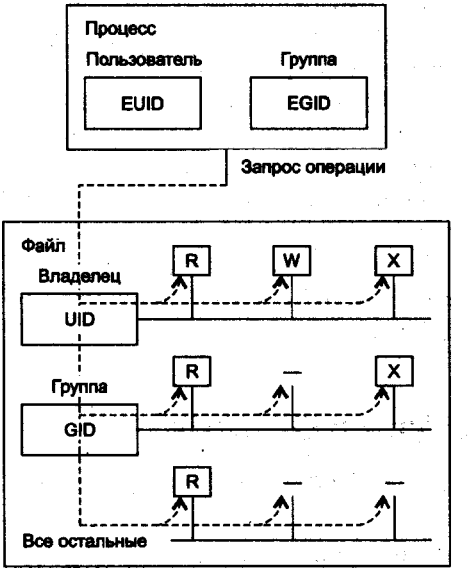


Рис. 7.32. Проверка прав доступа в Unix

Введение эффективных идентификаторов позволяет процессу выступать в некоторых случаях от имени пользователя и группы, отличных от тех, которые ему достались при рождении. В исходном состоянии эффективные идентификаторы совпадают с реальными.

Случаи, когда процесс выполняет системный вызов `exec` для запуска приложения, хранящегося в некотором файле, в Unix связаны со сменой процессом исполняемого кода. В рамках данного процесса начинает выполняться новый код, и если в характеристиках безопасности этого файла указаны признаки разрешения смены идентификаторов пользователя и группы, то происходит смена эффективных идентификаторов процесса. Файл имеет два признака разрешения смены идентификатора — Set User ID on execution (SUID) и Set Group ID on execution (SGID), которые разрешают смену идентификаторов пользователя и группы при выполнении данного файла.

Механизм эффективных идентификаторов позволяет пользователю получать некоторые виды доступа, которые ему явно не разрешены, но только с помощью вполне ограниченного набора приложений, хранящихся в файлах с установленными признаками смены идентификаторов. Пример такой ситуации приведен на рис. 7.33.

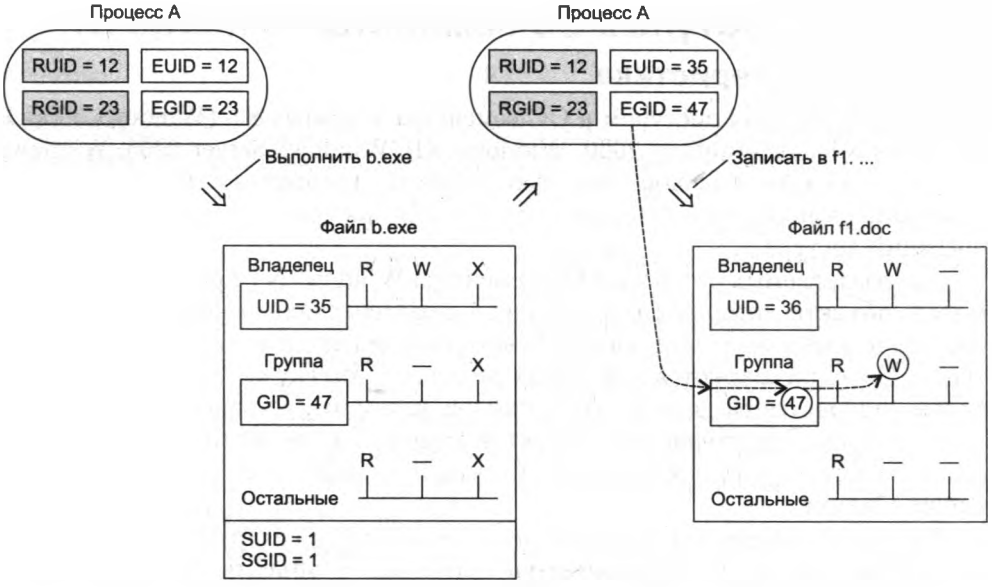


Рис. 7.33. Смена эффективных идентификаторов процесса

Первоначально процесс *A* имел эффективные идентификаторы пользователя и группы (12 и 23 соответственно), совпадающие с реальными. На каком-то этапе работы процесс запросил выполнение приложения из файла `b.exe`. Процесс может выполнить файл `b.exe`, хотя его эффективные идентификаторы не совпадают с идентификатором владельца и группы файла, так как выполнение разрешено всем пользователям.

Файл `b.exe` имеет установленные признаки смены идентификаторов SUID и SGID, поэтому одновременно со сменой кода процесс меняет значения эффективных идентификаторов (35 и 47). Вследствие этого при последующей попытке записать данные в файл `f1.doc` процессу *A* это удается, так как его новый эффективный идентификатор группы совпадает с идентификатором группы файла `f1.doc`. Без смены идентификаторов эта операция для процесса *A* была бы запрещена.

Описанный механизм преследует те же цели, что и рассмотренный ранее механизм подчиненных сегментов процессора Pentium.

Использование модели файла как универсальной модели разделяемого ресурса позволяет в Unix применять одни и те же механизмы для контроля доступа к файлам, каталогам, принтерам, терминалам и разделяемым сегментам памяти.

Система управления доступом ОС Unix была разработана в 70-е годы и с тех пор мало изменилась. Это достаточно простая система позволяет во многих случаях решить поставленные перед администратором задачи по предотвращению несанкционированного доступа, однако такое решение иногда требует слишком больших ухищрений или же вовсе не может быть реализовано.

Контроль доступа в ОС семейства Windows NT

Общая характеристика

Система управления доступом в ОС семейства Windows NT (Windows NT 3.1, Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003, Windows Vista) отличается высокой степенью гибкости, которая достигается за счет большого разнообразия субъектов и объектов доступа, а также детализации операций доступа.

Для разделяемых ресурсов в ОС семейства Windows NT применяется общая модель **объекта**, который содержит такие характеристики безопасности, как набор допустимых операций, идентификатор владельца, список управления доступом. Объекты создаются для любых ресурсов в том случае, когда они являются или становятся разделяемыми — файлов, каталогов, устройств, секций памяти, процессов. Характеристики объектов делятся на две части — общую часть, состав которой не зависит от типа объекта, и индивидуальную, определяемую типом объекта.

Все объекты хранятся в древовидных иерархических структурах, элементами которых являются **объекты-ветви** (каталоги) и **объекты-листья** (файлы). Для объектов файловой системы такая схема отношений является прямым отражением иерархии каталогов и файлов. Для объектов других типов иерархическая схема отношений имеет свое содержание, например, для процессов она отражает связи «родитель-потомок», а для устройств отражает принадлежность к определенному типу устройств и связи устройства с другими устройствами, например SCSI-контроллера с дисками.

Проверка прав доступа для объектов любого типа выполняется централизованно с помощью **монитора безопасности** (Security Reference Monitor), работающего в привилегированном режиме. Централизация функций контроля доступа повышает надежность средств защиты информации операционной системы по сравнению с распределенной реализацией, когда в различных модулях ОС имеются свои процедуры проверки прав доступа, и вероятность ошибки программиста от этого возрастает.

Для системы безопасности ОС семейства Windows NT характерно наличие большого количества различных предопределенных (встроенных) субъектов доступа — как отдельных пользователей, так групп. Так, в системе всегда имеются пользователи *Administrator*, *System* и *Guest*, а также группы *Users*, *Administrators*, *Account Operators*, *Server Operators*, *Everyone* и другие. Смысл этих встроенных пользователей и групп состоит в том, что они наделены некоторыми правами, облегчая администратору работу по созданию эффективной системы разграничения доступа. При добавлении нового пользователя администратору остается

только решить, к какой группе или группам отнести этого пользователя. Конечно, администратор может создавать новые группы, а также добавлять права к встроенным группам для реализации собственной политики безопасности, но во многих случаях встроенных групп оказывается вполне достаточно.

ОС семейства Windows NT поддерживает три класса *операций доступа*, которые отличаются типом субъектов и объектов, участвующих в этих операциях:

- разрешения;
- права;
- возможности пользователей.

Разрешения (permissions) — это множество операций, которые могут быть определены для субъектов всех типов по отношению к объектам любого типа: файлам, каталогам, принтерам, секциям памяти и т. д. Разрешения по своему назначению соответствуют правам доступа к файлам и каталогам в ОС Unix.

Права (user rights) определяются для субъектов типа группа на выполнение некоторых системных операций: установка системного времени, архивирование файлов, выключение компьютера и т. п. В этих операциях участвует особый объект доступа — операционная система в целом. В основном именно права, а не разрешения отличают одну встроенную группу пользователей от другой. Некоторые права у встроенной группы являются также встроенными — их у данной группы нельзя удалить. Остальные права встроенной группы можно удалять (или добавлять из общего списка прав).

Возможности пользователей (user abilities) определяются для отдельных пользователей на выполнение действий, связанных с формированием их операционной среды, например изменение состава главного меню программ, возможность пользоваться пунктом меню Run (Выполнить) и т. п. За счет уменьшения набора возможностей (доступных пользователю по умолчанию) администратор может «заставить» пользователя работать с той операционной средой, которую администратор считает наиболее подходящей и ограждающей пользователя от возможных ошибок.

Права и разрешения, данные группе, автоматически предоставляются ее членам, позволяя администратору рассматривать большое количество пользователей как единицу учетной информации и минимизировать свои действия.

Проверка разрешений доступа процесса к объекту производится в ОС семейства Windows NT в основном в соответствии с общей схемой доступа, представленной на рис. 7.31.

При входе пользователя в систему для него создается так называемый **токен**, или **маркер доступа** (access token), включающий:

- идентификатор пользователя;
- идентификаторы всех групп, в которые входит пользователь;
- список управления доступом (ACL) по умолчанию, который состоит из разрешений и применяется к создаваемым процессом объектам;
- список прав пользователя на выполнение системных действий.

Все объекты, включая файлы, потоки, события, даже токены доступа, когда они создаются, снабжаются дескриптором безопасности. **Дескриптор безопасности** содержит список управления доступом (ACL). Владелец объекта, обычно пользователь, который его создал, обладает правом избирательного управления доступом к объекту и может изменять ACL объекта, чтобы позволить или не позволить другим осуществлять доступ к объекту. Встроенный администратор в ОС семейства Windows NT в отличие от суперпользователя в ОС Unix может не иметь некоторых разрешений на доступ к объекту. Для реализации этой возможности идентификаторы администратора и группы администраторов могут входить в ACL, как и идентификаторы рядовых пользователей. Однако администратор все же имеет возможность выполнять любые операции с любыми объектами, так как он всегда может стать владельцем объекта, а затем уже как владелец получить полный набор разрешений.

ПРИМЕЧАНИЕ Чтобы компенсировать такое «ущемление» прав владельца, операционная система ограничивает администратора тем, что он не может вернуть владение предыдущему владельцу объекта. В результате пользователь-владелец всегда может узнать о том, что с его файлом или принтером работал администратор.

При запросе процессом некоторой операции доступа к объекту в ОС семейства Windows NT управление всегда передается монитору безопасности, который сравнивает идентификаторы пользователя и групп пользователей из токена доступа с идентификаторами, хранящимися в элементах ACL объекта. В отличие от Unix здесь в элементах ACL могут существовать как списки разрешенных, так и списки запрещенных для пользователя операций.

Система безопасности могла бы осуществлять проверку разрешений каждый раз, когда процесс использует объект. Но список ACL состоит из многих элементов, процесс в течение своего существования может иметь доступ ко многим объектам, и количество активных процессов в каждый момент времени также велико. Поэтому проверка выполняется только при каждом открытии, а не при каждом использовании объекта.

Для смены в некоторых ситуациях процессом своих идентификаторов в ОС семейства Windows NT используется механизм *олицетворения* (impersonation). В этих ОС существуют простые субъекты и субъекты-серверы. **Простой субъект** — это процесс, которому не разрешается смена токена доступа и, соответственно, смена идентификаторов. **Субъект-сервер** — это процесс, который работает в качестве сервера и обслуживает процессы своих клиентов (например, процесс файлового сервера). Поэтому такому процессу разрешается получить токен доступа у процесса-клиента, запросившего у сервера выполнения некоторого действия, и использовать его при доступе к объектам.

В ОС семейства Windows NT однозначно определены правила, по которым вновь создаваемому объекту назначается список ACL. Если вызывающий код во время создания объекта явно задает все права доступа к вновь создаваемому объекту, то система безопасности приписывает объекту этот список ACL.

Если же вызывающий код не снабжает объект списком ACL, а объект имеет имя, то применяется *принцип наследования* разрешений. Система безопасности просматривает ACL того каталога объектов, в котором хранится имя нового объекта. Некоторые из входов ACE каталога объектов могут быть помечены как наследуемые. Это означает, что они могут быть приписаны новым объектам, создаваемым в этом каталоге.

В том случае, когда процесс не задал явно список ACL для создаваемого объекта и объект-каталог не имеет наследуемых элементов ACE, используется список ACL по умолчанию из токена доступа процесса.

Наследование разрешений чаще всего применяется при создании нового объекта. Особенно эффективно наследование разрешений при создании файлов, так как эта операция самая распространенная в системе.

Разрешения на доступ к каталогам и файлам

В ОС семейства Windows NT администратор может управлять доступом пользователей к каталогам и файлам только в разделах диска, в которых установлена файловая система NTFS. Разделы FAT не поддерживаются средствами защиты, так как в FAT у файлов и каталогов отсутствуют атрибуты для хранения списков управления доступом. Доступ к каталогам и файлам контролируется за счет установки соответствующих разрешений.

В ОС семейства Windows NT предусмотрено два типа разрешений:

- **индивидуальные разрешения** относятся к элементарным операциям с каталогами и файлами;
- **стандартные разрешения** являются объединением нескольких индивидуальных разрешений.

В табл. 7.2 показано шесть индивидуальных разрешений (элементарных операций), смысл которых различается для каталогов и файлов.

Таблица 7.2. Индивидуальные разрешения для каталогов и файлов

Разрешение	Для каталога	Для файла
Read (R)	Чтение имен файлов и каталогов, входящих в данный каталог, а также атрибутов и владельца каталога	Чтение данных, атрибутов, имени владельца и разрешений файла
Write (W)	Добавление файлов и каталогов, изменение атрибутов каталога, чтение владельца и разрешений каталога	Чтение владельца и разрешений файла, изменение атрибутов файла, изменение и добавление данных файла
Execute (X)	Чтение атрибутов каталога, выполнение изменений в каталогах, входящих в данный каталог, чтение имени владельца и разрешений каталога	Чтение атрибутов файла, имени владельца и разрешений. Выполнение файла, если он хранит код программы
Delete (D)	Удаление каталога	Удаление файла

продолжение

Таблица 7.2 (*продолжение*)

Разрешение	Для каталога	Для файла
Change Permission (P)	Изменение разрешений каталога	Изменение разрешений файла
Take Ownership (O)	Вступление во владение каталогом	Вступление во владение файлом

Для файлов определено четыре стандартных разрешения: *No Access*, *Read*, *Change* и *Full Control*, которые объединяют индивидуальные разрешения, перечисленные в табл. 7.3.

Таблица 7.3. Стандартные разрешения, объединяющие индивидуальные разрешения

Стандартное разрешение	Индивидуальные разрешения
No Access	Ни одного
Read	RX
Change	RWXD
Full Control	Все

Разрешение *Full Control* отличается от *Change* тем, что дает право на изменение разрешений (*Change Permission*) и вступление во владение файлом (*Take Ownership*).

Для каталогов определено семь стандартных разрешений: *No Access*, *List*, *Read*, *Add*, *Add&Read*, *Change* и *Full Control*. В табл. 7.4 показано соответствие стандартных разрешений индивидуальным разрешениям для каталогов, а также то, каким образом эти стандартные разрешения преобразуются в индивидуальные разрешения для файлов, входящих в каталог в том случае, если файлы наследуют разрешения каталога.

Таблица 7.4. Соответствие стандартных разрешений индивидуальным разрешениям при наследовании

Стандартные разрешения	Индивидуальные разрешения для каталога	Индивидуальные разрешения для файлов каталога при наследовании
No Access	Ни одного	Ни одного
List	RX	Не определены
Read	RX	RX
Add	WX	Не определены
Add & Read	RWX	RX
Change	RWXD	RWXD
Full Control	Все	Все

При создании файла он наследует разрешения от каталога указанным способом только в том случае, если у каталога установлен признак наследования его разрешений. Стандартная оболочка ОС семейства Windows NT — Проводник Windows (Windows Explorer) — не позволяет установить такой признак для каждого разрешения отдельно (то есть задать маску наследования), управляя наследованием по принципу «все или ничего».

Существует ряд правил, которые определяют действие разрешений.

- Пользователи не могут работать с каталогом или файлом, если они не имеют явного разрешения на это или же они не относятся к группе, которая имеет соответствующее разрешение.
- Разрешения имеют накопительный эффект за исключением разрешения *No Access*, которое отменяет все остальные имеющиеся разрешения. Например, если группа *Engineering* имеет разрешение *Change* для какого-то файла, а группа *Finance* имеет для этого файла только разрешение *Read* и Петров является членом обеих групп, то у Петрова будет разрешение *Change*. Однако если разрешение для группы *Finance* изменится на *No Access*, то Петров не сможет использовать этот файл, несмотря на то, что он член группы, которая имеет доступ к файлу.

По умолчанию в окнах Проводника Windows находят свое отражение стандартные права, а переход к отражению индивидуальных прав происходит только при выполнении некоторых действий. Это стимулирует администратора и пользователей к получению тех наборов прав, которые разработчики ОС посчитали наиболее удобными.

Встроенные группы пользователей и их права

Гибкость системы безопасности ОС семейства Windows NT во многом определяется наличием в ней достаточно широкого набора прав групп пользователей на выполнение системных действий. Для иллюстрации этого утверждения в следующих двух табл. 7.5 и 7.6 приводятся списки изменяемых и встроенных прав для встроенных групп ОС семейства Windows NT.

Таблица 7.5. Изменяемые права встроенных групп

Права	Administrators	Server Operators	Account Operators	Print Operators	Backup Operators	Everyone	Users	Guests
Log on locally (локальный логический вход)	Есть	Есть	Есть	Есть	Есть	Нет	Нет	Нет
Access this computer from network (доступ к данному компьютеру через сеть)	Есть	Нет	Нет	Нет	Нет	Есть	Нет	Нет
Take ownership of files (установление прав собственности на файлы)	Есть	Нет	Нет	Нет	Нет	Нет	Нет	Нет

продолжение

Таблица 7.3 (*продолжение*)

Права	Administrators	Server Operators	Account Operators	Print Operators	Backup Operators	Everyone	Users	Guests
Manage auditing and security log (управление аудитом и учетом событий, связанных с безопасностью)	Есть	Нет	Нет	Нет	Нет	Нет	Нет	Нет
Change the system time (изменение системного времени)	Есть	Есть	Нет	Нет	Нет	Нет	Нет	Нет
Shutdown the system (останов системы)	Есть	Есть	Нет	Нет	Есть	Нет	Нет	Нет
Force shutdown from remote system (инициирование останова с удаленной системы)	Есть	Есть	Нет	Нет	Нет	Нет	Нет	Нет
Backup files and directories (резервное копирование файлов и каталогов)	Есть	Есть	Есть	Есть	Есть	Нет	Нет	Нет
Restore files and directories (восстановление файлов и каталогов со стримера)	Есть	Есть	Нет	Нет	Есть	Нет	Нет	Нет
Load and unload device drivers (загрузка и выгрузка драйверов устройств)	Есть	Нет	Нет	Нет	Нет	Нет	Нет	Нет
Add workstation to domain (добавление рабочих станций к домену)	Есть	Нет	Нет	Нет	Нет	Нет	Нет	Нет

Таблица 7.6. Встроенные права встроенных групп

Встроенные права	Administrators	Server Operators	Account Operators	Print Operators	Backup Operators	Everyone	Users	Guests
Create and manage user accounts (создание и управление пользовательской учетной информацией)	Есть	Нет	Есть	Нет	Нет	Нет	Нет	Нет
Create and manage global groups (создание и управление глобальными группами)	Есть	Нет	Есть	Нет	Нет	Нет	Нет	Нет

Встроенные права	Administrators	Server Operators	Account Operators	Print Operators	Backup Operators	Everyone	Users	Guests
Create and manage local groups (создание и управление локальными группами)	Есть	Нет	Есть	Нет	Нет	Нет	Нет	Нет
Assign user rights (назначение прав для пользователей)	Есть	Нет	Нет	Нет	Нет	Нет	Нет	Нет
Manage auditing of system events (управление аудитом системных событий)	Есть	Нет	Нет	Нет	Нет	Нет	Нет	Нет
Lock the server (блокирование сервера)	Есть	Есть	Нет	Нет	Нет	Есть	Нет	Нет
Override the lock of the server (преодоление блокировки сервера)	Есть	Есть	Нет	Нет	Нет	Нет	Нет	Нет
Format server's hard disk (форматирование жесткого диска сервера)	Есть	Есть	Нет	Нет	Нет	Нет	Нет	Нет
Create common groups (создание общих групп)	Есть	Есть	Нет	Нет	Нет	Нет	Нет	Нет
Keep local profile (хранение локального профиля)	Есть	Есть	Есть	Есть	Есть	Нет	Нет	Нет
Share and stop sharing directories (разделение и прекращение разделения каталогов)	Есть	Есть	Нет	Нет	Нет	Нет	Нет	Нет
Share and stop sharing printers (разделение и прекращение разделения принтеров)	Есть	Есть	Нет	Есть	Нет	Нет	Нет	Нет

При создании новых групп администратор может наделить их любым изменяемым правом, но распоряжаться встроенными правами он не может — они являются неотъемлемыми атрибутами встроенных и только встроенных групп.

Выводы

- Основными задачами подсистемы ввода-вывода являются:
  - организация параллельной работы процессора и устройств ввода-вывода при обеспечении приемлемого уровня реакции каждого драйвера и минимизации общей загрузки процессора;

- согласование скоростей работы процессора, оперативной памяти и устройств ввода-вывода, для чего применяется буферизация данных в оперативной памяти, на диске и в памяти контроллера;
- разделение устройств ввода-вывода между процессами;
- обеспечение удобного логического интерфейса к устройствам ввода-вывода.

- Подсистема ввода-вывода обычно имеет ярко выраженную многослойную структуру, которая помогает объединить большое количество разнотипных драйверов в систему с общим интерфейсом.
- Драйверы делятся на низкоуровневые, непосредственно управляющие работой контроллеров внешних устройств, и высокоуровневые, обеспечивающие логический интерфейс к устройствам, например драйверы файловых систем.
- Для координации работы драйверов в подсистеме ввода-вывода может выделяться особый модуль, называемый менеджером ввода-вывода.
- Аппаратные драйверы делятся на блок-ориентированные, обеспечивающие доступ к устройствам с поблочной непосредственной адресацией, и байт-ориентированные, управляющие устройствами, поддерживающими побайтный неадресуемый обмен.
- Файл — это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные, а также собственно хранимые в этой области данные и набор атрибутов, позволяющие ОС манипулировать этими данными.
- Файловая система представляет собой комплекс системных программных средств, реализующих различные операции с файлами, такие как создание, уничтожение, чтение, запись, именование и поиск файлов. Под файловой системой понимают также набор всех файлов и служебных структур данных, хранящихся на внешнем устройстве.
- Помимо типовых файлов, ОС обычно поддерживает такие типы файлов, как каталоги, символьные связи, именованные конвейеры и специальные файлы.
- Специальный файл является универсальной моделью устройства ввода-вывода, представляя его для остальной части операционной системы и прикладных процессов в виде неструктурированного набора байтов, то есть в виде обычного файла.
- Современные файловые системы имеют иерархическую структуру, упрощающую именование файлов и их поиск.
- Физическая организация файловой системы подразумевает способы размещения и адресации отдельных частей файлов в разделах и секторах дисковой памяти, а также способы организации служебной информации, описывающей размещение файлов и их атрибуты.

- Для синхронизации совместно работающих процессов, пытающихся использовать один и тот же файл одновременно, в файловых системах реализуется механизм блокировки файлов и отдельных записей в файлах.
- Механизм контроля доступа к файлам позволяет администраторам многопользовательских ОС задавать для отдельных пользователей и групп пользователей набор операций, которые им разрешается выполнять над отдельным файлом или группой файлов, объединенных в каталог.
- Управление доступом в ОС осуществляется на основе одного из двух базовых подходов: избирательного доступа, когда владелец файла самостоятельно может определить права доступа к файлу, и мандатного доступа, при котором права доступа определяются членством пользователя в определенной группе, и пользователь не может их произвольно изменять или делегировать.
- Права доступа к файлу могут присваиваться явно, а могут наследоваться у родительского каталога, что сокращает количество ручных операций и упрощает логику доступа.

Задачи и упражнения

1. За счет каких устройств удается распараллелить ввод-вывод даже в однопроцессорных системах?
2. Какие функции выполняет менеджер ввода-вывода?
3. Какие из следующих утверждений правильны:
   1) драйвер выполняет низкоуровневые функции по управлению устройством ввода-вывода;
   2) драйвер выполняет функции управления файловой системой;
   3) все функции драйвера вызываются по прерываниям;
   4) драйвер является частью подсистемы ввода-вывода;
   5) драйвер организует взаимодействие модулей ядра операционной системы;
   6) драйвер работает в привилегированном режиме.
4. Какие два типа ресурсов, связанных с диском, требуется выделить процессу, чтобы он выполнил запись данных на диск?
5. Каким из двух типов драйверов — блок-ориентированным или байт-ориентированным — обслуживается диск?
6. С какой целью в некоторых файловых системах характеристики файла отделяются от его имени?
7. Какие программные компоненты поддерживают структуру файла в тех ОС, где файл представлен последовательностью байтов?
8. С какого каталога начинается «раскрутка» полного имени файла?
9. Операционная система выделяет файлам пространство на диске:

1) секторами;
2) дорожками;
3) кластерами;
4) цилиндрами.

10. Выберите размер кластера для файловой системы FAT16, устанавливаемой в разделе, который разделен на секторы размером 512 байт и имеет общий объем 272 Мбайт. Оцените, сколько в этом случае кластеров будет содержать область данных, а также какой размер необходимо отвести таблице FAT. Учтите, что размер кластера должен быть равен степени двойки. Кроме того, примите во внимание, что стандартным размером корневого каталога для жестких дисков является размер в 32 сектора.
11. При каких условиях можно автоматически гарантированно восстановить в файловой системе FAT удаленный файл?
12. Сформулируйте основную цель введения в ОС системного вызова `open`?
13. В какой из типов систем управления доступом — избирательной или мандатной — пользователю предоставляется большая свобода действий?
14. Какой смысл имеет операция «выполнить каталог» в ОС Unix?
15. С помощью какого механизма пользовательский процесс может запускать на выполнение привилегированные утилиты операционной системы Unix?
16. Чем отличается разрешение *Full Control* для файлов от разрешения *Change* в ОС семейства Windows NT?
17. Какие действия по отношению к файлу *A* разрешены пользователю ОС Windows NT, если он лично имеет разрешение *Change*, а для группы, в которую он входит, задано разрешение *No Access*?

Глава 8

Дополнительные возможности файловых систем

Особая роль файловой системы, связанная с долговременным хранением информации, в том числе критически важных программ и данных пользователей и самой ОС, порождает повышенные требования к ее надежности и отказоустойчивости. Эти важные свойства обеспечиваются за счет применения восстанавливаемых файловых систем и отказоустойчивых дисковых массивов.

Модели файла и файловых операций, применяемые первоначально к хранимым на дисках данным, оказались удобным средством работы с данными любой природы, поэтому со временем они нашли применение и в других областях, таких как управление устройствами ввода-вывода и обмен данными между процессами. В то же время и на классическую файловую систему оказало влияние развитие других подсистем ОС, в частности подсистемы управления памятью, благодаря которой можно выполнять отображение файлов на оперативную память и работать с дисковыми данными как с обычными переменными.

Специальные файлы и аппаратные драйверы

Специальные файлы как универсальный интерфейс

Понятие «специальный файл» появилось в операционной системе Unix. **Специальный файл**, называемый также **виртуальным**, связан с некоторым устройством ввода-вывода и представляет его для остальной части операционной системы и прикладных процессов в виде неструктурированного набора байтов, то есть в виде обычного файла. Однако в отличие от обычного файла, специальный

файл не хранит статичные данные, а является интерфейсом к одному из аппаратных драйверов ОС.

Использование специальных файлов во многих случаях существенно упрощает программирование операций с внешними устройствами. Со специальным файлом можно работать так же, как с обычным, то есть открывать, считывать из него или же записывать в него определенное количество байтов, а после завершения операции закрывать. Для этого используются привычные многим программистам системные вызовы, применяемые для работы с обычными файлами: `open`, `create`, `read`, `write` и `close`. Помимо этих вызовов, имеется несколько системных вызовов, используемых только при работе со специальными файлами, например вызов `ioctl`, с помощью которого можно передать команду контроллеру устройства. Для того чтобы вывести на алфавитно-цифровой терминал, с которым связан специальный файл **/dev/tty3**, сообщение «Hello, friends!», достаточно открыть этот файл с помощью системного вызова `open`:

```
fd = open ("/dev/tty3", 2)
```

Затем можно вывести сообщение с помощью системного вызова `write`:

```
write(fd, "Hello, friends! ", 15)
```

Для устройств прямого доступа имеет смысл также указатель текущего положения в файле, которым можно управлять с помощью системного вызова `lseek`.

Очевидно, что представление устройства в виде файла и использование для управления устройством файловых системных вызовов позволяет выполнять только простые операции управления, которые сводятся к передаче в устройство последовательности байтов. Для некоторых устройств такие операции вполне адекватны — в основном это устройства, отображающие строки символов (алфавитно-цифровые терминалы, алфавитно-цифровые принтеры) или принимающие от пользователя строки символов (клавиатура). Форматирование ввода-вывода в устройствах этого класса осуществляется с помощью служебных символов начала кодовой таблицы и их последовательностей, например для перевода строки и возврата каретки принтера или терминала достаточно к последовательности символов текста добавить восьмеричные коды <12> <15>.

Для устройств с более сложной организацией информации, например графических дисплеев, от управляющего интерфейса требуется поддержка более сложных операций, таких как заполнение цветом области или вывод на экран основных графических примитивов, и аппаратные драйверы таких устройств их действительно выполняют. Тем не менее файловый интерфейс, оперирующий только неструктурированным потоком байтов, оказывается полезным и для устройств со сложной организацией информации. Такой интерфейс в силу своей простоты и универсальности дает возможность строить поверх него другой более сложный интерфейс с произвольной организацией.

Файловый интерфейс доступен пользователю, поэтому прикладной программист может воспользоваться им для создания собственного интерфейса к какому-либо устройству, обходя слои высокоуровневых драйверов, лежащие над аппаратным драйвером данного устройства. Например, если прикладного

программиста по какой-то причине не устраивают файловые системы, поддерживаемые некоторой операционной системой, то он может обратиться к диску как к устройству с помощью интерфейса специального файла, который будет вызывать аппаратный драйвер диска, поддерживающий модель диска в виде последовательности байтов (рис. 8.1). С помощью такого аппаратного драйвера прикладной программист может организовать данные в каком-либо разделе диска оригинальным способом, соответствующим его потребностям. При этом ему не нужно разрабатывать высокоуровневый драйвер для собственной файловой системы, что является более сложной задачей по сравнению с разработкой прикладной программы.

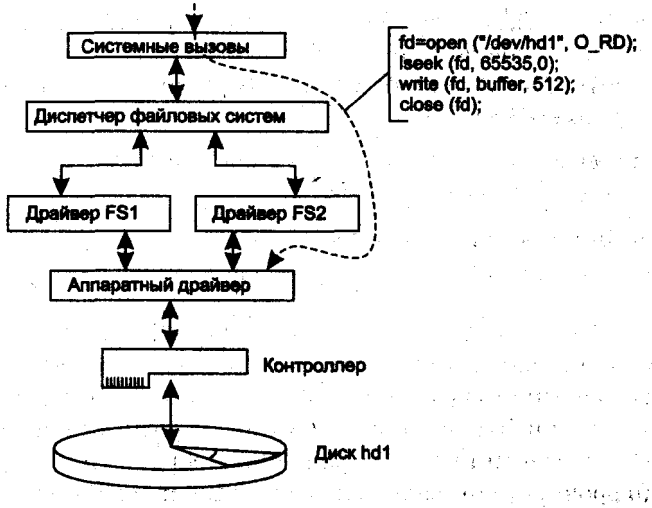


Рис. 8.1. Работа с диском как со специальным файлом

В Unix специальные файлы традиционно помещаются в каталог /dev, хотя ничто не мешает созданию их в любом каталоге файловой системы. При появлении нового устройства и, соответственно, нового драйвера администратор системы может создать новую запись с помощью команды `mknod`. Например, следующая команда создает блок-ориентированный специальный файл для представления третьего раздела на втором диске четвертого SCSI-контроллера:

```
mknod    /dev/dsk/scsi b 32 33
```

Связь специального файла с драйвером устанавливается за счет информации, находящейся в индексном дескрипторе специального файла.

Во-первых, в индексном дескрипторе хранится признак того, что файл является специальным, причем этот признак позволяет различить класс соответствующего устройству драйвера, то есть определяет, является ли драйвер байт-ориентированным или блок-ориентированным.

Во-вторых, в индексном дескрипторе хранится адресная информация, позволяющая выбрать нужные драйвер и устройство. Эта информация заменяет

стандартную адресную информацию обычного файла, состоящую из номеров блоков файла на диске.

Адресная информация специального файла состоит из двух элементов:

- *major* — номер драйвера;
- *minor* — номер устройства.

Номер драйвера определяет драйвер, обслуживающий данный специальный файл, а номер устройства передается драйверу в качестве параметра вызова и указывает ему на одно из нескольких однотипных устройств, которыми драйвер может управлять. Например, для дисковых драйверов номер устройства задает не только диск, но и раздел на диске.

В приведенном ранее примере команды создания специального файла `/dev/dsk/scsi` аргумент `b` определяет необходимость создания специального файла для блок-ориентированного драйвера, аргумент `32` задает номер драйвера, который будет вызываться при открытии устройства `/dev/dsk/scsi`, а аргумент `33` декодируется самим драйвером (в нем закодированы данные о том, что нужно управлять третьим разделом на втором диске четвертого SCSI-контроллера).

ОС Unix использует для хранения информации об установленных аппаратных драйверах две системные таблицы:

- *bdevsw* — таблица блок-ориентированных драйверов;
- *cdevsw* — таблица байт-ориентированных драйверов.

Номер драйвера (*major*) является индексом соответствующей таблицы. При открытии специального файла операционная система обнаруживает, что она имеет дело со специальным файлом только после того, как прочитает с диска или найдет в системном буфере его индексный дескриптор. При этом она узнает, является ли вызываемый драйвер блок- или байт-ориентированным, после чего использует номер драйвера для обращения к определенной строке одной из двух таблиц: *bdevsw* или *cdevsw* (рис. 8.2).

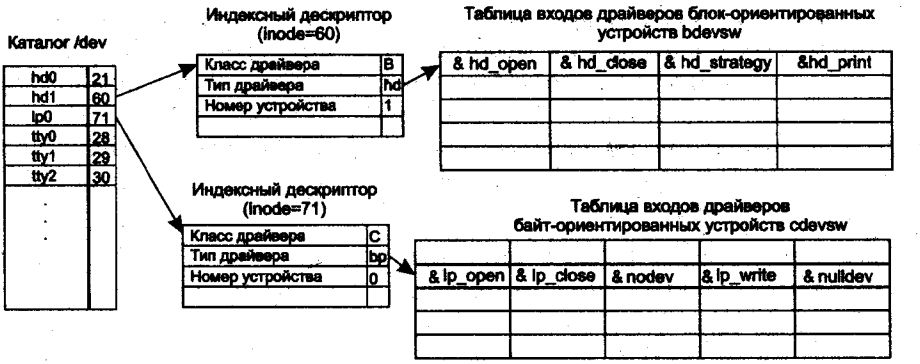


Рис. 8.2. Организация связи ядра Unix с драйверами

Таблицы *bdevsw* и *cdevsw* содержат адреса программных секций драйверов, причем одна строка таблицы описывает один драйвер. Такая организация логи-

ческой связи между ядром Unix и драйверами позволяет легко настраивать систему на новую конфигурацию внешних устройств путем модификации таблиц *bdevsw* и *cdevsw*.

Концепция специальных файлов ОС Unix была реализована во многих операционных системах, хотя для связи с драйверами в них часто используются механизмы, отличные от описанного. Так, в ОС семейства Windows NT для связи виртуальных устройств (аналогов специальных файлов) с драйверами задействуется механизм объектов. При этом в объектах-устройствах имеются ссылки на объекты-драйверы, за счет чего при открытии виртуального устройства система находит нужный драйвер.

Структурирование аппаратных драйверов

Аппаратные драйверы можно назвать «истинными» драйверами, так как, в отличие от высокоуровневых драйверов, они выполняют все традиционные функции по управлению устройствами, включая обработку прерываний и непосредственное взаимодействие с устройствами ввода-вывода.

Более точно, аппаратный драйвер имеет дело не с устройством, а с его контроллером.

Контроллер, как правило, выполняет достаточно простые функции, например, преобразует поток битов в блоки данных и осуществляют контроль и исправление возникающих в процессе обмена данными ошибок. Каждый контроллер имеет несколько регистров, которые используются для взаимодействия с центральным процессором. Обычно у контроллера имеются регистры данных, через которые осуществляется обмен данными между драйвером и устройством, и управляющие регистры, в которые драйвер помещает команды. В некоторых типах компьютеров регистры являются частью физического адресного пространства, при этом в таких компьютерах отсутствуют специальные инструкции ввода-вывода — их функции выполняют инструкции обмена с памятью. В других компьютерах адреса регистров ввода-вывода, называемых часто портами, образуют собственное адресное пространство за счет введения специальных операций ввода-вывода (например, команд `IN` и `OUT` в процессорах Intel Pentium).

Внешнее устройство в общем случае состоит из механических и электронных компонентов. Обычно электронная часть устройства сосредоточивается в его контроллере, хотя это и не обязательно. Некоторые контроллеры могут управлять несколькими устройствами. Если интерфейс между контроллером и устройством стандартизован, то независимые производители могут выпускать совместимые со стандартом контроллеры и устройства.

Аппаратный драйвер выполняет ввод-вывод данных, записывая команды в регистры контроллера. Например, контроллер диска персонального компьютера принимает такие команды, как `READ`, `WRITE`, `SEEK`, `FORMAT` и т. д. Когда команда принята, процессор оставляет контроллер и занимается другой работой. При завершении команды контроллер генерирует запрос прерывания, чтобы передать управление процессором операционной системе, которая должна

проверить результаты операции. Процессор получает результаты и данные о статусе устройства, читая информацию из регистров контроллера.

Аппаратные драйверы могут в своей работе опираться на **микропрограммные драйверы** (firmware drivers), поставляемые производителем компьютера и находящиеся в постоянной памяти компьютера (в персональных компьютерах это программное обеспечение получило название BIOS — Basic Input-Output System). Микропрограммное обеспечение представляет собой самый нижний слой программного обеспечения компьютера, управляющий устройствами. Модули этого слоя выполняют функции транслирующих драйверов и конверторов, экранирующих специфические интерфейсы аппаратуры данной компьютерной системы от операционной системы и ее драйверов.

Драйвер выполняет *операцию* ввода-вывода, которая представляет собой обмен с устройством заданным количеством байтов по заданному адресу оперативной памяти (и адресу устройства ввода-вывода в том случае, когда оно является адресуемым). Примерами операций ввода-вывода могут служить чтение нескольких смежных секторов диска или печать на принтере нескольких строк документа. Операция задается одним системным вызовом ввода-вывода, например `read` или `write`.

Операция отрабатывается драйвером в общем случае за несколько *действий*. Так, при выводе документа на принтер драйвер сначала выполняет некоторые начальные действия, приводящие принтер в состояние готовности к печати, затем выводит в буфер принтера первую порцию данных и ждет сигнала прерывания, который свидетельствует об окончании контроллером принтера печати этой порции данных. После этого в буфер выводится вторая порция данных и т. д.

Так как большинство действий драйвер выполняет асинхронно по отношению к вызвавшему драйвер процессу, то драйверу запрещается изменять контекст текущего процесса (который в общем случае отличается от вызвавшего). Кроме того, драйвер не может запрашивать у ОС выделения дополнительных ресурсов или отказываться от уже имеющихся у текущего процесса — драйвер должен пользоваться теми системными ресурсами, которые выделяются непосредственно ему (а не процессу) на этапе загрузки в систему или старта очередной операции ввода-вывода. Соблюдение этих условий необходимо для корректного распределения ресурсов между процессами — каждый получает то, что запрашивал и что непосредственно ему выделила операционная система.

В подсистеме ввода-вывода каждой современной операционной системы существует *стандарт* на структуру драйверов. Несмотря на специфику управляемых устройств, в любом драйвере можно выделить некоторые общие части, выполняющие определенный набор действий, такие как запуск операции ввода-вывода, обработка прерывания от контроллера устройства и т. п. Рассмотрим принципы структуризации драйверов на примере операционных систем семейства Windows NT и Unix.

Структура драйвера ОС семейства Windows NT

Особенностью ОС семейства Windows NT является общая структура драйверов любого уровня и расширенное толкование самого понятия «драйвер». В ОС семейства Windows NT и аппаратный драйвер диска, и высокоуровневый драйвер файловой системы построены единообразно, поэтому другие модули ОС взаимодействуют с драйверами одним и тем же способом.

Драйвер ОС семейства Windows NT состоит из следующих (не обязательно всех) процедур:

- *Процедура инициализации драйвера.* Эта процедура выполняется при загрузке драйвера в подсистему ввода-вывода, при этом создаются системные объекты, которые позволяют менеджеру ввода-вывода найти нужный драйвер для управления определенным устройством или выполнения некоторых высокоуровневых функций с информацией, получаемой от устройства или передаваемой в устройство.
- *Набор диспетчерских процедур.* Эти процедуры составляют основу драйвера, так как именно они выполняют операции ввода-вывода, поддерживаемые данным драйвером, например чтение данных, запись данных, перемотка ленты и т. п.
- *Стартовая процедура* предназначена для приведения устройства в исходное состояние перед началом очередной операции. Выполняет «открытие» (open) устройства.
- *Процедура обслуживания прерываний* (ISR) включает наиболее важные действия, которые нужно выполнить при возникновении очередного аппаратного прерывания от контроллера устройства. Процедура обслуживания прерывания драйвера имеет достаточно высокий приоритет запроса прерывания IRQL (к примеру, он выше приоритета диспетчера потоков), поэтому в нее рекомендуется включать только те функции, которые требуют незамедлительной реакции, чтобы не задерживать надолго работу других модулей и процессов. При завершении работы ISR может поставить в очередь диспетчера прерываний DPC-процедуру драйвера.
- *Процедура, выполняемая посредством механизма отложенных вызовов (DPC).* Эта процедура также состоит из функций, которые нужно выполнить при возникновении прерывания от контроллера устройства, однако эти функции не требуют такой быстрой реакции, как ISR-функции. В результате DPC-процедура обслуживается с более низким значением приоритета IRQL, давая возможность ISR-процедурам и другим приоритетным запросам обслуживаться в первую очередь. Обычно большая часть действий драйвера по обработке прерывания включается в DPC-процедуру.
- *Процедура завершения операции* уведомляет менеджер ввода-вывода о том, что операция завершена и данные находятся в системной области памяти. Менеджер при этом может вызвать драйвер более высокого уровня для продолжения обработки данных или же вызывать APC-процедуру (см. раздел

«Процедуры обработки прерываний и текущий процесс» в главе 4) для копирования данных из системной области в область памяти пользовательского процесса.

- *Процедура отмены ввода-вывода.* Для разных стадий выполнения операции могут существовать разные процедуры отмены.
- *Процедура выгрузки драйвера* вызывается при динамической выгрузке драйвера из подсистемы ввода-вывода. Удаляет созданные для драйвера объекты и освобождает системную память.
- *Процедура регистрации ошибок.* При возникновении ошибки в процессе выполнения операции данная процедура уведомляет о ней менеджер ввода-вывода, который, в свою очередь, делает соответствующую запись в журнале регистрации.

Адреса всех перечисленных процедур представляют собой точки входа в драйвер, известные менеджеру ввода-вывода. Эти адреса хранятся в объекте, создаваемом для каждого драйвера ОС семейства Windows NT, и менеджер использует такие объекты для вызова той или иной функции драйвера. Процедура диспетчеризации используется как общая точка входа для нескольких процедур обмена данными (чтение, запись, управление и т. п.), набор которых изменяется от драйвера к драйверу и, следовательно, не может быть стандартизован.

Большое количество стандартизованных функций драйвера ОС семейства Windows NT обусловлено желанием разработчиков этой ОС иметь единую модель для драйверов всех типов, от сравнительно простого аппаратного драйвера COM-порта до весьма сложного драйвера файловой системы NTFS. В результате некоторые функции для некоторого драйвера могут оказаться невостребованными. Например, для высокоуровневых драйверов не нужна процедура обслуживания прерываний (ISR), так как прерывания от устройства обрабатывает соответствующий низкоуровневый драйвер, который затем вызывает высокоуровневый драйвер с помощью менеджера ввода-вывода, не используя механизм прерываний.

Рассмотрим особенности вызова функций аппаратного драйвера ОС семейства Windows NT на примере выполнения *операции чтения с диска* (рис. 8.3). Диск рассматривается в этой операции как виртуальное устройство, следовательно, слой драйверов файловых систем в выполнении операции не участвует.

Пусть в некоторый момент времени выполнения в пользовательской фазе процесс *A* запрашивает с помощью соответствующего системного вызова чтение некоторого количества блоков диска, начиная с блока определенного номера. Процесс *A* при этом переходит в состояние ожидания завершения запрошенной операции, а планировщик/диспетчер Sh активизирует ожидавший выполнения процесс *B*.

При выполнении системного вызова управление с помощью менеджера ввода-вывода передается стартовой функции драйвера диска DD, которая проверяет, открыт ли виртуальный файл диска и готов ли контроллер диска к выполне-

нию операции обмена данными. После возврата управления от стартовой процедуры менеджер вызывает функцию диспетчеризации драйвера, которой передается пакет запроса ввода-вывода IRP, содержащий параметры операции — начальный адрес и количество блоков диска. В результате функция диспетчеризации драйвера вызывает внутреннюю функцию чтения данных с диска, которая передает контроллеру диска запрос на чтение первой порции запрошенных данных. На рисунке выполнение всех перечисленных функций показано как один этап работы драйвера DD. Тот факт, что драйвер DD выполняет работу для процесса *A*, отмечен на рисунке нижним индексом, то есть как DD_A.

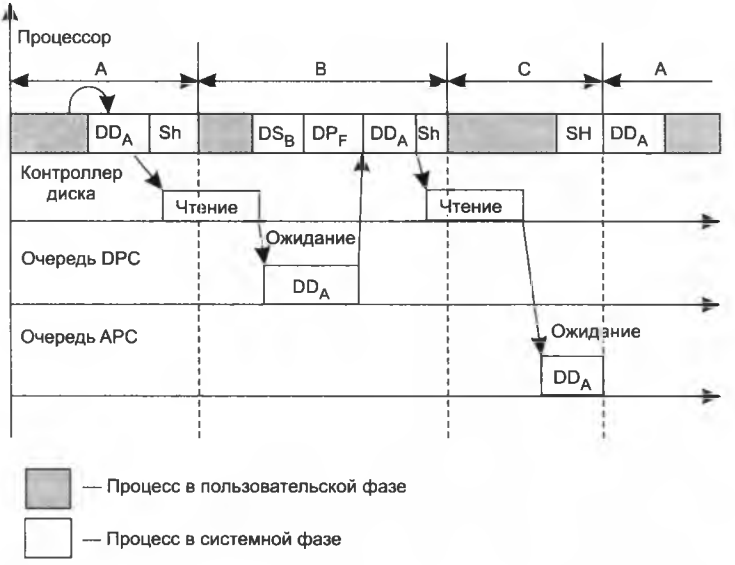


Рис. 8.3. Работа аппаратного драйвера ОС семейства Windows NT

После завершения чтения порции данных контроллер генерирует аппаратный запрос прерывания, который вызывает процедуру обработки прерываний драйвера диска ISR, имеющую высокий уровень IRQL. После короткого периода выполнения самых необходимых действий с регистрами контроллера (этот период для упрощения рисунка не показан) эта процедура делает запрос на выполнение менее срочной DPC-процедуры драйвера, которая должна выполнить передачу имеющейся у контроллера порции данных в системную область. Запрос на выполнение DPC-процедуры драйвера DD_A некоторое время стоит в очереди уровня DPC, так как в это время в процессоре выполняются более приоритетные ISR-процедуры DS_B (драйвера стриммера для процесса *B*) и DP_F (драйвера принтера для процесса *F*). После завершения этих процедур начинается выполнение DPC-процедуры драйвера DD_A, при этом текущим для ОС процессом является процесс *B*, сменивший процесс *A* и прерванный на время ISR-процедурами. Однако на выполнение DPC-процедуры драйвера диска это

обстоятельство не оказывает никакого влияния, так как данные перемещаются в системную область, общую для всех процессов.

Помимо перемещения данных DPC-процедура драйвера выдает контроллеру диска указание о чтении второй и последней для операции порции данных (если контроллер использует режим прямого доступа к памяти и самостоятельно перемещает данные из своего буфера в системный буфер, то запуск нового действия является единственной обязанностью DPC-процедуры). Контроллер выполняет чтение и выдает новый запрос прерывания, который снова вызывает процедуру обработки прерываний драйвера диска. Данная процедура ставит в IRQL-очереди две процедуры: DPC-процедуру, которая, как и в предыдущем цикле чтения, должна переписать данные из буфера контроллера в системный буфер, и APC-процедуру, которая должна переписать все полученные данные из системного буфера в заданную пользовательскую область памяти процесса *A*.

DPC-процедура вызывается раньше, так как имеет более высокий приоритет в очереди диспетчера прерываний. APC-процедура ждет дольше, так как она имеет более низкий приоритет и, кроме того, обязана ждать до тех пор, пока текущим процессом не станет процесс *A*. DPC-процедура после выполнения фиксирует в операционной системе событие — завершение операции ввода-вывода. При наступлении события вызывается планировщик потоков, который переводит процесс *A* в состояние готовности (но не ставит его на выполнение, так как текущий процесс *C* еще не исчерпал своего кванта времени). И только после того как планировщик снимает процесс *C* с выполнения и делает текущим процесс *A*, вызывается APC-процедура, которая вытесняет пользовательский код процесса *A*, имеющий низший приоритет IRQL. APC-процедура переписывает считанные с диска данные из системного буфера в область данных процесса *A*. Для доступа к системному буферу APC-процедура должна иметь нужный уровень привилегий. После завершения работы APC-процедуры управление возвращается пользовательскому коду процесса *A*, который обрабатывает запрошенные у диска данные.

Структура драйвера Unix

В ОС Unix вместо одной общей структуры драйвера существуют две стандартные структуры, одна — для блок-ориентированных драйверов, а другая — для байт-ориентированных. По этой причине в Unix используются две таблицы, *bdevsw* и *cdevsw*, хранящие точки входа в функции драйверов. Каждая из таблиц имеет свою структуру, соответствующую стандартным функциям блок- и байт-ориентированных драйверов.

Блок-ориентированные драйверы

Драйвер блок-ориентированного устройства состоит из следующих функций:

- `open` — выполняет процедуру логического открытия устройства;
- `close` — выполняет процедуру логического закрытия устройства;
- `strategy` — читает или записывает блок;

- `print` — выводит сообщение об ошибке;
- `size` — возвращает размер раздела, который представляет данное устройство.

Указатели на эти функции (то есть их адреса) составляют строку в таблице *bdevsw*, описывающую один драйвер системы. Ядро Unix вызывает нужную функцию драйвера, передавая ей параметры, необходимые для работы. Например, при вызове функции `open` ей передается номер устройства (*minor*), режим открытия (для чтения, для записи, для чтения и записи и т. д.), а также указатель на идентификаторы безопасности процесса, открывающего файл.

Процедуры обработки прерываний драйвера в таблице *bdevsw* не указываются, их адреса помещаются в специальную системную структуру — таблицу прерываний. В Unix все обработчики прерываний, в том числе и обработчики прерываний аппаратных драйверов, состоят из двух процедур, называемых соответственно *top_half* — верхняя часть обработчика прерываний и *bottom_half* — нижняя часть обработчика прерываний. Верхняя часть обработчика прерываний соответствует по назначению ISR-процедуре драйвера ОС семейства Windows NT — она вызывается при возникновении аппаратного запроса прерывания от устройства. В обязанности верхней части входит быстрая реакция на событие в устройстве, вызвавшее генерирование сигнала прерывания. При обработке верхних половин все прерывания с более низкими приоритетами блокируются аппаратно за счет управления контроллером прерываний (или аналогичным по назначению блоком компьютера). Верхняя половина отвечает также за постановку в очередь на выполнение нижней половины обработчика прерываний драйвера, который выполняет менее срочную и более трудоемкую работу.

Нижние половины драйверов выполняются с низким уровнем приоритета, так что любые запросы прерываний устройств могут прервать их обработку. Нижние половины обработчиков прерываний драйверов Unix по назначению соответствуют DPC-процедурам драйверов ОС семейства Windows NT. Часто единственной обязанностью верхней половины обработчика прерываний является постановка в очередь нижней половины для последующего выполнения.

Примером разделения функций между верхней и нижней половинами является обработчик прерываний от таймера. Верхняя часть, вызываемая 100 раз в секунду, наращивает переменную, хранящую количество тактов системных часов с момента последней загрузки системы, а также две переменные, подсчитывающие, сколько тактов прошло с момента последнего вызова нижней половины и сколько из них пришлось на период работы в режиме системы. Затем верхняя половина ставит в очередь диспетчера прерываний нижнюю половину и завершает свою работу. Нижняя половина обработчика прерываний таймера на основании данных о тактах, собранных верхней половиной, занимается вычислением статистики, в том числе рассчитывает среднюю загрузку системы в пользовательском и системном режимах. Кроме того, нижняя половина обновляет глобальную переменную системного времени и уменьшает оставшиеся значения кванта времени текущего процесса. Затем нижняя половина просмат-

ривает очередь процедур, ожидающих своего вызова по времени, в число которых входит и планировщик процессов.

Функция стратегии драйвера `strategy` выполняет чтение и запись блока данных на основании информации в буфере — особой структуре ядра с именем *buf*, управляющей обменом данных с диском. Функция `strategy` поддерживает обмен только с системной памятью, так как блок-ориентированный драйвер непосредственно не взаимодействует с пользовательским процессом. Между ним и пользовательским процессом всегда работает промежуточный программный слой или слои — либо слой дискового кэша вместе со слоем файловой системы, либо слой байт-ориентированного драйвера диска, с помощью которого пользовательский процесс может открыть специальный файл, соответствующий диску.

В число наиболее важных элементов структуры *buf* входят следующие:

- *b_flags* — набор битов, в котором задаются тип операции (чтение или запись), синхронный или асинхронный режим операции (при записи), признак активности операции с буфером, признак завершения операции, признак ожидания буфера процессом и некоторые другие;
- *b_forw*, *b_back* — указатели на последующий и предыдущий буферы в списке активных (используемых) буферов;
- *av_forw*, *av_back* — указатели на последующий и предыдущий буферы в списке свободных буферов;
- *b_dev* — номер драйвера (*major*) и номер устройства (*minor*) из индексного дескриптора специального устройства, для которого выполняется операция обмена данными;
- *b_bcount* — количество байтов, которые нужно передать;
- *b_addr* — адрес буфера памяти, куда нужно записать или откуда нужно прочитать данные;
- *b_blkno* — номер блока в разделе диска;
- *b_bufsize* — размер блока (в ранних версиях Unix использовался только одни размер блока — 512 байт, в версиях, основанных на коде System V Release 4, можно работать с блоками разного размера);
- *b_iodone* — указатель на функцию, которая вызывается при завершении операции ввода-вывода.

Функция `strategy` при вызове получает указатель на структуру *buf*, описывающую требуемую операцию. На рис. 8.4 приведен пример блок-схемы двух функций драйвера диска — стратегии (`hd_strategy`) и нижней половины обработчика прерываний (`hd_bottom`). Функция `hd_strategy` преобразует логический номер блока в номера цилиндра, головки и сектора и помещает эту информацию в *заголовок запроса операции* для передачи ее контроллеру диска. В заголовок запроса помещается также другая информация, необходимая для работы контроллера, — это операция чтения или записи, адрес системной памяти, куда нужно поместить прочитанную информацию или откуда контролеру

нужно считать записываемые данные. Драйвер ведет две очереди для передачи запросов на выполнение операций чтения и записи контроллеру диска: рабочую очередь, в которой находятся обрабатываемые контроллером запросы, и очередь приостановленных запросов, куда помещаются новые запросы в том случае, если рабочая очередь заполнена — а ее размер зависит от возможностей контроллера по параллельной обработке запросов.

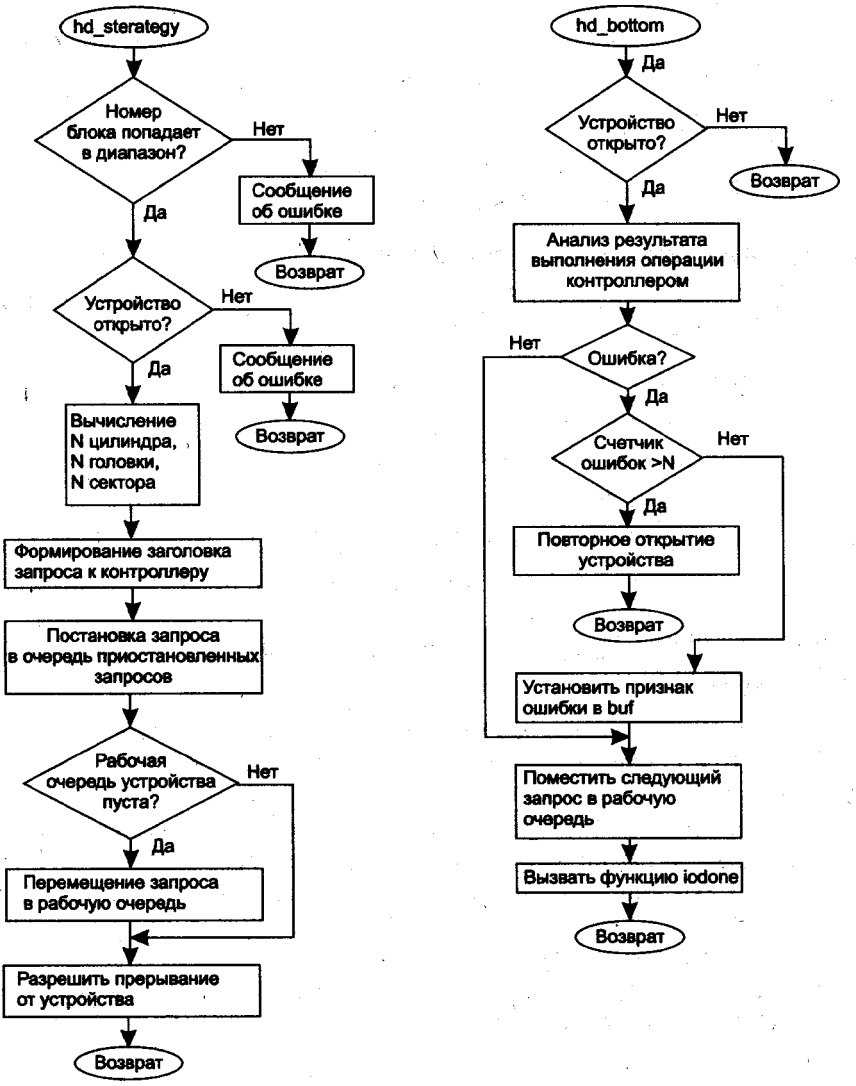


Рис. 8.4. Блок-схема драйвера диска

После помещения нового запроса в одну из очередей функция стратегии разрешает прерывания от данного устройства и завершает свою работу. Всю

дальнейшую работу по обслуживанию поставленных в очереди запросов выполняют контроллер и обработчики прерываний. После выполнения запроса, когда данные либо записаны в системную память (чтение), либо переписаны из системной памяти в блок диска (запись), контроллер генерирует сигнал прерывания. По этому сигналу вызывается верхняя часть обработчика прерываний дискового драйвера (на рисунке не показана), которая просто ставит в очередь диспетчера прерываний нижнюю часть обработчика прерываний диска `hd_bottom`. Эта процедура считывает данные из регистра управления контроллера для того, чтобы определить, корректно ли завершилась запрошенная операция. Если признак ошибки в регистре не установлен, то в рабочую очередь контроллера помещается следующий заголовок запроса из очереди приостановленных запросов, а при завершении операции вызывается функция `iodone`, указатель на которую имеется в буфере *buf* операции.

Байт-ориентированные драйверы

Драйвер байт-ориентированного устройства состоит из следующих стандартных функций:

- `open` — открывает устройство;
- `close` — закрывает устройство;
- `read` — читает данные из устройства;
- `write` — записывает данные в устройство;
- `ioctl` — управляет вводом-выводом;
- `poll` — опрашивает устройство для выяснения, не произошло ли некоторое событие;
- `mmap`, `segmap` — используются при отображении файла-устройства на виртуальную память.

Функции чтения и записи данных выполняют обмен заданной последовательности байтов из буфера в области пользователя с контроллером символьного устройства.

Функция управления `ioctl` обеспечивает интерфейс к драйверу устройства, который выходит за рамки возможностей функций `read` и `write`. С помощью функции `ioctl` обычно устанавливается режим работы устройства, например задаются параметры COM-порта, такие как разрядность символов, количество стоповых битов, режим проверки четности и т. п.

Функции, используемые для отображения специального файла на виртуальную память, рассматриваются далее в разделе «Отображаемые на память файлы».

Если драйвер не поддерживает какую-либо из стандартных функций, то в таблицу *bdevsw* помещается указатель на специальную функцию `nodev` ядра. Например, драйвер принтера может не поддерживать функцию `read`. Функция `nodev` при вызове просто возвращает код ошибки `ENODEV` и на этом завершает свою работу. Для тех случаев, когда функция должна обязательно поддержи-

ваться (примерами таких функций являются функции `open` и `close`), но она не выполняет никакой полезной работы, в операционной системе имеется функция `nulldev`, которая похожа на функцию `nodev`, но в отличие от нее возвращает значение 0, во всех системных вызовах означающее успешное завершение.

Рисунок 8.5 иллюстрирует взаимодействие функции записи драйвера байт-ориентированного устройства с обработчиком прерываний. Функция записи осуществляет передачу данных из пользовательского буфера процесса, выдавшего запрос на обмен, в системный буфер, организованный в виде очереди байтов. Передача байтов идет до тех пор, пока системный буфер не заполнится до некоторого заранее определенного в драйвере уровня. Затем функция записи

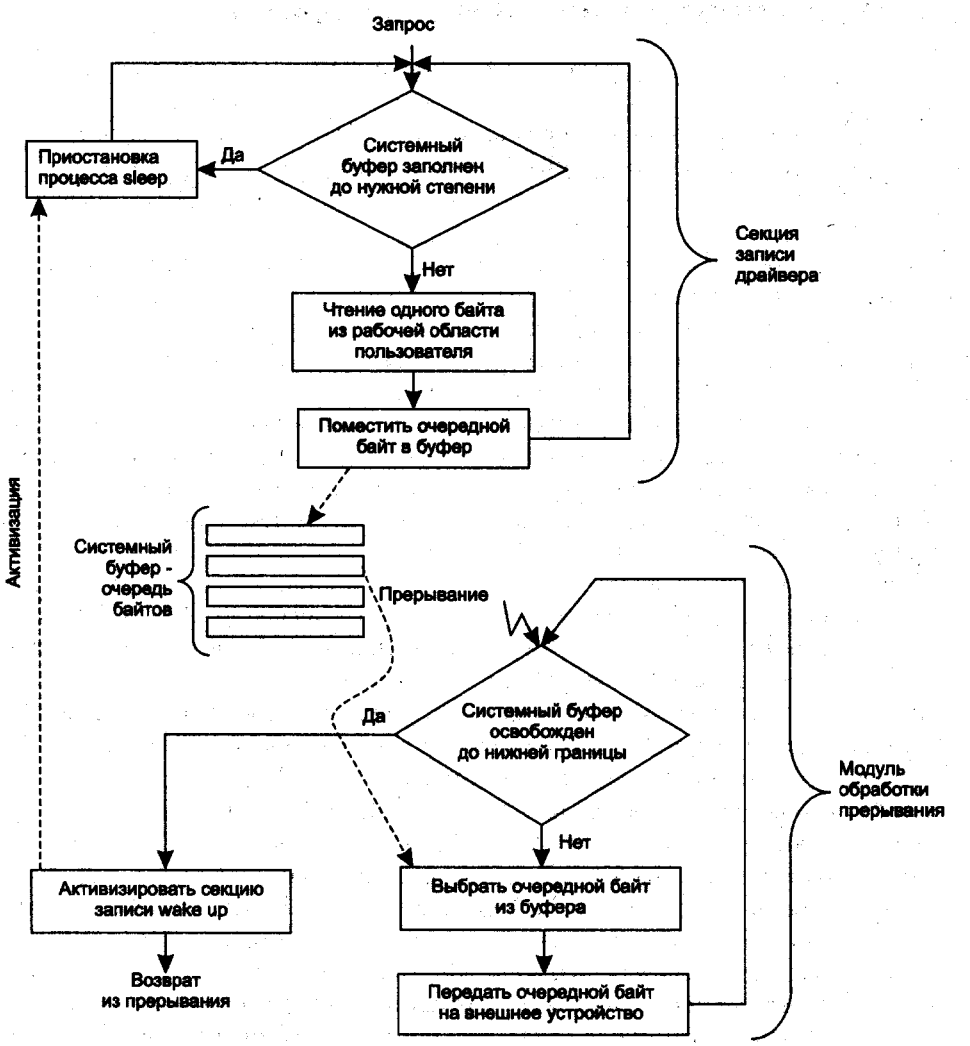


Рис. 8.5. Взаимодействие секции записи драйвера с модулем обработки прерывания

драйвера приостанавливается, выполнив системную функцию `sleep`, переводящую процесс, в рамках которого работает функция записи `write`, в состояние ожидания.

Если при очередном прерывании оказывается, что очередь байтов уменьшилась до определенной нижней границы, то обработчик прерываний активизирует секцию записи драйвера путем обращения к системной функции `wakeup` для перевода процесса в состояние готовности. Аналогично организована работа драйвера при чтении данных с устройства.

Отображаемые на память файлы

Отображение файла на виртуальное адресное пространство процесса позволяет упростить программирование. Благодаря такому отображению с данными файла можно работать с помощью адресных указателей как с обычными переменными программы без использования громоздких файловых функций `read`, `write` и `lseek`. При отображении файлов на память широко используются механизмы подсистемы виртуальной памяти.

Действительно, подсистема виртуальной памяти связывает некоторый сегмент виртуального адресного пространства процесса с некоторым файлом или частью файла. Так, кодовый сегмент и сегмент инициализированных данных всегда связаны с файлом, в котором находится исполняемый модуль приложения. Сегменты стека и неинициализированных данных связаны с выделенными им областями системного страничного файла. При обращении кода приложения к некоторой переменной сегмента данных подсистема виртуальной памяти читает с диска данные из блоков, соответствующих странице виртуального адресного пространства, содержащей эту переменную, и переносит данные в оперативную память, если на момент обращения эта страница там отсутствовала. В сущности, подсистема виртуальной памяти выполняет обмен данными с файлом по запросу, только этот запрос формулируется косвенно, а не путем явного описания области файла, с которой нужно выполнить обмен данными, как это происходит при выполнении операции `read` или `write`.

Механизм отображения файлов на память использует возможности системы виртуальной памяти для файлов, содержащих произвольные данные (а не только данные исполняемого модуля программы).

Отображение данных файла на память осуществляется с помощью системного вызова, который указывает, какую часть какого файла нужно отобразить, а также задает виртуальный адрес, с которого должен начинаться новый сегмент виртуальной памяти процесса. Подсистема управления виртуальной памятью создает по этому системному вызову новый сегмент процесса, в дескриптор которого помещает указатель на открытый отображаемый файл. При первом же обращении приложения по виртуальному адресу, принадлежащему новому сегменту, происходит страничный отказ, при обработке которого из отображаемого файла читается несколько блоков и данные из них помещаются в физическую страницу.

В Unix SystemV Release 4 отображение файла на память выполняется с помощью системного вызова mmap. Этот вызов имеет следующие аргументы:

- *addr* — виртуальный адрес начала сегмента, если он задается нулевым, то система сама выбирает подходящий адрес и возвращает его в качестве значения функции mmap;
- *len* — размер сегмента;
- *prot* — атрибуты защиты сегмента (только чтение, только запись и т. п.);
- *flags* — флаги, определяющие режим использования сегмента: разделяемый (shared) или закрытый (private);
- *fd* — дескриптор открытого файла, данные которого отображаются;
- *offset* — смещение в файле, с которого начинаются отображаемые данные.

Для сравнения рассмотрим две функции, которые выполняют одни и те же действия с файлом, но с помощью разных средств — функция ffile использует традиционные файловые операции, а функция fmap работает с отображенным на память файлом.

Пусть файл /data/base1.dat состоит из записей фиксированной длины, каждая из которых включает переменную, отражающую значение баланса предприятия (переменная balance) и признак типа баланса (переменная mode):

```
/* Структура записи */
struct record {
    unsigned long balance;
    unsigned short mode;
};

/* Функция ffile использует для работы файловые операции */
finction ffile ()
{
    int fd;
struct record r;
/* Открытие файла, "пролистывание" первых 24 записей и чтение 25-й */
fd = open("/data/base1.dat", O_RDWR, 0);
lseek (fd, 24 * sizeof(struct record), SEEK_SET);
read(fd, &r, sizeof(struct record));
/* Модификация данных 25-й записи */
r.balance = 1000000;
r.mode = 3;
/* Возвращение указателя на начало 25-й записи, так как в результате
чтения он переместился на начало 26-й */
lseek (fd, 24 * sizeof(struct record), SEEK_SET);
/* Запись модифицированных данных и закрытие файла*/
write (fd, &r, sizeof(struct record));
close (fd);
}

/* Функция fmap отображает файл на память */
function fmap();
{
    int fd;
    struct record *rp; /* rp является адресом записи recored */
```

продолжение

```
    int len = 1000 * sizeof(struct record); /* len хранит длину файла,
состоящего из 1000 записей */
/* Открытие файла и отображение его на память, которая доступна для
чтения и записи, причем доступна в режиме разделения другим процессам */
rp = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
/* Модификация 25-й записи простым присваиванием значений переменным */
    rp[24].balance = 1000000;
    rp[24].mode = 3;
    close(fd);
}
```

В некоторых операционных системах, например в версиях Unix, основанных на коде System V Release 4, можно отобразить на память не только обычные файлы, но и некоторые другие типы файлов, например специальные файлы. Отображение на память блок-ориентированного специального файла, то есть раздела или части раздела диска, позволяет легко получить доступ к любой области диска, рассматриваемого как последовательность байтов. При отображении байт-ориентированных устройств на оперативную память отображается внутренняя память контроллера устройства, например память сетевого адаптера Ethernet.

В общем случае не все типы файлов можно отобразить на память, например, в Unix SVR4 нельзя отображать каталоги и символьные связи.

Отображение файла эффективней непосредственного использования файловых операций в нескольких отношениях.

- Исключаются операции копирования данных из системной памяти в пользовательскую. При выполнении файловых операций `read` и `write` данные сначала попадают в системный буфер, а затем копируются в пользовательскую память, а при отображении они сразу копируются в страницы пользовательской памяти.
- Программист применяет более удобный интерфейс с адресными указателями.
- Уменьшается количество системных вызовов, так как при использовании файловых операций каждая операция обмена с файлом связана с выполнением системного вызова, а при отображении выполняется один системный вызов для всех последующих операций доступа к данным файла.
- Обеспечивается возможность обмена данными между процессами с помощью разделяемых сегментов памяти, соответствующих одному отображенному файлу, вместо многочисленных операций обмена данными между диском и памятью.

К недостаткам техники отображения файлов на память можно отнести то, что размер отображенного файла нельзя увеличить, в то время как файловые операции допускают это путем записи данных в конец файла.

Механизм отображения файлов на память используется большинством современных операционных систем.

Дисковый кэш

Во многих операционных системах запросы к блок-ориентированным внешним устройствам с прямым доступом (типичными и наиболее распространенными представителями которых являются диски) перехватываются промежуточным программным слоем — подсистемой буферизации, называемой также **дисковым кэшем**. Дисковый кэш располагается между слоем драйверов файловых систем и блок-ориентированными драйверами. При поступлении запроса на чтение некоторого блока диспетчер дискового кэша просматривает свой буферный пул, находящийся в системной области оперативной памяти, и если требуемый блок имеется в кэше, то диспетчер копирует его в буфер запрашивающего процесса. Операция ввода-вывода считается выполненной, хотя физического обмена с устройством не происходило, при этом *выигрыш во времени доступа к файлу* очевиден.

При записи данные также попадают сначала в буфер и только потом при необходимости освободить место в буферном пуле или же по требованию приложения они действительно переписываются на диск. Операция же записи считается завершенной при завершении обмена с кэшем, а не с диском. Данные блоков диска за время пребывания в кэше могут быть многократно прочитаны прикладными процессами и модулями ОС без выполнения дисковых операций ввода-вывода, что также существенно *повышает производительность* операционной системы.

Дисковый кэш обычно занимает достаточно большую часть оперативной системной памяти, чтобы максимально повысить вероятность кэш-попаданий при выполнении дисковых операций. В таких специализированных операционных системах, как NetWare 3.x и 4.x, дисковый кэш занимает большую часть физической памяти компьютера, что во многом и обеспечивает высокую скорость файлового обслуживания, в ходе которого значительная часть запрошенных по сети данных обнаруживается в кэше. Доля оперативной памяти, отводимой под дисковый кэш, зависит от специфики функций, выполняемых компьютером, например, сервер приложений выделяет под дисковый кэш меньше памяти, чем файловый сервер, чтобы обеспечить более высокую скорость работы приложений за счет предоставления им большего объема оперативной памяти.

Отрицательным последствием использования дискового кэша является *потенциальное снижение надежности* системы. При крахе системы, когда по разным причинам (сбой по питанию, ошибка кода ОС и т. д.) теряется информация, находившаяся в оперативной памяти, могут пропасть и данные, которые пользователь считает надежно сохраненными на диске, но которые фактически до диска еще не дошли и хранились в кэше. Обычно для предотвращения таких потерь все содержимое дискового кэша периодически переписывается («сбрасывается») на диск. Существуют специальные системные вызовы, которые позволяют организовать принудительное выталкивание всех модифицированных

блоков кэша на диск. Примером может служить системный вызов sync в ОС Unix.

Однако и в этом случае потери возможны, хотя вероятность их меньше — теряются данные, которые появились в кэше в период после последнего сброса и до краха. Потери данных, хранящихся в кэше, можно существенно сократить при использовании восстанавливаемых файловых систем, которые рассматриваются далее в разделе «Отказоустойчивость файловых и дисковых систем».

Существует два способа организации дискового кэша. Первый способ, который можно назвать традиционным, основан на автономном диспетчере кэша, обслуживающем набор буферов системной памяти и при необходимости самостоятельно организующим загрузку блока диска в буфер, не обращаясь за помощью к другим подсистемам ОС. По такому принципу функционировал дисковый кэш во многих ранних версиях Unix (традиционный дисковый кэш работает и в последних версиях Unix, но в качестве вспомогательного механизма), а также в ОС семейства NetWare.

Второй способ основан на возможностях подсистемы виртуальной памяти по отображению файлов на память, о чем рассказывалось в предыдущем разделе. При этом способе функции диспетчера дискового кэша значительно сокращаются, так как большую часть работы выполняет подсистема виртуальной памяти, а значит, уменьшается объем ядра ОС и повышается его надежность. Однако применение механизма отображения файлов имеет одно ограничение — во многих файловых системах существуют служебные данные, которые не относятся к файлам, а следовательно, не могут кэшироваться. Поэтому в таких случаях наряду с кэшем на основе виртуальной памяти применяется и традиционный кэш.

Традиционный дисковый кэш

Рассмотрим традиционный дисковый кэш на примере его организации в ОС Unix, где он появился в первых же версиях.

Любой запрос на ввод-вывод к блок-ориентированному устройству преобразуется в запрос к подсистеме буферизации, которая представляет собой буферный пул и комплекс программ управления этим пулом, называемых диспетчером дискового кэша.

Буферный пул состоит из буферов, находящихся в области ядра. Размер отдельного буфера равен размеру блока данных (сектора) диска. С каждым буфером связан заголовок, структура которого уже рассматривалась, — это та же структура *buf*, которая используется блок-ориентированным драйвером при выполнении операций чтения и записи. В заголовке структуры *buf* драйверу передаются такие параметры, как адрес самого буфера в системной памяти, тип операции (чтение или запись), а также адресная информация для нахождения блока на диске (номер диска, номер раздела диска и логический номер блока).

Диспетчер дискового кэша управляет пулом буферов данных дисковых блоков с помощью дважды связанного списка заголовков структуры *buf*, каждый из которых указывает на буфер данных (рис. 8.6).

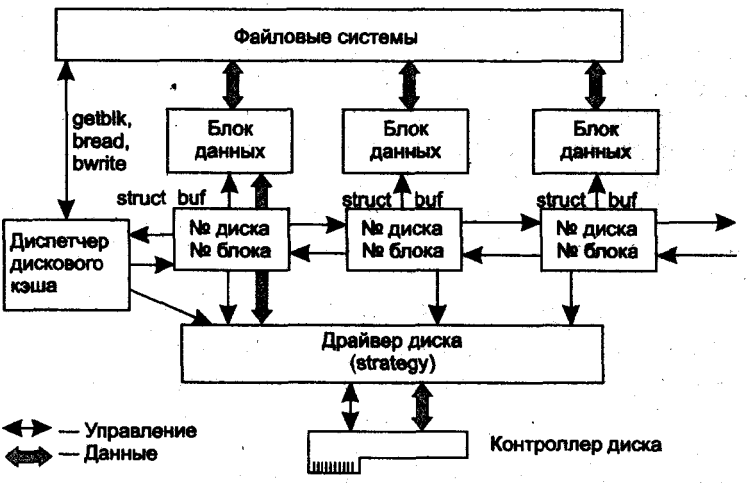


Рис. 8.6. Организация традиционного дискового кэша

При наличии кэша большая часть операций, которые выполняет функция `strategy` блок-ориентированного драйвера, производится с данными, размещенными в буферах дискового кэша, хотя драйверу все равно, кто инициировал данную операцию — диспетчер кэша или другой модуль ядра ОС, лишь бы она описывалась структурой *buf* с правильно заполненными полями. Отличие для драйвера заключается только в том, что диспетчер кэша всегда указывает в *buf* в качестве размера переписываемых данных размер блока диска (который чаще всего составляет 512 байт), а другие модули ОС могут указывать произвольный размер области данных. Например, менеджер виртуальной памяти запрашивает обмен страницами.

Таким образом, диспетчер кэша для фактической записи данных на диск использует интерфейс с блок-ориентированным драйвером, вызывая для этого его функцию `strategy`. Для вышележащих драйверов файловых систем диспетчер кэша предоставляет собственный интерфейс, который состоит из функций выделения буферов в кэше и функций чтения и записи данных из своих буферов. Функции диспетчера кэша не являются системными вызовами, они предназначены для внутреннего употребления, их используют модули операционной системы или же системные вызовы.

Интерфейс диспетчера кэша образует следующие функции.

- Функция `bwrite` при сброшенных признаках `B_ASYNC` и `B_DELWRI` в *buf* (рассматриваются далее) выполняет синхронную запись. В результате немедленно инициируется физический обмен с внешним устройством. Процесс, выдавший запрос, переходит в состояние ожидания результата выполнения операции ввода-вывода, используя функцию `sleep`. В данном случае в процессе может быть предусмотрена собственная реакция на ошибочную ситуацию. Такой тип записи используется тогда, когда необходима гарантия правильного завершения операции ввода-вывода.

- Функция `bwrite` при установленном признаке `B_ASYNC` и сброшенном признаке `B_DELWRI` выполняет асинхронную запись. Признак `B_ASYNC` задает асинхронный характер выполнения операции, при этом так же, как и в предыдущем случае, инициируется физический обмен с устройством, однако завершения операции ввода-вывода функция `bwrite` не дожидается и немедленно возвращает управление. В этом случае возможные ошибки ввода-вывода не могут быть переданы в процесс, выдавший запрос. Такая операция записи целесообразна при поточной обработке файлов, когда ожидание завершения операции ввода-вывода не обязательно, но есть уверенность в возможности повторения этой операции.
- Функция `bwrite` с установленными признаками `B_ASYNC` и `B_DELWRI` выполняет отложенную запись. При этом передача данных из системного буфера не производится, а в заголовке буфера делается отметка о том, что буфер заполнен и может быть выгружен, если потребуется его освободить. Управление немедленно возвращается вызвавшей функции.
- Функции `bread` и `getblk` обеспечивают соответственно чтение и получение блока. Каждая из этих функций ищет в пуле буфер, содержащий указанный блок данных (по номерам устройства и блока). Если такого блока в буферном пуле нет, то в случае функции `getblk` осуществляется поиск любого

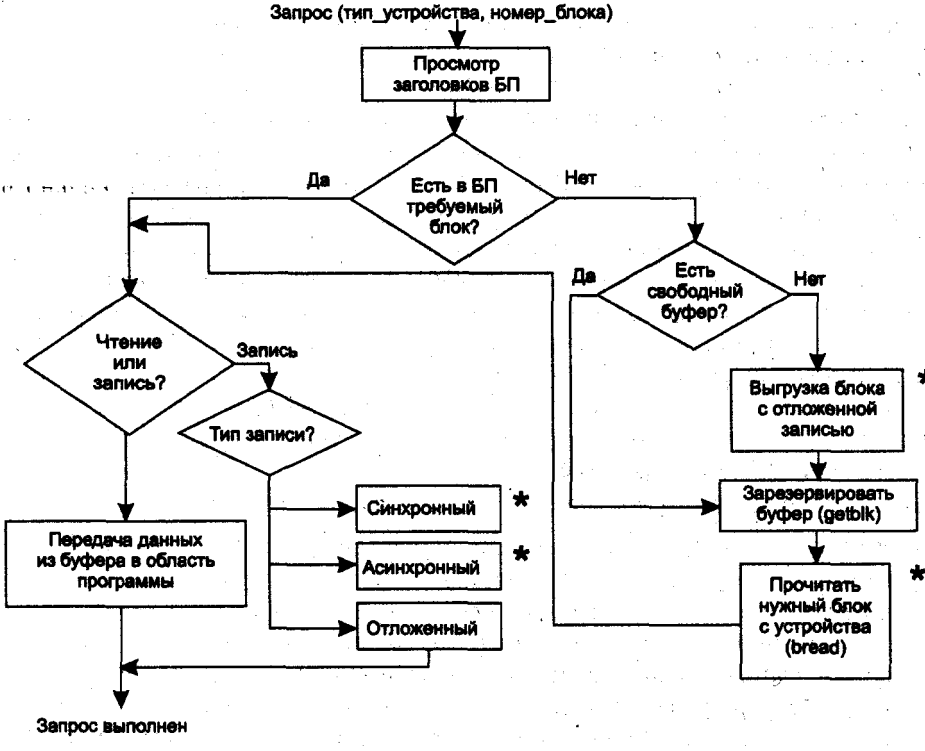


Рис. 8.7. Упрощенная схема выполнения запросов диспетчером дискового кэша

свободного буфера, а при его отсутствии возможна выгрузка на диск буфера, содержащего в заголовке признак отложенной записи. В случае функции `bread` при отсутствии заданного блока в буферном пуле сначала вызывается функция `getblk` для получения свободного буфера, а затем организуется загрузка в него данных путем вызова блок-ориентированного драйвера. Функция `getblk` используется тогда, когда содержимое зарезервированного блока несущественно, например, при записи на устройство блока данных.

Упрощенный алгоритм выполнения запросов к подсистеме буферизации приведен на рис. 8.7.

Отложенная запись является основным механизмом, за счет которого в системном буферном пуле задерживается определенное число блоков данных. На рисунке звездочками отмечены те операции, которые приводят к действительному обмену данных с диском, а остальные заканчиваются обменом данными в пределах оперативной памяти.

Дисковый кэш на основе виртуальной памяти

После того как средства виртуальной памяти получили в ОС широкое распространение, их стали использовать для кэширования файлов вместо специализированных традиционных механизмов автономного буферного пула. Ранее были рассмотрены механизмы отображения файлов на пользовательское виртуальное пространство по явному запросу приложения (например, с помощью системного вызова `mmap` в ОС Unix). При кэшировании же файл отображается не на пользовательскую, а на общую *системную* часть виртуального адресного пространства, а в остальном использование виртуальной памяти остается таким же. Для прикладного программиста кэширование файла путем отображения на системную область памяти остается прозрачным, оно выполняется неявно по инициативе ОС при выполнении обычных системных вызовов `read` и `write`.

В операционных системах Unix на основе кода SVR4 в виртуальном адресном пространстве системы существует специальный сегмент *segkmap*, на который отображаются данные всех открытых файлов. Диспетчер кэша поддерживает массив структур *smap*, каждая из которых хранит описание одного отображения. Одно отображение состоит из 8096 последовательных блоков одного файла. Отображаемый файл описывается в структуре *smap* парой *vnode/offset*, где *vnode* является указателем на виртуальный дескриптор операции с файлом (см. раздел «Файловые операции» в главе 7), а *offset* — смещением в файле на диске, начиная с которого отображаются данные файла. Кроме того, в структуре *smap* указывается виртуальный адрес внутри сегмента *segkmap*, на который отображаются данные файла.

При выполнении системного вызова `read` для чтения данных из некоторого открытого файла подсистема ввода-вывода вызывает функцию `segmap_getmap` диспетчера кэша, которой передается в качестве параметра пара *vnode/offset* (эти значения берутся из структуры *file*, описывающей операцию с файлом). Функция `segmap_getmap` ищет в массиве *smap* элемент, который содержит требуемую пару *vnode/offset*. Если такого элемента нет, это значит, что требуемый

участок файла еще не отображен на системную память и для нового отображения создается новый элемент *smap*, а значение виртуального адреса, на который он указывает, возвращается в `read`. После этого системный вызов `read` пытается скопировать данные из системной памяти, начиная с указанного адреса, в пользовательский буфер. Так как страницы, содержащей требуемый виртуальный адрес, в оперативной памяти пока нет, то при обращении к памяти возникает страничное прерывание, которое обслуживается соответствующим обработчиком прерываний. В результате блок-ориентированный драйвер читает блоки отсутствующей страницы (а при упреждающей загрузке — и нескольких окружающих ее страниц) с диска и помещает их в системную память. Затем системный вызов `read` продолжает свою работу, фактически копируя данные в пользовательский буфер. Последующие обращения с помощью вызова `read` к близкой к прочитанным данным области файла (в пределах 8096 блоков) уже не вызывают нового отображения, а страничные прерывания ведут к загрузке новых блоков файла в кэш по мере необходимости.

На основе таких же принципов работают диспетчеры кэша и в других современных ОС, поддерживающих виртуальную память, например ОС семейства Windows NT, OS/2.

Отказоустойчивость файловых и дисковых систем

Диски и файловые системы, используемые для упорядоченного хранения данных на дисках, часто представляют собой последний «островок стабильности», на котором находит спасение пользователь после неожиданного краха системы, разрушившего результаты его труда, полученные за последние несколько минут или даже часов, но не сохраненные на диске. Однако те данные, которые пользователь записывал в течение своего сеанса работы на диск, останутся, скорее всего, нетронутыми. Вероятность того, что при сбое питания или программной ошибке в коде какого-либо системного модуля система выполнит осмысленные действия по уничтожению файлов на диске, пренебрежимо мала. Поэтому при перезапуске операционной системы после краха большая часть данных, хранящихся в файлах на диске, остается корректной и доступной пользователю. Коды и данные операционной системы также хранятся в файлах, что и позволяет легко ее перезапустить после сбоя, не связанного с отказом диска или повреждением системных файлов.

Тем не менее диски тоже могут отказывать, например, по причине нарушения магнитных свойств отдельных областей поверхности. В данном разделе рассматриваются методы, которые повышают устойчивость вычислительной системы к отказам дисков за счет использования *избыточных дисков* и специальных алгоритмов управления массивами таких дисков.

Другой причиной недоступности данных после сбоя системы может стать нарушение целостности служебной информации файловой системы, произошедшее из-за незавершенности операций по изменению этой информации при

крахе системы. Примером такого нарушения может послужить несоответствие между адресной информацией файла, хранящейся в каталоге, и фактическим размещением кластеров файла. Для борьбы с этим явлением применяются так называемые *восстанавливаемые файловые системы*, которые обладают определенной степенью устойчивости к сбоям и отказам компьютера (при сохранении работоспособности диска, на котором расположена данная файловая система).

Комплексное применение отказоустойчивых дисковых массивов и восстанавливаемых файловых систем существенно повышает такой важный показатель вычислительной системы, как общая надежность.

Восстанавливаемость файловых систем

Причины нарушения целостности файловых систем

Восстанавливаемость файловой системы — это свойство, которое гарантирует, что в случае отказа питания или краха системы, когда все данные в оперативной памяти безвозвратно теряются, все начатые файловые операции будут либо успешно завершены, либо отменены безо всяких отрицательных последствий для работоспособности файловой системы.

Любая операция с файлом (создание, удаление, запись, чтение и т. д.) может быть представлена в виде некоторой последовательности *подопераций*. Последствия сбоя питания или краха ОС зависят от того, какая операция ввода-вывода выполнялась в этот момент, в каком порядке шли подоперации и до какой подоперации продвинулось выполнение операции к этому моменту.

Рассмотрим, например, последствия сбоя при удалении файла в файловой системе FAT. Для выполнения этой операции требуется пометить как недействительную запись об этом файле в каталоге, а также обнулить все элементы FAT, которые соответствуют кластерам удаляемого файла. Предположим, что сбой питания произошел после того, как была объявлена недействительной запись в каталоге и обнулено несколько (но не все) элементов FAT, занимаемых удаляемым файлом. В этом случае после сбоя файловая система сможет продолжать нормальную работу, если не считать того, что несколько последних кластеров удаленного файла «навечно» оказываются помечены как занятые. Хуже было бы, если бы операция удаления начиналась с обнуления элементов FAT, а корректировка каталога происходила бы после. Тогда при возникновении сбоя между этими подоперациями содержимое каталога не соответствовало бы действительному состоянию файловой системы: файл как будто существует, а на самом деле его нет. Ситуация, когда не исправленная запись в каталоге содержит адрес кластера, который уже объявлен свободным и может быть назначен другому файлу, может привести к разного рода коллизиям.

Некорректность файловой системы может возникать не только в результате насильственного прерывания операций ввода-вывода, выполняемых непосредственно с диском, но и в результате *нарушения работы дискового кэша*. Кэширование данных с диска предполагает, что в течение некоторого времени результаты операций ввода-вывода никак не сказываются на содержимом диска — все изменения происходят с копиями блоков диска, временно хранящихся в буферах

оперативной памяти. В этих буферах оседают данные из пользовательских файлов и служебная информация файловой системы, такая как каталоги, индексные дескрипторы, списки свободных, занятых и поврежденных блоков и т. п.

Для согласования содержимого кэша и диска время от времени выполняется запись всех модифицированных блоков, находящихся в кэше, на диск. Выталкивание блоков на диск может выполняться либо по инициативе менеджера дискового кэша, либо по инициативе приложения. Менеджер дискового кэша вытесняет блоки из кэша в следующих случаях:

- если необходимо освободить место в кэше для новых данных;
- если к менеджеру поступил запрос от какого-либо приложения или модуля ОС на запись указанных в запросе блоков на диск;
- при периодическом сбросе всех модифицированных блоков кэша на диск (как это происходит, например, в результате системного вызова `sync` в ОС Unix).

Кроме того, в распоряжение приложений обычно предоставляются средства, с помощью которых они могут запросить у подсистемы ввода-вывода операцию сквозной записи; при ее выполнении данные немедленно и практически одновременно записываются и на диск, и в кэш.

Несмотря на то что период полного сброса кэша на диск обычно выбирается весьма коротким (порядка 10–30 секунд), все равно остается высокая вероятность того, что после сбоя содержимое диска не в полной мере будет соответствовать действительному состоянию файловой системы — копии некоторых блоков с обновленным содержимым система может не успеть переписать на диск. Для восстановления некорректных файловых систем, использующих механизм кэширования диска, в операционных системах предусматриваются специальные утилиты, такие как fsck для файловых систем s5/uf, ScanDisk для FAT и Chkdsk для HPFS. Однако объем несоответствий может быть настолько большим, что восстановление файловой системы после сбоя с помощью стандартных системных средств становится невозможным.

Протоколирование транзакций

Проблемы, связанные с восстановлением файловой системы, могут быть решены при помощи техники протоколирования транзакций, которая сводится к следующему. В системе должны быть определены **транзакции** (transactions) — неделимые работы, которые не могут быть выполнены частично. Они либо выполняются полностью, либо вообще не выполняются.

Модель неделимой транзакции пришла из бизнеса. Пусть, например, идет переговорный процесс двух фирм о покупке-продаже некоторого товара. В процессе переговоров условия договора могут многократно меняться, уточняться. Пока договор еще не подписан обеими сторонами, каждая из них может от него отказаться. Но после подписания контракта сделка (транзакция) должна быть выполнена от начала и до конца. Если же контракт не подписан, то любые дей-

ствия, которые были уже проделаны, отменяются или объявляются недействительными.

В файловых системах такими транзакциями являются операции ввода-вывода, изменяющие содержимое файлов, каталогов или других системных структур файловой системы (например, индексных дескрипторов ufs или элементов FAT). Пусть к файловой системе поступает запрос на выполнения той или иной операции ввода-вывода. Эта операция включает несколько шагов, связанных с созданием, уничтожением и модификацией объектов файловой системы. Если все подоперации были благополучно завершены, то транзакция считается выполненной. Это действие называется **фиксацией** (committing) транзакции. Если же одна или более подопераций не успели выполниться из-за сбоя питания или краха ОС, тогда для обеспечения целостности файловой системы все измененные в рамках транзакции данные файловой системы должны быть возвращены точно в то состояние, в котором они находились до начала выполнения транзакции.

Так, например, транзакцией может быть представлена операция удаления файла. Действительно, для целостности файловой системы необходимо, чтобы все требуемые при выполнении данной операции изменения каталога и таблицы распределения дисковой памяти были сделаны в полном объеме. Либо, если во время операции произошел сбой, каталог и таблица распределения памяти должны быть приведены в исходное состояние.

В то же время в файловой системе существуют операции, которые не изменяют состояния файловой системы и которые, вследствие этого, нет необходимости рассматривать как транзакции. Примерами таких операций являются чтение файла, поиск файла на диске, просмотр атрибутов файла.

Незавершенная операция с диском несет угрозу целостности файловой системы. Каким же образом файловая система может реализовать свойство транзакций «все или ничего»? Очевидно, что решение в этом случае может быть одно — необходимо **протоколировать** (запоминать) все изменения, происходящие в рамках транзакции, чтобы на основе этой информации в случае прерывания транзакции можно было отменить все уже выполненные подоперации, то есть реализовать так называемый **откат** транзакции.

В файловых системах с кэшированием диска для восстановления системы после сбоя, помимо отката незавершенных транзакций, необходимо выполнить дополнительное действие — **повторение** зафиксированных транзакций. Когда происходит сбой по питанию или крах ОС, все данные, находящиеся в оперативной памяти, теряются, в том числе и модифицированные блоки данных, которые менеджер дискового кэша не успел вытолкнуть на диск. Единственный способ восстановить утерянные изменения данных — это повторить все завершенные транзакции, которые участвовали в модификации этих блоков. Чтобы обеспечить возможность повторения транзакций, система должна включать в протокол не только данные, которые могут быть использованы для отката транзакции, но и данные, которые позволят в случае необходимости повторить всю транзакцию.

ПРИМЕЧАНИЕ При восстановлении часто возникают ситуации, когда система пытается отменить транзакцию, которая уже была отменена или вообще не выполнялась. Аналогично, при повторении некоторой транзакции может оказаться, что она уже была выполнена. Учитывая это, необходимо так определить подоперации транзакции, чтобы многократное выполнение каждой из этих подопераций не давало никакого добавочного эффекта по сравнению с первым выполнением этой подоперации. Такое свойство операции называется идемпотентностью (idempotency). Примером идемпотентной операции может служить, например, многократное присвоение переменной некоторого значения (сравните с операциями инкремента и декремента).

Для восстановления файловой системы требуется упреждающее протоколирование транзакций. Оно заключается в том, что перед изменением какого-либо блока данных на диске или в дисковом кэше производится запись в специальный системный файл — **журнал транзакций** (log file), где отмечается, какая транзакция делает изменения, какие файл и блок изменяются и каковы старое и новое значения изменяемого блока. Изменения в исходных блоках делаются только после успешной регистрации всех подопераций в журнале. Если транзакция прерывается, то информация журнала регистрации используется для приведения файлов, каталогов и служебных данных файловой системы в исходное состояние, то есть производится откат. Если транзакция фиксируется, то и об этом делается запись в журнал регистрации, но новые значения измененных данных сохраняются в журнале еще некоторое время, чтобы сделать возможным повторение транзакции, если это потребуется.

Восстанавливаемость файловой системы NTFS

Файловая система NTFS является восстанавливаемой файловой системой, однако восстанавливаемость обеспечивается только для системной информации файловой системы, то есть каталогов, атрибутов безопасности, битовой карты занятости кластеров и других системных файлов. Сохранность данных пользовательских файлов, работа с которыми выполнялась в момент сбоя, в общем случае не гарантируется.

Для повышения производительности файловая система NTFS использует дисковый кэш, то есть все изменения файлов, каталогов и управляющей информации выполняются сначала с копиями соответствующих блоков в буферах оперативной памяти и только спустя некоторое время переносятся на диск. Однако кэширование, как уже было сказано, повышает риск разрушения файловой системы. В таких условиях NTFS обеспечивает отказоустойчивость с помощью технологии протоколирования транзакций и восстановления системных данных. Пользовательские данные, которые в момент краха находились в дисковом кэше и не успели записаться на диск, в NTFS не восстанавливаются.

Журнал регистрации транзакций в NTFS делится на две части: область рестарта и область протоколирования (рис. 8.8).

- *Область рестарта* содержит информацию о том, с какого места необходимо будет начать читать журнал транзакций для проведения процедуры восстановления системы после сбоя или краха ОС. Эта информация представляет собой указатель на определенную запись в области протоколирования. Для надежности в файле журнала регистрации хранится две копии области рестарта.
- *Область протоколирования* содержит записи обо всех изменениях в системных данных файловой системы, произошедших в результате выполнения транзакций в течение некоторого достаточно большого периода. Все записи идентифицируются логическим порядковым номером (Logical Sequence Number, LSN). Записи о подоперациях, принадлежащих одной транзакции, образуют связанный список: каждая последующая запись содержит номер предыдущей записи. Заполнение области протоколирования идет циклически: после исчерпания всей памяти, отведенной под область протоколирования, новые записи помещаются на место самых старых.

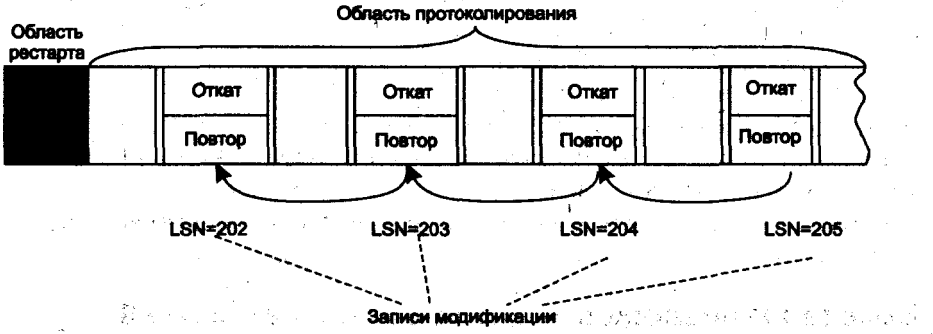


Рис. 8.8. Записи модификации в журнале транзакций

Существует несколько типов записей в журнале транзакций: запись модификации, запись контрольной точки, запись фиксации транзакции, запись таблицы модификации, запись таблицы модифицированных страниц.

Запись модификации заносится в журнал транзакций относительно каждой подоперации, которая модифицирует системные данные файловой системы. Эта запись состоит из двух частей: одна содержит информацию, необходимую системе для повторения этого действия, а другая — информацию для его отмены. Информация о модификации хранится в двух формах — в физическом и логическом описаниях. Логическое описание используется программным обеспечением уровня приложений и формулируется в терминах операций, например, «выделить файловую запись в MFT» или «удалить имя из корневого индекса». На нижнем уровне программного обеспечения, к которому относятся модули самой системы NTFS, применяется менее компактное, но более простое физическое описание, сводящееся к указанию диапазона байтов на диске, в которые необходимо поместить определенные значения.

Пусть, например, регистрируется транзакция создания файла lotus.doc, которая включает, как показано на рисунке, три подоперации. Тогда в журнале будут сделаны три записи модификации, содержимое которых приведено в табл. 8.1.

Таблица 8.1. Структура записи модификации

Запись модификации	Информация для повторения транзакции	Информация для отката транзакции
LSN = 202	Выделить и инициировать запись для нового файла lotus.doc в таблице MFT	Удалить запись о файле lotus.doc из таблицы MFT
LSN = 203	Добавить имя файла в индекс	Исключить имя файла из индекса
LSN = 204	Установить биты 3–9 в битовой карте	Обнулить биты 3–9 в битовой карте

Журнал транзакций, как и все остальные файлы, кэшируется в буферах оперативной памяти и периодически сбрасывается на диск.

Файловая система NTFS все действия с журналом транзакций выполняет только путем запросов к специальной службе LFS (Log File Service). Эта служба размещает в журнале новые записи, сбрасывает на диск все записи до некоторого заданного номера, считывает записи в прямом и обратном порядке и выполняет некоторые другие действия над записями журнала.

Прежде чем выполнить любую транзакцию, NTFS вызывает службу журнала транзакций LFS для регистрации всех подопераций в журнале транзакций. И только после этого описанные подоперации действительно выполняются с копиями блоков данных файловой системы, находящимися в кэше. Когда все подоперации транзакции выполнены, с помощью службы LFS транзакция фиксируется. Это выражается в том, что в журнал заносится специальный вид записи — *запись фиксации транзакции*.

Параллельно с регистрацией и выполнением транзакций происходит процесс выталкивания блоков кэша на диск. Сброс на диск измененных блоков выполняется в два этапа: сначала сбрасываются блоки журнала, а потом — модифицированные блоки транзакций. Такой порядок реализуется следующим образом. Каждый раз, когда диспетчер кэша принимает решение о том, что определенные модифицированные блоки (не обязательно все) должны быть вытеснены на диск, он сообщает об этом службе LFS. В ответ на это сообщение LFS обращается к диспетчеру кэша с запросом о записи на диск всех измененных блоков журнала. После того как блоки журнала сброшены на диск, сбрасываются на диск модифицированные блоки транзакций, среди которых могут быть, конечно, и блоки системных данных файловой системы.

Такая двухэтапная процедура сброса данных кэша на диск делает возможным восстановление файловой системы, если во время записи модифицированных блоков из кэша на диск происходит сбой. Действительно, какие бы неприятности ни произошли во время записи модифицированных блоков на диск, вся

информация об изменениях, произведенных в этих блоках, уже записана на диск в файл журнала транзакций. Заметим, что все действия, которые были выполнены файловой системой в интервале между последним сбросом данных на диск и сбоем, отменяются сами собой, поскольку все они проводятся только над блоками в кэше и не вызывают никаких изменений содержимого диска.

Какие же дефекты может иметь файловая система после сбоя?

Во-первых, это несогласованность системных данных, возникшая в результате незавершенности транзакций, которые были начаты еще до момента последнего сброса данных из кэша на диск. На рис. 8.9 показана транзакция *A*, две подоперации которой, a_1 и a_2, были сделаны до сброса кэша, а еще две, a_3 и a_4, — после сброса. К моменту сбоя результаты первых двух подопераций могли быть записаны на диск, в то время как изменения, вызванные подоперациями a_3 и a_4, отразились только на копиях блоков файловой системы в кэше и были потеряны в результате сбоя. Чтобы устранить несогласованность, вызванную этой причиной, требуется сделать откат для всех транзакций, незафиксированных к моменту последнего сброса кэша. Для примера, изображенного на рисунке, такими транзакциями являются транзакции *A* и *C*. В каждый момент времени NTFS располагает списком незафиксированных транзакций, называемым **таблицей незавершенных транзакций** (transaction table). Для каждой незавершенной транзакции эта таблица содержит порядковый номер (LSN) последней по времени подоперации, выполненной в рамках данной транзакции. По этому номеру может быть найдена вся цепочка подопераций транзакции.

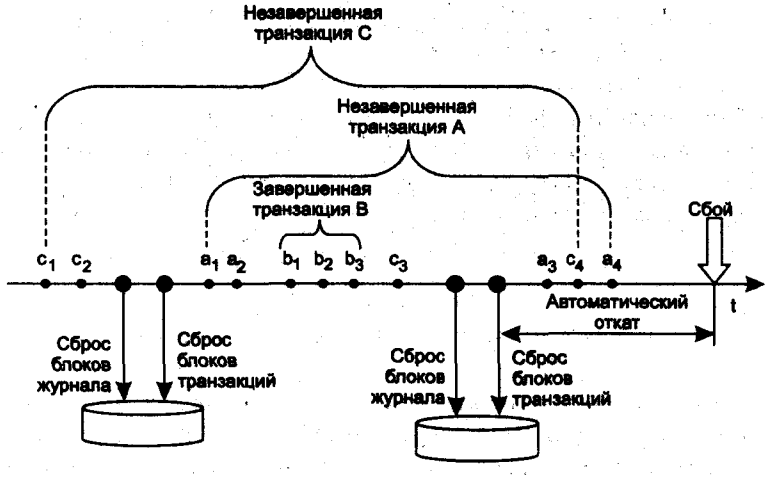


Рис. 8.9. Временная диаграмма событий в файловой системе

Во-вторых, противоречия в файловой системе могут быть вызваны потерей тех изменений, которые были сделаны транзакциями, завершившимися еще до сброса кэша, но которые не были записаны на диск в ходе последнего сброса. На рисунке таковой может оказаться транзакция *B*. Чтобы определить, какие завершенные транзакции надо повторять, система ведет **таблицу модифицированных**

страниц[1] (dirty page table), находящихся в данный момент в кэше. В таблице для каждой модифицированной страницы указывается, какая транзакция вызвала эти изменения. Повторение транзакций, которые имели дело со страницами, указанными в данной таблице, гарантирует, что ни одно изменение не будет потеряно.

Таблицы модифицированных страниц и незавершенных транзакций создаются NTFS на основании записей журнала транзакций и поддерживаются в оперативной памяти. Следует подчеркнуть, что обе эти таблицы не добавляют новой информации в журнал транзакций, они лишь представляют информацию, содержащуюся в записях журнала, в концентрированном виде, более удобном для восстановления. Содержимое таблиц фиксируется в журнале транзакций во время выполнения операции *контрольная точка*.

Операция контрольная точка выполняется каждые 5 секунд и включает следующие действия (рис. 8.10). Сначала в области протоколирования журнала транзакций создаются две записи — запись таблицы незавершенных транзакций и запись таблицы модифицированных страниц, содержащие копии соответствующих таблиц. Затем номера этих записей включаются в запись контрольной точки, которая также создается в области протоколирования журнала транзакций. Сделав запись контрольной точки, NTFS помещает ее номер LSN в область рестарта.

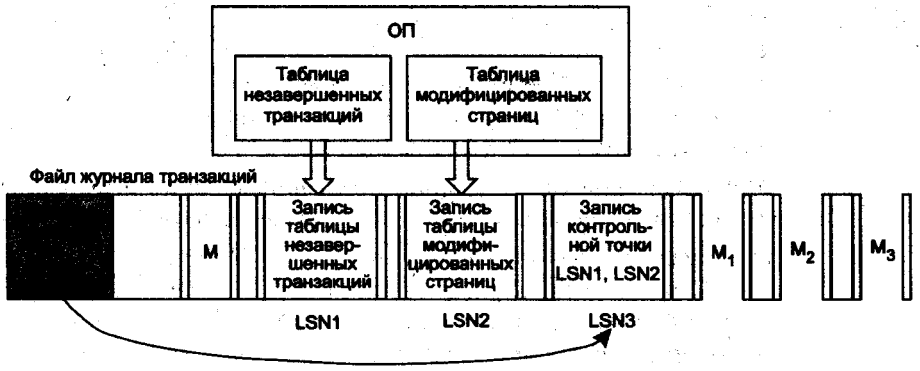


Рис. 8.10. Записи операции контрольная точка

Заметим, что процессы создания контрольных точек и сброса блоков данных из кэша на диск протекают асинхронно. Когда в результате сбоя из оперативной памяти исчезает вся информация, в том числе из таблиц незавершенных транзакций и модифицированных страниц, состояние этих таблиц, хотя и несколько устаревшее, сохраняется на диске в файле журнала транзакций. Кроме того, здесь же имеется несколько более поздних записей, которые были сделаны в период между сохранением таблиц и сбросом кэша (на рисунке это записи

[1] Дисковый кэш в ОС семейства Windows NT основан на использовании виртуальной памяти, поэтому он оперирует не блоками, а страницами.

M_1, M_2, M_3). При восстановлении файловая система обрабатывает эти записи и вносит изменения в таблицы незавершенных транзакций и модифицированных страниц, сохраненные в журнале. Так, например, если запись M_1 является записью фиксации транзакции, то соответствующая транзакция исключается из таблицы незавершенных транзакций, а если это запись модификации, то в таблицу модифицированных страниц заносится информация еще об одной странице.

Процесс восстановления файловой системы включает следующие шаги.

1. Чтение области рестарта из файла журнала транзакций и определение номера самой последней по времени записи контрольная точка.
2. Чтение записи контрольная точка и определение номеров записей таблицы незавершенных транзакций и таблицы модифицированных страниц.
3. Чтение и корректировка таблиц незавершенных транзакций и модифицированных страниц на основании записей, сделанных в журнале транзакций уже после сохранения таблиц в журнале, но еще до записи журнала на диск (рис. 8.11).
4. Анализ таблицы модифицированных страниц, определение номера самой ранней записи модификации страницы.
5. Чтение журнала транзакций в прямом направлении, начиная с самой ранней записи о модификации, найденной при анализе таблицы модифицированных страниц. При этом система выполняет *повторение завершенных транзакций*, в результате которого устраняются все несоответствия файловой системы, вызванные потерями модифицированных страниц в кэше во время сбоя или краха операционной системы.
6. Анализ таблицы незавершенных транзакций, определение номера самой поздней подоперации, выполненной в рамках незавершенной транзакции.
7. Чтение журнала транзакций в обратном направлении. Учитывая, что все подоперации каждой транзакции связаны в список, система легко переходит от одной записи модификации к другой, извлекает из них информацию, необходимую для отмены, и выполняет *откат незавершенных транзакций*.

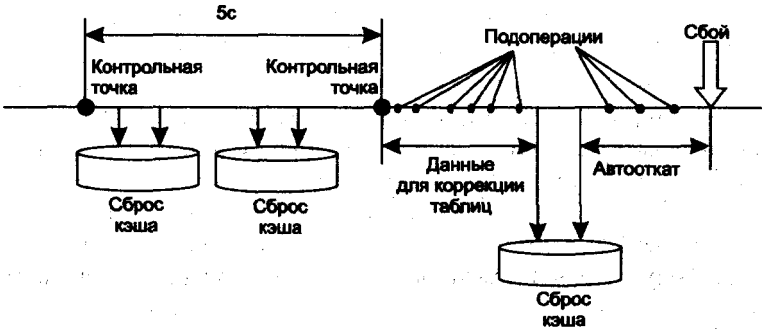


Рис. 8.11. Асинхронность процессов сброса кэша и создания контрольных точек

Операция отмены сама является транзакцией, поскольку связана с модификацией системных блоков файловой системы, поэтому она протоколируется обычным образом в журнале транзакций. По отношению к ней также могут быть применены операции повторения или отката.

Избыточные дисковые подсистемы RAID

В основе средств обеспечения отказоустойчивости дисковой памяти лежит общий для всех отказоустойчивых систем принцип избыточности, и дисковые подсистемы RAID (Redundant Array of Inexpensive Disks, дословно — «избыточный массив недорогих дисков») являются примером реализации этого принципа. Идея технологии **RAID-массивов** состоит в том, что для хранения данных используется несколько дисков даже в тех случаях, когда для таких данных хватило бы места на одном диске. Организация совместной работы нескольких централизованно управляемых дисков позволяет придать их совокупности новые свойства, отсутствовавшие у каждого диска в отдельности.

RAID-массив может быть создан на базе нескольких обычных дисковых устройств, управляемых обычными контроллерами, в этом случае для организации управления всей совокупностью дисков в операционной системе должен быть установлен специальный драйвер. В ОС семейства Windows NT, например, таким драйвером является FtDisk — драйвер отказоустойчивой дисковой подсистемы. Существуют также различные модели дисковых систем, в которых технология RAID полностью реализуется аппаратными средствами, в этом случае массив дисков управляется общим специальным контроллером.

Дисковый массив RAID представляется для пользователей и прикладных программ единым логическим диском. Такое логическое устройство может обладать различными качествами в зависимости от стратегии, заложенной в алгоритмы работы средств централизованного управления и размещения информации на всей совокупности дисков. Это логическое устройство может, например, обладать повышенной отказоустойчивостью или иметь производительность, значительно более высокую, чем у отдельно взятого диска, либо обладать обоими этими свойствами. Различают несколько вариантов RAID-массивов, называемых также *уровнями*: RAID-0, RAID-1, RAID-2, RAID-3, RAID-4, RAID-5 и некоторые другие.

При оценке эффективности RAID-массивов чаще всего используются следующие критерии:

- степень избыточности хранимой информации (или тесно связанная с этим критерием стоимость хранения единицы информации);
- производительность операций чтения и записи;
- степень отказоустойчивости.

В логическом устройстве **RAID-0** (рис. 8.12) общий для дискового массива контроллер при выполнении операции записи расщепляет данные на блоки и передает их параллельно на все диски, при этом первый блок данных записывается на первый диск, второй — на второй и т. д. Различные варианты реализа-

ции технологии RAID-0 могут отличаться размерами блоков данных, например, в наборах с чередованием, представляющих собой программную реализацию RAID-0 в ОС семейства Windows NT, на диски поочередно записываются полосы (strips) данных по 64 Кбайт. При чтении контроллер мультиплексирует блоки данных, поступающие со всех дисков, и передает их источнику запроса.

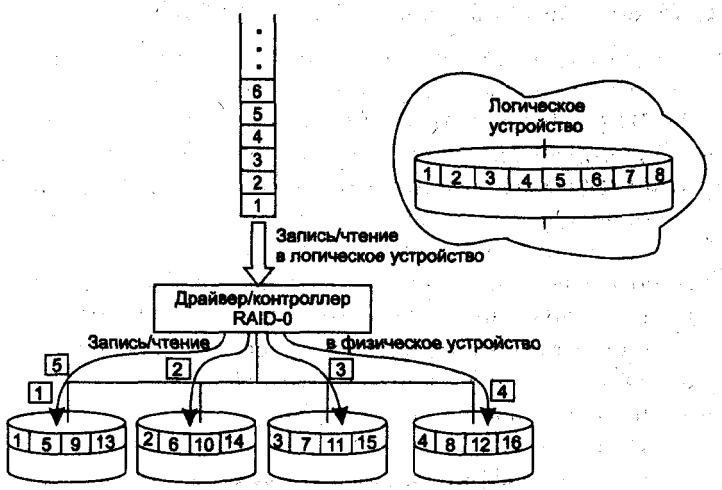


Рис. 8.12. Организация массива RAID-0

По сравнению с одиночным диском, в котором данные записываются на диск и считываются с диска последовательно, производительность дисковой конфигурации RAID-0 значительно выше за счет одновременности операций записи/чтения по всем дискам массива.

Уровень RAID-0 не обладает избыточностью данных, а значит, не обеспечивает повышение отказоустойчивости. Если при считывании произойдет сбой, то данные будут безвозвратно испорчены. Более того, отказоустойчивость даже снижается, поскольку если один из дисков выйдет из строя, то восстанавливать придется все диски массива. Имеется еще один недостаток — если при работе с RAID-0 объем памяти логического устройства потребуется изменить, то сделать это путем простого добавления еще одного диска к уже имеющимся в RAID-массиве дискам будет невозможно без полного перераспределения информации по всему изменившемуся набору дисков.

Уровень **RAID-1** (рис. 8.13) реализует подход, называемый **зеркальным копированием** (mirroring). Логическое устройство в этом случае образуется на основе одной или нескольких пар дисков, в которых один диск является основным, а другой диск (зеркальный) дублирует информацию, находящуюся на основном диске. Если основной диск выходит из строя, зеркальный продолжает сохранять данные, тем самым обеспечивается повышенная отказоустойчивость логического устройства. За это приходится платить избыточностью — все данные хранятся на логическом устройстве RAID-1 в двух экземплярах, в результате дисковое пространство используется лишь на 50 %.

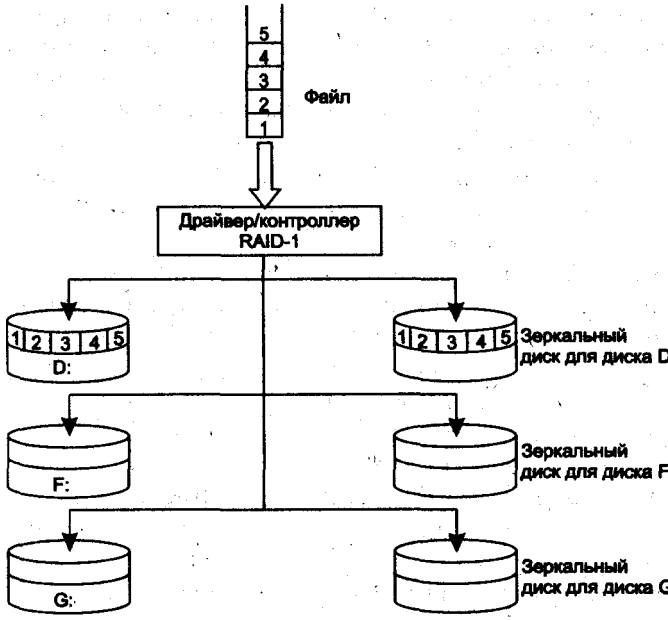


Рис. 8.13. Организация массива RAID-1

При внесении изменений в данные, расположенные на логическом устройстве RAID-1, контроллер (или драйвер) массива дисков одинаковым образом модифицирует и основной и зеркальный диски, при этом дублирование операций абсолютно прозрачно для пользователя и приложений. Удвоение количества операций записи снижает, хотя и не очень значительно, производительность дисковой подсистемы, поэтому во многих случаях наряду с дублированием дисков дублируются и их контроллеры. Такое **дублирование** (duplexing), помимо повышения скорости операций записи, повышает надежность системы — данные на зеркальном диске останутся доступными не только при сбое диска, но и в случае сбоя дискового контроллера.

Некоторые современные контроллеры (например, SCSI-контроллеры) обладают способностью ускорять выполнение операций чтения с дисков, связанных в зеркальный набор. При высокой интенсивности ввода-вывода контроллер распределяет нагрузку между двумя дисками так, что две операции чтения могут быть выполнены одновременно. В результате распараллеливания работы по считыванию данных между двумя дисками время операции чтения может быть снижено в два раза! Таким образом, некоторое снижение производительности, возникающее при выполнении операций записи, с лихвой компенсируется повышением скорости операций чтения.

Уровень **RAID-2** расщепляет данные побитно: первый бит записывается на первый диск, второй бит — на второй диск и т. д. Отказоустойчивость реализуется в RAID-2 путем использования для кодирования данных корректирующего кода Хэмминга, который обеспечивает исправление однократных ошибок

и обнаружение двукратных ошибок. Избыточность достигается за счет нескольких дополнительных дисков, куда записывается код коррекции ошибок. Так, массив с числом основных дисков от 16 до 32 должен иметь три дополнительных диска для хранения кода коррекции. Массив RAID-2 обеспечивает высокую производительность и надежность, но он применяется в основном в мэйнфреймах и суперкомпьютерах. В сетевых файловых серверах из-за высокой стоимости реализации RAID-2 в настоящее время практически не используется.

В массивах **RAID-3** используется **расщепление** (stripping) данных на массивы дисков с выделением одного диска на весь набор для контроля четности. То есть если имеется массив из N дисков, то запись на $N - 1$ из них производится параллельно с побайтным расщеплением, а диск N используется для записи контрольной информации о четности. Диск четности является резервным. Если какой-либо диск выходит из строя, то данные остальных дисков плюс данные о четности резервного диска позволяют не только определить, какой из дисководов массива вышел из строя, но и восстановить утраченную информацию. Это восстановление может выполняться динамически по мере поступления запросов или в результате выполнения специальной процедуры восстановления, когда содержимое отказавшего диска заново генерируется и записывается на резервный диск.

Рассмотрим пример динамического восстановления данных. Пусть массив RAID-3 состоит из четырех дисков: три из них — ДИСК 1, ДИСК 2 и ДИСК 3 — хранят данные, а ДИСК 4 хранит контрольную сумму по модулю 2 (XOR). И пусть на логическое устройство, образованное этими дисками, записывается последовательность байтов, каждый из которых имеет значение, равное его порядковому номеру в последовательности. Тогда первый байт 0000 0001 попадет на ДИСК 1, второй байт 0000 0010 — на ДИСК 2, а третий по порядку байт — на ДИСК 3. На четвертый диск будет записана сумма по модулю 2, равная в данном случае 0000 0000 (рис. 8.14). Вторая строка таблицы, приведенной на рисунке, соответствует следующим трем байтам и их контрольной сумме и т. д. Представим, что ДИСК 2 вышел из строя.

ДИСК 1	ДИСК 2	ДИСК 3	ДИСК 4
0000 0001	0000 0010	0000 0011	0000 0000
0000 0100	0000 0101	0000 0110	0000 0111
0000 0111	0000 1000	0000 1001	0000 0110
0000 1010	0000 1011	0000 1100	0000 1101

Рис. 8.14. Пример распределения данных по дискам массива RAID-3

При поступлении запроса на чтение, например, пятого байта (он выделен жирным шрифтом), контроллер дискового массива считывает данные, относящиеся к этой строке со всех трех оставшихся дисков (байты 0000 0100, 0000 0110, 0000 0111), и вычисляет для них сумму по модулю 2. Значение контрольной суммы 0000 0101 и будет являться восстановленным значением потерянного

из-за неисправности пятого байта. Конечно, эта схема восстановления работает и для произвольных значений байтов, записываемых в произвольном порядке на диски, назначенные для хранения данных.

Если же требуется записать данные на отказавший диск, то эта операция физически не выполняется, вместо этого корректируется контрольная сумма — она получает такое значение, как если бы данные были действительно записаны на этот диск.

Однако динамическое восстановление данных снижает производительность дисковой подсистемы. Для полного восстановления исходного уровня производительности необходимо заменить вышедший из строя диск и провести регенерацию всех данных, которые хранились на отказавшем диске.

Минимальное количество дисков, необходимое для создания конфигурации RAID-3, равно трем. В этом случае избыточность достигает максимального значения — 33 %. При увеличении числа дисков степень избыточности снижается, так, для 33 дисков она составляет менее 1 %.

Уровень RAID-3 для файлов с длинными записями позволяет выполнять одновременное чтение данных с нескольких дисков или запись данных на несколько дисков, однако следует подчеркнуть, что в каждый момент выполняется только один запрос на ввод-вывод, то есть RAID-3 позволяет распараллеливать ввод-вывод в рамках только одного процесса (рис. 8.15). Таким образом, уровень RAID-3 повышает как надежность, так и скорость обмена информацией.

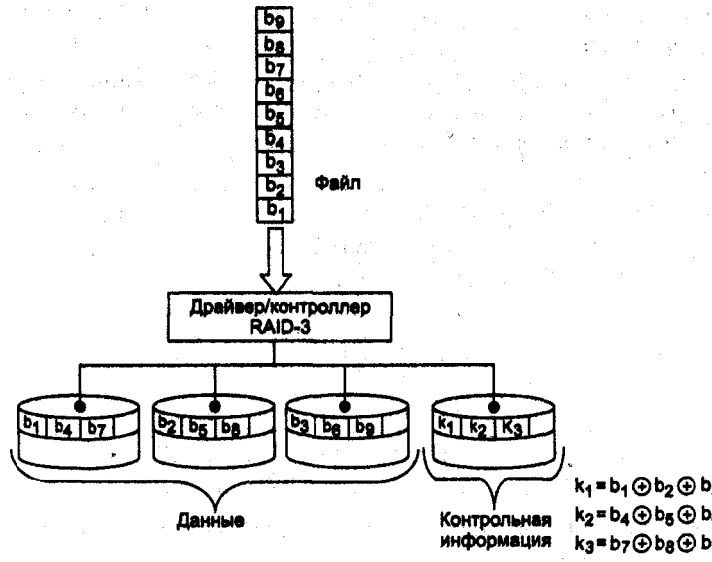


Рис. 8.15. Организация массива RAID-3

Организация **RAID-4** аналогична RAID-3 за тем исключением, что данные распределяются на дисках не побайтно, а блоками. За счет этого может происходить независимый обмен с каждым диском. Для хранения контрольной

информации также используется один дополнительный диск. Эта реализация удобна для файлов с очень короткими записями и большей частотой операций чтения по сравнению с операциями записи, поскольку в этом случае при подходящем размере блоков диска возможно одновременное выполнение нескольких операций чтения.

Однако по-прежнему допустима только одна операция записи в каждый момент времени, так как все операции записи используют один и тот же дополнительный диск для вычисления контрольной суммы. Действительно, информация о четности должна корректироваться каждый раз, когда выполняется операция записи. Контроллер должен сначала считать старые данные и старую контрольную информацию, а затем, объединив их с новыми данными, вычислить новое значение контрольной суммы и записать его на диск, предназначенный для хранения контрольной информации. Если требуется выполнить запись в более чем один блок, то возникает конфликт по обращению к диску с контрольной информацией. Все это приводит к тому, что скорость выполнения операций записи в массиве RAID-4 снижается.

В уровне **RAID-5** (рис. 8.16) используется метод, аналогичный RAID-4, но данные о контроле четности распределяются по всем дискам массива. При выполнении операции записи требуется в три раза больше оперативной памяти. Каждая команда записи инициирует ту же последовательность «считывание — модификация — запись» в нескольких дисках, как и в методе RAID-4. Наибольший выигрыш в производительности достигается при операциях чтения. Поскольку информация о четности может быть считана с нескольких дисков и записана на несколько дисков одновременно, скорость записи по сравнению с уровнем RAID-4 увеличивается, однако она все еще гораздо ниже скорости отдельного диска уровня RAID-1 или RAID-3.

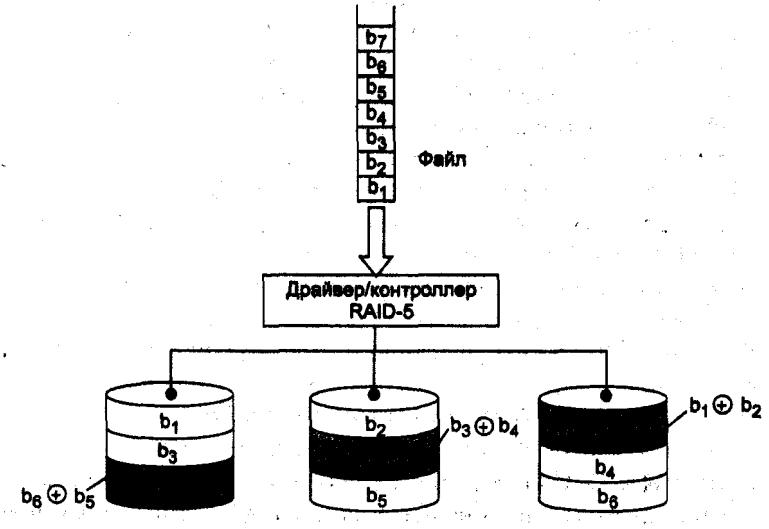


Рис. 8.16. Организация массива RAID-5

Помимо рассмотренных имеются еще и другие варианты организации совместной работы избыточного набора дисков, среди них можно особо отметить технологию **RAID-10**, которая представляет собой комбинированный подход, при котором данные «расщепляются» (RAID-0) и зеркально копируются (RAID-1) без вычисления контрольных сумм. Обычно две пары «зеркальных» массивов объединяются и образуют один массив RAID-0. Этот подход целесообразно применять при работе с большими файлами.

В табл. 8.2 сведены основные характеристики некоторых конфигураций избыточных дисковых массивов.

Таблица 8.2. Характеристики уровней RAID

Конфигурация RAID	Избыточность	Отказоустойчивость	Скорость чтения	Скорость записи
RAID-0	Нет	Нет	Повышенная	Повышенная
RAID RAID-1	50 %	Есть	Повышенная	Пониженная (в варианте без дуплексирования)
RAID-3, RAID-4, RAID-5	До 33 %	Есть	Повышенная	Пониженная (в разной степени)
RAID-10	50 %	Есть	Повышенная	Повышенная

Обмен данными между процессами и потоками

Если абстрагироваться от вопросов синхронизации, то обмен данными между потоками одного процесса не представляет никакой сложности — имея общее адресное пространство и общие открытые файлы, потоки получают беспрепятственный доступ к данным друг друга. Другое дело — обмен данными потоков, выполняющихся в рамках разных процессов. Для защиты процессов друг от друга ОС возводит мощные изолирующие преграды, которые не только защищают процессы, но и не позволяют им передавать друг другу данные. Потоки разных процессов работают в разных адресных пространствах. Однако операционная система имеет доступ ко всем областям памяти, поэтому она может играть роль посредника в информационном обмене прикладных потоков. При необходимости в обмене данными поток обращается с запросом к ОС. По этому запросу ОС, пользуясь своими привилегиями, создает различные системные средства связи, такие, например, как конвейеры или очереди сообщений.

Эти средства, так же как рассмотренные ранее средства синхронизации процессов, относятся к классу **средств межпроцессного взаимодействия** (Inter-Process Communications, IPC).

Многие из средств межпроцессного обмена данными выполняют также и функции синхронизации: в том случае, когда данные для процесса-получателя

отсутствуют, последний переводится в состояние ожидания средствами ОС, а при поступлении данных от процесса-отправителя процесс-получатель активизируется.

Набор средств межпроцессного обмена данными в большинстве современных ОС выглядит следующим образом:

- конвейеры (pipes);
- именованные конвейеры (named pipes);
- очереди сообщений (message queues);
- разделяемая память (shared memory).

Кроме этого, достаточно стандартного набора средств, в конкретных ОС часто имеются и более специфические средства межпроцессного обмена, например средства среды STREAMS для различных версий Unix или почтовые ящики (mail slots) в ОС Windows.

Конвейеры

Конвейеры как средство межпроцессного обмена данными впервые появились в операционной системе Unix. Системный вызов `pipe` позволяет двум процессам обмениваться неструктурированным потоком байтов. Конвейер представляет собой буфер в оперативной памяти, поддерживающий очередь байтов по алгоритму FIFO. Для программиста, использующего системный вызов `pipe`, этот буфер выглядит как безымянный файл, в который можно писать и из которого можно читать, осуществляя тем самым обмен данными.

Системный вызов `pipe` имеет одно существенное ограничение — обмениваться данными могут только родственные процессы, точнее процессы, которые имеют общего прародителя, создавшего данный конвейер. Если операционная система поддерживает потоки, то это же ограничение будет означать, что конвейером могут воспользоваться только потоки, относящиеся к такого рода процессам. Ограничение проистекает из-за того, что указанный конвейер не имеет имени, а обращение к нему происходит по дескриптору файла, который, как уже отмечалось, имеет локальное для каждого процесса значение.

При выполнении системного вызова `pipe` в процесс возвращаются два дескриптора файла, один для записи данных в конвейер, другой для чтения данных из конвейера. Обычно для выполнения некоторой общей работы ведущий процесс сначала создает конвейер, а затем — несколько процессов-потомков с помощью соответствующего системного вызова. В результате механизм наследования процессов копирует для всех процессов-потомков значения дескрипторов, указывающих на один и тот же конвейер, так что все кооперирующиеся процессы, включая процесс-прародитель, могут использовать этот конвейер для обмена данными. Данные читаются из конвейера с помощью системного вызова `read` с использованием первого из возвращенных вызовом `pipe` дескрипторов файла, а записываются в конвейер с помощью системного вызова `write` с использованием второго дескриптора. Синтаксис системных вызов `read` и `write` тот же, что и при работе с обычными файлами.

Конвейер обеспечивает автоматическую синхронизацию процессов — если при использовании системного вызова `read` в буфере конвейера нет данных, то процесс, обратившийся к ОС с системным вызовом `read`, переводится в состояние ожидания и активизируется при появлении данных в буфере.

Механизм конвейеров доступен не только программистам, но и пользователям большинства современных операционных систем. Именно системные вызовы `pipe` задействуются оболочкой (командным процессором) операционной системы для организации конвейера команд, когда выходные данные одной команды пользователя становятся входными данными для другой команды. Примером такого конвейера команд может служить показанная ниже строка командного интерпретатора shell ОС Unix, которая передает выходные данные команды `ls` (чтение списка имен файлов текущего каталога) на вход команды `wc` (подсчет слов) с ключом `-l`:

```
ls | wc -l
```

Результатом работы этой командной строки будет количество файлов в текущем каталоге.

Именованные конвейеры

Именованные конвейеры представляют собой развитие механизма обычных конвейеров. Такие конвейеры имеют имя, которое является записью в каталоге файловой системы ОС, поэтому они пригодны для обмена данными между двумя произвольными процессами или потоками этих процессов.

Именованный конвейер является специальным файлом типа FIFO и не имеет области данных на диске. Создается именованный конвейер с помощью того же системного вызова, который используется и для создания файлов любого типа, но только с указанием в качестве типа файла параметра FIFO. Системный вызов порождает в каталоге запись о файле типа FIFO с заданным именем, после чего любой процесс может открыть этот файл и передавать данные другому процессу, также открывшему файл с этим именем.

Ввиду того что именованные конвейеры основаны на файловой системе, обычные конвейеры, создаваемые системным вызовом `pipe`, иногда называют программными конвейерами (software-pipes). Следует иметь в виду, что именованные конвейеры используют файловую систему только для хранения имени конвейера в каталоге, а данные между процессами передаются через буфер в оперативной памяти, как и в случае программного конвейера.

Очереди сообщений

Механизм **очередей сообщений** похож на механизм конвейеров с тем отличием, что он позволяет процессам и потокам обмениваться структурированными сообщениями. При этом синхронизация осуществляется по сообщениям, то есть процесс, пытающийся прочитать сообщение, переводится в состояние ожидания в том случае, если в очереди нет ни одного полного сообщения. Очереди сообщений являются глобальными средствами коммуникаций для процессов операционной системы, как и именованные конвейеры, так как каждая очередь

имеет в пределах ОС уникальное имя. В ОС Unix в качестве такого имени применяется числовое значение — так называемый ключ. Ключ является числовым аналогом имени файла, при использовании одного и того же значения ключа процессы будут работать с одной и той же очередью сообщений. Существует также функция, которая преобразует произвольное символьное имя в значение ключа, что позволяет программисту для указания уникальных очередей задействовать имена вместо трудно запоминаемых чисел.

Для работы с очередью сообщений процесс должен выполнить системный вызов `msgget`, указав в качестве параметра значение ключа. Если очередь с данным ключом в настоящий момент не используется ни одним процессом, то для нее резервируется область памяти, а затем процессу возвращается идентификатор очереди, который, как и дескриптор файла, имеет локальное для процесса значение. Если же очередь уже используется, то процессу просто возвращается ее идентификатор. Системный администратор может управлять параметрами операционной системы для изменения максимального объема памяти, отводимой очереди, а также максимального размера сообщения.

После открытия очереди процесс может помещать в него сообщения с помощью вызова `msgsnd` или читать сообщения с помощью вызова `msgrsv`. Программист может влиять на то, как ОС будет обрабатывать ситуацию, когда процесс пытается читать сообщения, которые еще не поступили в очередь, то есть на синхронизацию процесса с данными. При задании в системных вызовах `msgsnd` и `msgrcv` параметра `IPC_NOWAIT` операционная система в любом случае будет возвращать управление в вызывающий процесс, даже если он пытается прочитать несуществующее сообщение (в последнем случае в процесс возвращается код ошибки). Без этого параметра процесс при отсутствии данных переводится в состояние ожидания. Параметр `IPC_NOWAIT` используется не только в очередях сообщений, но и в некоторых других средствах IPC, например в семафорах. При применении параметра `IPC_NOWAIT` программист должен самостоятельно организовать ожидание данных.

Разделяемая память

Разделяемая память представляет собой сегмент физической памяти, отображенной на виртуальное адресное пространство двух или более процессов. Механизм разделяемой памяти поддерживается подсистемой виртуальной памяти, которая настраивает таблицы отображения адресов для процессов, запросивших разделение памяти, так что одни и те же адреса некоторой области физической памяти соответствуют виртуальным адресам разных процессов.

Выводы

- Механизм специальных файлов дает упрощенное представление устройства в виде неструктурированного набора байтов. Эта модель может непосредственно использоваться при работе с символьными устройствами или служить

универсальной основой для построения более сложной логической модели устройств других типов.

- В подсистеме ввода-вывода каждой современной операционной системы существуют стандарты на структуру драйверов и способ взаимодействия драйверов с приложениями и остальными модулями ОС, а также друг с другом. Наличие таких стандартов упрощает разработку новых драйверов независимыми производителями и повышает стабильность ОС.
- Стандарт на структуру драйвера определяет состав и назначение основных процедур драйвера, в том числе инициализации, старта операции, чтения и записи данных, обработки прерываний и завершения операции.
- Процедура обработки прерываний драйвера работает в общем случае в контексте процесса, отличающегося от того, для которого выполняется операция, поэтому ей запрещается пользоваться ресурсами текущего процесса или влиять на их распределение.
- Механизм отображаемых на память файлов является удобным для программиста средством доступа к файлам, при котором обращение к данным файла осуществляется тем же способом, что и к переменным, находящимся в оперативной памяти. Большую часть работы по отображению файлов на память выполняют стандартные модули подсистемы виртуальной памяти ОС, поэтому данный механизм очень экономичен в отношении реализации.
- Для ускорения обмена данными с дисковыми накопителями в операционных системах используется дисковый кэш, который располагается между слоем драйверов файловых систем и блок-ориентированными драйверами дисков.
- Негативным последствием кэширования диска может быть потеря при сбое системы или отключении питания той части кэшируемых данных, которая не успела переписаться на диск. Для борьбы с этим явлением периодически осуществляется принудительное выталкивание данных кэша на диск, используются также восстанавливаемые файловые системы.
- Для организации дискового кэша в большинстве современных ОС с помощью подсистемы виртуальной памяти происходит отображение частей файлов, к которым происходит обращение, на системную область памяти.
- Восстанавливаемость файловой системы — это свойство, которое гарантирует, что в случае отказа питания или краха системы, когда все данные в оперативной памяти безвозвратно теряются, все начатые файловые операции будут либо успешно завершены, либо отменены без всяких отрицательных последствий для работоспособности файловой системы. Это означает, что операции с диском рассматриваются как транзакции, которые протоколируются в специальном журнале.
- Свойство восстанавливаемости может распространяться на данные двух типов: пользовательские данные и служебную информацию файловой системы. В некоторых файловых системах, например NTFS, поддерживается только

восстанавливаемость служебной информации, что обеспечивает работоспособность и непротиворечивость файловой системы после сбоев, но не гарантирует полной сохранности данных в файлах пользователей.

- Для обеспечения устойчивости хранимых данных к отказам самих дисков в вычислительных системах применяются избыточные дисковые массивы, известные под названием RAID. При отказе одного из дисков для восстановления его данных используются данные, хранящиеся на оставшихся дисках массива.
- Существует несколько уровней RAID (наиболее часто применяемые — RAID-0, RAID-1, RAID-3 и RAID-5), отличающихся степенью избыточности хранимой информации, производительностью операций чтения и записи, а также степенью отказоустойчивости.
- В операционных системах существуют специальные средства межпроцессного обмена, которые позволяют преодолеть защитные барьеры, разделяющие адресные пространства различных процессов. К этим средствам относятся простые и именованные конвейеры, очереди сообщений, разделяемые сегменты памяти и ряд других механизмов.

Задачи и упражнения

1. На какой стадии операция записи данных в специальный файл начинает отличаться от операций записи в дисковый файл в ОС Unix?
2. Если при поступлении запроса от приложения к файлу система обнаруживает требуемые данные в системном буфере (кэше), то время доступа приложения к нужным ему данным будет:
   1) таким же, как время доступа к его внутренним переменным;
   2) немного больше, чем время доступа к его внутренним переменным;
   3) таким же, как время доступа к данным на диске.
3. Какая секция блок-ориентированного драйвера ОС Unix выполняет вывод данных?
4. Какие преимущества связаны с включением в модель драйвера большого количества секций различного типа?
5. Чем отличаются функции `nodev` и `nulldev` драйвера ОС Unix?
6. Чем принципиально отличается отображение файла на память от кэширования файла с помощью средств менеджера виртуальной памяти?
7. Все ли типы файлов можно отображать на память в ОС Unix?
8. В каких ситуациях целесообразно использовать асинхронные операции записи в файл?
9. Какие дополнительные меры должны предприниматься при восстановлении файловой системы при наличии дискового кэша?
10. Какие параметры операции с файлом фиксируются в журнале транзакций?

11. Восстанавливаются ли пользовательские данные в NTFS?
12. Из каких двух частей состоит запись о модификации в журнале транзакций ОС семейства Windows NT?
13. Можно ли организовать дисковый массив RAID без специального контроллера?
14. В чем преимущество дисковых массивов RAID-0 по сравнению с обычными дисками?
15. Скорость какого типа операций повышается при использовании дисковых массивов RAID-1?
16. Что такое «динамическое восстановление данных»?
17. В каких случаях обмен данными между процессами можно выполнить только с помощью именованных конвейеров?

Глава 9

Сеть как транспортная система

В главе 2 мы рассмотрели структуру сетевых средств ОС и отметили, что их можно разделить на два уровня: сетевые службы (клиенты и серверы) и транспортные средства ОС. Сетевые службы предоставляют пользователям компьютеров такие сервисы, как доступ к файлам и веб-страницам, размещенным на удаленных серверах, обмен почтовыми сообщениями между компьютерами сети, прослушивание радиопрограмм через Интернет и многое другое. Для того чтобы сетевые клиенты и серверы могли общаться между собой, они обращаются к еще одному компоненту сетевой ОС, а именно — к сетевым транспортным средствам ОС.

В этой главе мы рассмотрим организацию сетевых транспортных средств ОС, в том числе:

- принцип коммутации пакетов, позволяющий эффективно передавать через каналы связи пульсирующий трафик компьютеров;
- семиуровневую модель OSI, стандартизующую многоуровневый подход к организации транспортных протоколов;
- организацию стека протоколов TCP/IP, являющихся основой Интернета и любых современных составных сетей;
- основы технологии Ethernet, доминирующей в области построения локальных сетей.

Во второй части главы рассматривается реализация этих принципов в транспортных средствах универсальных ОС, а также основные особенности операционных систем маршрутизаторов на примере Cisco IOS.

Роль сетевых транспортных средств ОС

Сетевые транспортные средства операционной системы передают сообщения между компьютерами через сеть. Современные сети представляют собой сложную систему: они состоят из большого количества подсетей, каждая из которых

в общем случае построена на коммуникационном оборудовании различного типа, использующем разнообразные сетевые технологии и объединенном коммуникационными каналами разного типа по различной топологии. Интернет является наиболее ярким примером такой сети, объединяющей огромное количество подсетей по всему миру.

Транспортные сетевые средства ОС должны экранировать серверные и клиентские компоненты ОС от всех деталей сети и обеспечивать эти высокоуровневые компоненты ОС стабильными и простыми процедурами передачи сообщений, работающими одинаково хорошо и в небольшой локальной сети, и в крупной корпоративной сети, отдельные подсети которой соединены через Интернет.

Для понимания принципов построения транспортных средств сетевых ОС необходимо знать основы современных сетевых технологий. Собственно, *транспортные средства ОС отдельного компьютера* являются интегральной частью коммуникационных средств компьютерной сети, которые помимо конечных узлов сети — компьютеров — включают промежуточные узлы, такие как маршрутизаторы и коммутаторы. Маршрутизаторы и коммутаторы компьютерной сети работают под управлением собственного программного обеспечения, которое во многих случаях имеет достаточно сложную организацию, что дает право называть его *специализированными сетевыми ОС*.

Совокупность сетевых средств универсальных ОС компьютеров, являющихся узлами сети, и сетевых средств специализированных ОС, установленных на маршрутизаторах и коммутаторах, можно рассматривать как единую программную коммуникационную систему, которая обеспечивает информационные связи пользователей и приложений в сети.

Коммутация пакетов

Пакеты

Современные компьютерные сети работают на основе техники **коммутации пакетов**. Эта техника была специально разработана для эффективной передачи компьютерного трафика. Телефонные сети, которые до появления компьютеров были основным видом телекоммуникационных сетей, используют другую технику коммутации, называемую **коммутацией каналов**. Мы не будем вдаваться в ее детали, но в данном контексте главным является то, что в телефонной сети между терминальным оборудованием — телефонами — создается составной канал с *фиксированной пропускной способностью*, которая выделяется на время соединения именно этой паре терминалов. Такой принцип отвечает особенностям телефонии. Например, современный цифровой телефон порождает постоянный поток битов, передаваемых со скоростью 64 Кбит/с, поэтому в цифровых телефонных сетях между абонентами создаются каналы с фиксированной пропускной способностью в 64 Кбит/с, и эта пропускная способность полностью используется.

Эксперименты по созданию первых компьютерных сетей на основе традиционных телефонных каналов показали, что этот вид связи не позволяет достичь высокой общей производительности сети из-за *высокого уровня пульсации компьютерных данных*. Действительно, работа пользователя, который сканирует веб-ресурсы Интернета, очевидным образом порождает неравномерный трафик. При загрузке очередной страницы в компьютер скорость резко возрастает, а после окончания загрузки падает практически до нуля.

Коэффициент пульсации трафика отдельного пользователя сети определяется как отношение пиковой скорости на каком-либо небольшом интервале времени к средней скорости обмена данными на длительном интервале времени и может достигать значений 100:1. Если для описанного веб-сеанса организовать коммутацию канала между компьютером пользователя и сервером, то большую часть времени канал будет простаивать. В то же время часть производительности сети останется закрепленной за данной парой абонентов и недоступной другим пользователям сети. Сеть в такие периоды похожа на пустой эскалатор метро, который движется, но полезную работу не делает, другими словами, «перевозит воздух». Поэтому для компьютерных сетей и был предложен и воплощен в жизнь принципиально иной принцип коммутации, который получил название коммутации пакетов.

При коммутации пакетов все передаваемые пользователем сети данные разбиваются в исходном узле на сравнительно небольшие части, называемые **пакетами**, **кадрами** или **ячейками**, — в данном контексте различия в значении этих терминов не существенны (рис. 9.1). Каждый пакет снабжается **заголовком**, в котором указывается адрес, необходимый для доставки пакета узлу назначения. Наличие адреса в каждом пакете является одним из важнейших свойств техники коммутации пакетов, так как каждый пакет может быть обработан коммутатором независимо от других пакетов информационного потока. Помимо заголовка у пакета имеется еще одно дополнительное поле, которое обычно размещается в конце пакета и поэтому называется **концевиком**. В концевике помещается **контрольная сумма**, которая позволяет проверить, была ли искажена информация при передаче через сеть или нет.

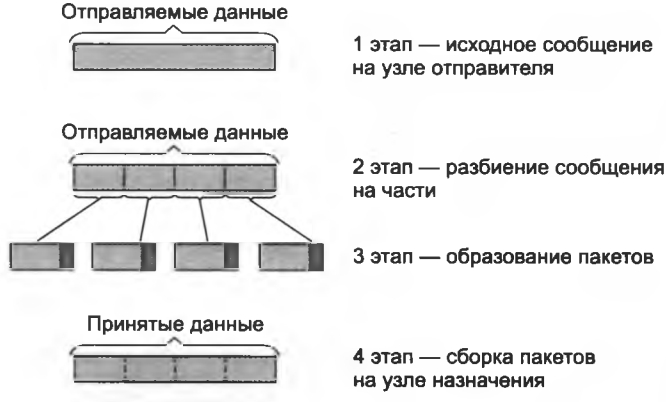


Рис. 9.1. Разбиение потока данных на пакеты

Пакеты поступают в сеть *без предварительного резервирования линий связи* и *не с фиксированной заданной скоростью*, как это происходит в сетях с коммутацией каналов, а в том темпе, в котором их генерирует источник. Предполагается, что сеть с коммутацией пакетов, в отличие от сети с коммутацией каналов, всегда готова принять пакет от конечного узла.

Буферы и очереди

Сеть с коммутацией пакетов так же, как и сеть с коммутацией каналов, состоит из коммутаторов, связанных физическими линиями связи. Однако коммутаторы функционируют в этих сетях по-разному. Главное отличие состоит в том, что коммутаторы в сетях с коммутацией пакетов, которые далее мы будем называть **коммутаторами пакетов**, имеют *внутреннюю буферную память* для временного хранения пакетов. Действительно, коммутатор пакетов не может принять решения о продвижении пакета, не имея в своей памяти всего пакета. Коммутатор проверяет контрольную сумму, и только если она говорит о том, что данные пакета не искажены, начинает обрабатывать пакет и по адресу назначения определяет следующий коммутатор. Поэтому каждый пакет последовательно бит за битом помещается во **входной буфер**. Имея в виду это свойство, говорят, что сети с коммутацией пакетов используют технику **сохранения с продвижением** (store-and-forward). Заметим, что для этой цели достаточно иметь буфер размером в один пакет.

Буферизация необходима коммутатору пакетов также для согласования скорости поступления пакетов со скоростью их коммутации. Если коммутирующий блок не успевает обрабатывать пакеты, то на интерфейсах коммутатора возникают **входные очереди**. Очевидно, что для хранения входной очереди объем буфера должен превышать размер одного пакета. Существуют различные подходы к построению коммутирующего блока. Традиционный способ основан на одном центральном процессоре, который обслуживает все входные очереди коммутатора. Такой способ построения может приводить к большим очередям, так как производительность процессора разделяется между несколькими очередями. Современные подходы к построению коммутирующего блока подразумевают многопроцессорность, когда каждый интерфейс имеет свой встроенный процессор для обработки пакетов. Кроме того, существует также центральный процессор, координирующий работу интерфейсных процессоров. Использование интерфейсных процессоров повышает производительность коммутатора и уменьшает очереди на входных интерфейсах. Однако такие очереди все равно могут возникать, так как центральный процессор по-прежнему остается «узким местом».

Наконец, буферы нужны для согласования скоростей передачи данных в каналах, подключенных к коммутатору пакетов[1]. Действительно, если скорость

[1] Здесь термин «коммутатор пакетов» понимается в широком смысле — как любое устройство, реализующее коммутацию пакетов, то есть он в равной степени относится и к IP-маршрутизатору, Ethernet-коммутатору, и к ATM-коммутатору, а также коммутаторам любых других сетевых технологий.

поступления пакетов из одного канала в течение некоторого периода превышает пропускную способность того канала, в который эти пакеты должны быть направлены, то во избежание потерь пакетов на целевом интерфейсе необходимо организовать **выходную очередь** (рис. 9.2).

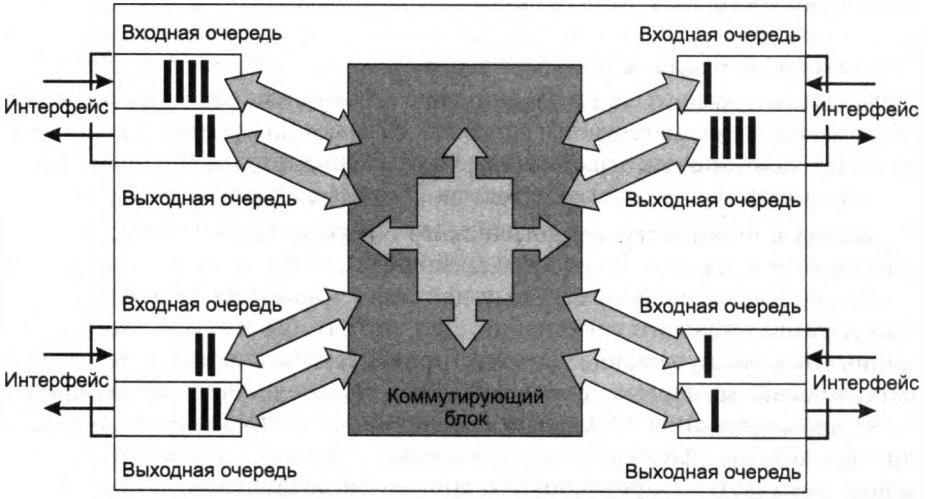


Рис. 9.2. Очереди в пакетном коммутаторе

В сети с коммутацией пакетов пульсации трафика отдельных абонентов в соответствии с законом больших чисел распределяются во времени так, что их пики чаще всего не совпадают. Поэтому коммутаторы постоянно и достаточно равномерно загружены работой, если число обслуживаемых ими абонентов действительно велико. Обычно коммутаторы более высокого уровня иерархии, которые обслуживают соединения между коммутаторами нижнего уровня, загружены более равномерно, и магистральные каналы, соединяющие коммутаторы верхнего уровня, имеют близкие к максимальным коэффициенты использования. Буферизация сглаживает пульсации, поэтому коэффициент пульсации на магистральных каналах гораздо ниже, чем на каналах абонентского доступа.

Поскольку объем буферов в коммутаторах ограничен, иногда происходит потеря пакетов из-за **переполнения** буферов при временной **перегрузке** части сети, когда совпадают периоды пульсации нескольких информационных потоков. Так как потеря пакетов является неотъемлемым свойством сети с коммутацией пакетов, то для нормальной работы таких сетей разработан ряд механизмов, которые компенсируют этот эффект, например, можно выполнять повторную пересылку пакета при истечении тайм-аута подтверждения факта его получения узлом назначения.

Методы продвижения пакетов

Решение о том, на какой интерфейс передать пришедший пакет, принимается на основании одного из трех методов продвижения пакетов.

- При **дейтаграммной передаче** соединение не устанавливается, и все передаваемые пакеты *продвигаются* (передаются от одного узла сети другому) *независимо* друг от друга на основании одних и тех же правил. Процедура обработки пакета определяется только значениями параметров, которые он несет в себе, и текущим состоянием сети (например, в зависимости от ее нагрузки пакет может стоять в очереди на обслуживание большее или меньшее время). Однако никакая информация об уже переданных пакетах сетью не хранится и в ходе обработки очередного пакета во внимание не принимается. То есть каждый отдельный пакет рассматривается сетью как совершенно независимая единица передачи — **дейтаграмма**. Примерами дейтаграммных технологий являются такие доминирующие сегодня технологии, как Ethernet и IP (см. далее).
- **Передача с установлением логического соединения** распадается на так называемые сеансы, или логические соединения. Процедура обработки определяется не для отдельного пакета, а для всего множества пакетов, передаваемых в рамках каждого соединения. Для того чтобы реализовать дифференцированное обслуживание пакетов, принадлежащих разным соединениям, сеть должна, во-первых, присвоить каждому соединению **идентификатор**, во-вторых, запомнить параметры соединения, то есть значения, определяющие процедуру обработки пакетов в рамках данного соединения. Эта информация называется **информацией о состоянии соединения**. Фиксированный маршрут не является обязательным параметром соединения. Пакеты, принадлежащие одному и тому же соединению, даже имеющие одни и те же адреса отправления и назначения, могут перемещаться по разным независимым друг от друга маршрутам. Протокол IP, который в сетях TCP/IP отвечает за надежную передачу пакетов между конечными узлами сети, является примером передачи с установлением логического соединения.
- **Передача с установлением виртуального канала.** Если в число параметров соединения *входит* маршрут, то все пакеты, передаваемые в рамках данного соединения, должны проходить по указанному пути. Такой единственный заранее проложенный фиксированный маршрут, соединяющий конечные узлы в сети с коммутацией пакетов, называют **виртуальным каналом** (virtual circuit, или virtual channel). Техника виртуальных каналов используется в технологиях Frame Relay и ATM.

Протоколы, модель OSI и стек протоколов TCP/IP

Протокол и стек протоколов

Организация взаимодействия между устройствами сети является сложной задачей. Для решения сложных задач используется известный универсальный прием — **декомпозиция**, то есть разбиение одной сложной задачи на несколько более простых задач-модулей. Декомпозиция состоит в четком определении функций каждого модуля, а также порядка их взаимодействия (то есть межмодульных интерфейсов).

В сетях применяют еще одну эффективную концепцию, развивающую идею декомпозиции, а именно **многоуровневый подход**. После представления исходной задачи в виде множества модулей эти модули группируют и упорядочивают по уровням, образующим иерархию.

Многоуровневое представление средств сетевого взаимодействия имеет свою специфику, связанную с тем, что в процессе обмена сообщениями участвуют, по меньшей мере, *две стороны*, то есть в данном случае необходимо организовать согласованную работу двух иерархий аппаратных и программных средств, работающих на разных компьютерах. Оба участника сетевого обмена должны принять множество соглашений. Например, они должны согласовать уровни и форму электрических сигналов, способ определения размера сообщений, договориться о методах контроля достоверности и т. п. Другими словами, соглашения должны быть приняты на всех уровнях, начиная от самого низкого — уровня передачи битов, и заканчивая самым высоким, реализующим обслуживание пользователей сети.

На рис. 9.3 показана модель взаимодействия двух узлов. С каждой стороны средства взаимодействия представлены четырьмя уровнями. Каждый уровень поддерживает интерфейсы двух типов. Во-первых, это интерфейсы с вышеи нижележащим уровнями «своей» иерархии средств. Во-вторых, это интерфейс со средствами взаимодействия другой стороны, расположенными на том же уровне иерархии. Этот тип интерфейса называют **протоколом**. Таким образом, протокол всегда является одноранговым интерфейсом.

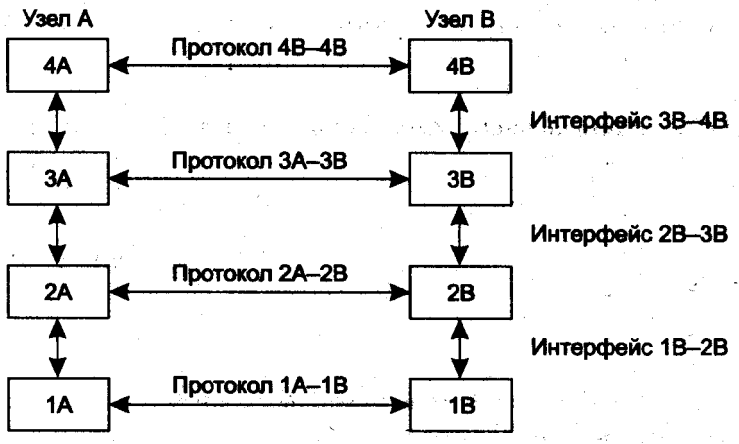


Рис. 9.3. Взаимодействие двух узлов

ПРИМЕЧАНИЕ В сущности, термины «протокол» и «интерфейс» выражают одно и то же понятие — формализованное описание процедуры взаимодействия двух объектов, но традиционно в сетях за ними закрепили разные области действия: протоколы определяют правила взаимодействия модулей одного уровня в разных узлах, а интерфейсы — правила взаимодействия модулей соседних уровней в одном узле.

Иерархически организованный набор протоколов, достаточный для организации взаимодействия узлов в сети, называется **стеком протоколов**.

Протоколы нижних уровней часто реализуются комбинацией программных и аппаратных средств, а протоколы верхних уровней, как правило, программными средствами.

Программный модуль, реализующий некоторый протокол, называют протокольной сущностью, или, для краткости, тоже протоколом. Понятно, что один и тот же протокол может быть реализован с разной степенью эффективности. Именно поэтому при сравнении протоколов следует учитывать не только логику их работы, но и качество программной реализации. Более того, на эффективность взаимодействия устройств в сети влияет качество всей совокупности протоколов, составляющих стек, в частности то, насколько рационально распределены функции между протоколами разных уровней и насколько хорошо определены интерфейсы между ними.

Протокольные сущности одного уровня двух взаимодействующих сторон обмениваются сообщениями в соответствии с определенным для них протоколом. Сообщения состоят из заголовка и поля данных (иногда оно может отсутствовать). Обмен сообщениями является своеобразным языком общения, с помощью которого каждая из сторон «объясняет» другой стороне, что необходимо сделать на каждом этапе взаимодействия. Работа каждого протокольного модуля состоит в интерпретации заголовков поступающих к нему сообщений и выполнении связанных с этим действий. Заголовки сообщений разных протоколов имеют разную структуру, что соответствует различиям в их функциональности. Понятно, что чем сложнее структура заголовка сообщения, тем более сложные функции возложены на соответствующий протокол.

Семиуровневая модель OSI

Из того, что протокол является соглашением, принятым двумя взаимодействующими узлами сети, совсем не следует, что он обязательно является стандартным. Но на практике при реализации сетей стремятся использовать стандартные протоколы. Это могут быть фирменные, национальные или международные стандарты.

В начале 80-х годов ряд международных организаций по стандартизации, в частности International Organization for Standardization (ISO), разработали стандартную модель **взаимодействия открытых систем** (Open System Interconnection, OSI). Эта модель сыграла значительную роль в развитии компьютерных сетей.

К концу 70-х годов в мире уже существовало большое количество фирменных стеков коммуникационных протоколов, среди которых можно назвать, например, такие популярные стеки, как DECnet, TCP/IP и SNA. Такое разнообразие средств межсетевого взаимодействия вывело на первый план проблему несовместимости устройств, использующих разные протоколы. Одним из путей разрешения этой проблемы в то время виделся всеобщий переход на единый, общий для всех систем стек протоколов, созданный с учетом недостатков уже

существующих стеков. Именно с такой целью и была разработана модель OSI. Эта модель, в обобщенном виде представляющая средства сетевого взаимодействия, стала своего рода универсальным языком сетевых специалистов и именно поэтому ее называют справочной моделью.

Модель OSI определяет, во-первых, уровни взаимодействия систем в сетях с коммутацией пакетов, во-вторых, стандартные названия уровней, в-третьих, функции, которые должен выполнять каждый уровень. Модель OSI не содержит описаний реализации конкретного набора протоколов.

В модели OSI средства взаимодействия делятся на *семь* уровней: прикладной, представления, сеансовый, транспортный, сетевой, канальный и физический (рис. 9.4). Каждый уровень имеет дело с совершенно определенным аспектом взаимодействия сетевых устройств.

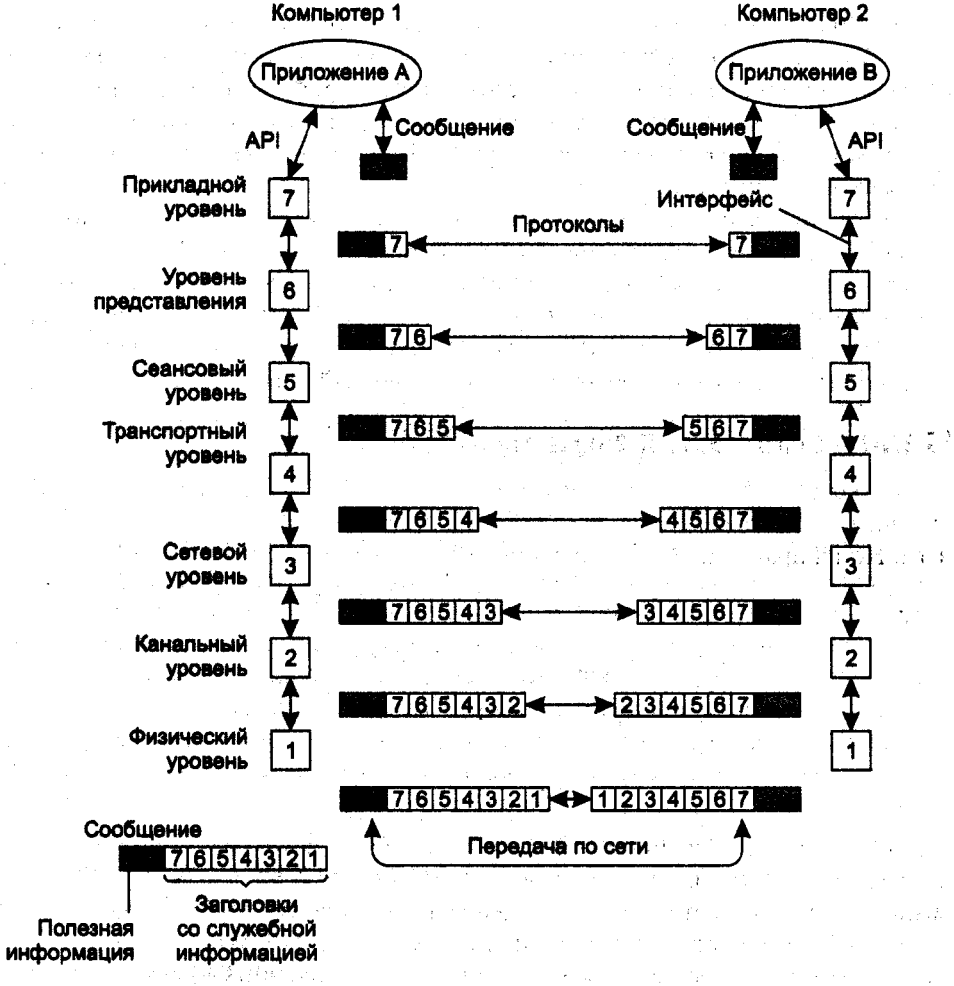


Рис. 9.4. Модель взаимодействия открытых систем ISO/OSI

Модель OSI описывает только *системные* средства взаимодействия, реализуемые операционной системой, системными утилитами, системными устройствами. Модель не включает средства взаимодействия по сети приложений конечных пользователей. Важно различать уровень взаимодействия приложений и прикладной уровень семиуровневой модели.

Приложения могут реализовывать собственные протоколы взаимодействия, используя для этих целей многоуровневую совокупность системных средств. Именно для этого в распоряжение программистов предоставляется **прикладной программный интерфейс** (Application Program Interface, API). В соответствии с идеальной схемой модели OSI приложение может обращаться с запросами к самому верхнему уровню — прикладному, однако на практике многие стеки коммуникационных протоколов предоставляют возможность программистам напрямую обращаться к сервисам, или службам, нижележащих уровней.

Например, некоторые СУБД имеют встроенные средства удаленного доступа к файлам. В этом случае приложение, выполняя доступ к удаленным ресурсам, не использует системную файловую службу; оно обходит верхние уровни модели OSI и обращается непосредственно к ответственным за транспортировку сообщений по сети системным средствам, которые располагаются на нижних уровнях модели OSI.

Итак, пусть приложение узла *A* хочет взаимодействовать с приложением узла *B*. Для этого приложение *A* обращается с запросом к *прикладному уровню*, например к файловой службе. На основании этого запроса программное обеспечение прикладного уровня формирует сообщение стандартного формата. Но для того чтобы доставить эту информацию по назначению, предстоит решить еще много задач, ответственность за которые несут нижележащие уровни.

После формирования сообщения прикладной уровень направляет его вниз по стеку *уровню представления*. Протокол уровня представления на основании информации, полученной из заголовка сообщения прикладного уровня, выполняет требуемые действия и добавляет к сообщению собственную служебную информацию — заголовок уровня представления, в котором содержатся указания для протокола уровня представления машины-адресата. Полученное в результате сообщение передается вниз *сеансовому уровню*, который, в свою очередь, добавляет свой заголовок и т. д. (Некоторые реализации протоколов помещают служебную информацию не только в начале сообщения в виде заголовка, но и в конце в виде концевика.) Наконец, сообщение достигает нижнего, *физического уровня*, который собственно и передает его по линиям связи машине-адресату. К этому моменту сообщение «обрастает» заголовками всех уровней (рис. 9.5).

Физический уровень помещает сообщение на физический выходной интерфейс компьютера 1 (см. рис. 9.4), и оно начинает свое «путешествие» по сети (до этого момента сообщение передавалось от одного уровню другому в пределах компьютера 1).

Когда сообщение по сети поступает на входной интерфейс компьютера 2, оно принимается его физическим уровнем и последовательно перемещается

вверх с уровня на уровень. Каждый уровень анализирует и обрабатывает заголовок своего уровня, выполняя соответствующие функции, а затем удаляет этот заголовок и передает сообщение вышележащему уровню.

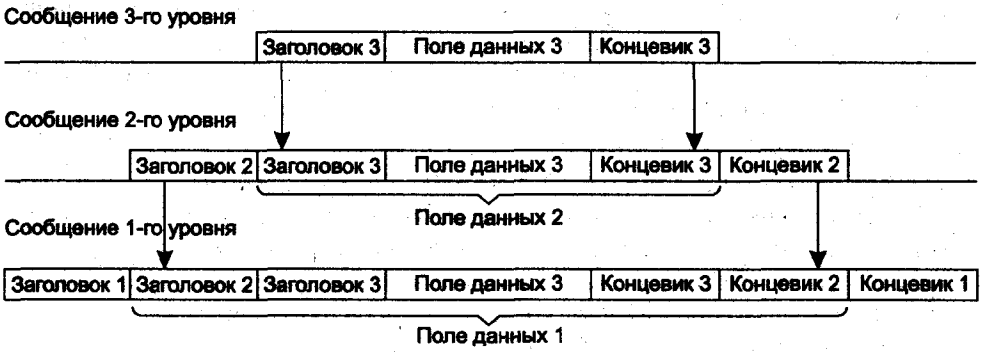


Рис. 9.5. Вложенность сообщений различных уровней

Как видно из описания, протокольные сущности одного уровня не общаются между собой непосредственно, в этом общении всегда участвуют посредники — средства протоколов нижележащих уровней. И только физические уровни различных узлов взаимодействуют непосредственно.

В стандартах ISO для обозначения единиц обмена данными, с которыми имеют дело протоколы разных уровней, используется общее название **протокольная единица данных** (Protocol Data Unit, PDU). Для обозначения единиц обмена данными конкретных уровней часто применяют *специальные* названия, в частности **сообщение**, **кадр**, **пакет**, **дейтаграмма**, **сегмент**.

Физический уровень

Физический уровень (physical layer) имеет дело с передачей потока битов по физическим каналам связи, таким как коаксиальный кабель, витая пара, оптоволоконный кабель или цифровой территориальный канал.

Функции физического уровня реализуются на всех устройствах, подключенных к сети. Со стороны компьютера функции физического уровня выполняются сетевым адаптером или последовательным портом.

Физический уровень не вникает в смысл информации, которую он передает. Для него эта информация представляет собой однородный поток битов, которые нужно доставить без искажений и в соответствии с заданной тактовой частотой (интервалом между соседними битами).

Канальный уровень

Канальный уровень (data link layer) является первым уровнем (если идти снизу вверх), который работает в режиме коммутации пакетов. На этом уровне PDU обычно носит название **кадр** (frame).

Функции средств канального уровня определяются по-разному для локальных и глобальных сетей.

- В **локальных сетях** (Local Area Networks, LANs) канальный уровень должен обеспечивать доставку кадра между *любыми* узлами сети. При этом предполагается, что сеть имеет типовую топологию, например, общая шина, кольцо, звезда или дерево (иерархическая звезда). Примерами технологий локальных сетей, применение которых ограничено типовыми топологиями, являются Ethernet, FDDI, Token Ring.
- В **глобальных сетях** (Wide Area Networks, WANs) канальный уровень должен обеспечивать доставку кадра только между двумя *соседними* узлами, соединенными индивидуальной линией связи. Примерами двухточечных протоколов (как часто называют эти протоколы) могут служить широко распространенные протоколы PPP и HDLC. На основе двухточечных связей могут быть построены сети произвольной топологии.

Для связи локальных сетей между собой или для доставки сообщений между любыми конечными узлами глобальной сети используются средства более высокого сетевого уровня.

Одной из функций канального уровня является поддержание интерфейсов с нижележащим физическим уровнем и вышележащим сетевым уровнем. Сетевой уровень направляет канальному уровню пакет для передачи в сеть или принимает от него пакет, полученный из сети. Физический уровень используется канальным как инструмент, который принимает и передает в сеть последовательности битов.

Рассмотрим работу канального уровня, начиная с момента, когда сетевой уровень отправителя передает канальному уровню пакет, а также указание, какому узлу его передать. Для решения этой задачи канальный уровень создает кадр, который имеет поле данных и заголовок. Канальный уровень помещает (*инкапсулирует*) пакет в поле данных кадра и заполняет соответствующей служебной информацией заголовок кадра. Важнейшей информацией заголовка кадра является адрес назначения, на основании которого коммутаторы сети будут продвигать пакет.

Одной из задач канального уровня является *обнаружение и коррекция ошибок*. Для этого канальный уровень фиксирует границы кадра, помещая специальную последовательность битов в его начало и конец, а затем добавляет к кадру контрольную сумму, которая называется также **контрольной последовательностью кадра** (Frame Check Sequence, FCS). Контрольная сумма вычисляется по некоторому алгоритму как функция от всех байтов кадра. По FCS узел назначения сможет определить, были или нет искажены данные кадра в процессе передачи по сети.

Однако прежде, чем переправить кадр физическому уровню для непосредственной передачи данных в сеть, канальному уровню может потребоваться решить еще одну важную задачу. Если в сети используется разделяемая среда, то прежде чем физический уровень начнет передавать данные, канальный уровень

должен *проверить доступность среды*. Функции проверки доступности разделяемой среды иногда выделяют в отдельный подуровень **управления доступом к среде** (Medium Access Control, MAC).

Если разделяемая среда освободилась (при отсутствии разделения такая проверка, конечно, пропускается), кадр передается средствами физического уровня в сеть, проходит по каналу связи и поступает в виде последовательности битов в распоряжение физического уровня узла назначения.

Чтобы кадр дошел до узла назначения, он должен включать адрес этого узла. В локальных сетях такой адрес называют **MAC-адресом**. В глобальных сетях адрес канального уровня обычно не имеет большого значения, так как в двухточечном соединении кадр доходит до узла назначения и без указания адреса.

MAC-адрес предназначен для однозначной идентификации сетевых интерфейсов в локальных сетях, он имеет 6-байтный формат. MAC-адреса являются глобально уникальными, каждый производитель оборудования имеет свое уникальное значение 3-байтного префикса адреса, обеспечивая уникальность значений трех младших байтов самостоятельно. MAC-адреса уникальны независимо от технологии, которую они поддерживают, то есть MAC-адрес интерфейса Ethernet всегда имеет свое уникальное значение не только среди всех интерфейсов Ethernet, но и интерфейсов Token Ring и любой другой технологии локальной сети.

MAC-адрес является плоским адресом, то есть он не структурирован как, например, почтовый адрес, включающий поля названий города, улицы и номеров дома и квартиры. MAC-адрес обычно используется только аппаратурой, поэтому его стараются сделать по возможности компактным и записывают в виде двоичного или шестнадцатеричного числа, например 0081005e24a8. При задании MAC-адресов не требуется ручной работы, так как они обычно встраиваются в аппаратуру компанией-изготовителем, поэтому их называют также **аппаратными адресами** (hardware addresses). Использование плоских адресов является жестким решением — при замене аппаратуры, например сетевого адаптера, изменяется и адрес сетевого интерфейса компьютера. На MAC-уровне поддерживается также специальный широковещательный адрес FFFFFFFFFFFF, состоящий из двоичных единиц во всех разрядах. Кадры, имеющие такой адрес, направляются всем компьютерам сети.

После получения битов кадра из сети физический уровень узла назначения передает их «наверх» канальному уровню своего узла. Последний группирует биты в кадры, снова вычисляет контрольную сумму полученных данных и сравнивает результат с контрольной суммой, переданной в кадре. Если они совпадают, кадр считается правильным. Если же контрольные суммы не совпадают, фиксируется ошибка. В функции канального уровня входит не только обнаружение ошибок, но и их исправление за счет повторной передачи поврежденных кадров. Однако эта функция не является обязательной и в некоторых реализациях канального уровня она не поддерживается, например в Ethernet, Token Ring, FDDI и Frame Relay.

Протоколы канального уровня реализуются компьютерами, мостами, коммутаторами и маршрутизаторами. В компьютерах функции канального уровня реализуются совместными усилиями сетевых адаптеров и их драйверов.

Протокол канального уровня обычно работает в пределах сети, являющейся одной из сетей более крупной составной сети, объединенной протоколами сетевого уровня. Адреса, с которыми работает протокол канального уровня, используются для доставки кадров только в пределах этой сети, а для перемещения пакетов между сетями применяются уже адреса следующего, сетевого, уровня.

В локальных сетях канальный уровень поддерживает весьма мощный и законченный набор функций по пересылке сообщений между узлами сети. В некоторых случаях протоколы канального уровня локальных сетей оказываются самодостаточными транспортными средствами и могут допускать работу непосредственно поверх себя протоколов прикладного уровня или приложений без привлечения средств сетевого и транспортного уровней. Тем не менее для качественной передачи сообщений в сетях с произвольной топологией функций канального уровня оказывается недостаточно.

Это утверждение в еще большей степени справедливо для глобальных сетей, в которых протокол канального уровня реализуют достаточно простую функцию передачи данных между соседними узлами.

Сетевой уровень

Сетевой уровень (network layer) служит для образования единой транспортной системы, объединяющей несколько сетей и называемой **составной сетью**, или **интернетом**[1].

Технология, позволяющая соединять в единую сеть множество сетей, в общем случае построенных на основе разных технологий, называется технологией **межсетевого взаимодействия** (internetworking).

На рис. 9.6 показано несколько сетей, каждая из которых использует собственную технологию канального уровня: Ethernet, FDDI, Token Ring, ATM, Frame Relay. На базе этих технологий каждая из указанных сетей может связывать между собой любых пользователей, но только *своей* сети и не способна обеспечить передачу данных в другую сеть.

Причина такого положения вещей очевидна и кроется в существенных отличиях одной технологии от другой. Например, технологии LAN (включая Ethernet, FDDI, Token Ring), имеющие одну и ту же систему адресации (адреса подуровня MAC, называемые MAC-адресами), отличаются друг от друга форматом используемых кадров и логикой работы протоколов. Еще больше отличий между технологиями LAN и WAN. Во многих технологиях WAN задействована техника предварительно устанавливаемых виртуальных каналов, идентифика-

[1] Не следует путать интернет (со строчной буквы) с Интернетом (с прописной буквы); Интернет — это самая известная и охватывающая весь мир реализация составной сети, построенная на основе технологии TCP/IP.

торы которых применяются в качестве адресов. Все технологии имеют собственные форматы кадров (в технологии ATM кадр даже называется иначе — ячейкой) и, конечно, собственные стеки протоколов.

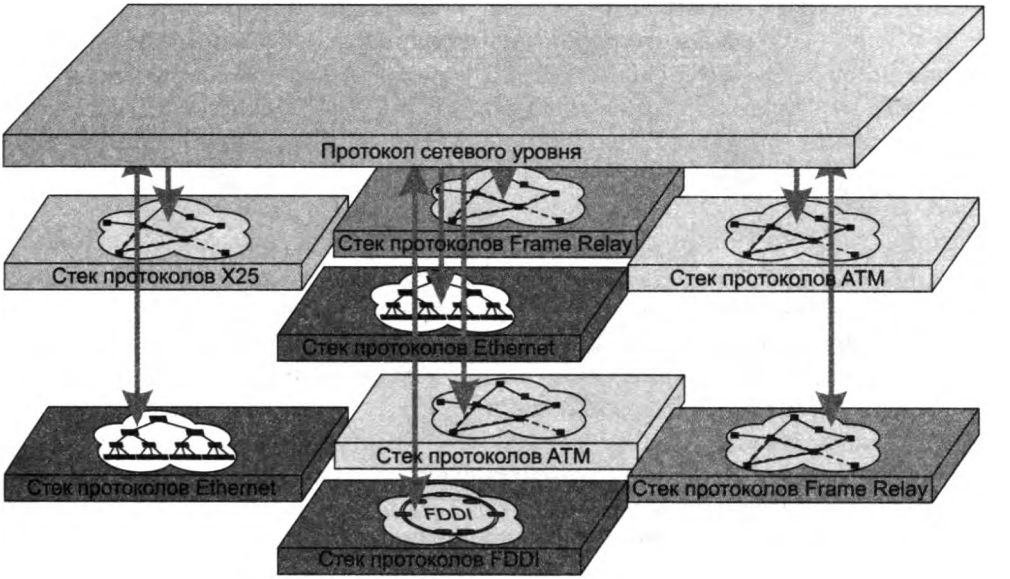


Рис. 9.6. Необходимость сетевого уровня

Чтобы связать между собой сети, построенные на основе столь отличающихся технологий, нужны *дополнительные средства*, и такие средства предоставляет сетевой уровень.

Функции сетевого уровня реализуются:

- группой протоколов;
- специальными устройствами — **маршрутизаторами**.

Одной из функций маршрутизатора является *физическое соединение сетей*. Маршрутизатор имеет несколько сетевых интерфейсов, подобных интерфейсам компьютера, к каждому из которых может быть подключена одна сеть. Таким образом, все интерфейсы маршрутизатора можно считать узлами разных сетей. Маршрутизатор может быть реализован программно на базе универсального компьютера (например, типовая конфигурация Unix или Windows включает программный модуль маршрутизатора). Однако чаще маршрутизаторы реализуются на базе специализированных аппаратных платформ. В состав программного обеспечения маршрутизатора входят протокольные модули сетевого уровня.

Итак, чтобы связать сети, показанные на рис. 9.6, необходимо соединить все эти сети маршрутизаторами и установить протокольные модули сетевого уровня на все конечные узлы пользователей, которые хотели бы связываться через составную сеть (рис. 9.7).

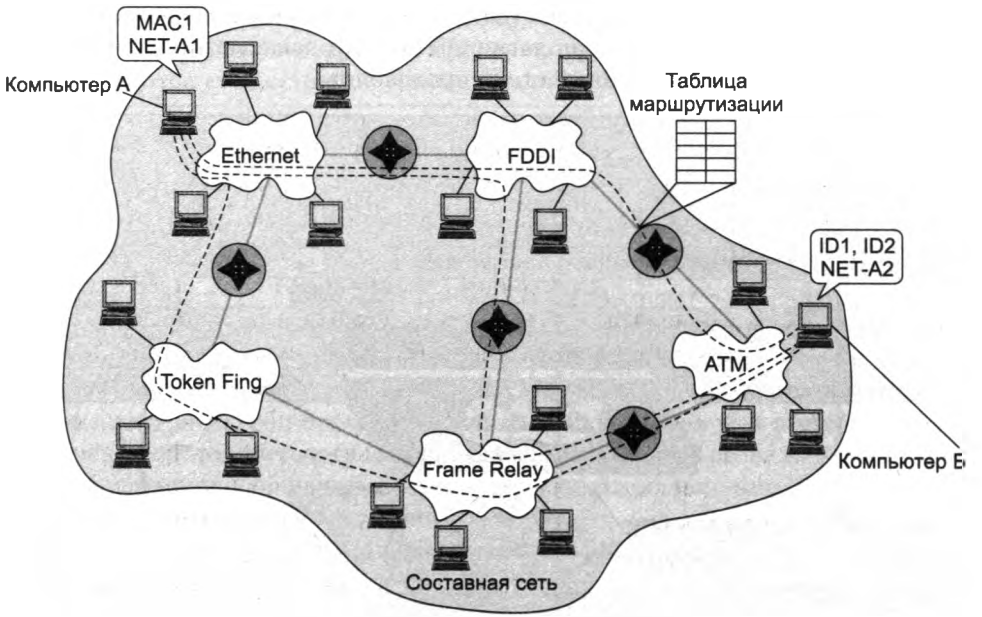


Рис. 9.7. Пример составной сети

Данные, которые необходимо передать через составную сеть, поступают на сетевой уровень от вышележащего транспортного уровня. Эти данные снабжаются заголовком сетевого уровня. Данные вместе с заголовком образуют **пакет** — так называется PDU сетевого уровня. Заголовок пакета сетевого уровня имеет унифицированный формат, не зависящий от форматов кадров канального уровня тех сетей, которые могут входить в составную сеть, и наряду с другой служебной информацией этот заголовок несет данные об адресе назначения пакета.

Для того чтобы протоколы сетевого уровня могли доставлять пакеты любому узлу составной сети, эти узлы должны иметь адреса, уникальные в пределах данной составной сети. Такие адреса называются **сетевыми**, или **глобальными**. Каждый узел составной сети, который намерен обмениваться данными с другими узлами составной сети, должен иметь сетевой адрес наряду с адресом, назначенным ему на канальном уровне. Например, на рис. 9.7 компьютер в сети Ethernet, входящей в составную сеть, имеет адрес канального уровня MAC1 и адрес сетевого уровня NET-A1; аналогично в сети ATM узел, адресуемый идентификаторами виртуальных каналов ID1 и ID2, имеет сетевой адрес NET-A2.

Сетевые адреса обычно являются иерархическими, то есть состоят из поля адреса сети, входящей в составную сеть, и адреса узла в пределах этой сети.

В пакете в качестве адреса назначения должен быть указан адрес сетевого уровня, на основании которого определяется маршрут пакета. *Определение маршрута* является важной задачей сетевого уровня. Маршрут описывается последовательностью сетей (или маршрутизаторов), через которые должен пройти

пакет, чтобы попасть к адресату. Например, на рис. 9.7 штриховой линией показано 3 маршрута, по которым могут быть переданы данные от компьютера *А* к компьютеру *Б*. Маршрутизатор собирает информацию о топологии связей между сетями и на ее основании строит таблицы коммутации, которые в данном случае носят специальное название **таблиц маршрутизации**.

Иерархическая структура сетевого адреса позволяет эффективно организовывать таблицы маршрутизации, так как в них можно указывать только старшую часть адреса, то есть адрес сети, опуская адрес узла. Действительно, для передачи пакета между сетями, входящими в составную сеть, адрес узла в пределах сети назначения не имеет значения, он нужен только на последнем этапе маршрута, когда пакет достиг сети назначения и его нужно доставить определенному узлу этой сети.

В соответствии с многоуровневым подходом сетевой уровень для решения своей задачи обращается к нижележащему канальному уровню. Весь путь через составную сеть разбивается на участки от одного маршрутизатора до другого, причем каждый участок соответствует пути через отдельную сеть.

Для того чтобы передать пакет через очередную сеть, сетевой уровень помещает его в поле данных кадра соответствующей канальной технологии, указывая в заголовке кадра канальный адрес интерфейса следующего маршрутизатора. Сеть, используя свою канальную технологию, доставляет кадр с инкапсулированным в него пакетом по заданному адресу. Маршрутизатор извлекает пакет из прибывшего кадра и после необходимой обработки передает пакет для дальнейшей транспортировки в следующую сеть, предварительно упаковав его в новый кадр канального уровня в общем случае другой технологии. Таким образом, сетевой уровень играет роль координатора, организующего совместную работу сетей, построенных на основе разных технологий.

В общем случае функции сетевого уровня шире, чем обеспечение обмена в пределах составной сети. Так, сетевой уровень решает задачу создания надежных и гибких барьеров на пути нежелательного трафика между сетями.

В заключение отметим, что на сетевом уровне определяются два вида протоколов. Первый вид — **маршрутизируемые протоколы** — реализуют продвижение пакетов через сеть. Именно эти протоколы обычно имеют в виду, когда говорят о протоколах сетевого уровня. Однако часто к сетевому уровню относят и другой вид протоколов, называемых **маршрутизирующими протоколами**, или **протоколами маршрутизации**. С помощью этих протоколов маршрутизаторы собирают информацию о топологии межсетевых соединений, на основании которой осуществляется выбор маршрута продвижения пакетов.

Транспортный уровень

На пути от отправителя к получателю пакеты могут быть искажены или утеряны. Хотя некоторые приложения имеют собственные средства обработки ошибок, существуют и такие, которые предпочитают сразу иметь дело с надежным соединением.

Транспортный уровень (transport layer) обеспечивает для приложений или верхних уровней стека — прикладного, представления и сеансового — передачу данных с той степенью надежности, которая им требуется. Модель OSI определяет пять **классов транспортного сервиса** от низшего класса 0 до высшего класса 4. Эти виды сервиса отличаются качеством предоставляемых услуг: срочностью, возможностью восстановления прерванной связи, наличием средств мультиплексирования нескольких соединений между различными прикладными протоколами через общий транспортный протокол, а главное — способностью к обнаружению и исправлению ошибок передачи, таких как искажение, потеря и дублирование пакетов.

Выбор класса сервиса транспортного уровня определяется, с одной стороны, тем, в какой степени задача обеспечения надежности решается самими приложениями и протоколами более высоких, чем транспортный, уровней. С другой стороны, этот выбор зависит от того, насколько надежной является система транспортировки данных в сети, обеспечиваемая уровнями, расположенными ниже транспортного, — сетевым, канальным и физическим. Так, если качество каналов передачи очень высокое и вероятность возникновения ошибок, не обнаруженных протоколами более низких уровней, невелика, то разумно воспользоваться одним из облегченных сервисов транспортного уровня, не обремененных многочисленными проверками, квитированием и другими приемами повышения надежности. Если же транспортные средства нижних уровней очень ненадежны, то целесообразно обратиться к наиболее развитому сервису транспортного уровня, который использует максимум средств для обнаружения и устранения ошибок, включая предварительное установление логического соединения, контроль доставки сообщений по контрольным суммам и циклической нумерации пакетов, установление тайм-аутов доставки и т. п.

Все протоколы, начиная с транспортного уровня и выше, реализуются программными средствами конечных узлов сети — компонентами их сетевых операционных систем. В качестве примера транспортных протоколов можно привести протоколы TCP и UDP стека TCP/IP, а также протокол SPX стека Novell.

Протоколы нижних четырех уровней обобщенно называют сетевым транспортом, или транспортной подсистемой, так как они полностью решают задачу транспортировки сообщений с заданным уровнем качества в составных сетях с произвольной топологией и различными технологиями. Оставшиеся три верхних уровня решают задачи предоставления прикладных сервисов, используя нижележащую транспортную подсистему.

Сеансовый уровень

Сеансовый уровень (session layer) обеспечивает управление взаимодействием сторон: фиксирует, какая из сторон является активной в настоящий момент, и предоставляет средства синхронизации сеанса. Эти средства позволяют в ходе длинных передач сохранять информацию о состоянии этих передач в виде контрольных точек, чтобы в случае отказа можно было вернуться назад к послед-

ней контрольной точке, а не начинать все с начала. На практике немногие приложения используют сеансовый уровень, и он редко реализуется в виде отдельных протоколов. Функции этого уровня часто объединяют с функциями прикладного уровня и реализуют в одном протоколе.

Уровень представления

Уровень представления (presentation layer), как явствует из его названия, обеспечивает представление передаваемой по сети информации, не меняя при этом ее содержания. За счет уровня представления информация, передаваемая прикладным уровнем одной системы, всегда понятна прикладному уровню другой системы. С помощью средств данного уровня протоколы прикладных уровней могут преодолеть синтаксические различия в представлении данных или же различия в кодах символов, например кодов ASCII и EBCDIC. На этом уровне могут выполняться шифрование и дешифрирование данных, благодаря которым секретность обмена данными обеспечивается сразу для всех прикладных служб. Примером такого протокола является протокол SSL (Secure Socket Layer — слой защищенных сокетов), который обеспечивает секретный обмен сообщениями для протоколов прикладного уровня стека TCP/IP.

Прикладной уровень

Прикладной уровень (application layer) — это в действительности просто набор разнообразных протоколов, с помощью которых пользователи сети получают доступ к разделяемым ресурсам, таким как файлы, принтеры или гипертекстовые веб-страницы, а также организуют свою совместную работу, например, по протоколу электронной почты. Единица данных, которой оперирует прикладной уровень, обычно называется **сообщением**.

Существует очень большое разнообразие протоколов и соответствующих служб прикладного уровня. Приведем в качестве примера несколько наиболее распространенных реализаций сетевых файловых служб: NFS и FTP в стеке TCP/IP, SMB в Microsoft Windows, NCP в операционной системе Novell NetWare.

Ethernet

Ethernet является сегодня доминирующей технологией канального уровня в локальных сетях, где она практически вытеснила все остальные технологии, популярные в конце 80-х начале 90-х. Более того, эта технология становится все более и более популярной в глобальных сетях, где пока еще преобладают два протокола, PPP и HDLC, специально разработанные для учета специфики территориальных двухточечных каналов[1].

Технология Ethernet прошла длинный путь развития с тех пор, как была изобретена в 1973 году сотрудником компании Xerox PARC *Робертом Меткалфом* (Robert Metcalf). После этого в технологии многое изменилось, так что некоторые специалисты высказывают сомнения, имеем ли мы дело с Ethernet,

когда сталкиваемся с сетевым адаптером Ethernet-1000T, работающим со скоростью 1 Гбит/с через локальную сеть Ethernet-коммутаторов. И дело не только в совершенном новом диапазоне скоростей, поддерживаемым сегодня Ethernet (от классической скорости в 10 Мбит/с до 10 Гбит/с и не так уж далекой перспективой в 100 Гбит/с).

Изменился принцип доступа компьютеров, оснащенных сетевыми Ethernet-адаптерами, к сети или точнее к среде передачи данных, используемой сетью. В классической технологии Ethernet использовался общий коаксиальный кабель для объединения компьютеров в сеть, так что среда передачи данных была *разделяемой*. Для того чтобы обеспечить доступ к этой среде в каждый момент времени только одного компьютера, для Ethernet был разработан алгоритм, получивший длинное название **CSMA/CD** (Carrier Sense with Multiple Access and Collision Detection). Этот алгоритм реализуется MAC-уровнем каждого Ethernet-узла. Собственно, алгоритм доступа к разделяемой среде долгое время был самым главным отличительным признаком для разных технологий локальных сетей, то есть тем, чем технология Token Ring отличалась от Ethernet, а Arcnet — от FDDI.

Сегодня в единственном «выжившем» представителе технологий локальных сетей 80–90-х годов, Ethernet, практически не используются разделяемые среды и, соответственно, алгоритм CSMA/CD. Технология Ethernet стала *коммутируемой*.

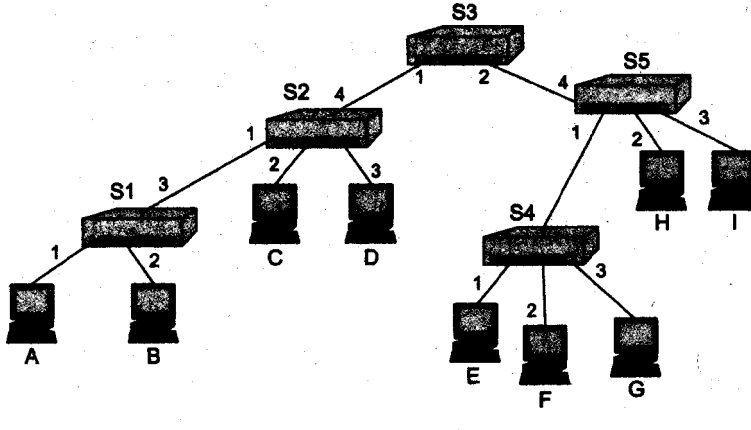


Рис. 9.8. Коммутируемая технология Ethernet

Принципиальное отличие **коммутируемой** (switched) технологии Ethernet от Ethernet на разделяемой среде состоит в том, что связи компьютера с коммутатором стали индивидуальными. Пример такой сети показан на рис. 9.8. Индивидуальный характер связей означает, что, например, сетевой Ethernet-адаптер компьютера *A* может в любой момент послать кадр любому другому компьютеру сети, например, компьютеру *F* через Ethernet-коммутатор *S1*, к которому он подключен индивидуальным кабелем, поддерживающим дуплексный режим работы (то есть позволяющим передавать кадры от компьютера и в компьютер

одновременно). Коммутатор *S1* обязан принять кадр от компьютера *A* и передать его следующему коммутатору, через который проходит маршрут от компьютера *A* к компьютеру *F*.

Для того чтобы передавать кадры через сеть по рациональным маршрутам, которые обеспечивают, в конечном счете, доставку кадров адресатам, Ethernet-коммутаторы автоматически строят **таблицы продвижения** (forwarding tables) на основании информации, получаемой при слежении за кадрами, проходящими через коммутаторы. Этот алгоритм построения таблицы продвижения был разработан давно для так называемых прозрачных мостов локальных сетей (алгоритм закреплен стандартом **IEEE 802.1D**). Мост называется прозрачным, потому что конечный узел «не замечает» присутствия такого моста в сети и ведет себя совершенно так же, как если бы он непосредственно был соединен с узлом-адресатом. То есть компьютер *A* в нашем примере просто выдает на свой выходной порт кадр с двумя MAC-адресами: MAC-A (собственный адрес) в качестве адреса источника и MAC-F (адрес компьютера *F*) в качестве адреса назначения. В классической сети с разделяемой средой, где все компьютеры сети были объединены одним и тем же коаксиальным кабелем, кадр попал бы в буферы сетевых адаптеров всех компьютеров, но только компьютер *F* распознал бы свой собственный адрес MAC-F и начал бы обработку принятого кадра.

В сети с коммутаторами процесс передачи кадра является более сложным. В начальном состоянии после первого включения в сеть каждый коммутатор имеет пустую таблицу продвижения, так как администратор сети не конфигурирует ее вручную, оставляя эту работу самим коммутаторам. Если коммутатор *S1* получает кадр с адресами {MAC-A, MAC-F}, то он просто копирует его на все порты, кроме того, на который он получил этот кадр. В нашем примере коммутатор *S1* копирует кадр на порты 2 и 3. Первое копирование получается очевидно безрезультатным, так как кадр попадает компьютеру *B*, которому он не предназначался (и который отбрасывает этот кадр). А вот копирование на порт 3 имеет смысл, так как кадр попадает в коммутатор *S2*, который находится на пути кадра к компьютеру *F*.

Читатель может самостоятельно продолжить цепочку рассуждений, которая приводит, в конечном счете, к тому, что кадр достигает компьютера *F*.

Однако это не очень эффективный способ передачи кадров, так как сеть оказывается буквально затопленной копиями исходного кадра — подобный режим работы прозрачного моста поэтому и называют «затоплением» (flooding). Но этот режим не является основным, он нужен только для того, чтобы коммутаторы начали строить свои таблицы продвижения. Таблица строится на основании адресов источника кадров, проходящих через коммутатор. Так, коммутатор *S1*, передавая кадр на порты 2 и 3, запоминает тот факт, что кадр от компьютера с адресом MAC-A пришел на порт 1. Поэтому он помещает в свою пустую таблицу продвижения первую запись.

Адрес	Порт
MAC-A	1

Имея такую таблицу, коммутатор *S1* сможет обрабатывать кадр, пришедший от любого компьютера сети для компьютера *A*, более рационально: он не будет копировать его на все порты, а передаст единственную копию на порт 1. Именно так он поступит с ответом компьютера *F* на кадр компьютера *A*, если такой ответ придет (напомним, что поддержание осмысленного взаимодействия компьютеров, в ходе которого устанавливается логический сеанс и на кадры-запросы присылаются кадры-ответы, обычно является делом прикладного, а не канального уровня).

Ответ от компьютера *F* поможет коммутатору *S1* добавить новую запись, в результате его таблица приобретет следующий вид.

Адрес	Порт
MAC-A	1
MAC-F	3

После этого кадры, посланные для компьютера *F*, также начинают продвигаться коммутатором *S1* рационально: единственная копия передается только на порт 3.

Как видно из описания, Ethernet-коммутаторы действительно прозрачны для компьютеров, кроме того, их способность к самообучению без какого-либо предварительного ручного конфигурирования удобна администратору сети.

Коммутируемая технология Ethernet явилась большим шагом вперед в отношении сетевой производительности по сравнению с Ethernet на разделяемой среде, так как она исключила периоды ожидания среды за счет параллельного продвижения кадров от всех компьютеров сети. Высокие скорости работы коммутаторов по обработке кадров и возросшие с 10 до 100 и даже до 1000 Мбит/с скорости передачи битов по каналам связи привели к тому, что локальные сети стали очень эффективной и сравнительно простой средой взаимодействия компьютеров.

Однако простота организации коммутируемых сетей Ethernet имеет и ряд недостатков.

- Появление кадров с неизвестными ранее адресами приводит к так называемым *широковещательным штормам*, так как эти кадры затопляют сеть своими копиями. Сеть Ethernet не может эффективно предотвращать такие ситуации, если какой-либо из компьютеров начинает работать некорректно и генерирует кадры с ошибочным адресом. Говорят, что сеть, построенная на Ethernet-коммутаторах, является «плоской», имея в виду тот факт, что в сети нет естественных барьеров на пути распространения ошибочного широковещательного трафика.
- Плоские Ethernet-адреса не очень удобны для адресации узлов больших сетей, так как в этом случае таблицы продвижения включают *слишком много записей*. Как уже отмечалось, многоуровневый адрес сетевого уровня в этом отношении гораздо более эффективен.

- Самообучение Ethernet-коммутаторов на основе наблюдения за проходящим трафиком приводит к тому, что коммутируемые сети Ethernet эффективно работают только при древовидной топологии сети, когда в сети *нет петель*. Иначе кадры начинают зацикливаться и размножаться, а таблицы продвижения не достигают устойчивого состояния, постоянно перестраиваясь, что, конечно, не дает сети нормально работать. Алгоритм **покрывающего дерева** (spanning tree) решает задачу автоматического нахождения и активизации древовидных структур в сети с произвольной топологией, но не решает проблему полностью, так как в этом случае *невозможно использовать альтернативные маршруты*, если они существуют в сети, для повышения ее производительности. Кроме того, при отказах каналов или коммутаторов новая древовидная топология находится алгоритмом покрывающего дерева достаточно медленно.

Ограничения Ethernet приводят к тому, что на основе этой технологи строят отдельные сети, пока в основном локальные, а для организации крупной сети используют сетевой уровень и строят составную сеть.

Протоколом номер один для построения составных сетей благодаря своей мощной функциональности (а также успеху Интернета) является сегодня протокол IP (Internet Protocol). Вместе с ним работают также другие протоколы стека TCP/IP, которые рассматриваются в следующем разделе.

Стек TCP/IP

Структура стека

Стек **TCP/IP** был разработан по инициативе Министерства обороны США (Department of Defense, DoD) почти 30 лет назад для связи экспериментальной сети ARPAnet с другими сетями в виде набора общих протоколов для разнородной вычислительной среды. Большой вклад в развитие стека TCP/IP, который получил свое название по популярным протоколам IP и TCP, внес университет Беркли, реализовав протоколы стека в своей версии ОС UNIX. Популярность этой операционной системы привела к широкому распространению протоколов TCP, IP и других протоколов стека.

Поскольку стек TCP/IP изначально создавался для Интернета, он имеет много особенностей, дающих ему преимущество перед другими протоколами, когда речь заходит о построении сетей, включающих глобальные связи. В частности, очень полезным свойством, делающим возможным применение этого протокола в больших сетях, является его *способность фрагментировать пакеты*. Действительно, большая составная сеть часто состоит из сетей, построенных на совершенно разных принципах. В каждой из этих сетей может быть собственная величина максимальной длины единицы передаваемых данных (кадра). В таком случае при переходе из одной сети, имеющей большую максимальную длину, в сеть с меньшей максимальной длиной может возникнуть необходимость деления передаваемого кадра на несколько частей. Протокол IP стека TCP/IP эффективно решает эту задачу.

Другой особенностью технологии TCP/IP является *гибкая система адресации*, позволяющая проще, чем другие протоколы аналогичного назначения, включать в составную сеть сети разных технологий. Это свойство также способствует применению стека TCP/IP для построения больших гетерогенных сетей.

В стеке TCP/IP очень *экономно используются широковещательные рассылки*. Это свойство совершенно необходимо при работе на медленных каналах связи, характерных для территориальных сетей.

Однако, как и всегда, за получаемые преимущества надо платить, и платой здесь оказываются *высокие требования к ресурсам* и *сложность администрирования* IP-сетей. Мощные функциональные возможности протоколов стека TCP/IP требуют для своей реализации больших вычислительных затрат. Гибкая система адресации и отказ от широковещательных рассылок приводят к наличию в IP-сети разнообразных централизованных служб типа DNS, DHCP и т. п. Каждая из этих служб направлена на облегчение администрирования сети, но в то же время сама требует пристального внимания со стороны администраторов.

Можно приводить и другие доводы за и против стека протоколов Интернета, однако факт остается фактом — сегодня это самый популярный стек протоколов, широко используемый как в глобальных, так и локальных сетях.

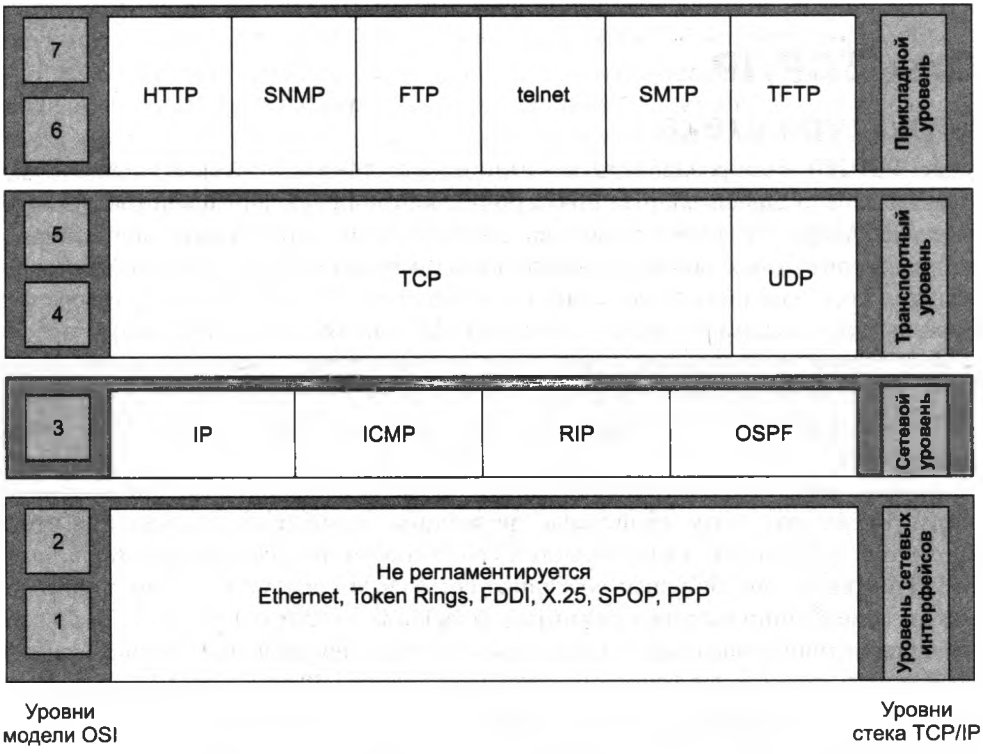


Рис. 9.9. Архитектура стека TCP/IP

На рис. 9.9 приведена структура стека TCP/IP. Так как стек TCP/IP был разработан до появления модели ISO/OSI, то, хотя он также имеет многоуровневую структуру, соответствие уровней стека TCP/IP уровням модели OSI достаточно условно.

В стеке TCP/IP определены 4 уровня.

Прикладной уровень стека TCP/IP соответствует трем верхним уровням модели OSI: прикладному, представления и сеансовому. Он объединяет службы, предоставляемые системой пользовательским приложениям. За долгие годы использования в сетях различных стран и организаций стек TCP/IP вобрал в себя большое количество протоколов и служб прикладного уровня. К ним относятся такие распространенные протоколы, как протокол передачи файлов (File Transfer Protocol, FTP), протокол эмуляции терминала (telnet), простой протокол передачи электронной почты (Simple Mail Transfer Protocol, SMTP), протокол передачи гипертекста (HyperText Transfer Protocol, HTTP) и многие другие. Протоколы прикладного уровня развертываются на хостах[1].

Транспортный уровень стека TCP/IP может предоставлять вышележащему уровню два типа сервиса:

- гарантированную доставку обеспечивает **протокол управления передачей** (Transmission Control Protocol, TCP);
- доставку по возможности, или с максимальными усилиями, обеспечивает **протокол пользовательских дейтаграмм** (User Datagram Protocol, UDP).

Для того чтобы обеспечить надежную доставку данных, протокол TCP предусматривает установление логического соединения, что позволяет ему нумеровать пакеты, подтверждать их прием квитанциями, в случае потери организовывать повторные передачи, распознавать и уничтожать дубликаты, доставлять прикладному уровню пакеты в том порядке, в котором они были отправлены. Этот протокол позволяет объектам на компьютере-отправителе и компьютере-получателе поддерживать обмен данными в дуплексном режиме. TCP дает возможность без ошибок доставить сформированный на одном из компьютеров поток байтов на любой другой компьютер, входящий в составную сеть. TCP делит поток байтов на фрагменты и передает их нижележащему уровню межсетевого взаимодействия. После того как эти фрагменты доставляются средствами уровня межсетевого взаимодействия в пункт назначения, протокол TCP снова собирает их в непрерывный поток байтов.

Второй протокол этого уровня — UDP — является простейшим дейтаграммным протоколом, который используется в том случае, когда задача надежного обмена данными либо вообще не ставится, либо решается средствами более высокого уровня — прикладным уровнем или пользовательскими приложениями.

В функции протоколов транспортного уровня TCP и UDP входит также исполнение роли связующего звена между прилегающими к ним прикладным уровнем и уровнем межсетевого взаимодействия. От прикладного протокола

[1] В Интернете конечный узел традиционно называют хостом, а маршрутизатор — шлюзом. В этой главе мы будем также придерживаться такой терминологии.

транспортный уровень принимает задание на передачу данных с тем или иным качеством, а после выполнения рапортует ему об этом. Нижележащий уровень межсетевого взаимодействия протоколы TCP и UDP рассматривают как своего рода инструмент, не очень надежный, но способный перемещать пакет в свободном и рискованном путешествии по составной сети.

Программные модули, реализующие протоколы TCP и UDP, подобно модулям протоколов прикладного уровня, устанавливаются на хостах.

Сетевой уровень, называемый также **уровнем интернета**, является стержнем всей архитектуры TCP/IP. Именно этот уровень, функции которого соответствуют сетевому уровню модели OSI, обеспечивает перемещение пакетов в пределах составной сети, образованной объединением множества сетей. Протоколы сетевого уровня поддерживают интерфейс с вышележащим транспортным уровнем, получая от него запросы на передачу данных по составной сети, а также с нижележащим уровнем сетевых интерфейсов, о функциях которого мы расскажем далее.

Основным протоколом сетевого уровня является **межсетевой протокол** (Internet Protocol, IP). В его задачу входит продвижение пакета между сетями — от одного маршрутизатора до другого до тех пор, пока пакет не попадет в сеть назначения. В отличие от протоколов прикладного и транспортного уровней, протокол IP устанавливается не только на хостах, но и на всех маршрутизаторах. Протокол IP — это дейтаграммный протокол, работающий без установления соединения по принципу доставки с максимальными усилиями.

К сетевому уровню TCP/IP часто относят протоколы, выполняющие вспомогательные функции по отношению к IP. Это, прежде всего, **протоколы маршрутизации** RIP, OSPF, IS-IS и BGP, занимающиеся изучением топологии сети, определением маршрутов и составлением таблиц маршрутизации, на основании которых протокол IP перемещает пакеты в нужном направлении. По этой же причине к сетевому уровню могут быть отнесены еще два протокола: протокол межсетевых управляющих сообщений (Internet Control Message Protocol, ICMP), предназначенный для передачи маршрутизатором источнику информации об ошибках, возникших при передаче пакета, и протокол групповой адресации (Internet Group Management Protocol, IGMP), использующийся для направления пакета сразу по нескольким адресам.

Идеологическим отличием архитектуры стека TCP/IP от многоуровневой организации других стеков является интерпретация функций самого нижнего уровня — **уровня сетевых интерфейсов**.

Напомним, что нижние уровни модели OSI (канальный и физический) реализуют большое количество функций доступа к среде передачи, формирования кадров и согласования уровней электрических сигналов, кодирования, синхронизации и некоторые другие. Все эти весьма конкретные функции составляют суть таких протоколов обмена данными, как Ethernet, Token Ring, PPP, HDLC и многих других.

У нижнего уровня стека TCP/IP задача существенно проще — он отвечает только за *организацию взаимодействия с технологиями сетей, входящих в со-*

ставную сеть. TCP/IP рассматривает любую сеть, входящую в составную сеть, как средство транспортировки пакетов до следующего на пути маршрутизатора.

Задачу обеспечения интерфейса между технологией TCP/IP и любой другой технологией промежуточной сети упрощенно можно свести к определению способов:

- упаковки (инкапсуляции) IP-пакета в единицу передаваемых данных промежуточной сети;
- преобразования сетевых адресов в адреса технологии данной промежуточной сети.

Такой подход делает составную сеть TCP/IP открытой для включения *любой* сети, какую бы внутреннюю технологию передачи данных эта сеть не использовала, при условии, что для этой технологии разработаны средства взаимодействия с сетевым уровнем стека TCP/IP.

Поскольку условием интегрируемости каждой вновь появившейся технологии в сеть TCP/IP является наличие соответствующих интерфейсных средств, функции уровня сетевых интерфейсов нельзя определить раз и навсегда. В связи с этим уровень сетевых интерфейсов в стеке TCP/IP не регламентируется. Он поддерживает все популярные технологии; для локальных сетей — это Ethernet, Token Ring, FDDI, Fast Ethernet, Gigabit Ethernet, для глобальных сетей — протоколы двухточечных соединений SLIP и PPP, технологии X.25, Frame Relay, ATM.

Обычно при появлении новой технологии локальных или глобальных сетей она быстро включается в стек TCP/IP путем разработки соответствующего документа RFC (Request For Comments), определяющего метод инкапсуляции IP-пакетов в ее кадры (например, спецификация RFC 1577, определяющая работу протокола IP через сети ATM, появилась в 1994 году вскоре после принятия основных стандартов ATM).

ПРИМЕЧАНИЕ Стек TCP/IP позволяет включать в составную сеть сети независимо от того, каким количеством уровней описывается используемая в них технология. Так, перемещение данных в сети X.25 обеспечивают собственные протоколы физического, канального и сетевого уровней (в терминологии OSI). Тем не менее стек TCP/IP рассматривает сеть X.25 наравне с другими технологиями в качестве средства транспортировки IP-пакетов между двумя пограничными шлюзами. Уровень сетевых интерфейсов обычным образом предоставляет для этой технологии способ инкапсуляции IP-пакета в пакет X.25, а также средства преобразования сетевых IP-адресов в адреса сетевого уровня X.25. Если рассматривать такую организацию сети в строгом соответствии с моделью OSI, то налицо явное противоречие — один сетевой протокол (IP) работает поверх другого сетевого протокола (X.25). Однако для стека TCP/IP это — нормальное явление.

Каждый коммуникационный протокол оперирует некоторой единицей передаваемых данных. Названия этих единиц иногда закрепляются стандартом,

а чаще просто определяются традицией. В стеке TCP/IP за многие годы его существования образовалась устоявшаяся терминология в этой области.

Потоком данных, или просто **потоком**, называют данные, поступающие от приложений на вход протоколов транспортного уровня — TCP и UDP.

Протокол TCP «нарезает» из потока данных **сегменты**.

Единицу данных протокола UDP часто называют **дейтаграммой**, или **датаграммой**. Дейтаграмма — это общее название для единиц данных, которыми оперируют протоколы, работающие без установления соединений. К таким протоколам относится и протокол IP, поэтому его единицу данных также называют дейтаграммой. Однако очень часто используется и другой термин — **пакет**.

В стеке TCP/IP **кадрами**, или **фреймами**, принято называть единицы данных любых технологий, в которые упаковываются IP-пакеты для последующей передачи их через сети составной сети. При этом не имеет значения, какое название используется для этой единицы данных в технологии каждой из объединяемых сетей. Для TCP/IP фреймом является и Ethernet-кадр, и ATM-ячейка, и пакет X.25, так как все они выступают в качестве контейнеров, в которых IP-пакеты переносятся через составную сеть.

IP-адреса

Чтобы технология TCP/IP могла решать свою задачу объединения сетей, ей необходима собственная глобальная система адресации, *не зависящая от способов адресации узлов в отдельных сетях*. Эта система адресации должна позволять универсальным и однозначным способом идентифицировать любой интерфейс составной сети. Очевидным решением является уникальная нумерация всех сетей составной сети, а затем нумерация всех узлов в пределах каждой из этих сетей. Пара, состоящая из **номера сети** и **номера узла**, отвечает поставленным условиям и может служить в качестве **сетевого адреса**, называемого также **IP-адресом**.

В протоколе IP значение номера узла выбирается независимо от адреса этого узла на канальном уровне, то есть от его MAC-адреса. Это дает сетевому уровню большую степень независимости от технологий нижележащих уровней, хотя также возможно принципиально другое решение, когда MAC-адрес используется в качестве номера узла протокола IP.

Каждый раз, когда пакет направляется адресату через составную сеть, в его заголовке указывается IP-адрес узла назначения. По номеру сети назначения каждый очередной маршрутизатор находит IP-адрес следующего маршрутизатора. Перед тем как отправить пакет в следующую сеть, маршрутизатор должен определить на основании найденного IP-адреса следующего маршрутизатора его локальный MAC-адрес. Для этой цели протокол IP обращается к **протоколу разрешения адресов** (Address Resolution Protocol, ARP).

Протокол ARP работает в локальной сети на основе широковещательных запросов. Запрос содержит известный IP-адрес узла и принимается всеми узлами сети. Узел, распознавший в запросе свой IP-адрес, отвечает кадром, в котором указывает свой MAC-адрес. Обычно, чтобы ускорить работу, каждый узел хранит таблицу соответствия IP- и MAC-адресов своей сети.

Наиболее распространенной формой представления IP-адреса является запись в виде четырех чисел, представляющих значения каждого байта в десятичной форме и разделенных точками, например:

128.10.2.30

Этот же адрес может быть представлен в двоичном формате:

10000000 00001010 00000010 00011110

Или в шестнадцатеричном формате:

80.0A.02.1D

Заметим, что запись адреса не предусматривает специального разграничительного знака между номерами сети и узла. Вместе с тем при передаче пакета по сети часто возникает необходимость разделить адрес на эти две части. Например, маршрутизация, как правило, осуществляется на основании номера сети, поэтому каждый маршрутизатор, получая пакет, должен прочитать из соответствующего поля заголовка адрес назначения и выделить из него номер сети. Каким образом маршрутизаторы определяют, какая часть из 32 бит, отведенных под IP-адрес, относится к номеру сети, а какая — к номеру узла?

Можно предложить несколько вариантов решения этой проблемы.

- Простейший из них состоит в использовании **фиксированной границы**. При этом все 32-разрядное поле адреса заранее делится на две части не обязательно равной, но фиксированной длины, в одной из которых всегда должен размещаться номер сети, а в другой — номер узла. Решение очень простое, но хорошее ли? Поскольку поле, которое отводится для хранения номера узла, имеет фиксированную длину, максимальное число узлов всех сетей оказывается одинаковым. Если, например, под номер сети отвести один первый байт, то все адресное пространство распадется на сравнительно небольшое (2^8) число сетей огромного размера (2^{24} узлов). Если границу передвинуть дальше вправо, то сетей станет больше, но все равно их размеры будут одинаковыми. Очевидно, что такой жесткий подход не позволяет дифференцированно удовлетворять потребности отдельных предприятий и организаций. Именно поэтому он не нашел применения, хотя и использовался на начальном этапе существования технологии TCP/IP (RFC 760).
- Второй подход (RFC 950, RFC 1518) основан на применении *маски*, которая позволяет максимально гибко устанавливать границу между номерами сети и узла. При таком подходе адресное пространство можно использовать для создания множества сетей разного размера.

  Маска — это число, применяемое в паре с IP-адресом, причем двоичная запись маски содержит непрерывную последовательность единиц в тех разрядах, которые должны в IP-адресе интерпретироваться как номер сети. Граница между последовательностями единиц и нулей в маске соответствует границе между номером сети и номером узла в IP-адресе.

- И, наконец, наиболее распространенный до недавнего времени способ решения данной проблемы заключается в использовании **классов адресов** (RFC 791). Этот способ представляет собой компромисс по отношению к двум предыдущим: размеры сетей, хотя и не могут быть произвольными, как при применении масок, не должны быть одинаковыми, как при установлении фиксированных границ. Вводится пять классов адресов: *A*, *B*, *C*, *D* и *E*. Три из них — *A*, *B* и *C* — используются для адресации сетей, а два — *D* и *E* — имеют специальное назначение. Для каждого класса сетевых адресов определено собственное положение границы между номерами сети и узла.

Классы IP-адресов

Признаком, на основании которого IP-адрес относится к тому или иному классу, являются значения нескольких первых битов адреса. Таблица 9.1 иллюстрирует структуру IP-адресов разных классов.

Таблица 9.1. Классы IP-адресов

Класс	Первые биты	Наименьший номер сети	Наибольший номер сети	Максимальное число узлов в сети
A	0	1.0.0.0 (0 — не используется)	126.0.0.0 (127 — зарезервирован)	2^{24}, поле 3 байта
B	10	128.0.0.0	191.255.0.0	2^{16}, поле 2 байта
C	110	192.0.0.0	223.255.255.0	2^{8}, поле 1 байт
D	1110	224.0.0.0	239.255.255.255	Групповые адреса
E	11110	240.0.0.0	247.255.255.255	Зарезервировано

- К **классу A** относится адрес, в котором старший бит имеет значение 0. В адресах класса *A* под идентификатор сети отводится 1 байт, а остальные 3 байта интерпретируются как номер узла в сети. Сети, все IP-адреса которых имеют значение первого байта в диапазоне от 1 (00000001) до 126 (01111110), называются сетями класса *A*. Значение 0 (00000000) первого байта не используется, а значение 127 (01111111) зарезервировано для специальных целей, о чем будет рассказано далее. Сетей класса *A* сравнительно немного, зато количество узлов в них может достигать 2^{24}, то есть 16 777 216 узлов.
- К **классу B** относятся все адреса, старшие два бита которых имеют значение 10. В адресах класса *B* под номера сети и узла отводится по 2 байта. Сети, значения первых двух байтов адресов которых находятся в диапазоне от 128.0. (10000000 00000000) до 191.255 (10111111 11111111), называются сетями класса *B*. Ясно, что сетей класса *B* больше, чем сетей класса *A*, а размеры их меньше. Максимальное количество узлов в сетях класса *B* составляет 2^{16} (65 536).
- К **классу C** относятся все адреса, старшие три бита которых имеют значение 110. В адресах класса *C* под номер сети отводится 3 байта, а под номер

узла — 1 байт. Сети, старшие три байта которых находятся в диапазоне от 192.0.0 (11000000 00000000 00000000) до 223.255 (11011111 11111111 11111111), называются сетями класса *C*. Сети класса *C* наиболее распространены и имеют наименьшее максимальное число узлов — 2^8 (256).

- Если адрес начинается с последовательности 1110, то он является адресом **класса D** и обозначает особый **групповой адрес** (multicast address). В то время как адреса классов *A*, *B* и *C* используются для идентификации отдельных сетевых интерфейсов, то есть являются **индивидуальными адресами** (unicast address), групповой адрес идентифицирует группу сетевых интерфейсов, которые в общем случае могут принадлежать разным сетям. Интерфейс, входящий в группу, получает наряду с обычным индивидуальным IP-адресом еще один групповой адрес. Если при отправке пакета в качестве адреса назначения указан адрес класса *D*, то такой пакет должен быть доставлен всем узлам, которые входят в группу.
- Если адрес начинается с последовательности 11110, это значит, что данный адрес относится к **классу E**. Адреса этого класса зарезервированы для будущих применений.

Чтобы получить из IP-адреса номера сети и узла, требуется не только разделить адрес на две соответствующие части, но и дополнить каждую из них нулями до четырех полных байтов. Возьмем, например, адрес класса *B* 129.64.134.5. Первые два байта идентифицируют сеть, а последующие два — узел. Таким образом, номером сети является адрес 129.64.0.0, а номером узла — адрес 0.0.134.5.

Использование масок

Снабжая каждый IP-адрес маской, можно отказаться от понятий классов адресов и сделать более гибкой систему адресации.

Пусть, например, для IP-адреса 129.64.134.5 указана маска 255.255.128.0, то есть в двоичном виде IP-адрес 129.64.134.5 — это:

10000001.01000000.10000110.00000101,

а маска 255.255.128.0 — это:

11111111.11111111.10000000.00000000.

Если игнорировать маску и интерпретировать адрес 129.64.134.5 на основе классов, то номером сети является 129.64.0.0, а номером узла — 0.0.134.5 (поскольку адрес относится к классу *B*).

Если же использовать маску, то 17 последовательных двоичных единиц в маске 255.255.128.0, «наложенные» на IP-адрес 129.64.134.5, делят его на две части:

- номер сети: 10000001.01000000.1;
- номер узла: 0000110.00000101.

В десятичной форме записи номера сети и узла, дополненные нулями до 32 бит, выглядят, соответственно, как 129.64.128.0 и 0.0.6.5.

Наложение маски можно интерпретировать как выполнение логической операции «И» (AND). Так, в предыдущем примере номер сети из адреса 129.64.134.5 является результатом выполнения логической операции AND с маской 255.255.128.0:

10000001 01000000 10000110 00000101
AND
11111111.11111111.10000000.00000000

Для стандартных классов сетей маски имеют следующие значения:

- класс *A* — 11111111.00000000.00000000.00000000 (255.0.0.0);
- класс *B* — 11111111.11111111.00000000.00000000 (255.255.0.0);
- класс *C* — 11111111.11111111.11111111.00000000 (255.255.255.0).

ПРИМЕЧАНИЕ Для записи масок используются и другие форматы. Например, удобно интерпретировать значение маски, записанной в шестнадцатеричном коде: FF.FF.00.00 — маска для адресов класса B. Еще чаще встречается обозначение 185.23.44.206/16 — эта запись говорит о том, что маска для этого адреса содержит 16 единиц или что в указанном IP-адресе под номер сети отведено 16 двоичных разрядов.

Механизм масок широко распространен в IP-маршрутизации, причем маски могут использоваться для самых разных целей. С их помощью администратор может разбить одну выделенную ему поставщиком услуг сеть определенного класса на несколько других, не требуя от него дополнительных номеров сетей — эта операция называется *разделением на подсети* (subnetting).

Частные и публичные IP-адреса

Для IP-сетей, являющихся частью Интернета, уникальность нумерации IP-адресов обеспечивается специально созданными для этого органами Интернета: центральной корпорацией ICANN (Internet Corporation for Assigned Names and Numbers) и *региональными органами* ARIN (Америка), RIP (Европа), LACNIC (Латинская Америка и страны Карибского бассейна), AfriNIC (Африка) и APNIC (Азия и Океания).

В небольшой автономной IP-сети условие глобальной уникальности номеров сетей не обязательно, а локальную уникальность номеров сетей и узлов должен обеспечивать сетевой администратор организации. В этом случае в распоряжении администратора имеется все адресное пространство, так как совпадение IP-адресов в не связанных между собой сетях не вызовет никаких отрицательных последствий.

Однако при таком подходе исключается возможность в будущем подсоединить данную сеть к Интернету. Действительно, произвольно выбранные адреса данной сети могут совпасть с адресами Интернета, назначенными централизо-

ванно. Для того чтобы избежать коллизий, связанных с такого рода совпадениями, в стандартах Интернета определено несколько так называемых **частных адресов** (private addresses), рекомендуемых для автономного использования:

- в классе *A* — сеть 10.0.0.0;
- в классе *B* — диапазон из 16 номеров сетей 172.16.0.0–172.31.0.0;
- в классе *C* — диапазон из 255 сетей 192.168.0.0–192.168.255.0.

Эти адреса, исключенные из множества централизованно распределяемых, составляют огромное адресное пространство, достаточное для нумерации узлов автономных сетей практически любых размеров. Заметим также, что частные адреса, как и при произвольном выборе адресов, в разных автономных сетях могут совпадать. В то же время использование частных адресов для адресации автономных сетей делает возможным корректное подключение их к Интернету. Применяемые при этом специальные технологии трансляции адресов (Network Address Translation, NAT) исключают коллизии адресов.

Символьные имена и DNS

Пользователь компьютера может применять IP-адреса для того, чтобы работать с другими компьютерами, подключенными к Интернету или к его корпоративной IP-сети. Однако это не очень удобно, так как приходится запоминать ничего не говорящие последовательности цифр, разделенные точками. Гораздо проще запоминать символические адреса, например www.amazon.com или www.rutube.ru.

В сетях TCP/IP символьные имена, как и IP-адреса, имеют иерархическую многоуровневую структуру, но число уровней символьного имени не ограничивается. Пример такой структуры приведен на рис. 9.10.

Иерархия доменных имен аналогична иерархии имен файлов, принятой во многих популярных файловых системах. Дерево имен начинается с корня, обозначаемого здесь точкой (.). Затем следует старшая символьная часть имени, вторая по старшинству символьная часть имени и т. д. Младшая часть имени соответствует конечному узлу сети. В отличие от имен файлов, при записи которых сначала указывается самая старшая составляющая, затем составляющая более низкого уровня и т. д., запись доменного имени начинается с самой младшей составляющей, а заканчивается самой старшей. Составные части доменного имени отделяются друг от друга точкой. Например, в имени partnering.microsoft.com составляющая partnering является именем одного из компьютеров в домене microsoft.com.

Разделение имени на части позволяет *разделить административную ответственность* за назначение уникальных имен между различными людьми или организациями в пределах своего уровня иерархии. Так, для примера, приведенного на рис. 9.10, один человек может нести ответственность за то, чтобы все имена с окончанием «ru» имели уникальную, следующую вниз по иерархии часть. Если этот человек справляется со своими обязанностями, то все имена типа www.ru, mail.mmt.ru или m2.zil.mmt.ru будут отличаться второй по старшинству частью.

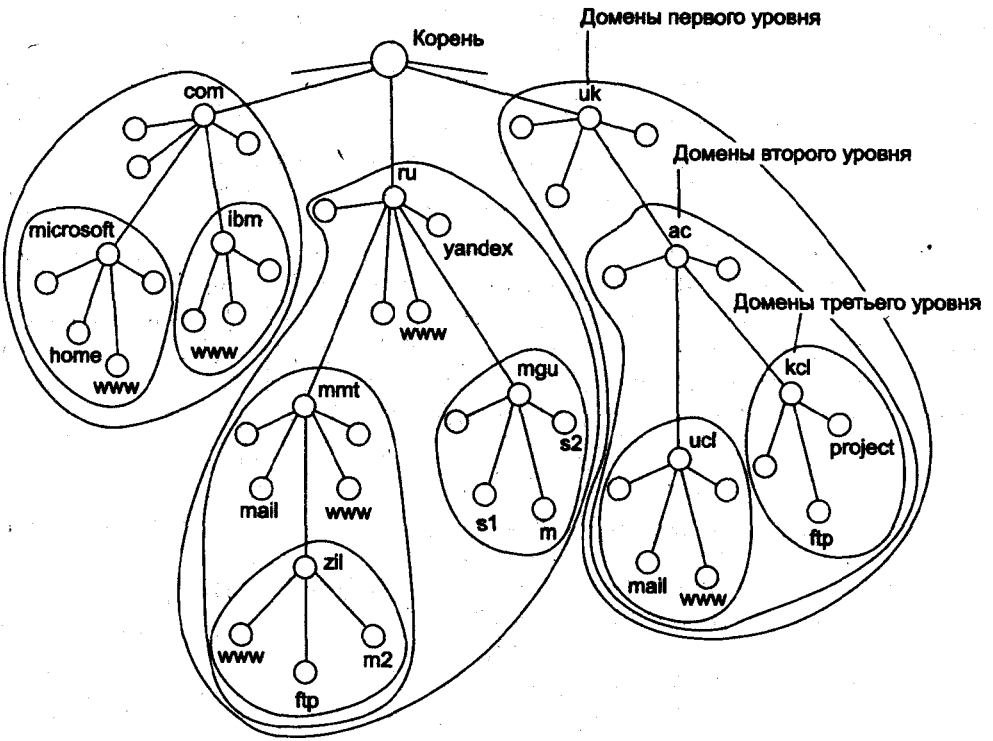


Рис. 9.10. Пространство доменных имен

Разделение административных обязанностей позволяет решить проблему образования уникальных имен без взаимных консультаций между организациями, отвечающими за имена одного уровня иерархии. Очевидно, что должна существовать одна организация, отвечающая за назначение имен верхнего уровня иерархии.

Совокупность имен, у которых несколько старших составных частей совпадают, образуют **домен имен** (name domain). Например, имена www1.zil.mmt.ru, ftp.zil.mmt.ru, yandex.ru и s1.mgu.ru входят в домен ru, так как все они имеют одну общую старшую часть — имя ru. Другим примером является домен mgu.ru. Из представленных на рис. 9.10 имен в него входят имена s1.mgu.ru, s2.mgu.ru и m.mgu.ru. Этот домен образуют имена, у которых две старшие части всегда равны mgu.ru. Администратор домена mgu.ru несет ответственность за уникальность имен следующего уровня, входящего в домен, то есть имен s1, s2 и m. Образованные домены s1.mgu.ru, s2.mgu.ru и m.mgu.ru являются **поддоменами** домена mgu.ru, так как имеют общую старшую часть имени. Часто поддомены для краткости называют только младшей частью имени, то есть s1, s2 и m.

ПРИМЕЧАНИЕ Термин «домен» очень многозначен, поэтому его нужно трактовать в рамках определенного контекста. Помимо доменов имен стека TCP/IP в компьютерной литературе также часто упоминаются домены ОС семейства Windows NT, домены коллизий и некоторые другие. Общим у всех этих терминов является то, что они описывают некоторое множество компьютеров, обладающее каким-либо определенным свойством.

Если в каждых домене и поддомене обеспечивается уникальность имен следующего уровня иерархии, то и вся система имен состоит из уникальных имен.

По аналогии с файловой системой в доменной системе имен различают краткие, относительные и полные доменные имена. **Краткое имя** — это имя конечного узла сети: хоста или порта маршрутизатора. Краткое имя — это лист дерева имен. **Относительное имя** — это составное имя, начинающееся с некоторого уровня иерархии, но не самого верхнего. Например, www1.zil — это относительное имя. **Полное доменное имя** (Fully Qualified Domain Name, FQDN) включает составляющие всех уровней иерархии, начиная от краткого имени и кончая корневой точкой: www1.zil.mmt.ru.

Корневой домен управляется центральными органами Интернета IANA/ICANN. Домены верхнего уровня назначаются для каждой страны, а также для различных типов организаций. Имена этих доменов должны следовать международному стандарту ISO 3166. Для обозначения стран используются трехбуквенные и двухбуквенные аббревиатуры, например ru (Россия), uk (Великобритания), fi (Финляндия), us (Соединенные Штаты), а для различных типов организаций такие, например, обозначения:

- com — коммерческие организации (например, microsoft.com);
- edu — образовательные организации (например, mit.edu);
- gov — правительственные организации (например, nsf.gov);
- org — некоммерческие организации (например, fidonet.org);
- net — сетевые организации (например, nsf.net).

Каждый домен администрирует отдельная организация, которая обычно разбивает свой домен на поддомены и передает функции администрирования этих поддоменов другим организациям. Чтобы получить доменное имя, необходимо зарегистрироваться в какой-либо организации, которой орган IANA/ICANN делегировал свои полномочия по распределению имен доменов.

ВНИМАНИЕ Компьютеры входят в домен в соответствии со своими составными именами, при этом они могут иметь абсолютно независимые друг от друга IP-адреса, принадлежащие различным сетям и подсетям. Например, в домен mgu.ru могут входить хосты с адресами 132.13.34.15, 201.22.100.33 и 14.0.0.6.

Доменная система имен реализована в Интернете, но она может работать и как автономная система имен в любой крупной корпоративной сети, которая хотя и использует стек TCP/IP, никак не связана с Интернетом.

Широковещательный способ установления соответствия между символьными именами и локальными адресами, подобный протоколу ARP, хорошо работает только в небольшой локальной сети, не разделенной на подсети. В крупных сетях, где возможность всеобщей широковещательной рассылки не поддерживается, нужен другой способ разрешения символьных имен. Хорошей альтернативой широковещательной рассылке является применение централизованной службы, поддерживающей соответствие между различными типами адресов всех компьютеров сети. Например, компания Microsoft для своей корпоративной операционной системы Windows NT разработала централизованную службу WINS, которая поддерживала базу данных NetBIOS-имен и соответствующих им IP-адресов.

В сетях TCP/IP соответствие между доменными именами и IP-адресами может устанавливаться средствами как локальных хостов, так и централизованной службы.

На раннем этапе развития Интернета на каждом хосте вручную создавался текстовый файл с известным именем hosts.txt. Этот файл состоял из некоторого количества строк, каждая из которых содержала одну пару «доменное имя — IP-адрес», например:

rhino.acme.com — 102.54.94.97

По мере роста Интернета файлы hosts.txt также увеличивались в объеме, и создание *масштабируемого* решения для разрешения имен стало необходимостью.

Таким решением стала **централизованная служба DNS** (Domain Name System — система доменных имен), основанная на распределенной базе отображений доменных имен на IP-адреса. Служба DNS использует в своей работе DNS-серверы и DNS-клиенты. DNS-серверы поддерживают распределенную базу отображений, а DNS-клиенты обращаются к серверам с запросами на разрешение доменных имен в IP-адрес.

Служба DNS использует текстовые файлы почти такого формата, как и файл hosts, и эти файлы администратор также подготавливает вручную. Однако служба DNS опирается на иерархию доменов, и каждый DNS-сервер хранит только часть имен сети, а не все имена, как это происходит при использовании файлов hosts. При росте количества узлов в сети проблема масштабирования решается созданием новых доменов и поддоменов имен и добавлением в службу DNS новых серверов.

Для каждого домена имен создается свой DNS-сервер. Имеется два распределения имен на серверах. В первом случае сервер может хранить отображения доменных имен на IP-адреса для всего домена, включая все его поддомены. Однако такое решение оказывается плохо масштабируемым, так как при добавлении новых поддоменов нагрузка на этот сервер может превысить его возможности.

Чаще используется другой подход, когда сервер домена хранит только имена, которые заканчиваются на следующем вниз уровне иерархии по сравнению с именем домена. (Аналогично каталогу файловой системы, который содержит записи о файлах и подкаталогах, непосредственно в него «входящих».) Именно при такой организации службы DNS нагрузка по разрешению имен распределяется более-менее равномерно между всеми DNS-серверами сети. Например, в первом случае DNS-сервер домена mmt.ru будет хранить отображения для всех имен, заканчивающихся символами «mmt.ru» (www1.zil.mmt.ru, ftp.zil. mmt.ru, mail.mmt.ru и т. д.). Во втором случае этот сервер будет хранить отображения только имен типа mail.mmt.ru, www.mmt.ru, а все остальные отображения должны храниться на DNS-сервере поддомена zil.

Каждый DNS-сервер помимо таблицы отображений имен содержит ссылки на DNS-серверы своих поддоменов. Эти ссылки связывают отдельные DNS-серверы в единую службу DNS. Ссылки представляют собой IP-адреса соответствующих серверов. Для обслуживания корневого домена выделено несколько дублирующих друг друга DNS-серверов, IP-адреса которых являются широко известными (их можно узнать, например, в IANA/ICANN).

Процедура разрешения DNS-имени во многом аналогична процедуре поиска файловой системой адреса файла по его символьному имени. Действительно, в обоих случаях составное имя отражает иерархическую структуру организации соответствующих справочников — каталогов файлов или DNS-таблиц. Здесь домен и доменный DNS-сервер являются аналогом каталога файловой системы. Для доменных имен, так же как и для символьных имен файлов, характерна независимость именования от физического местоположения.

Процедура поиска адреса файла по символьному имени заключается в последовательном просмотре каталогов, начиная с корневого. При этом предварительно проверяются кэш и текущий каталог. Для определения IP-адреса по доменному имени также необходимо просмотреть все DNS-серверы, обслуживающие цепочку поддоменов, входящих в имя хоста, начиная с корневого домена.

Существенным отличием файловой системы от службы DNS является то, что первая расположена на одном компьютере, а вторая по своей природе является *распределенной*.

Существует две основные схемы разрешения DNS-имен. В первом варианте работу по поиску IP-адреса координирует DNS-клиент:

1. DNS-клиент обращается к корневому DNS-серверу с указанием полного доменного имени.
2. DNS-сервер отвечает клиенту, указывая адрес следующего DNS-сервера, обслуживающего домен верхнего уровня, заданный в следующей старшей части запрошенного имени.
3. DNS-клиент делает запрос следующего DNS-сервера, который отсылает его к DNS-серверу нужного поддомена и т. д., пока не будет найден DNS-сервер, в котором хранится соответствие запрошенного имени IP-адресу. Этот сервер дает окончательный ответ клиенту.

Такая процедура разрешения имени, в которой клиент сам итеративно выполняет последовательность запросов к разным серверам имен, называется **нерекурсивной**. Эта схема загружает клиента достаточно сложной работой, и она применяется редко.

Во втором варианте реализуется **рекурсивная** процедура.

1. DNS-клиент запрашивает локальный DNS-сервер, то есть тот сервер, обслуживающий поддомен, которому принадлежит имя клиента.
2. Далее возможны два варианта действий:
   - если локальный DNS-сервер знает ответ, то он сразу же возвращает его клиенту (это может произойти, когда запрошенное имя входит в тот же поддомен, что и имя клиента, или когда сервер уже узнавал данное соответствие для другого клиента и сохранил его в своем кэше);
   - если локальный сервер не знает ответ, то он выполняет итеративные запросы к корневому серверу и т. д. точно так же, как это делал клиент в предыдущем варианте, а получив ответ, передает его клиенту, который все это время просто ждет его от своего локального DNS-сервера.

В этой схеме клиент перепоручает работу своему серверу, поэтому схема называется косвенной, или рекурсивной. Практически все DNS-клиенты используют рекурсивную процедуру.

Для ускорения поиска IP-адресов DNS-серверы широко применяют *кэширование* проходящих через них ответов. Чтобы служба DNS могла оперативно отрабатывать изменения, происходящие в сети, ответы кэшируются на относительно короткое время — обычно от нескольких часов до нескольких дней.

Протокол DHCP

Долгое время в IP-сетях адреса узлов назначались исключительно вручную. Эта практика популярна и теперь, однако у администратора сети сейчас есть выбор, так как он может автоматизировать эту процедуру, воспользовавшись **протоколом динамического конфигурирования хостов** (Dynamic Host Configuration Protocol, DHCP).

Работа протокола DHCP основана на том, что в сети имеется сервер, который распределяет IP-адреса конечным узлам, запрашивающим это назначение у DHCP-сервера. Администратор в этом случае конфигурирует только сам DHCP-сервер, который может быть как конечным узлом сети, так и маршрутизатором.

DHCP-сервер может работать в различных режимах:

- ручное назначение статических адресов;
- автоматическое назначение статических адресов;
- автоматическое распределение динамических адресов.

Во всех режимах администратор при конфигурировании DHCP-сервера сообщает ему один или несколько диапазонов IP-адресов, причем все эти адреса

относятся к одной сети, то есть имеют одно и то же значение в поле номера сети.

В *ручном режиме* администратор, помимо пула доступных адресов, снабжает DHCP-сервер информацией о жестком соответствии IP-адресов физическим адресам или другим идентификаторам клиентских узлов. DHCP-сервер, пользуясь этой информацией, всегда выдает определенному DHCP-клиенту один и тот же назначенный ему администратором IP-адрес (а также набор других конфигурационных параметров).

В режиме *автоматического назначения статических адресов* DHCP-сервер самостоятельно, без вмешательства администратора произвольным образом выбирает клиенту IP-адрес из пула наличных IP-адресов. Адрес дается клиенту из пула в постоянное пользование, то есть между идентифицирующей информацией клиента и его IP-адресом по-прежнему, как и при ручном назначении, существует постоянное соответствие. Оно устанавливается в момент первого назначения DHCP-сервером IP-адреса клиенту. При всех последующих запросах сервер возвращает клиенту тот же самый IP-адрес.

При *динамическом* распределении адресов DHCP-сервер выдает адрес клиенту на ограниченное время, называемое **сроком аренды**. Когда компьютер, являющийся DHCP-клиентом, удаляется из подсети, назначенный ему IP-адрес автоматически освобождается. Когда компьютер подключается к другой подсети, то ему автоматически назначается новый адрес. Ни пользователь, ни сетевой администратор не вмешиваются в этот процесс.

Это дает возможность впоследствии повторно использовать этот IP-адрес для назначения другому компьютеру. Таким образом, помимо основного преимущества DHCP — автоматизации рутинной работы администратора по конфигурированию стека TCP/IP на каждом компьютере, — динамическое разделение адресов в принципе позволяет строить IP-сеть, количество узлов в которой превышает количество имеющихся в распоряжении администратора IP-адресов.

DHCP-сервер может назначать хостам не только IP-адреса, но и другие параметры конфигурации их стека TCP/IP, например адреса DNS-серверов, адрес маршрутизатора по умолчанию, имена домена, к которому принадлежит хост.

Таблицы маршрутизации

Рассмотрим, как маршрутизаторы организуют доставку IP-пакетов от узла-источника до узла назначения. Для этого мы будем использовать пример составной IP-сети, показанной на рис. 9.11.

Составная сеть в нашем примере состоит из пяти сетей *N*1–*N*5 и трех маршрутизаторов *R*1–*R*3. Все сети являются сетями Ethernet, построенными на коммутаторах. Для упрощения рассуждений будем считать, что все интерфейсы коммутаторов и маршрутизаторов представляют собой интерфейсы Fast Ethernet. Все сети имеют номера, причем маски этих адресов соответствуют классу номера сети, опять же для того, чтобы не усложнять картину. Это значит, что сети класса *C* 193.201.5.0, 195.20.12.0 и 212.107.200.0 имеют маску 255.255.255.0,

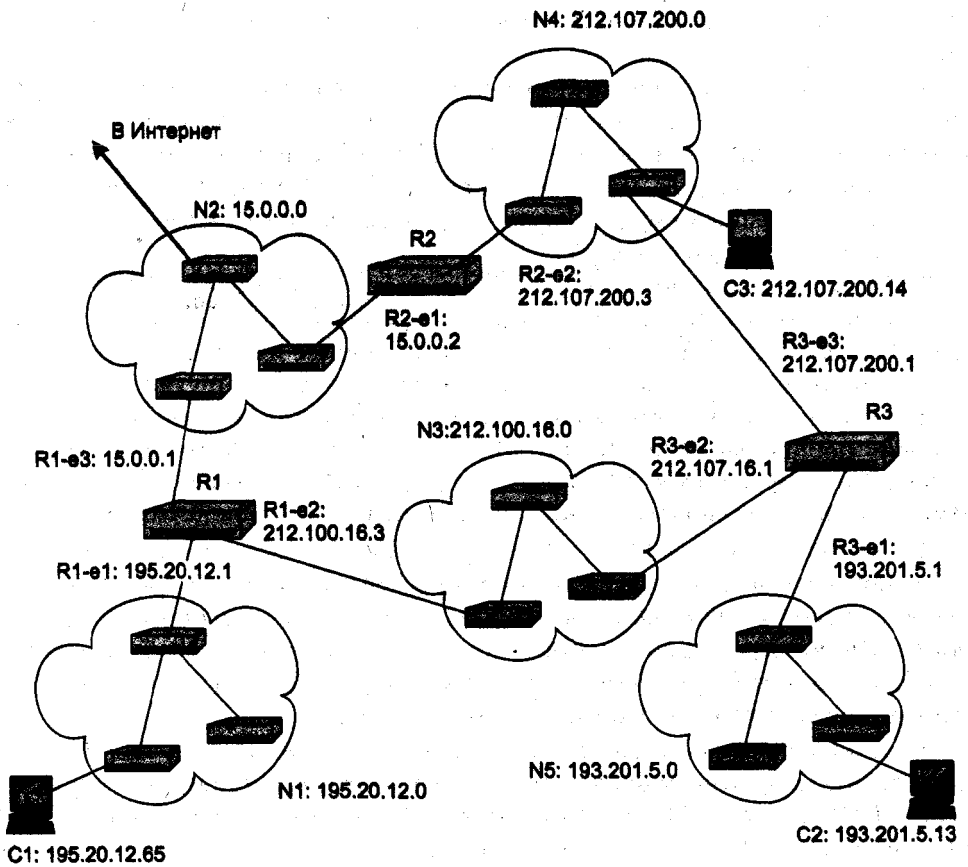


Рис. 9.11. Маршрутизация пакетов в составной IP-сети

а сеть класса *A* 15.0.0.0 имеет маску 255.0.0.0. На рисунке также подразумевается, что маршрутизатор *R2* связан с некоторым (не показанным на рисунке) маршрутизатором интернет-провайдера, через который идет обмен трафиком данной составной сети с остальной частью Интернета (границы Интернета — понятие условное, раз наша составная сеть доступна для обмена со всякой другой сетью, также подключенной к Интернету, то и она является частью Интернета).

Как видно из рисунка, топология сети не является древовидной, она включает петлю *R1* — *N2* — *R2* — *N4* — *R3* — *N3* — *R1*. Поддержка составных сетей с произвольной топологией связей является одним из важных и очень полезных свойств IP-сетей, это повышает производительность и надежность составной сети и облегчает ее проектирование и развитие. Как вы помните, таким свойством обладают не все технологии, мы рассматривали коммутируемую технологию Ethernet, в которой топология сети всегда должна быть древовидной, то есть без петель, иначе простой способ построения таблиц продвижения ком-

мутаторов на основе слежения за проходящим трафиком перестанет правильно работать.

Для того чтобы поддерживать произвольную топологию связей и обеспечивать при этом рациональное продвижение IP-пакетов от источника до узла назначения, IP-маршрутизаторы используют таблицы маршрутизации, позволяющие выяснить, на какой выходной интерфейс нужно передать пришедший IP-пакет. Основой таблиц маршрутизации являются записи наподобие следующей.

IP-адрес назначения	Маска адреса назначения	IP-адрес следующего маршрутизатора
212.107.200.0	255.255.255.0	15.0.0.2

Эта запись говорит о том, что если в маршрутизатор поступает пакет с адресом назначения, у которого старшая часть равна 212.107.200, то его нужно передать следующему маршрутизатору с адресом интерфейса 15.0.0.2. Мы еще вернемся к деталям таблиц маршрутизации и их использования, а сейчас рассмотрим их более общее свойство.

Таблицы маршрутизации напоминают таблицы продвижения Ethernet-коммутаторов, так как те и другие оперируют адресами назначения (пакетов или кадров). Однако между этими таблицами имеется и несколько принципиальных отличий, одно из которых состоит в том, что таблицы маршрутизации строятся иным способом, нежели пассивное наблюдение за проходящим трафиком.

Существует два основных способа построения таблиц маршрутизации: ручной и автоматический, при этом формат результирующей таблицы является одним и тем же.

Ручное конфигурирование таблиц

При ручном способе администратор сети изучает топологию составной сети и находит рациональные маршруты продвижения для каждого маршрутизатора сети и для каждой сети, входящей в состав данной составной сети.

Руководствуясь этим принципом, давайте построим таблицу маршрутизации для маршрутизатора *R*1.

Начнем с сети 193.201.5.0. Топология составной сети показывает, что возможны два рациональных маршрута от маршрутизатора *R*1 до сети 193.201.5.0: первый через сеть *N*3 и маршрутизатор *R*3, второй через сеть *N*2, маршрутизатор *R*2, сеть *N*4 и маршрутизатор *R*3.

Первый маршрут проходит через меньшее число промежуточных сетей и, соответственно, промежуточных маршрутизаторов, чем второй. Меньшее число промежуточных маршрутизаторов означает более быструю доставку пакета адресату, если все промежуточные сети обладают одинаковой скоростью передачи. В нашем случае это так, поэтому логично выбрать первый маршрут для сети назначения 193.201.5.0. Такой критерий выбора маршрута получил название

«количества хопов», где под «хопом» (hop — прыжок, перелет) понимается промежуточная сеть.

В общем случае сети, входящие в состав составной сети, могут обладать различной пропускной способностью, и этот фактор будет влиять на время доставки пакета по тому или иному маршруту. Для того чтобы учесть влияние пропускной способности сетей, часто используют другой критерий выбора маршрута, называемый «расстоянием». С каждой сетью связывается расстояние, значение которого обратно пропускной способности сети. Например, если для сетей с пропускной способностью 1000 Мбит/с выбрать расстояние в одну условную единицу, то сети с пропускной способностью 100 Мбит/с будут связаны с расстоянием в 10 условных единиц, а сети с пропускной способностью 10 Мбит/с — в 100 таких единиц. При выборе маршрута суммарное расстояние от источника до адресата вычисляется простым суммированием расстояний промежуточных сетей и выбирается маршрут с минимальным суммарным расстоянием.

Так как в нашем примере все сети имеют одинаковую пропускную способность, то первый маршрут оказывается предпочтительным и по критерию хопов, и по критерию минимального расстояния.

Соответствующая запись об этом маршруте будут выглядеть так.

IP-адрес назначения	Маска адреса назначения	IP-адрес следующего маршрутизатора
193.201.5.0	255.255.255.0	212.100.16.1

Эта запись описывает действия маршрутизатора не для одного адреса назначения, как это было в случае таблицы продвижения Ethernet-коммутатора, а для всего набора IP-адресов, относящихся к сети 193.201.5.0.

Маска в записи маршрута позволяет задействовать адреса подсетей. Общее правило использования маски состоит в том, что при приходе IP-пакета маршрутизатор извлекает из него IP-адрес назначения и накладывает на него маску, выделяя с помощью логической операции «И» (AND) номер подсети. Затем этот адрес сравнивается с адресами назначения, имеющимися в таблице маршрутизации. Если для какой-либо записи эти адреса совпадают, то считается, что маршрут найден. Для его отработки используется последнее поле записи, то есть IP-адрес следующего маршрутизатора, в нашем примере — это 212.100.16.1, то есть адрес интерфейса *e*2 маршрутизатора *R*3.

Аналогичным образом создаются записи таблицы маршрутизации для остальных четырех сетей, в результате чего таблица приобретает законченный вид (табл. 9.2).

Таблица 9.2. Пример таблицы маршрутизации

IP-адрес назначения	Маска адреса назначения	IP-адрес следующего маршрутизатора
15.0.0.0	255.0.0.0	15.0.0.2
193.201.5.0	255.255.255.0	212.100.16.1

IP-адрес назначения	Маска адреса назначения	IP-адрес следующего маршрутизатора
212.107.200.0	255.255.255.0	15.0.0.2
212.100.16.0	255.255.255.0	Непосредственно присоединенная сеть
195.20.12.0	255.255.255.0	Непосредственно присоединенная сеть
Default		

Отличие записей для сетей 212.100.16.0 и 195.20.12.0 от трех первых записей состоит в том, что для них не задается адрес следующего маршрутизатора, так как эти сети непосредственно присоединены к маршрутизатору *R*1.

В таблице маршрутизации присутствует еще одна запись, вместо адреса сети назначения имеющая значение Default (то есть «по умолчанию»). Эта запись является рабочей для всех адресов сетей назначения, которые явно не указаны в таблице маршрутизации. Запись Default является достаточно элегантным средством решения задачи масштабирования таблиц маршрутизации. Как правило, такая запись очень эффективна для маршрутизаторов, работающих на нижних уровнях иерархии Интернета — в сетях организаций или небольших провайдеров. Она обычно указывает на единственный путь «наверх», ведущий к провайдеру более высокого уровня. Маршруты, выбираемые при этом, оказываются рациональными, и «коллапс» таблицы маршрутизации не наносит никакого ущерба работе составной сети.

Посмотрим, как используется таблица маршрутизации маршрутизатором *R*1 на примере обработки пакета, отправленного компьютером *C*1 компьютеру *C*2. Этот пакет содержит IP-адрес источника 195.20.12.65 и IP-адрес назначения 193.201.5.13. Так как компьютер *C*1 и маршрутизатор *R*1 связаны сетью Ethernet, то для доставки пакета маршрутизатору *R*1 компьютер *C*1 должен упаковать сформированный IP-пакет в Ethernet-кадр и поместить в этот кадр MAC-адрес интерфейса *e*1 маршрутизатора *R*1, который условно обозначим MAC-e1. В этом случае кадр должен пройти через сеть *N*1 Ethernet-коммутаторов и быть принят интерфейсом *e*1.

Возникает вопрос, каким образом компьютер *C*1 узнает MAC-адрес маршрутизатора *R*1? В принципе, он узнает его уже описанным способом за счет того, что администратор сконфигурирует на компьютере таблицу маршрутизации, подобную табл. 9.2. Однако из-за того, что обычно компьютер подключен к сети нижнего уровня, имеющей только один маршрутизатор, через который проходит путь ко всем внешним сетям, на компьютерах используются вырожденные таблицы маршрутизации, состоящие из одной записи Default. Традиционно такую единственную запись называют не таблицей маршрутизации, а просто **адресом маршрутизатора по умолчанию** (default gateway address). В компьютере *C*1 этот адрес должен быть указан как 195.20.12.1. Зная этот адрес, стек TCP/IP компьютера находит с помощью ARP-запроса соответствующий ему MAC-адрес, то есть MAC-адрес интерфейса *e*1 маршрутизатора *R*1.

После прохождения через сеть Ethernet-коммутаторов и приема его в буфер интерфейса *e*1 маршрутизатора *R*1 кадр, сделавший свое дело, отбрасывается, и маршрутизатор работает далее только с полями IP-пакета. В частности, он оперирует адресом назначения 193.201.5.13, содержащимся в пакете. Это оперирование заключается в том, что маршрутизатор последовательно просматривает записи его таблицы маршрутизации, извлекая из них маску и накладывая на данный адрес назначения. Полученный номер сети в каждом случае сравнивается с номером сети назначения, имеющимся в записи.

Для нашего примера первая запись таблицы сравнения не дает, так как после наложения маски образуется номер сети, равный 193.0.0.0, что не равно 15.0.0.0. Вторая проверка приводит к успеху, поэтому маршрутизатор извлекает из этой записи адрес следующего маршрутизатора 212.100.16.1, которому необходимо направить пакет.

Однако непосредственно этот адрес использовать для отправки пакета маршрутизатору *R*3 нельзя, как и в случае с компьютером *C*1 пакет должен быть упакован в Ethernet-кадр и только после этого направлен по сети *N*3 интерфейсу *e*1 маршрутизатора *R*3. Эта передача будет происходить через выходной интерфейс *e*2 маршрутизатора, который выбирается на основании того, что его адрес принадлежит той же сети, что и адрес назначения. Для нахождения MAC-адреса интерфейса *e*1 маршрутизатора *R*3 маршрутизатор *R*1 использует ARP-запрос, распространяемый по сети *N*3 интерфейсом *e*2.

После приема пакета маршрутизатор *R*3 обнаруживает по своей таблице маршрутизации, что его IP-адрес назначения принадлежит сети 193.201.5.0, которая непосредственно подключена к интерфейсу *e*1 этого маршрутизатора. Поэтому маршрутизатор *R*3 не направляет пакет следующему маршрутизатору, а с помощью ARP-запроса находит MAC-адрес, соответствующий IP-адресу назначения, указанному в пакете. После этого пакет упаковывается в Ethernet-кадр с найденным MAC-адресом и отправляется компьютеру *C*2.

Протоколы маршрутизации

Ручное конфигурирование маршрутизаторов становится чересчур обременительным делом для администратора в том случае, когда составная сеть включает большое число подсетей со сложной топологией связей. Администратор может автоматизировать этот процесс, активизировав на маршрутизаторах протоколы маршрутизации.

Сегодня существует несколько протоколов маршрутизации, применяемых в IP-сетях. Несмотря на их различия, их объединяет тот факт, что все они децентрализованные, то есть каждый маршрутизатор строит свою таблицу маршрутизации самостоятельно, в сети нет центрального сервера, который строит и распределяет таблицы маршрутизации по маршрутизаторам для исполнения. Децентрализованный подход повышает масштабируемость протокола, а также его надежность.

Еще одним общим свойством протоколов маршрутизации является то, что они используют служебные пакеты, периодически посылаемые каждым мар-

шрутизатором своим соседям. В этих пакетах маршрутизатор описывает свое видение топологии сети, при этом степень детализации топологии зависит от протокола. На основании этой информации, принимаемой от соседей, каждый маршрутизатор узнает о сетях, входящих в составную сеть, а также о расстояниях до них и автоматически строит свою версию таблицы маршрутизации. Мы не будем рассматривать в данной книге принципы работы и детали каждого протокола маршрутизации, это дело специальных книг по компьютерным сетям. Здесь же только перечислим применяемые сегодня протоколы маршрутизации IP-сетей с краткой характеристикой их достоинств и недостатков.

- **RIP** (Routing Information Protocol) — «старейший» протокол маршрутизации, появившийся в начале 70-х годов и применяющийся не только в IP-сетях. Очень простой протокол для реализации, требует небольшой доли вычислительной мощности маршрутизатора для своей работы. Использует только метрику в «хопах» при определении расстояния между сетями, что во многих случаях является очевидным недостатком. Не передает данные о подробной топологии сети своим соседям, что приводит к длительным периодам несогласованной работы маршрутизаторов при отказах линий связи или самих маршрутизаторов. Основная область применения — небольшие сети с простой топологией.
- **OSPF** (Open Shortest Path First) — очень мощный протокол маршрутизации, разработанный для больших сетей со сложной топологией. Его достоинством является возможность использования метрики, учитывающей производительность сетей, а расширения этого протокола позволяют даже учитывать степень загруженности линий связей сетей при выборе рационального маршрута. Протокол передает полные сведения о топологии сети, что сокращает периоды нестабильной работы маршрутизаторов при отказах в сети. Недостаток протокола связан с его большой вычислительной сложностью, что может перегрузит процессор маршрутизатора.
- **IS-IS for IP** (Intermediate System–Intermediate System for IP) — этот протокол близок по функциональности к OSPF. Он появился раньше, чем OSPF, и первоначально был создан не для IP-сетей, но затем был доработан для применения и в IP-сетях.
- **BGP** (Border Gateway Protocol) — единственный стандартный протокол для обмена маршрутной информацией между маршрутизаторами различных **автономных систем Интернета** (Internet Autonomous Systems). Достаточно давно архитекторы Интернета поняли, что для создания такой глобальной сети нужна гибкость в использовании протоколов маршрутизации. В результате появилась концепция автономной системы, которая представляет собой набор сетей под административным контролем одной организации и с единым протоколом маршрутизации внутри этого набора, позволяющим маршрутизаторам автономной системы общаться между собой и строить эффективные таблицы маршрутизации. В то же время между автономными системами должен применяться общий протокол маршрутизации, своего

рода эсперанто, на котором должны «разговаривать» пограничные маршрутизаторы автономных систем. Такой протокол маршрутизации обычно называют внешним (exterior) протоколом маршрутизации, в отличие от внутренних (interior) протоколов маршрутизации, применяемых внутри автономных систем. В настоящее время таким единым стандартным внешним протоколом маршрутизации является BGP. Все остальные из перечисленных протоколов маршрутизации являются внутренними.

Усложняет картину построения таблицы маршрутизации тот факт, что в маршрутизаторе могут работать одновременно несколько протоколов маршрутизации, которые совместными усилиями создают общую таблицу. Такая ситуация может быть связана с существованием в большой составной сети различных доменов, в которых исторически использовались различные протоколы маршрутизации (эти домены не обязательно представляют собой разные автономные системы, это могут быть, например, находящиеся в введении различных департаментов части одной и той же корпоративной сети, представляющей собой одну автономную систему).

Кроме того, таблица, созданная автоматически протоколом или протоколами маршрутизации, может дополняться записями, задаваемыми администратором сети вручную. Такие записи будут, естественно, статическими в отличие от динамических записей, создаваемых протоколами.

Реализация стека протоколов в универсальной ОС

В данном разделе мы рассмотрим организацию и конфигурирование транспортных сетевых средств универсальной ОС для наиболее массового варианта работы компьютера в сети, а именно для варианта, в котором компьютер является конечным узлом сети. В таком варианте компьютер выполняет только две операции с пакетами: прием пакетов, адресованных данному узлу, и отправку в сеть пакетов, сгенерированных данным узлом. Положение конечного узла сети существенно упрощает последнюю задачу, так как у конечного узла чаще всего существует только один маршрутизатор — сосед по подсети, через которого он может пересылать пакеты компьютерам других сетей, поэтому задача поддержания разветвленной таблицы маршрутизации и ее использования при каждой отправке пакета с конечного узла снимается. Отсутствие потоков транзитных пакетов, которые могут быть весьма интенсивными, а также отсутствие необходимости работы с таблицей маршрутизации заметно снижают требования к быстродействию транспортных протоколов в универсальных ОС и делают возможным их реализацию в виде стандартных драйверов системы ввода-вывода универсальной ОС.

Компьютер может также играть роль промежуточного узла сети, являясь маршрутизатором, для этого к его операционной системе должно быть добавлено несколько программных модулей, в первую очередь модуль IP-маршрутизации, который обеспечивает транзитную передачу IP-пакетов через компьютер.

Как было сказано ранее, такие маршрутизаторы относятся к классу программных; они были достаточно популярны в 80-е и в начале 90-х годов. Однако сравнительно невысокая скорость маршрутизации в универсальных ОС, в которых программный маршрутизатор работал наряду с обычными программами без оптимизации его специфических операций манипуляции пакетами, привела к постепенной замене программных маршрутизаторов аппаратными, такими как маршрутизаторы компаний Cisco или Juniper.

ПРИМЕЧАНИЕ Названия — «программные маршрутизаторы» и «аппаратные маршрутизаторы» — не вполне точны, так как, в сущности, все маршрутизаторы являются программно-аппаратными. Правильнее было бы сказать, что программные маршрутизаторы — это такие устройства, в которых программное обеспечение работает на универсальной аппаратной платформе (универсальном компьютере). Аппаратные маршрутизаторы, в свою очередь, являются устройствами, в которых программное обеспечение работает на специализированной аппаратной платформе.

Особенности организации программного обеспечения аппаратного маршрутизатора на примере маршрутизаторов компании Cisco мы рассмотрим в следующем разделе, а пока сосредоточимся на транспортных средствах универсальной ОС.

Структура транспортных средств универсальной ОС

Сетевые компоненты универсальной ОС обычно реализуются в виде многоуровневых драйверов системы ввода-вывода и фоновых служб (процессов-демонов в терминологии Unix). Обычно в виде драйверов выполняются коммуникационные протоколы канального, сетевого и транспортного уровней, а прикладные протоколы работают как службы, например, как служба удаленного файлового доступа или веб-сервер.

В главе 7 уже рассматривалась общая структура системы ввода-вывода, здесь мы будем пользоваться введенными там понятиями и терминами.

В вертикальной подсистеме сетевых устройств, приведенной на рис. 9.12, аппаратными драйверами являются драйверы сетевых адаптеров, которые выполняют функции низкоуровневых канальных протоколов, таких как Ethernet, Frame Relay, ATM и других. Эти драйверы выполняют простые функции — они организуют передачу кадров данных между компьютерами одной сети. Над ними располагается слой модулей, которые реализуют функции более интеллектуальных протоколов сетевого уровня IP и IPX, обеспечивающих взаимодействие компьютеров разных сетей с произвольной топологией связей. Модули IP и IPX также могут быть оформлены как драйверы, хотя они находятся в промежуточном программном слое и непосредственно с аппаратурой не взаимодействуют. Вообще, вертикальная подсистема управления сетевыми устройствами является примером эффективного многоуровневого подхода к организации драйверов — просто в силу того, что в ее основе лежит хорошо продуманная

семиуровневая модель взаимодействия открытых систем OSI. И хотя все семь уровней модели OSI обычно не выделяются в самостоятельные программные уровни, четыре уровня драйверов в подсистеме управления сетевыми устройствами чаще всего присутствуют. Над слоем драйверов сетевых протоколов располагается слой драйверов транспортных протоколов, таких как TCP/UDP, SPX и NetBEUI, которые отвечают за надежную связь между компьютерами сети. Еще выше расположен слой служб протоколов прикладного уровня (на рисунке — http, ftp и SMB), которые предоставляют пользователям сети конечные услуги по доступу к гипертекстовой информации, архивам файлов и многие другие.

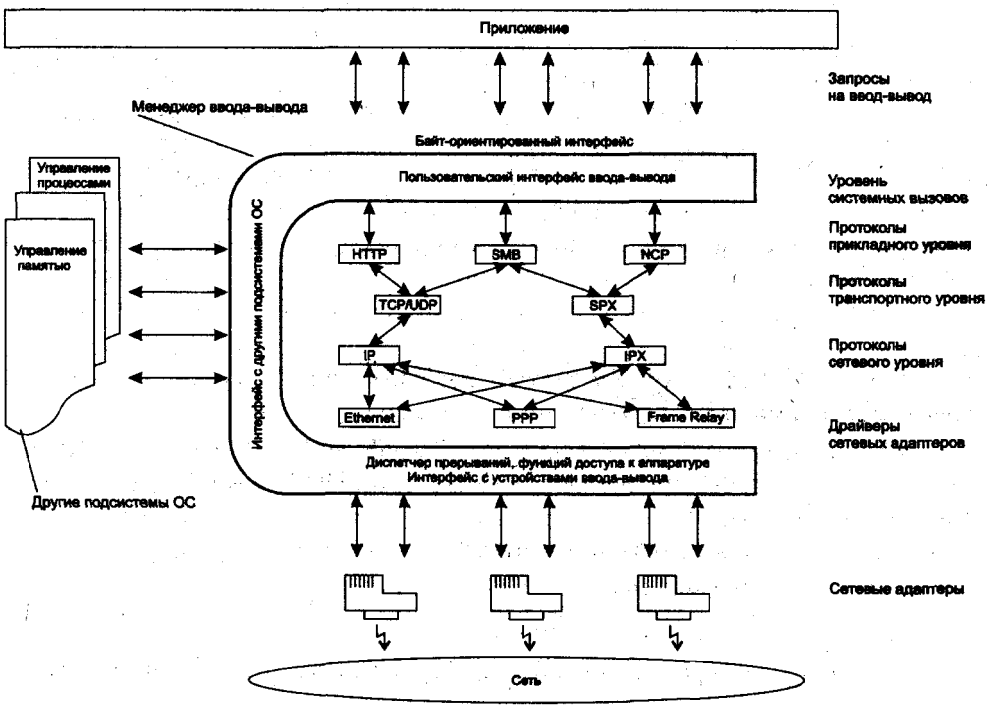


Рис. 9.12. Реализация коммуникационных протоколов в универсальной ОС

В современных ОС система ввода-вывода автоматически создает связи между многоуровневыми драйверами при их установке, так что администратору обычно не требуется выполнять какие-либо операции по ручному конфигурированию взаимосвязей между отдельными драйверами, реализующими коммуникационные протоколы.

Конфигурирование параметров стека TCP/IP

В рассмотренном ранее примере мы затронули вопросы конфигурирования стека TCP/IP компьютера, упомянув о необходимости задания адреса маршрутизатора по умолчанию. Однако это не единственный параметр, который нужен

ОС компьютера, чтобы ее стек TCP/IP работал нормально. Рассмотрим типичный случай конфигурирования параметров стека TCP/IP компьютера на примере сети, изображенной на рис. 9.13.

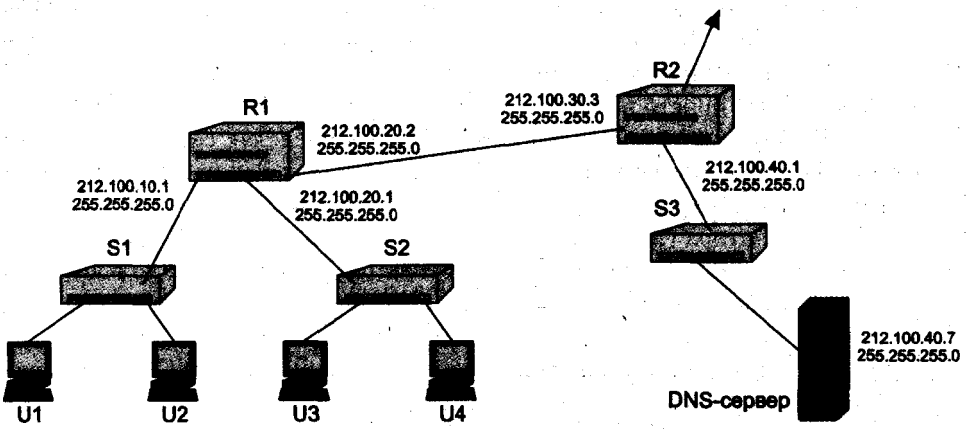


Рис. 9.13. Параметры стека TCP/IP универсальной ОС

Рассмотрим два случая: ручного конфигурирования стека TCP/IP компьютера *U*1 и вариант использования для этого DHCP-сервера, работающего на маршрутизаторе *R*1. Все коммутаторы, показанные на рисунке, являются Ethernet-коммутаторами, а маршрутизаторы — IP-маршрутизаторами.

В первом случае администратор должен решить, какие значения должны быть статически назначены по крайней мере следующим параметрам стека IP компьютера *U*1:

- IP-адрес сетевого адаптера компьютера;
- маска IP-адреса компьютера;
- IP-адрес маршрутизатора по умолчанию;
- IP-адрес DNS-сервера, который будет обслуживать запросы данного компьютера.

IP-адрес компьютера *U*1 должен принадлежать диапазону IP-адресов подсети, в которую входит данный компьютер, и иметь уникальное значение номера узла в пределах этой подсети. Как видно из рисунка, компьютер *U*1 принадлежит подсети 212.100.10.0 с маской 255.255.255.0, в которой номер узла 1 уже использован для интерфейса маршрутизатора *R*1. Предположим, что администратор выбрал для компьютера *U*1 следующий номер узла, тогда он должен присвоить IP-адресу значение 212.100.10.2 с маской 255.255.255.0.

Что касается IP-адреса маршрутизатора по умолчанию, то он известен — это 212.100.10.1, так как это должен быть интерфейс маршрутизатора, который подключен к той же подсети, что и сетевой адаптер компьютера. В то же время если бы администратор задал в качестве такого адреса, например, значение 212.100.30.3, то пакеты от компьютера *U*1 не смогли бы уходить за пределы его

подсети. Причина в том, что сетевой адаптер компьютера использует IP-адрес маршрутизатора по умолчанию только для того, чтобы с помощью ARP-запроса найти его MAC-адрес и отправить IP-пакет, который предназначен для отправки за пределы его подсети, упакованным в Ethernet-кадр с корректным MAC-адресом интерфейса маршрутизатра. Если бы администратор задал IP-адрес 212.100.30.3 в качестве адреса маршрутизатора по умолчанию, то сетевой адаптер компьютера *U*1 просто не получил бы ответа на ARP-запрос с этим адресом, так как ARP-запросы распространяются только в пределах сети Ethernet-коммутаторов, которые обязаны распространять запросы с широковещательным MAC-адресом назначения через все свои порты. Эти запросы доходят до интерфейса маршрутизатора (в данном случае — до интерфейса 212.100.10.1) и дальше не распространяются, так как маршрутизатор обрабатывает только кадры, направленные непосредственно одному из интерфейсов маршрутизатора по его MAC-адресу.

После задания адреса маршрутизатора по умолчанию стек TCP/IP компьютера *U*1 может работать, но только в том случае, если не заданы символьные имена хостов, то есть если пользователь или приложение обращается, например, к сайту компании Cisco не по его имени http://www.cisco.com, а непосредственно по IP-адресу, например, задав URL-адрес http://198.133.219.25. Если адрес корректен (он был корректен на момент написания этих строк), то стек будет работать. Однако намного удобнее и надежнее работать с символьными именами (поскольку адрес может измениться или же сайт для баланса нагрузки и обеспечения надежности может поддерживать несколько IP-адресов, что без символьных имен невозможно).

Для того чтобы стек TCP/IP компьютера *U*1 начал работать с символьными именами IP-хостов, нужно задать, по крайней мере, один адрес DNS-сервера, который будет принимать от компьютера *U*1 запросы, содержащие искомое символьное имя (например, www.cisco.com), и разрешать их, используя иерархию DNS-серверов Интернета или же данные своего кэша, если запрошенное имя в нем имеется. В данном случае администратор может задать адрес 212.100.40.7 DNS-сервера, который относится к той же организации. Принадлежность DNS-сервера к какой-либо сети или организации не принципиальна, но нужно иметь в виду, что многие администраторы конфигурируют свои DNS-серверы так, чтобы те не обслуживали запросы от «чужих» клиентов, выполняя фильтрацию на основе диапазона IP-адресов. Поэтому при конфигурировании клиентов нужно использовать адрес того DNS-сервера, про который точно известно, что он обслуживает запросы данной подсети.

Желательно также указать не один, а как минимум два адреса различных DNS-серверов, просто на тот случай, если один из них временно окажется неработоспособным. В данном случае это должен быть адрес внешнего DNS-сервера, так как в сети организации нет второго DNS-сервера. Таким DNS-сервером чаще всего служит DNS-сервер провайдера, обеспечивающего подключение данной сети к Интернету. На этом ручное конфигурирование стека TCP/IP заканчивается.

При использовании DHCP-сервера (а им в данном случае является маршрутизатор *R1*) администратор должен задать пул IP-адресов, который затем этот сервер задействует для ответов на запросы клиентов-компьютеров. Мы ограничимся рассмотрением только режима автоматического назначения динамических адресов, наиболее часто используемого при конфигурировании клиентских компьютеров (для серверов статические адреса удобнее, так как они позволяют упростить работу службы DNS; что касается клиентов, для них это обстоятельство не имеет большого значения, так как к клиентским компьютерам обычно никто по сети не обращается).

В нашем примере администратор должен задать пул адресов 212.100.10.0 с маской 255.255.255.0 и исключить из него те адреса, которые использованы или должны быть использованы при ручном конфигурировании. В данном случае необходимо из пула исключить адрес 212.100.10.1, принадлежащий интерфейсу маршрутизатора. Возможно, администратор захочет исключить также несколько младших адресов для ручного назначения в будущем, например, если в сети 212.100.0.0 появится сервер. После задания пула адресов клиентов на DHCP-сервере для этого пула необходимо задать адрес маршрутизатора по умолчанию и адреса DNS-серверов. Вся эта конфигурационная информация будет передаваться клиентским компьютерам подсети в ответ на их DHCP-запросы. Для того чтобы клиентский компьютер начал генерировать эти запросы, достаточно активизировать на нем режим DHCP — это будет единственное ручное действие по конфигурированию стека TCP/IP на клиентском компьютере, все остальное происходит в автоматическом режиме.

Cisco IOS

Маршрутизаторы представляют собой ключевой элемент любой современной крупной сети. Интернет построен на IP-маршрутизаторах, которые объединяют в единое целое сотни тысяч сетей. Компания Cisco Systems была и остается лидером в этом секторе сетевого оборудования, большая часть маршрутизаторов, работающих в Интернете, представляют собой продукты именно этой компании.

Сердцем маршрутизаторов Cisco является специализированная операционная система IOS (Internetwork Operating System — межсетевая операционная система), которая, как и любая ОС, выполняет функции по организации программного обеспечения и экранированию его от особенностей аппаратной платформы. Особенности Cisco IOS в том, что само прикладное программное обеспечение решает весьма узкоспециализированные задачи, а именно задачи маршрутизации пакетов, делая это в реальном времени, так как задержка маршрутизации пакета может серьезно ухудшить качество работы распределенного сетевого приложения, например приложения интернет-телефонии.

Функциональная схема маршрутизатора

Для того чтобы лучше понять, какие функции должна выполнять операционная система маршрутизатора, рассмотрим обобщенную функциональную схему

маршрутизатора, которую можно отнести как к маршрутизаторам Cisco, так как и к маршрутизаторам других производителей (рис. 9.14).

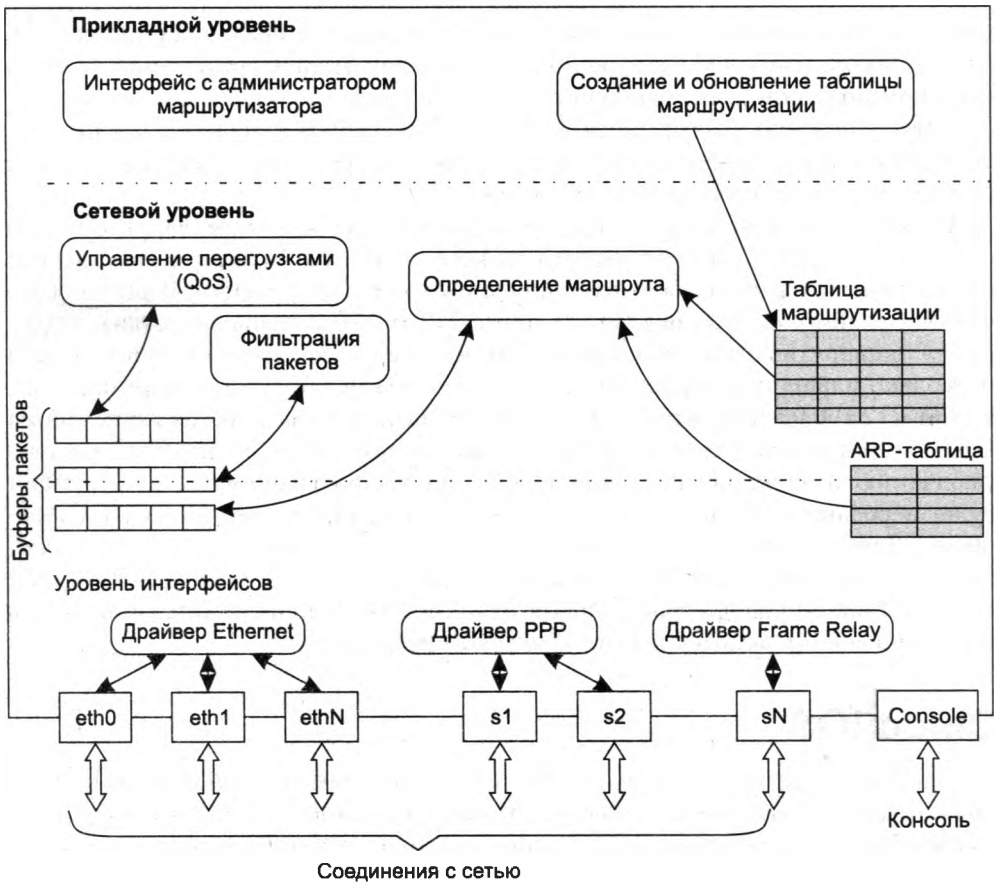


Рис. 9.14. Функциональная схема маршрутизатора

Как видно на этой схеме, все функции маршрутизатора делятся на уровни в соответствии с многоуровневым подходом, принятым при построении стека коммуникационных протоколов. Однако на рисунке показаны укрупненные уровни, которыми при описании работы маршрутизатора удобнее оперировать, чем семиуровневой моделью.

Нижний уровень — *уровень интерфейсов*. Он представлен портами маршрутизатора и драйверами протоколов канального уровня. Порт маршрутизатора представляет собой аппаратный элемент, с помощью которого маршрутизатор через соответствующий кабель соединяется с сетью и обменивается данными со своими непосредственными соседями — компьютерами, коммутаторами и другими маршрутизаторами. Порт оснащен контроллером — устройством, которое управляет логикой работы порта совместно с соответствующим драйвером.

Разделение функций между контроллером и драйвером зависит от реализации порта, в данной книге мы не будем вдаваться в детали этого разделения. Для нас главное здесь то, что контроллер и драйвер вместе обеспечивают поддержку некоторого протокола канального уровня, например Ethernet. На рис. 9.14 показаны также порты *s1–sN*, которые поддерживают такие протоколы глобальных каналов связи, как PPP и Frame Relay.

Работа порта (контроллера и драйвера) по приему пакетов заключается в том, что он принимает из канала связи биты кадра, инкапсулирующего пакет в соответствии с правилами канального протокола, который он реализует. Так, порт Ethernet принимает в свой внутренний буфер биты Ethernet-кадра, и в том случае, когда MAC-адрес назначения кадра совпадает с собственным MAC-адресом порта, выполняет дальнейшую обработку кадра — проверяет его целостность и корректность по контрольной сумме, имеющейся в кадре. Если полученный кадр корректен, он помещается в один из буферов пакетов, организованных в памяти маршрутизатора. Буферы пакетов являются одними из важнейших элементов маршрутизатора, они позволяют ему обрабатывать пульсации трафика в тех случаях, когда скорость поступления пакетов временно превышает возможности маршрутизатора по их продвижению.

Аналогично работают и порты других протоколов, разница только в том, что они поддерживают адресацию и форматы кадров другого стандарта.

После того как пакет попадает в буфер памяти, с ним начинают работать модули, реализующие функции *сетевого уровня*. Естественно, основной функцией сетевого уровня является маршрутизация. Сегодня это в подавляющем большинстве случаев IP-маршрутизация, но встречаются и другие сетевые протоколы, например IPX, поэтому маршрутизатор обычно является многопротокольным (этот термин применяется к протоколам сетевого уровня).

Если маршрутизатор определяет, что пришедший пакет является IP-пакетом, то для его обработки вызывается модуль протокола IP. Этот модуль выполняет все описанные ранее действия по маршрутизации пакета: он анализирует таблицу маршрутизации и пытается найти в ней номер сети, совпадающий с номером сети, взятым из адреса назначения обрабатываемого пакета. При совпадении этих номеров маршрут считается найденным, и сетевой протокол извлекает из таблицы маршрутизации IP-адрес следующего хопа — по этому адресу определяется порт маршрутизатора, на который нужно передать пакет. Затем просматривается ARP-таблица[1] и в ней по найденному IP-адресу ищется MAC-адрес следующего хопа, необходимый для формирования кадра и отправки его в сеть через выходной порт. Если в ARP-таблице нет соответствия для искомого IP-адреса, то сетевой протокол инициирует ARP-запрос, в результате чего ARP-таблица пополняется новым отображением.

[1] Это описание соответствует случаю, когда выходной порт поддерживает Ethernet. Для других протоколов канального уровня этот этап может либо требовать просмотра таблицы другого типа, например таблицы отображения IP-адресов на метки Frame Relay, либо вообще отсутствовать — для тех протоколов, где поиск канального адреса не нужен, например для протокола PPP, который работает на двухточечных соединениях, не требующих адресации.

Работа сетевого протокола заканчивается тем, что пакет помещается в выходной буфер соответствующего порта, куда также помещается найденный MAC-адрес. Далее этот пакет, инкапсулированный в кадр соответствующего канального протокола, порт побитно передает в канал связи.

Помимо описанной основной функции по маршрутизации пакетов, сетевой уровень выполняет также ряд дополнительных функций, из которых наиболее важными являются фильтрации пакетов и управление перегрузками.

Фильтрация заключается в проверке дополнительных условий продвижения пакета, которые администратор сети может сконфигурировать для какой-либо подсети назначения или отдельного хоста. Это очень мощный механизм первичного рубежа защиты сети, так как он позволяет, например, пропускать во внутреннюю сеть организации пакеты, исходящие только от филиалов этой организации. Обычно такие фильтры строятся на основе известных диапазонов IP-адресов, поэтому проверка условий таких фильтров — прямая обязанность сетевого уровня маршрутизатора.

Управление перегрузками (congestion management) требуется в тех случаях, когда маршрутизатор не успевает своевременно обрабатывать потоки поступающих пакетов, в результате чего возникают очереди в буферах пакетов, а при их переполнении — и потери пакетов (то есть пакеты, для которых нет места в буфере, отбрасываются). Такие ситуации могут временно возникать в маршрутизаторе даже в том случае, когда в среднем его производительность оказывается достаточной для обработки входящего трафика — причиной являются уже упомянутые пульсации компьютерного трафика. Для того чтобы справляться с временными перегрузками, в маршрутизаторах реализуются так называемые механизмы поддержания **качества обслуживания** (Quality of Service, QoS). Мы не будем здесь останавливаться на их детальном описании, отсылая интересующегося читателя к специальной литературе по этой теме (см., например, [1] или [8]), а ограничимся только краткой характеристикой этого подхода.

Механизмы QoS основаны на существовании нескольких выходных очередей к каждому порту вместо одной очереди, как при обычной обработке пакетов. Применяются два основных механизма обслуживания набора очередей — приоритетное обслуживание и взвешенное обслуживание. При приоритетном обслуживании очереди имеют различный приоритет, так что пакеты, помещаемые в более приоритетную очередь, передаются в выходной порт раньше пакетов из менее приоритетных очередей. В самую приоритетную очередь направляются пакеты приложений, которые наиболее чувствительны к задержкам и потерям пакетов, например приложений видеоконференций или IP-телефонии. Очевидным недостатком этого механизма является возможность дискриминации пакетов в менее приоритетных очередях, время ожидания в которых теоретически может достигать бесконечности. Для предотвращения таких ситуаций маршрутизаторы ограничивают входной поток высокоприоритетного трафика до уровня, заданного администратором, то есть логика здесь такая: «ваши пакеты будут обслуживаться лучше, чем пакеты других приложений, но

только в том случае, если ваш трафик не окажется слишком интенсивным и не будет вытеснять все другие потоки».

Взвешенное обслуживание является более «справедливым» механизмом, так как все очереди обслуживаются циклически, причем у каждой очереди есть свой «вес», соответствующий доле времени, которая выделяется этой очереди для передачи ее пакетов в выходной порт в каждом цикле. Например, администратор может организовать три взвешенные очереди к порту eth1, задав для них веса 10, 50 и 40 % соответственно. Если, например, eth1 имеет пропускную способность 100 Мбит/с, то первой очереди достанется 10 Мбит/с, второй — 50 Мбит/с, а третьей — 40 Мбит/с пропускной способности в среднем. Зная эти доли, администратор может направлять в различные очереди трафик различных приложений, обеспечивая приемлемое время ожидания пакетов в каждой из очередей в соответствии с типом приложений. Например, в первую очередь администратор может направить трафик очень чувствительных к задержкам приложений с суммарной интенсивностью в 5 Мбит/с, обеспечив тем самым трафику достаточно комфортные условия обслуживания, так как эта очередь может обслуживать пакеты с максимальной скоростью в 10 Мбит/с. Во вторую очередь администратор может направить трафик менее чувствительных приложений, ограничив суммарную интенсивность потоков в 40 Мбит/с. Это создаст для пакетов второй очереди не такие комфортные условия, как для пакетов первой очереди, так как пропускная способность этой очереди будет использоваться в среднем на 80 % (40 Мбит/с при требуемом значении в 50 Мбит/с), но, по крайней мере, приемлемые и контролируемые. В третью очередь целесообразно направить трафик всех остальных приложений без контроля интенсивности суммарного потока, так что перегрузки в этой очереди могут быть значительными, а задержки и потери пакетов неконтролируемыми.

Как видно из описания, маршрутизатор предоставляет в распоряжение администратора несколько базовых механизмов QoS, таких как приоритетные и взвешенные очереди. Эти механизмы служат для ограничения потоков пакетов, направляемых в каждую очередь, однако их применение требует продуманной стратегии и дополнительного ручного конфигурирования.

На *прикладном уровне* маршрутизатора реализуется несколько функций, к наиболее важным из которых относятся создание и модификация таблицы маршрутизации, а также поддержка интерфейса с администратором.

Таблица маршрутизации создается и поддерживается двумя способами: ручным конфигурированием, то есть администратором с использованием текстового или графического интерфейса, и автоматически, протоколами маршрутизации. Протоколов маршрутизации может работать несколько, даже если маршрутизатор сконфигурирован на поддержку только одного сетевого протокола IP (как чаще всего происходит сегодня). Это связано с тем, что интерфейсы маршрутизатора могут быть подключены к сетям, в которых используются различные протоколы маршрутизации, кроме того, маршрутизатор может дополнительно поддерживать протокол BGP, если он участвует в маршрутизации трафика между автономными системами.

Интерфейс администратора обеспечивает доступ к параметрам операционной системы маршрутизатора локально (через текстовый или графический терминал, подключенный к консольному порту маршрутизатора) или удаленно, через сеть. В последнем случае обычно используется протокол удаленного управления telnet стека TCP/IP. Telnet поддерживает эмуляцию алфавитно-цифрового экрана и практически является стандартом при удаленном управлении коммуникационными устройствами через IP-сеть. Графические интерфейсы поддерживаются реже, так как требуют больших затрат пропускной способности сети и усложняют программное обеспечение маршуртизатора.

Описанные функции составляют базовый набор функций любого маршрутизатора; естественно, помимо них могут поддерживаться и дополнительные функции, например управление по протоколу SNMP или CMIP, сбор статистики о передаваемом трафике и т. д.

Основные характеристики Cisco IOS

Операционная система Cisco IOS прошла длинный путь развития от библиотеки функций, написанных на языке высокого уровня, до модульной специализированной ОС, работающей на сотне типов различных аппаратных платформ. Хорошее представление о зрелости Cisco IOS дает номер ее текущей версии: 12.4. Основные принципы организации этой ОС отражают специфические требования по управлению маршрутизаторами, при этом часть этих требований взаимно противоречива, так что найденные разработчиками Cisco IOS решения часто представляют собой некоторый компромисс.

Рассмотрим основные требования, которые повлияли на организацию Cisco IOS.

- *Поддержка широкого спектра аппаратных платформ*, отличающихся как организацией, так и характеристиками элементов. На одном конце спектра аппаратных платформ находятся такие простые устройства, как, например, маршрутизаторы Cisco 800, имеющие один Ethernet-порт с пропускной способностью 10 Мбит/с, один последовательный порт с пропускной способностью до 2 Мбит/с, оперативную память на 4 Мбайта и флэш-память на 8 Мбайт для хранения файлов с кодом операционной системы и файлов конфигурации. Очевидно, что такая «скромная» аппаратная платформа требует, чтобы ОС была компактной, умещающейся в небольшой памяти и работающей достаточно быстро на сравнительно медленном процессоре. В то же время требования к набору функций такой ОС не очень велики: ей нужно маршрутизировать не более 14 800 пакетов в секунду (максимальная скорость поступления Ethernet-пакетов составляет 10 Мбит/с), таблица маршрутизации такого маршрутизатора вряд ли содержит более десятка записей, маршрутизатору нужно поддерживать IP на сетевом уровне, а также Ethernet и одну из технологий канального уровня глобальных сетей (например, PPP или HDLC) для последовательного порта.

  На другом конце спектра у компании Cisco сейчас находится платформа Cisco CRS-1 (Carrier Routing System — система маршрутизации операторов

связи), которая является модульной, многопроцессорной и в своей максимальной конфигурации может маршрутизировать пакеты с суммарной скоростью до 92 Тбит/с. Между этими двумя платформами располагаются промежуточные модели, отличающиеся как производительностью, так и аппаратной организацией.

Очевидно, что создание ОС, способной эффективно работать на платформах всего спектра, является непростым делом.

- *Поддержка широкого спектра протоколов.* Хотя на сетевом уровне сегодня в подавляющем числе случаев достаточно поддержки IP, список протоколов маршрутизации, а также канальных протоколов по-прежнему велик. Коммуникационные протоколы являются для ОС маршрутизатора приложениями, которые нужно выполнять.
- *Работа в реальном времени.* Мы уже говорили о том, что маршрутизатор работает в реальном масштабе времени и что задержки пакетов (особенно неравномерные задержки, характерные для обработки пульсирующего компьютерного трафика) могут привести к снижению качества выполнения сетевых приложений, например, появлению «снега» на экране при воспроизведении видео или эха при разговоре по IP-телефону. Потери пакетов страшны не только для чувствительных к задержкам приложений, но и для традиционных «эластичных» приложений, таких как загрузка файлов, так как потеря одного пакета ведет к негодности большого документа. Поэтому ОС маршрутизатора должна вносить минимальные и ограниченные каким-то верхним пределом задержки при выполнении своих приложений.

Cisco IOS удовлетворяет перечисленным требованием за счет следующих особенностей организации.

- *Модульность на этапе разработки и сборки системы, монолитный образ (файл кода) операционной системы на этапе эксплуатации маршрутизатора.* Цель — высокая скорость работы ОС и приложений как единой программы на этапе эксплуатации маршрутизатора с сохранением модульности ОС для работы на разных аппаратных платформах и поддержки разного набора протоколов.
- *Отказ от использования виртуальной памяти с хранением страниц на диске.* Операционная система и ее приложения-протоколы размещаются полностью в физической памяти маршрутизатора. Цель — ускорение работы ОС.
- *Упрощенная файловая система.* Все файлы образов системы и конфигурации хранятся в одном каталоге одного из разделов флэш-памяти. Цель — упрощение ОС и ускорение ее работы.
- *Работа ядра ОС и приложений-протоколов в одном и том же режиме процессора.* Цель — ускорение работы ОС.
- *Индексирование таблицы маршрутизации различными способами.* Цель — ускорение процесса маршрутизации.

Рассмотрим эти свойства Cisco IOS подробнее.

Модульная структура IOS

Cisco IOS имеет модульную структуру, которая достаточно традиционна — она состоит из модулей (сервисов в терминологии Cisco) ядра и сетевых модулей, которые соответствуют приложениям универсальной ОС (рис. 9.15).

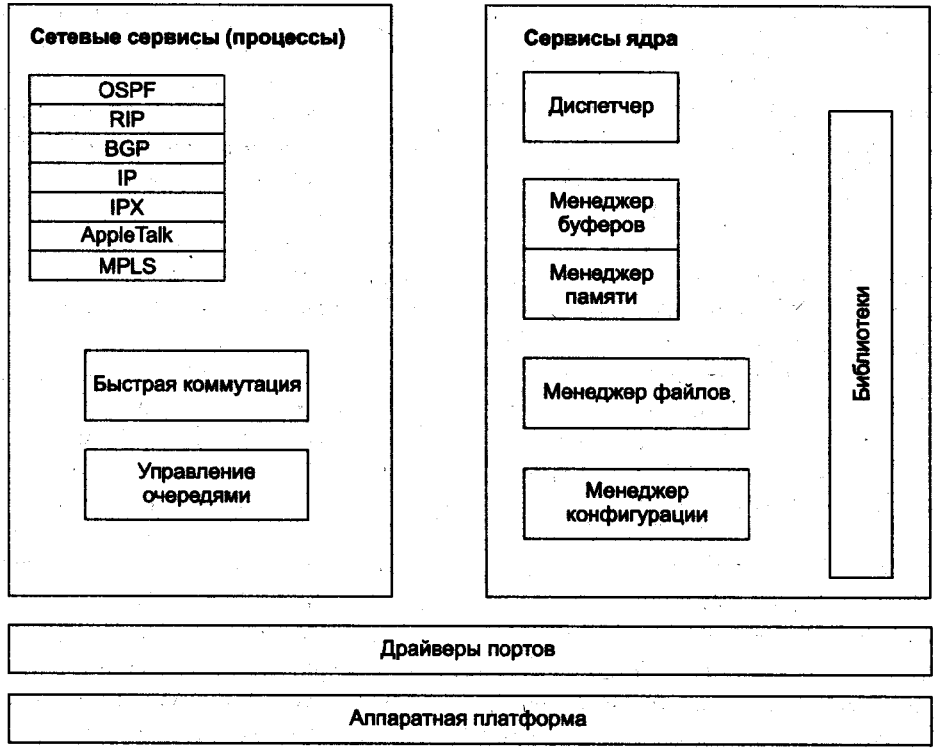


Рис. 9.15. Модульная структура Cisco IOS

Однако в отличие от универсальной ОС в Cisco IOS отсутствует поддержка приложений, написанных программистами других компаний. Универсальная ОС должна обеспечивать выполнение приложений, написанных любым программистом в соответствии с правилами, существующими для данной ОС. Например, любой программист может написать приложение для Unix или Windows, используя интерфейсы прикладного программирования этих ОС. После этого он получает в свое распоряжение средства разработчика, которые позволяют ему оттранслировать написанную программу и создать исполняемый модуль в виде файла определенной структуры. Этот модуль операционная система должна легко загружать и выполнять наряду с другими приложениями, написанными различными программистами. Для такой специализированной ОС, как ОС маршрутизатора, ситуация существенно отличается: новые модули появляются в результате работы исключительно программистов компании, выпускающей маршрутизаторы, в нашем примере — программистов Cisco. Добав-

ление приложений обычными пользователями маршрутизатора не является необходимым свойством маршрутизатора, поэтому все необходимые этапы компиляции новых программных модулей и включения их в состав ОС могут быть выполнены заранее, на этапе выпуска маршрутизатора.

Большая часть сетевых сервисов Cisco IOS реализуются как процессы, вызываемые диспетчером ядра. Модули сетевых сервисов выполняют функции коммуникационных протоколов, таких как сетевые протоколы IP, IPX или же протоколы маршрутизации RIP, OSPF или BGP. Кроме того, в состав сетевых сервисов входят некоторые модули, реализующие общие для коммуникационных протоколов функции, например управление очередями (поддержка функций QoS) или быструю коммутацию (см. далее раздел «Программная маршрутизация и ускоренная коммутация»).

Помимо диспетчера в состав модулей ядра входят также менеджер памяти, менеджер буферов пакетов, менеджер файлов и менеджер конфигурации маршрутизатора, набор библиотек функций ядра, а также рад других вспомогательных модулей, не показанных на рисунке.

Модульное строение Cisco IOS «видно» только разработчикам IOS, так как для потребителя маршрутизатора, которым является администратор сети, конфигурирующий маршрутизатор, модульность скрыта за монолитным образом IOS, который представляет собой исполняемый файл, хранится во флэш-памяти маршрутизатора и загружается в оперативную память при его старте. Так как пользователь маршрутизатора не может изменить состав сетевых сервисов после покупки и установки определенного образа IOS, то для того, чтобы обеспечить необходимую степень гибкости, существует несколько версий образов Cisco IOS для каждой аппаратной платформы, которые отличаются наборами реализованных сетевых сервисов, называемых также наборами свойств (feature sets) IOS. Например, в базовый набор свойств образа IOS для маршрутизаторов младшего семейства моделей Cisco 800 (называемый IP) входит только поддержка сетевого протокола IP. В расширенный набор свойств IOS для этой же модели (IP Plus) входит также ряд дополнительных модулей, например агент измерения параметров качества обслуживания (задержек и потерь пакетов), называемый Cisco SAA. Для того чтобы маршрутизатор Cisco 805 поддерживал не только протокол IP, но и IPX, нужно использовать образ с набором свойств IP/IPX Plus.

Специфика аппаратной платформы маршрутизатора учитывается наличием различных версий модулей ядра, а также модулей некоторых сетевых сервисов. Например, в модулях ядра IOS для платформы Cisco 12000 учтен тот факт, что интерфейсные карты маршрутизаторов этого семейства обладают собственными процессорами. Очевидно, что модули ядра для платформы Cisco 800 намного проще и компактней соответствующих модулей платформы Cisco 800.

Таким образом, чтобы выбрать нужный образ IOS для какого-либо маршрутизатора Cisco на сайте www.cisco.com, администратор сети должен сначала указать интересующую его аппаратную платформу, а затем выбрать образ IOS, набор свойств которого включает все необходимые для данного маршрутизатора

функции. Если же со временем выясняется, что маршрутизатору нужна новая функциональность, не поддерживаемая в выбранном образе IOS, то необходимо найти такой образ, который поддерживает искомую функцию, приобрести его и загрузить во флэш-память маршрутизатора.

Выбранный подход к обеспечению модульности Cisco IOS возможно не является достаточно удобным с точки зрения администратора сети, зато он позволяет IOS эффективно работать на принципиально различных аппаратных платформах и поддерживать набор нужных функций без излишней избыточности.

Прерывания и управление процессами

В Cisco IOS существует понятие процесса, а в ядре имеется специальный модуль — диспетчер, который управляет переключением между процессами. Механизм программных потоков (нитей) IOS не поддерживает. В виде процессов в Cisco IOS функционируют в первую очередь программные модули, реализующие сетевые протоколы и протоколы маршрутизации. Например, выполнив с консоли маршрутизатора Cisco команду `show process`, можно с большой вероятностью ожидать нахождения в полученном списке выполняющихся процессов имен IP Input, ARR Input, RIP Router — реализаций протоколов IP, ARP и RIP соответственно.

Как ядро, так и процессы работают в одном и том же режиме процессора, так что IOS не поддерживает защиту ядра от процессов и процессов друг от друга с помощью привилегированного режима процессора. Такой подход к организации ОС предъявляет высокие требования к качеству написания кода процесса, чтобы этот код не разрушил ядро и коды других процессов и не привел к краху системы. Если такое условие соблюдено, то работа ядра и процессов в одном режиме дает очевидное преимущество в реактивности системы благодаря отсутствию постоянного переключения между контекстами привилегированного и обычного режимов. А, как мы знаем, повышение реактивности системы является одним из главных требований, предъявляемых к ОС маршрутизатора, который должен быстро реагировать на приход очередного пакета.

Еще одним фактором, направленным на сокращение потерь времени при переключении с процесса на процесс, является использование в Cisco IOS режима невытесняющей многозадачности. В главе 4 мы рассматривали этот способ организации многозадачного режима ОС и отмечали, что его основным достоинством является минимизация накладных расходов ОС на переключение процессов по сравнению с режимом вытесняющей многозадачности. В то же время очевидным недостатком этого решения является потенциальное снижение надежности и устойчивости системы, так как ошибка в коде какого-либо процесса замедляет выполнение других процессов или ведет к их полной остановке.

В Cisco IOS с этой потенциальной опасностью борются двумя способами: помимо уже упоминавшейся тщательной отладки модулей-процессов, используется сторожевой таймер, который запускается при очередной передаче управления какому-либо процессу. Сторожевой таймер каждые 2 секунды получает управление по прерыванию, и если во второй раз застает процесс незавершен-

ным, таймер принудительно останавливает выполнение процесса и передает управление диспетчеру.

Диспетчер IOS ведет четыре очереди готовых к выполнению процессов, отличающихся приоритетами. В порядке убывания приоритета это очереди:

- критического приоритета;
- высокого приоритета;
- среднего приоритета;
- низкого приоритета.

Большая часть процессов работает со средним приоритетом, например процессы IP Input, ARP Input и RIP Router. Критический приоритет имеют некоторые системные процессы, например процесс измерения нагрузки маршрутизатора. Высокий приоритет назначается чаще всего тем процессам, которые выполняют срочную часть работы коммуникационного протокола (например, процесс RIP Send периодически посылает маршрутные объявления в сеть). Фоновые процессы, такие как проверка устаревших записей кэша, получают низкий приоритет.

Диспетчер проверяет очереди готовых процессов, начиная с очереди критического приоритета. Если она оказывается пустой, диспетчер переходит к просмотру очереди процессов с высоким приоритетом, затем — со средним, и только если все эти очереди пусты, — к очереди процессов с низким приоритетом. Чтобы предотвратить слишком долгое ожидание процессов с низким приоритетом, диспетчер вызывает на выполнение процесс из этой очереди через каждые 15 запусков процессов с более высокими приоритетами.

Прерывания широко используются в Cisco IOS для быстрой реакции на внешние события, основным из которых является приход пакета в буфер. При обработке прерывания IOS пытается проделать максимум возможного за выделенное на это время, стремясь, чтобы обработка прерывания в среднем заканчивалась раньше возникновения запроса на следующее прерывание, иначе очередь необработанных прерываний быстро переполнится.

Как мы увидим далее, при определенной организации данных в таблице маршрутизации и ARP-таблице Cisco IOS успевает выполнить все операции по продвижению пакета за один цикл обработки прерываний.

Организация памяти

Cisco IOS не использует механизм виртуальной памяти, так как в данном случае этот механизм и не желателен, и не обязателен.

Он не желателен потому, что режим реального времени требует, чтобы все коды всех активных процессов постоянно находились в памяти для быстрого выполнения и переключения с процесса на процесс. Наличие механизмов виртуальной памяти с ее подкачкой страниц из дисковой памяти в виртуальную создавало бы слишком длительные случайные паузы в работе процессов — ситуация, приемлемая для универсальных ОС, но крайне нежелательная для систем реального времени, к которым относится Cisco IOS.

Механизм виртуальной памяти не обязателен для Cisco IOS, потому что размеры образов IOS в памяти для каждой аппаратной платформы известны, они не меняются в ходе эксплуатации, так как, как уже было сказано, у пользователя IOS нет возможности добавлять новые модули к установленному образу IOS. А раз максимальный размер образа IOS известен, то достаточно оснастить аппаратную платформу физической оперативной памятью требуемого размера, позволяющего загрузить туда коды всех процессов для максимально возможного набора сетевых сервисов, оставив место для буферов пакетов. Например, модели Cisco 800 оснащены оперативной памятью в 8 Мбайт, а размеры образов IOS для этой модели колеблются в диапазоне 2–4 Мбайт.

На рис. 9.16 обобщенно показана организация памяти маршрутизаторов Cisco. Оперативная память разделена на две части: основную память и память ввода-вывода. Отличие этих двух областей памяти заключается в том, что к основной памяти доступ имеет только процессор, в то время как к памяти ввода-вывода могут обращаться как процессор, так и контроллеры ввода-вывода в режиме прямого доступа к памяти. Это свойство памяти ввода-вывода позволяет организовать параллельную перепись в буферы памяти ввода-вывода пакетов, поступивших во внутренние буферы контроллеров, без участия процессора, который в это время может заниматься задачами маршрутизации пакетов и корректировки таблицы маршрутизации.

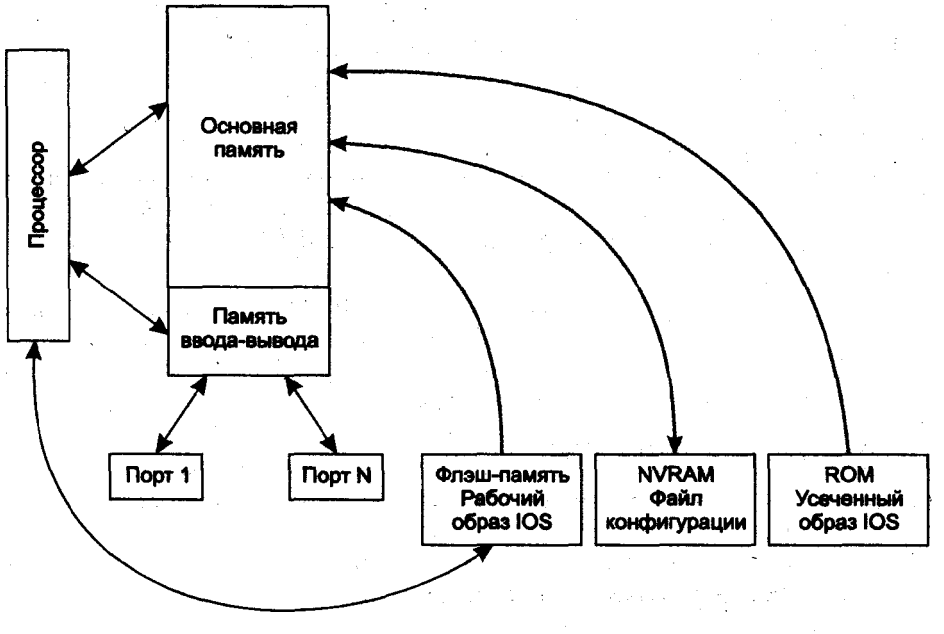


Рис. 9.16. Организация памяти в маршрутизаторах Cisco

Память ввода-вывода полностью отводится под буферы пакетов, а основная память процессора делится на несколько областей, соответствующих типовой

структуре программы: области кодов IOS и областей инициализированных и неинициализированных переменных IOS. В основной памяти есть также область буферов пакетов, дополняющая область буферов пакетов памяти ввода-вывода.

Маршрутизаторы Cisco поддерживают также несколько типов памяти для хранения файлов: флэш-память, постоянную память (ROM) и не разрушающуюся память (NVRAM). Основное назначение флэш-памяти — хранение образа IOS, загружаемого при старте маршрутизатора в основную память. Так как флэш-память обеспечивает произвольный доступ при чтении и поблочное стирание, коды некоторых образов IOS (или некоторых программ образа) выполняются непосредственно из флэш-памяти без загрузки в основную память. Очевидно, что для этого программы IOS должны быть написаны определенным образом, а именно быть реентерабельными. Для поддержания этой функции за флэш-памятью закрепляют определенный диапазон адресов адресного пространства маршрутизатора.

В постоянной памяти хранится усеченный образ IOS, который может быть вызван на выполнение, когда основной образ IOS, хранящийся во флэш-памяти, по каким-то причинам не работает. Это своего рода аварийная версия IOS, которая позволяет администратору сети решить проблему неработающей основной версии IOS: найти ошибку в конфигурации и исправить ее или же полностью заменить образ основной версии IOS новым.

NVRAM (Non-Volatile Random Access Memory — не разрушающаяся при отключении питания память) хранит файл конфигурации IOS. В этом файле записана информация о том, какие протоколы должны быть активными и с какими параметрами. Например, для каждого интерфейса файл конфигурации хранит большое количество параметров, среди которых находится IP-адрес этого интерфейса, но используется IP-адрес только в том случае, если для данного интерфейса активен протокол IP. Файл конфигурации при старте системы считывается кодом IOS в оперативную память и активизирует соответствующие переменные IOS.

Файловые системы флэш-памяти, ROM и NVRAM очень просты в отношении организации файлов: в них существует только корневой каталог, в котором и размещаются все файлы данного типа внешней памяти. Такая организация упрощает операционную систему Cisco IOS и ускоряет ее работу, а отсутствие иерархии каталогов не приводит к проблемам в работе администратора сети, так как в каждом из накопителей хранится совсем немного файлов, например, один-два образа IOS во флэш-памяти и два-три файла конфигурации в памяти NVRAM.

Работа с буферами пакетов

Управление оперативной памятью в Cisco IOS происходит на двух уровнях: уровне пулов памяти и уровне буферов пакетов. На первом уровне управление памятью осуществляется *менеджером памяти*, на втором — *менеджеров буферов*. Пул памяти IOS — это непрерывная область памяти произвольного размера.

Пулы памяти управляются менеджером памяти и выделяются по запросу модулям IOS; в этом отношении функции IOS не отличаются от функций универсальной ОС.

Уровень буферов пакетов представляет собой специфический механизм IOS. Его наличие отражает тот факт, что операции с пакетами в оперативной памяти (перемещение пакетов между буферами контроллеров интерфейсов и памятью ввода-вывода, перемещение пакетов между памятью ввода-вывода и основной памятью) являются критичными операциями IOS, существенно влияющими на скорость маршрутизации.

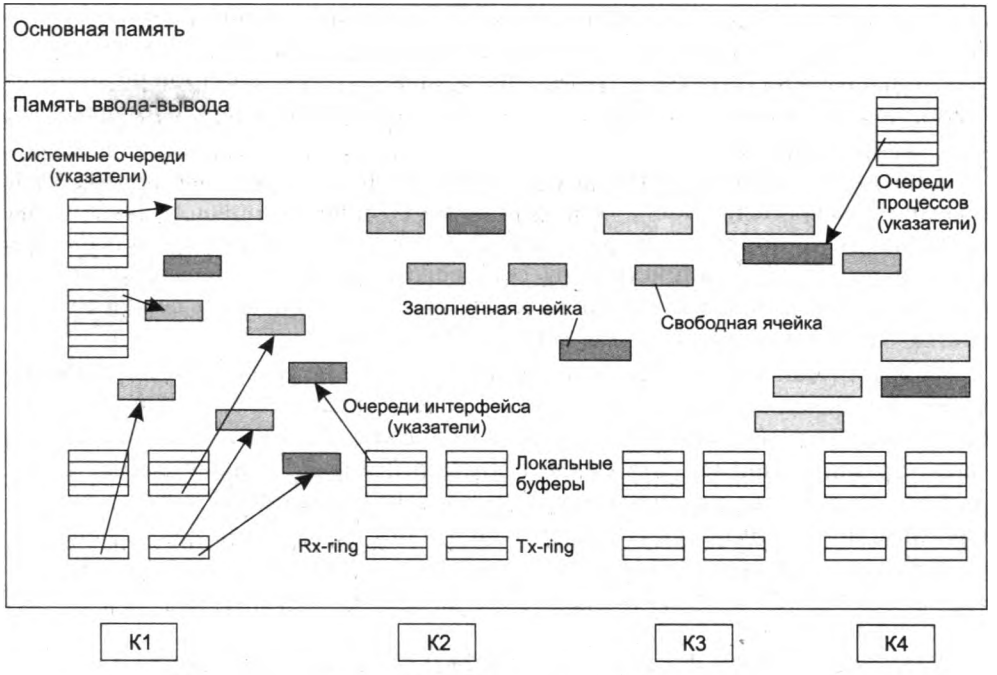


Рис. 9.17. Буферы пакетов IOS

Буферы образуют очереди пакетов, в которых пакеты ожидают своей обработки (рис. 9.17). Каждый буфер состоит из области памяти, предназначенной для хранения пакета, и указателя на эту область. На рисунке области памяти названы ячейками. Как видно из рисунка, очереди образуются за счет группировки указателей, в то время как ячейки памяти, хранящие пакеты одной и той же очереди, могут располагаться произвольно в любом месте памяти. Такая организация очередей весьма экономична, так как при перемещении пакета из одной очереди в другую нет необходимости копировать данные пакета из ячейки в ячейку — достаточно просто изменить значения соответствующих указателей очередей. IOS поддерживает несколько типов буферов и связанных с ними очередей пакетов: кольца интерфейсов Tx-ring и Rx-ring, локальные и системные.

С каждым интерфейсом связаны две *очереди Tx-ring* и *Rx-ring*, предназначенные соответственно для передачи пакетов непосредственно в контроллер и приема пакетов от контроллера данного интерфейса. Отличие очередей этого типа состоит в том, что к их указателям имеет доступ не только процессор маршрутизатора, но и контроллер данного интерфейса. За счет этого экономится процессное время при перемещении пакетов между памятью контроллера и буферами IOS. Например, когда пакет оказывается полностью принятым контроллером интерфейса, последний не инициирует прерывание, чтобы известить процессор о необходимости переписать биты пакета в буфер IOS, а выполняет эту операцию сам. Для этого контроллер читает значение указателя первой ячейки очереди Rx-ring, и если этот указатель имеет признак «свободен», то контроллер записывает пакет по адресу, хранящемуся в указателе. Структуры указателей очередей Tx-ring и Rx-ring зависят от типа интерфейса, размеры ячеек этих очередей также зависят от типа интерфейса — они выбираются таким образом, чтобы в ячейку мог поместиться кадр максимального для данного типа интерфейса размера. Например, ячейки интерфейсов Ethernet имеют размер 1524 байта, что позволяет поместить туда 1500 байт поля данных Ethernet-кадра (максимальная длина этого поля), а также все байты заголовка кадра.

Если после приема кадра контроллер интерфейса не может найти в очереди Rx-ring указатель на свободную ячейку, то фиксируется переполнение кольца Rx-ring, и кадр теряется.

Помимо очередей Tx-ring и Rx-ring, с каждым интерфейсом связана также пара очередей (входная и выходная), которые называются **локальными очередями** данного интерфейса, так как другие интерфейсы ими пользоваться не могут. Эти очереди контроллеру интерфейса недоступны, они доступны только кодам, выполняемым процессором маршрутизатора. Локальные очереди также включают ячейки фиксированного размера, зависящего от максимального размера кадра интерфейса. Как правило, количество буферов в очередях Tx-ring и Rx-ring невелико, локальные очереди содержат больше буферов, что позволяет маршрутизатору не терять пакеты даже при сравнительно длительных перегрузках. Локальные очереди, как и кольца Tx-ring и Rx-ring, состоят из фиксированного количества буферов, но это количество можно до определенного предела увеличивать администратору, используя команды конфигурирования маршрутизатора.

И наконец, существуют так называемые **системные очереди** пакетов, главная особенность которых состоит в том, что они могут быть использованы для хранения пакетов любых интерфейсов. Системные очереди нужны для того, чтобы хранить пакеты в периоды перегрузки какого-либо интерфейса, когда локальная очередь этого интерфейса полностью заполняется. В этом случае код IOS пытается найти свободный указатель в одной из системных очередей.

Так как системные очереди обслуживают интерфейсы различных типов, то они должны рационально хранить кадры различного размера. Cisco IOS решает эту проблему двумя способами. Для младших моделей маршрутизаторов Cisco организуется несколько системных очередей, каждая из которых поддерживает

ячейки определенного размера. В этом случае в памяти ввода-вывода организуется семь очередей с размерами из следующего ряда: 104, 600, 1524, 4520, 5024 и 18 024.

В старших моделях маршрутизаторов для уменьшения эффекта фрагментации ячейка системной очереди может состоять из нескольких фрагментов, расположенных в памяти произвольным образом. Эти фрагменты с помощью дополнительных указателей динамически связываются в ячейку необходимой длины. Механизм фрагментов позволяет более эффективно использовать память, но усложняет организацию IOS, поэтому в младших моделях с относительно медленным процессором и небольшим объемом памяти этот механизм не применяется.

Перечисленные три типа очередей образуют три уровня, которые взаимодействуют следующим образом. В исходном состоянии все очереди инициализируются указателями на свободные ячейки памяти ввода-вывода. Процесс записи кадра в очередь Tx-ring интерфейса мы уже рассмотрели. После завершения этой операции контроллер интерфейса, который выполнил ее автономно, генерирует прерывание.

Код ядра IOS, вызываемый по этому прерыванию, пытается переместить ячейку с принятым кадром из очереди Tx-ring в локальную входную очередь интерфейса или же, если она полна, в одну из системных очередей. Для этого коду ядра сначала нужно вернуть свободную ячейку в очередь Tx-ring. Для этого код ядра просматривает локальную входную очередь данного интерфейса, а если в ней нет свободных ячеек, то и системную очередь. Если он находит в одной из этих очередей указатель на свободную ячейку, он копирует его значение в указатель очереди Tx-ring, делая тем самым этот буфер свободным для приема кадра. После этого адрес ячейки с принятым кадром помещается в указатель на свободную ячейку локальной или системной очереди. Таким образом, ячейка переходит из очереди в очередь без реального перемещения кадра в памяти, что значительно сокращает время этой операции.

В том случае, когда ни в локальной, ни в системной очередях нет указателей на свободные ячейки, кадр отбрасывается. Для этого признак указателя на ячейку, содержащую кадр, меняет свое значение с «заполненная ячейка» на «свободная ячейка».

После того как пакет, находящийся в составе кадра, попадает в локальную или системную очередь, его можно маршрутизировать.

Программная маршрутизация и ускоренная коммутация

Маршрутизаторы Cisco поддерживают несколько вариантов маршрутизации, которые можно разделить на две группы.

В первую группу входят классические варианты маршрутизации, характерные для любого маршрутизатора на основе универсальной ОС и называемые **программной маршрутизацией** (например, IP- или IPX-маршрутизация).

Во вторую группу входит несколько режимов IP-маршрутизации, которых объединяет то, что они реализуют разными способами одну и ту же идею ускорения маршрутизации за счет предварительного создания дополнительных структур данных, ускоряющих поиск необходимых записей в таблице маршрутизации и ARP-таблице. В результате создается новая таблица, которая содержит не IP-адрес следующего хопа, а его MAC-адрес, то есть конечную цель продвижения пакета. Так как эта структура маршрутной информации похожа на таблицу продвижения коммутаторов, методы, использующие такие таблицы, получили название **ускоренной коммутации**.

Рассмотрим сначала, как происходит программная маршрутизация, на примере IP-маршрутизации. После того как кадр, содержащий пакет, подлежащий маршрутизации, попадает в локальную или системную очередь, программа обработки прерываний определяет, какому протоколу сетевого уровня соответствует пришедший пакет, и перемещает его во входную очередь процесса, реализующего этот протокол. Для нашего примера это будет очередь пакетов, ожидающих процесс IP Input, который занимается маршрутизацией IP-пакетов. Перемещение в эту очередь может потребовать для некоторых платформ физического перемещения кадра, содержащего пакет, из памяти ввода-вывода в основную память — это, естественно, значительно замедляет маршрутизацию. Для других платформ эта операция выполняется так же, как и в случае перемещения кадра из очереди Tx-ring, то есть за счет изменения значений указателей, но без физического копирования кадра.

После перемещения кадра в очередь процесса маршрутизации работа секции обработки прерывания заканчивается. Кадр ожидает, когда процесс IP Input, в очередной раз получив управление, обнаружит данный кадр в начале очереди. Далее процесс IP Input производит необходимые операции с таблицей маршрутизации и ARP-таблицей. Просмотр таблицы маршрутизации может занять много времени, если маршрутизатор работает на магистрали Интернета, так как в этом случае таблица может включать несколько десятков тысяч записей. После определения MAC-адреса назначения и выходного интерфейса, через который нужно передать кадр в сеть, процесс IP Input формирует новый кадр, помещая найденный адрес в поле адреса назначения кадра. Кадр, готовый к отправке в сеть, процесс IP Input помещает либо сразу в очередь Tx-ring выходного интерфейса, если она содержит свободный буфер, либо в локальную или системную очередь в противном случае. В первом случае контроллер интерфейса автономно обнаруживает факт наличия нового пакета в очереди Tx-ring и переписывает его в свой буфер, а затем побитно пересылает в канал связи. Во втором случае код обработки прерывания ядра, который вызывается контроллером интерфейса каждый раз после передачи очередного кадра в сеть, переписывает кадр из локальной или системной очереди в очередь Tx-ring.

Теперь посмотрим, как работают методы ускоренной коммутации на примере двух технологий, поддерживаемых маршрутизаторами Cisco: быстрой коммутации и экспресс-продвижения Cisco.

Быстрая коммутация (fast switching) основана на кэшировании информации, находимой процессом IP Input при маршрутизации пакета с определенным адресом назначения при просмотре таблицы маршрутизации и ARP-таблицы. Эта информация связывает IP-адрес назначения с выходным интерфейсом и MAC-адресом следующего хопа, то есть содержит необходимый минимум для формирования и отправки кадра следующему маршрутизатору или узлу назначения.

Если маршрутизатор поддерживает режим быстрого кэширования (он может быть включен по умолчанию или же активизирован вручную), то при маршрутизации очередного пакета маршрутизатор сначала просматривает кэш, а в случае отсутствия там IP-адреса назначения пакета просматривается таблица маршрутизации обычным способом. В том и другом случаях аргументом поиска является IP-адрес назначения, извлеченный из маршрутизируемого пакета.

Для быстрого просмотра кэша Cisco IOS выполняет детерминированное отображение IP-адреса на элемент таблицы кэша. Ранние реализации технологии быстрой коммутации использовали для этого хэширование (см. раздел «Отображение основной памяти на кэш» в главе 5). Для получения номера элемента кэша над старшими и младшими частями IP-адреса дважды применяется побитовая операция «исключающее ИЛИ». Начиная с версии 10, IOS применяет для ускорения поиска в кэше дополнительные структуры индексов, называемые двоичными деревьями.

Сокращение времени маршрутизации при использовании кэша позволило все операции по маршрутизации пакета выполнять в режиме обработки прерывания от контроллера интерфейса без обращения к процессу IP Input. Это, в свою очередь, приводит к дополнительной экономии времени, так как исключает период ожидания очередной итерации процесса IP Input. Таким образом, программа обработки прерываний пытается выполнить все операции по маршрутизации пакета самостоятельно, используя таблицу кэша маршрутов, и только в том случае, когда в кэше нужный адрес отсутствует, пакет помещается в очередь процесса IP Input и обрабатывается так же, как и в случае программной маршрутизации.

Размер кэша для протокола IP определяется автоматически в зависимости от размера свободной памяти, оставшейся после размещения там кодов IOS и буферов пакетов. При изменении состояния таблицы маршрутизации или ARP-таблицы кэш перестраивается. Кроме того, из кэша периодически удаляются «устаревшие» записи, то есть записи, срок жизни которых превысил определенный порог.

Кэширование в режиме быстрой коммутации значительно ускоряет маршрутизацию пакетов в тех случаях, когда состояние кэша является стабильным, то есть его не нужно часто перестраивать, так как в последнем случае затраты на перестройку кэша могут превысить экономию времени от его использования. Практика показывает, что в маршрутизаторах, работающих на периферии Интернета, то есть в корпоративных сетях, набор IP-адресов, к которым обращаются пользователи, является достаточно стабильным и не очень большим. В ре-

зультате все эти адреса попадают в кэш, который перестраивается редко, так что его применение для таких сетей эффективно.

Маршрутизаторы, работающие на магистрали Интернет, сталкиваются чаще всего с ситуацией, когда кэширование в режиме быстрой коммутации не дает желаемого эффекта из-за постоянной перестройки кэша. Это происходит потому, что пакеты, проходящие через магистраль, адресуются практически ко всему диапазону IP-адресов, так что стабильное подмножество IP-адресов назначения, на которое рассчитано кэширование в режиме быстрой коммутации, просто отсутствует.

Для магистральных маршрутизаторов Cisco IOS предлагает другой метод ускоренной маршрутизации, названный **экспресс-продвижением Cisco** (Cisco Express Forwarding, CEF). В этом методе также используются записи, аналогичные по структуре записям кэша метода быстрой коммутации, то есть записи, связывающие IP-адрес назначения с выходным интерфейсом и MAC-адресом следующего хопа. Однако основное отличие метода CEF заключается в том, что такие записи, называемые здесь записями таблицы связей (adjacency table), создаются для *каждой* записи таблицы маршрутизации, а не только для тех записей, которые активно использовались последними. Записи таблицы связей создаются *заранее* при инициации режима CEF, а не при приходе пакета с определенным адресом, как это происходит в методе быстрой коммутации. В результате таблица связей в режиме CEF перестраивается у магистральных маршрутизаторов гораздо реже, чем кэш в режиме быстрой коммутации, — она перестраивается при изменении адреса и интерфейса следующего хопа, что может произойти, например, из-за отказа линии связи или какого-либо магистрального маршрутизатора.

Для быстрого поиска нужной записи в таблице связей CEF по IP-адресу назначения IOS использует M-деревья — древовидную структуру индексов, в которой каждому возможному значению IP-адреса соответствует свой путь до искомой записи таблицы связей. В M-дереве у каждого элемента индекса существует 256 потомков, по количеству значений одного байта IP-адреса. Корневой элемент этого дерева указывает на 256 потомков, соответствующих 256 значениям старшего байта адреса, то есть ветви, ведущие от корня, направляются к адресам, имеющим в старшем байте значение 1.0.0.0, 2.0.0.0, 3.0.0.0 и т. д. вплоть до 255.0.0.0. При использовании такого дерева индексов путь по дереву до конечного элемента, указывающего на соответствующую запись таблицы связей, состоит из одного и того же количества промежуточных ветвей, а именно из четырех (по количеству байтов в IP-адресе). Этот метод индексации может показаться слишком громоздким, так как в принципе он ведет к дереву, включающему пути ко всем возможным значениям IP-адресов, то есть более чем к 4 миллиардам конечных элементов. Однако эта громоздкость кажущаяся, многие ветви дерева заканчиваются нулевыми (то есть не ведущими далее) указателями, причем это могут быть не только ветви нижнего, четвертого, уровня, но и ветви более высоких уровней. Усечение дерева происходит благодаря тому, что оно ведет к элементам таблицы связей, а их количество равно количеству

записей таблицы маршрутизации, которое даже для магистральных маршрутизаторов Интернета не превышает, как правило, нескольких десятков тысяч (результат агрегирования IP-адресов).

Поддержка QoS

Cisco IOS поддерживает большое количество (более 100) механизмов QoS, причем иногда одного и того же результата можно добиться разными способами. Это объясняется тем, что за долгое время своего существования и развития механизмы QoS встраивались в IOS по мере необходимости и в соответствие с различными стандартами. Такая несколько запутанная ситуация с поддержанием качества обслуживания в Cisco IOS значительно упростилась с появлением модульного интерфейса QoS (Modular QoS Command line interface, MQC). MQC стандартизует правила использования механизмов QoS и делает операции конфигурирования маршрутизаторов Cisco по поддержке QoS гораздо более понятными.

Для организации дифференцированного обслуживания пактов MQC использует дополнительные выходные очереди интерфейсов. Если в стандартном режиме для каждого интерфейса существует одна локальная выходная очередь, то при активизировании механизмов QoS таких очередей организуется несколько. MQC поддерживает для каждого интерфейса одну приоритетную очередь, называемую **очередью с низкой задержкой** (Low Latency Queue, LLQ), и несколько взвешенных очередей. Как и было описано ранее, пакеты, находящиеся в приоритетной очереди, выбираются из нее в первую очередь, а пакеты, находящиеся во взвешенных очередях, выбираются из них только в том случае, если приоритетная очередь пуста.

При конфигурировании QoS на маршрутизаторах Cisco нужно иметь в виду, что пакеты из очередей QoS попадают в контроллер интерфейса через единственную очередь Tx-ring, обойти ее никак нельзя, так как использование очереди Tx-ring — это единственный способ для контроллера интерфейса переписать пакеты из памяти ввода-вывода в свою внутреннюю память. Наличие очереди Tx-ring несколько искажает картину обслуживания пакетов несколькими очередями QoS, так как контроллер выбирает пакеты из очереди Tx-ring в том порядке, в котором они туда были записаны. Поэтому приоритетный пакет, который поступил в приоритетную очередь в тот момент, когда в очереди Tx-ring уже находилось, например, два неприоритетных пакета, будет вынужден ожидать, пока эти два пакета не будут переданы в сеть. Для сокращения времени ожидания приоритетных пакетов максимальную длину очереди Tx-ring рекомендуется сократить до двух пакетов.

В завершение рассмотрим пример конфигурирования QoS для Ethernet-интерфейса с помощью MQC-команд. Пусть мы хотим обслуживать трафик IP-телефонии в приоритетной очереди LLQ; кроме того, мы хотим, чтобы трафик некоторого корпоративного приложения ALPHA, направляемый серверу с IP-адресом 194.104.25.17 на TCP-порт 21403, получал как минимум 30 % пропускной способности порта в периоды перегрузок. Кроме того, мы бы хотели ограничить

приоритетный трафик так, чтобы он никогда не получал более 10 % пропускной способности Ethernet-интерфейса — на основе анализа возможной максимальной нагрузки, создаваемой абонентами IP-телефонии в нормальном режиме работы, мы знаем, что этой величины всегда будет достаточно для данного типа трафика, и хотим защитить трафик других приложений от случайных ошибок или преднамеренной атаки, когда кто-то начнет интенсивно генерировать трафик, имеющий все параметры трафика IP-телефонии, но им не являющийся.

С помощью MQC-команд эта задача решается в три этапа.

1. *Классификация трафика.* Необходимо описать содержащиеся в пакетах признаки, на основе которых маршрутизатор будет распознавать трафик и направлять его в ту или иную выходную очередь. Для нашего примера классификация может быть выполнена следующим образом:

```
class-map ip-tel
    match ip rtp 16384 10
class-map alpha
    match access-group 102

ip access-list 102
    permit tcp any host 194.104.25.17 eq 21403
```

Первые две команды определяют класс трафика IP-телефонии, названный `ip-tel`. Команда `match` говорит о том, что в этот класс нужно включать все пакеты, которые переносят протокол RTP, функционирующий в приложениях IP-телефонии поверх протокола UDP. В данном случае нас интересуют пакеты, в которых протокол RTP занимает для своей работы диапазон, состоящий из 10 UDP-портов, причем первый порт в диапазоне имеет значение 16384.

Следующие четыре команды позволяют задать признаки классификации для трафика нашего корпоративного приложения. Класс `alpha`, заданный этими командами, включает все пакеты, которые имеют IP-адрес назначения 194.104.25.17, а также переносят данные протокола TCP с портом назначения 21403.

2. *Задание параметров обслуживания классов трафика.* После того как классы трафика определены, необходимо описать способ их обслуживания в очереди и параметры этого обслуживания. В терминах интерфейса MQC этот этап выглядит так:

```
policy-map myqos
    class ip-tel
        priority percent 10
    class alpha
        bandwidth percent 30
```

Строка `priority percent 10` говорит маршрутизатору, что пакеты, которые классифицируются как принадлежащие классу `ip-tel`, нужно направлять в приоритетную очередь, контролируя при этом скорость их поступления в очередь и отбрасывая пакеты, если эта скорость превысит 10 % от пропускной способности интерфейса.

Строка `bandwidth percent 30` говорит маршрутизатору, что пакеты, которые классифицируются как принадлежащие классу `alpha`, нужно направлять во взвешенную очередь (об этом говорит ключевой параметр `bandwidth`) с весом 30 %.

3. *Активизация QoS на выбранном интерфейсе.* После того как правила классификации классов и их обработки заданы, их можно применить к тому или иному интерфейсу. В нашем примере для этого необходимо в режиме конфигурирования Ethernet-интерфейса задать команду:

```
service-policy output myqos
```

Данная команда говорит о том, что правила обслуживания, заданные командой `policy-map` с именем `myqos`, должны быть применены к этому интерфейсу.

Выводы

- Коммутация пакетов является основным механизмом передачи данных в компьютерных сетях. Этот механизм позволяет более эффективно передавать пульсирующий компьютерный трафик, чем механизм коммутации каналов, используемый в телефонных сетях.
- Пакет является единицей коммутации в компьютерных сетях. Пакет переносит порцию данных некоторого сетевого приложения, он наделен адресом назначения, что позволяет передавать его по сети от одного коммутационного устройства к другому независимо от других пакетов.
- Буферы пакетов используются во всех коммуникационных устройствах компьютерных сетей. Буферизация трафика позволяет сглаживать его пульсации и не терять пакеты во время перегрузок. В то же время буферизация приводит к очевидному отрицательному эффекту — случайным задержкам пакетов в очередях, создающихся в буферах в периоды перегрузок. Задержки снижают качество работы сетевых приложений реального времени, например пакетной телефонии.
- При организации средств взаимодействия между узлами сети используется многоуровневая декомпозиция. Интерфейс между средствами взаимодействия двух узлов сети, расположенными на одном и том же уровне иерархии, называется протоколом.
- Иерархически организованный набор протоколов, достаточный для организации взаимодействия узлов в сети, называется стеком протоколов.
- Модель взаимодействия открытых систем (Open System Interconnection, OSI) в обобщенном виде описывает функции семи уровней средств сетевого взаимодействия. Эта модель является международным стандартом; она разрабатывалась в качестве своего рода универсального языка сетевых специалистов, поэтому ее называют справочной. Модель OSI не описывает какие-либо конкретные протоколы, но при изучении любого протокола полезно знать, функции какого уровня этой модели он реализует.

- Транспортные функции сети реализуют три нижних уровня модели OSI: физический, канальный и сетевой. Сетевой уровень оперирует пакетами, которые инкапсулируются в кадры протокола канального уровня. Сетевой уровень нужен для передачи пакетов между сетями, входящими в состав составной сети. Канальный уровень обеспечивает передачу кадров в пределах одной сети.
- Технология Ethernet реализует функции двух нижних уровней модели OSI: физического и канального. Ethernet сегодня доминирует в локальных сетях и становится все более популярной в глобальных сетях. Ethernet поддерживает иерархию битовых скоростей 10, 100, 1000 и 10 000 Мбит/с, используя для адресации узлов MAC-адреса, которые состоят из 6 байт и имеют уникальные значения в глобальном масштабе, что обеспечивается соглашением между производителями сетевого оборудования.
- Ethernet-коммутаторы используют алгоритм автоматического построения таблиц продвижения кадров, не требующий предварительного ручного конфигурирования. Ethernet-коммутаторы являются прозрачными, так как не «видны» конечным узлам сети, которую коммутаторы образуют.
- Стек TCP/IP является стеком протоколов, на которых построен Интернет. Популярность Интернета наряду с функциональной гибкостью протоколов стека TCP/IP привели к его доминированию в сетях организаций и в домашних сетях.
- В стеке TCP/IP используется три типа адресов: локальные (например, MAC-адреса Ethernet-интерфейсов), IP-адреса и символьные доменные имена.
- IP-адрес имеет длину 4 байта и состоит из номеров сети и узла. Для разделения IP-адреса на эти два компонента используется связанная с этим адресом маска. Двоичная запись маски содержит единицы в тех разрядах, которые в данном адресе должны интерпретироваться как номер сети.
- Установление соответствия между IP- и MAC-адресами осуществляется протоколом ARP. Для этого ARP использует широковещательные запросы в пределах подсети.
- Назначение IP-адресов узлам сети может выполняться либо вручную, либо автоматически с помощью протокола DHCP. В последнем случае администратор заранее назначает DHCP-серверу диапазон свободных IP-адресов, из которого сервер выбирает очередной адрес при поступлении запроса от узла.
- Соответствие между доменными именами и IP-адресами может устанавливаться как средствами конечного узла с использованием файла hosts, так и с помощью службы DNS, основанной на распределенной базе отображений доменных имен на IP-адреса.
- Маршрутизация IP-пакетов осуществляется на основе таблиц маршрутизации, задающих для каждого IP-адреса назначения IP-адрес следующего хопа.

- Таблицы маршрутизации могут быть построены как вручную, так и автоматически с помощью протоколов маршрутизации. В стеке TCP/IP существует несколько протоколов маршрутизации, таких как RIP, OSPF и BGP.
- Коммуникационные протоколы реализуются в универсальных ОС как многоуровневые драйверы.
- Маршрутизатор представляет собой программу, работающую либо на стандартной компьютерной аппаратной платформе под управлением универсальной ОС, либо на специализированной аппаратной платформе под управлением специализированной ОС.
- Cisco IOS является специализированной ОС, работающей на широком спектре аппаратных платформ маршрутизаторов компании Cisco.
- Особенностями Cisco IOS являются:
  - модульная структура на этапе разработки и сборки IOS и монолитная структура на этапе эксплуатации маршрутизатора;
  - работа всех модулей в одном режиме процессора;
  - отсутствие поддержки виртуальной памяти;
  - режим невытесняющей многозадачности с приоритетами.
- Специальный модуль Cisco IOS организует эффективное управление буферами пакетов, минимизируя перемещение пакетов в памяти ввода-вывода маршрутизатора.
- Cisco IOS поддерживает несколько режимов ускоренной маршрутизации, основанных на использовании объединения информации из таблицы маршрутизации и ARP-таблицы.

Задачи и упражнения

1. Какие элементы сетей с коммутацией пакетов позволяют сглаживать пульсации компьютерного трафика?
2. Используется ли буферизация в сетях с коммутацией каналов?
3. Опишите основные свойства коммуникационного протокола.
4. Что стандартизует модель OSI?
5. На каком уровне модели OSI работает прикладная программа?
6. Должны ли маршрутизаторы поддерживать протоколы транспортного уровня?
7. В чем отличие терминов «пакет» и «кадр»?
8. За счет чего обеспечивается уникальность MAC-адресов?
9. В чем состоит «прозрачность» алгоритма, который поддерживают Ethernet-коммутаторы?
10. В чем заключается опасность «широковещательного шторма»?
11. Опишите назначение сетевого уровня модели OSI.

12. Что определяет маска IP-адреса?
13. Почему таблица маршрутизации использует в качестве адреса следующего маршрутизатора его IP-адрес, а не непосредственно его MAC-адрес?
14. Какой механизм использует протокол ARP для нахождения соответствия между IP- и MAC-адресами узла сети?
15. Какие параметры стека TCP/IP необходимо сконфигурировать в ОС компьютера для того, чтобы он смог начать работать в сети IP?
16. Какие главные цели преследовали разработчики Cisco IOS?
17. Почему IOS не использует привилегированный режим работы процессора?
18. В чем специфика очередей Tx-ring и Rx-ring маршрутизаторов Cisco?
19. За счет чего сокращается время обработки пакетов в режимах ускоренной маршрутизации Cisco IOS?

Глава 10

Концепции распределенной обработки в сетевых ОС

Объединение компьютеров в сеть предоставляет возможность программам, работающим на отдельных компьютерах, оперативно взаимодействовать и сообща решать задачи пользователей. Связь между некоторыми программами может быть настолько тесной, что их удобно рассматривать в качестве частей одного приложения, которое называют в этом случае **распределенным**, или **сетевым**.

Распределенные приложения обладают рядом потенциальных преимуществ по сравнению с локальными. Среди этих преимуществ можно отметить более высокие показатели производительности, отказоустойчивости и масштабируемости.

Модели сетевых служб и распределенных приложений

Значительная часть приложений, работающих в компьютерах сети, является сетевыми, но, конечно, не все. Действительно, ничто не мешает пользователю запустить на своем компьютере полностью локальное приложение, которое никак не задействует имеющиеся сетевые коммуникационные возможности. Достаточно типичным является сетевое приложение, состоящее из двух частей. Например, одна часть приложения работает на компьютере, хранящем базу данных большого объема, а вторая — на компьютере пользователя, который хочет видеть на экране некоторые статистические характеристики данных, хранящихся в базе. Первая часть приложения выполняет поиск в базе записей, отвечающих определенным критериям, а вторая занимается статистической обработкой этих данных, представлением их в графической форме на экране, а также поддерживает диалог с пользователем, принимая от него новые запросы на вычисление тех или иных статистических характеристик. Можно представить себе

случаи, когда приложение распределено и между большим числом компьютеров.

Распределенным в сетях может быть не только прикладное, но и системное программное обеспечение — компоненты операционных систем. Как и в случае локальных служб, программы, которые выполняют некоторые общие и часто встречающиеся в распределенных системах функции, обычно становятся частями операционных систем и называются **сетевыми службами**.

Целесообразно выделить три основных параметра организации работы приложений в сети. К ним относятся:

- способ разделения приложения на части, выполняющиеся на разных компьютерах сети;
- выделение специализированных серверов в сети, на которых выполняются некоторые общие для всех приложений функции;
- способ взаимодействия между частями приложений, работающих на разных компьютерах.

Разделение приложений на части

Очевидно, что можно предложить различные схемы разделения приложений на части, причем для каждого конкретного приложения можно предложить свою схему. Существуют и типовые модели распределенных приложений. В следующей, достаточно детальной модели предлагается разделить приложение на шесть функциональных частей:

- *средства представления* данных на экране, например средства графического пользовательского интерфейса;
- *логика представления* данных на экране описывает правила и возможные сценарии взаимодействия пользователя с приложением: выбор в системе меню, выбор элемента в списке и т. п.;
- *прикладная логика* — набор правил для принятия решений, вычислительные процедуры и операции;
- *логика данных* — операции с данными, хранящимися в некоторой базе, которые нужно выполнить для реализации прикладной логики;
- *внутренние операции базы данных* — действия СУБД, вызываемые в ответ на выполнение запросов логики данных, такие как поиск записи по определенным признакам;
- *файловые операции* — стандартные операции с файлами и файловой системой, которые обычно являются функциями операционной системы.

На основе этой модели можно построить несколько схем распределения частей приложения между компьютерами сети.

Двухзвенные схемы

Распределение приложения между большим числом компьютеров может повысить качество его выполнения (скорость, количество одновременно обслужи-

ваемых пользователей и т. д.), но при этом существенно усложняется организация самого приложения, что может просто не позволить воспользоваться потенциальными преимуществами распределенной обработки. Поэтому на практике приложение обычно разделяют на две или три части и достаточно редко — на большее число частей. Наиболее распространенной является *двухзвенная схема*, распределяющая приложение между двумя компьютерами. Перечисленные типовые функциональные части приложения можно разделить между двумя компьютерами различными способами.

Рассмотрим сначала два крайних случая двухзвенной схемы, когда нагрузка в основном ложится на один узел — либо на центральный компьютер, либо на клиентскую машину.

В *централизованной схеме* (рис. 10.1, *а*) компьютер пользователя работает как терминал, выполняющий лишь функции представления данных, тогда как все остальные функции передаются центральному компьютеру. Ресурсы компьютера пользователя задействуются в этой схеме в незначительной степени, загруженными оказываются только графические средства подсистемы ввода-вывода ОС, отображающие на экране окна и другие графические примитивы по командам центрального компьютера, а также сетевые средства ОС, принимающие из сети команды центрального компьютера и возвращающие данные о нажатии клавиш и координатах указателя мыши. Программа, работающая на компьютере пользователя, часто называется **эмулятором терминала** — графическим или текстовым, в зависимости от поддерживаемого режима. Фактически, эта схема повторяет организацию многотерминальной системы на базе мэйнфрейма, с тем лишь отличием, что вместо терминалов используются компьютеры, подключенные не через локальный интерфейс, а через сеть, локальную или глобальную.

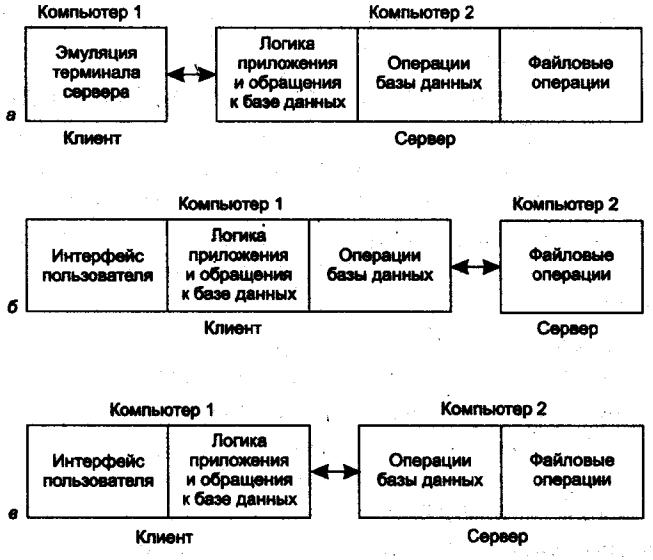


Рис. 10.1. Варианты распределения частей приложения по двухзвенной схеме

Главным и очень серьезным недостатком централизованной схемы является ее *недостаточная масштабируемость* и *плохая отказоустойчивость*. Производительность центрального компьютера всегда будет ограничителем количества пользователей, работающих с данным приложением, а отказ центрального компьютера приведет к прекращению работы всех пользователей. Именно из-за этих недостатков централизованные вычислительные системы, представленные мэйнфреймами, уступили место сетям, состоящим из мини-компьютеров, RISC-серверов и персональных компьютеров. Тем не менее централизованная схема иногда применяется благодаря простой структуре программы, которая почти целиком работает на одном компьютере, и наличию большого парка локальных приложений.

В схеме *файлового сервера* (рис. 10.1, *б*) на клиентской машине выполняются все части приложения, кроме файловых операций. В сети имеется достаточно мощный компьютер, имеющий дисковую подсистему большого объема. Этот компьютер хранит файлы, доступ к которым необходим большому числу пользователей. Он играет роль файлового сервера, представляя собой централизованное хранилище данных, находящихся в разделяемом доступе. Распределенное приложение в этой схеме мало отличается от полностью локального приложения. Единственным отличием является обращение не к локальным, а к удаленным файлам. Для того чтобы в этой схеме можно было использовать локальные приложения, в сетевые операционные системы ввели такой компонент сетевой файловой службы, как **редиректор**, который перехватывает обращения к удаленным файлам (с помощью специальной нотации для сетевых имен, например //server1/doc/file1.txt) и направляет запросы в сеть, освобождая приложение от необходимости явно задействовать сетевые системные вызовы.

Файловый сервер представляет собой компонент наиболее популярной сетевой службы — сетевой файловой системы, которая лежит в основе многих распределенных приложений и некоторых других сетевых служб. Первые сетевые ОС (NetWare компании Novell, IBM PC LAN Program, Microsoft MS-Net) обычно поддерживали две сетевые службы — файловую службу и службу печати, оставляя реализацию остальных функций разработчикам распределенных приложений.

Такая схема обладает *хорошей масштабируемостью*, так как дополнительные пользователи и приложения лишь незначительно увеличивают нагрузку на центральный узел — файловый сервер. Однако эта архитектура имеет и свои недостатки:

- во многих случаях резко возрастает сетевая нагрузка (например, многочисленные запросы к базе данных могут приводить к загрузке всей базы данных в клиентскую машину для последующего локального поиска нужных записей), что приводит к *увеличению времени реакции* приложения;
- компьютер клиента должен обладать *высокой вычислительной мощностью*, чтобы справляться с представлением данных, логикой приложения, логикой данных и поддержкой операций базы данных.

Другие варианты двухзвенной модели более равномерно распределяют функции между клиентской и серверной частями системы. Наиболее часто используется схема, в которой на серверный компьютер возлагаются функции проведения внутренних операций базы данных и файловых операций (рис. 10.1, *в*). Клиентский компьютер при этом выполняет все функции, специфические для данного приложения, а сервер — функции, реализация которых не зависит от специфики приложения, из-за чего эти функции могут быть оформлены в виде сетевых служб. Поскольку функции управления базами данных нужны далеко не всем приложениям, то в отличие от файловой системы они чаще всего не реализуются в виде службы сетевой ОС, а являются независимой распределенной прикладной системой. Система управления базами данных (СУБД) является одним из наиболее часто применяемых в сетях распределенных приложений. Не все СУБД являются распределенными, но практически все мощные СУБД, позволяющие поддерживать большое число сетевых пользователей, построены в соответствии с описанной моделью клиент-сервер. Сам термин **клиент-сервер** справедлив для любой двухзвенной схемы распределения функций, но исторически он оказался наиболее тесно связан со схемой, в которой сервер выполняет функции управления базами данных (и, конечно, файлами, в которых хранятся эти базы) и часто используется как синоним этой схемы.

Трехзвенные схемы

Трехзвенная архитектура позволяет еще лучше сбалансировать нагрузку на различные компьютеры в сети, а также способствует дальнейшей специализации серверов и средств разработки распределенных приложений. Примером трехзвенной архитектуры может служить такая организация приложения, в которой на клиентской машине выполняются средства представления и логика представления, а также поддерживается программный интерфейс для вызова частей приложения второго звена — промежуточного сервера (рис. 10.2).

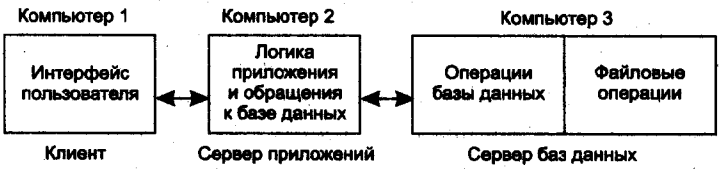


Рис. 10.2. Трехзвенная схема распределения частей приложения

Промежуточный сервер называют в этом варианте **сервером приложений**, так как на нем реализуются прикладная логика и логика обработки данных, представляющих собой наиболее специфические и важные части большинства приложений. Часть приложения, реализующая логику обработки данных, вызывает внутренние операции базы данных, которые выполняет третье звено схемы — **сервер баз данных**.

Сервер баз данных, как и в двухзвенной модели, выполняет функции двух последних функциональных частей приложения — операции внутри базы данных и файловые операции. Примером такой схемы может служить неоднородная архитектура, включающая клиентские компьютеры под управлением Windows Vista Professional, сервер приложений с монитором транзакций TUXEDO в среде Solaris и сервер баз данных Oracle в среде Windows Server 2003 Datacenter.

Централизованная реализация логики приложения решает проблему недостаточной вычислительной мощности клиентских компьютеров для сложных приложений, а также упрощает администрирование и сопровождение. В том случае, когда сервер приложений сам становится узким местом, в сеть можно включить несколько серверов приложений, распределив каким-то образом запросы пользователей между ними. Упрощается и разработка крупных приложений, так как в этом случае четко разделяются платформы и инструменты для реализации интерфейса и прикладной логики, что позволяет с наибольшей эффективностью реализовывать их силами специалистов узкого профиля.

Монитор транзакций представляет собой популярный пример программного обеспечения, не входящего в состав сетевой ОС, но выполняющего функции, полезные для большого количества приложений. Такой монитор управляет транзакциями, проводимыми с базой данных, и поддерживает целостность распределенной базы данных.

Трехзвенные схемы часто применяются для централизованной реализации в сети некоторых общих для распределенных приложений функций, отличных от файлового сервиса и управления базами данных. Программные модули, выполняющие такие функции, относят к *промежуточному слою* (middleware), располагающемуся между индивидуальной для каждого приложения логикой и сервером баз данных.

В крупных сетях для связи клиентских и серверных частей приложений используется также ряд других средств, относящихся к промежуточному слою, в том числе:

- средства асинхронной обработки сообщений (Message-Oriented Middleware, MOM);
- средства удаленного вызова процедур (Remote Procedure Call, RPC);
- брокеры запроса объектов (Object Request Broker, ORB), которые находят объекты, хранящиеся на различных компьютерах, и помогают их использовать в одном приложении или документе.

Эти средства помогают улучшить качество взаимодействия клиентов с серверами за счет промышленной реализации достаточно важных и сложных функций, а также упорядочить поток запросов от множества клиентов к множеству серверов, играя роль регулировщика, распределяющего нагрузку на серверы.

Сервер приложений должен базироваться на мощной аппаратной платформе (мультипроцессорные системы, специализированные кластерные архитектуры). ОС сервера приложений должна обеспечивать высокую производительность

вычислений, а значит, поддерживать многопоточную обработку, вытесняющую многозадачность, мультипроцессирование, виртуальную память и наиболее популярные прикладные среды.

Механизм передачи сообщений в распределенных системах

Единственным по-настоящему важным отличием распределенных систем от централизованных является способ взаимодействия между процессами. Принципиально взаимодействие между процессами может осуществляться одним из двух способов:

- путем совместного использования одних и тех же данных (разделяемая память);
- путем передачи друг другу данных в виде сообщений.

В *централизованных системах* связь между процессами, как правило, предполагает наличие разделяемой памяти. Типичный пример — задача «поставщик-потребитель». В этом случае один процесс пишет в разделяемый буфер, а другой читает из него. Даже наиболее простая форма синхронизации — семафор — требует, чтобы хотя бы одно слово (переменная самого семафора) было разделяемым. Аналогичным образом происходит взаимодействие не только между пользовательскими процессами, но и между приложением и операционной системой — процесс в пользовательском режиме запрашивает у ОС выполнения некоторой операции с помощью системного вызова, помещая в доступную ему часть оперативной памяти параметры этого системного вызова (например, имя файла, смещение от его начала и количество байтов, которые необходимо прочитать). После этого модуль ядра ОС считывает эти параметры из пользовательской памяти (ядру в привилегированном режиме доступна вся память, как ее системная часть, так и пользовательская) и выполняет системный вызов. Взаимодействие и в этом случае происходит за счет непосредственно доступной обоим участникам области памяти.

В *распределенных системах* не существует памяти, непосредственно доступной процессам, работающим на разных компьютерах, поэтому взаимодействие процессов (находящихся как в пользовательской фазе, так и в системной, то есть выполняющих код операционной системы) может осуществляться только путем передачи сообщений через сеть. Как было показано в разделе «Сетевые службы и сетевые сервисы» главы 2, на основе механизма передачи сообщений работают все сетевые службы, предоставляющие пользователям сети разнообразные услуги — доступ к удаленным файлам, принтерам, почтовым ящикам и т. п. В сообщениях переносятся запросы от клиентов некоторой службы к соответствующим серверам, например, запрос на просмотр содержимого определенного каталога файловой системы, расположенной на сетевом сервере. Сервер возвращает ответ — набор имен файлов и подкаталогов, входящих в данный каталог, также помещая его в сообщение и отправляя по сети клиенту.

Сообщение — это блок информации, отформатированный процессом-отправителем таким образом, чтобы он был понятен процессу-получателю. Сообщение состоит из заголовка, обычно фиксированной длины, и набора данных определенного типа переменной длины. В заголовке, как правило, содержатся следующие элементы.

- *Адрес* — набор символов, уникально определяющих отправляющий и получающий процессы. Адресное поле таким образом состоит из двух частей — адреса процесса-отправителя и адреса процесса-получателя. Адрес каждого процесса может, в свою очередь, иметь некоторую структуру, позволяющую найти нужный процесс в сети, состоящей из большого количества компьютеров.
- *Порядковый номер* сообщения, являющийся его идентификатором. Используется для идентификации потерянных сообщений и дубликатов сообщений в случае отказов в сети.
- Структурированная информация, состоящая в общем случае из нескольких частей: полей *типа данных, длины данных* и *значения данных* (то есть собственно данных). Поле типа данных определяет, например, что данные являются целым числом или же представляют собой строку символов (это поле может также содержать признак нерезидентности данных, то есть указатель на данные, которые хранятся где-то вне данного сообщения). Второе поле определяет длину передаваемых в сообщении данных (обычно в байтах), то есть размер следующего поля сообщения. Сообщение может включать несколько элементов, состоящих из описанных трех полей. В тех случаях, когда сообщение всегда переносит данные одного и того же типа, поле типа может быть опущено. То же касается поля длины данных — для тех типов сообщений, которые переносят данные фиксированного формата, но такая ситуация характерна только для протоколов низкого уровня (например, протокола ATM, имеющего фиксированный размер поля данных в 48 байт).

В любой сетевой ОС поддерживается **система передачи сообщений**, которая использует стек коммуникационных протоколов, описанных в главе 9. Назначение этой системы — экранировать детали сложных сетевых протоколов от программиста. Система передачи сообщений позволяет процессам взаимодействовать посредством достаточно простых **примитивов**. В самом простом случае системные средства обеспечения связи могут быть сведены к двум основным коммуникационным примитивам, один `send` (отправить) — для посылки сообщения, другой `receive` (получить) — для получения сообщения. В дальнейшем на их базе могут быть построены более мощные средства сетевых коммуникаций, такие как распределенная файловая система или служба вызова удаленных процедур, которые, в свою очередь, также могут служить основой для работы других сетевых служб.

Взаимодействие системы передачи сообщений с коммуникационными протоколами стека TCP/IP иллюстрирует рис. 10.3.

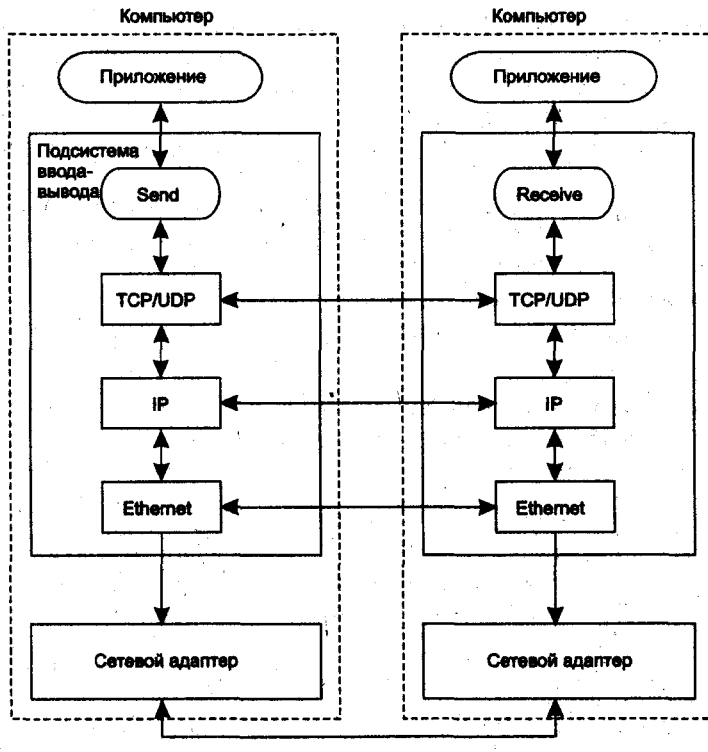


Рис. 10.3. Примитивы обмена сообщениями и транспортные средства подсистемы ввода-вывода

Несмотря на концептуальную простоту примитивов `send` и `receive`, существуют различные варианты их реализации, от правильного выбора которых зависит эффективность работы сети. В частности, эффективность зависит от способа задания адреса получателя. Не менее важны при реализации примитивов передачи сообщений ответы на другие вопросы: В сети всегда имеется один получатель или их может быть несколько? Должна ли доставка сообщений гарантироваться? Должен ли отправитель дождаться ответа на свое сообщение, прежде чем продолжать работу? Как отправитель, получатель и подсистема передачи сообщений должны реагировать на отказы узла или коммуникационного канала во время взаимодействия? Что делать, если приемник не готов принять сообщение, отбросить его или сохранить в буфере? А если сохранить, то как быть, если буфер уже заполнен? Разрешено ли приемнику изменять порядок обработки сообщений в соответствии с их важностью? Ответы на подобные вопросы составляют семантику конкретного протокола передачи сообщений.

Синхронизация

Центральным вопросом взаимодействия процессов в сети является способ их синхронизации, который полностью определяется используемыми в операци-

онной системе коммуникационными примитивами. В этом отношении коммуникационные примитивы делятся на **блокирующие (синхронные)** и **неблокирующие (асинхронные)**, причем смысл данных терминов в целом соответствует смыслу аналогичных терминов, применяемых при описании системных вызовов (см. раздел «Системные вызовы» в главе 4) и операций ввода-вывода (см. раздел «Синхронный и асинхронный режимы» в главе 7). В отличие от локальных системных вызовов (а именно такие системные вызовы были рассмотрены в главах 4 и 7), при выполнении коммуникационных примитивов завершение запрошенной операции в общем случае зависит от работы не только локальной, но и удаленной ОС.

Коммуникационные примитивы могут быть оформлены в операционной системе двояко: как *внутренние процедуры ядра ОС* (в этом случае они могут работать только с модулями ОС) или как *системные вызовы* (доступные в этом случае процессам в пользовательском режиме).

При использовании блокирующего примитива `send` процесс, выдавший запрос на его выполнение, приостанавливается до момента получения по сети сообщения-подтверждения о том, что приемник получил отправленное сообщение. А вызов блокирующего примитива `receive` приостанавливает вызывающий процесс до момента, когда он получит сообщение. При использовании неблокирующих примитивов `send` и `receive` управление возвращается вызывающему процессу немедленно, сразу после того, как ядру передается информация о том, где в памяти находится буфер, в который нужно поместить сообщение, отправляемое в сеть или ожидаемое из сети. Преимуществом этой схемы является параллельное выполнение вызывающего процесса и процедур передачи сообщения (не обязательно работающих в контексте вызвавшего соответствующий примитив процесса).

Важным вопросом при использовании неблокирующего примитива `receive` является выбор способа уведомления процесса-получателя о том, что сообщение пришло и помещено в буфер. Обычно для этой цели применяется один из двух подходов.

- **Опрос** (polling). Этот подход предусматривает наличие еще одного базового примитива `test` (проверить), с помощью которого процесс-получатель может анализировать состояние буфера.
- **Прерывание** (interrupt). В этом случае программное прерывание уведомляет процесс-получателя о том, что сообщение помещено в буфер. Хотя такой подход очень эффективен (он исключает многократные проверки состояния буфера), у него имеется существенный недостаток — усложненное программирование, связанное с прерываниями пользовательского уровня, то есть прерываниями, по которым вызываются процедуры пользовательского режима (например, вызов APC-процедур в ОС Windows NT при завершении операции ввода-вывода, рассмотренный в главе 8).

При использовании блокирующего примитива `send` может возникнуть ситуация, когда процесс-отправитель оказывается заблокированным навсегда, например, если процесс получатель потерпел крах или же отправленное сообще-

ние было утеряно из-за сетевой ошибки. Чтобы предотвратить такую ситуацию, блокирующий примитив `send` часто использует *механизм тайм-аута*. То есть определяется интервал времени, после которого операция `send` завершается со статусом «ошибка».

Механизм тайм-аута может использоваться также блокирующим примитивом `receive` для предотвращения блокировки процесса-получателя на неопределенное время, когда процесс-отправитель потерпел крах или сообщение было потеряно вследствие сетевой ошибки.

Если при взаимодействии двух процессов оба примитива `send` и `receive` являются блокирующими, говорят, что процессы взаимодействуют по сети *синхронно* (рис. 10.4), в противном случае взаимодействие считается *асинхронным* (рис. 10.5).

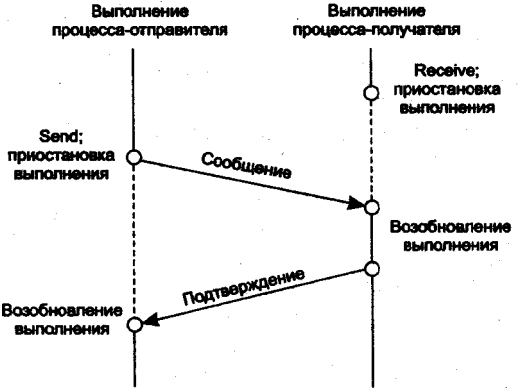


Рис. 10.4. Синхронное взаимодействие с помощью блокирующих примитивов send и receive

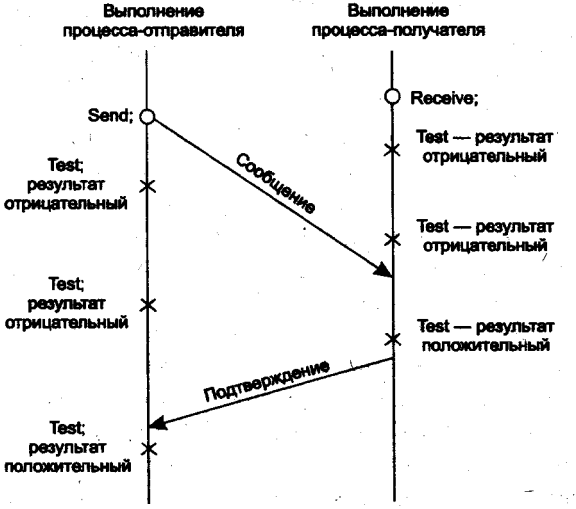


Рис. 10.5. Асинхронное взаимодействие с помощью неблокирующих примитивов send и receive

По сравнению с асинхронным взаимодействием синхронное проще и его легче реализовать. Оно также надежнее, так как гарантирует процессу-отправителю, возобновившему свое выполнение, что его сообщение было получено. Главный же недостаток — ограниченный параллелизм и возможность возникновения клинчей.

Обычно в ОС поддерживается один из двух видов примитивов, но ОС является более гибкой, если поддерживает как блокирующие, так и неблокирующие примитивы.

Буферизация в примитивах передачи сообщений

При передаче сообщений могут возникнуть ситуации, когда принимающий процесс оказывается неготовым обработать сообщение при его прибытии, но для процесса было бы желательным, чтобы операционная система на время сохранила поступившее сообщение в буфере для последующей обработки.

Способ буферизации тесно связан со способом синхронизации процессов при обмене сообщениями. При использовании синхронных, то есть блокирующих, примитивов, можно вообще обойтись без буферизации сообщений операционной системой. При этом возможны два варианта организации работы примитивов. В первом случае процесс-отправитель подготавливает сообщение в своей памяти и обращается к примитиву `send`, после чего процесс блокируется. Операционная система отправителя ждет, когда процесс-получатель выполнит на своем компьютере примитив `receive`, в результате чего ОС получателя направит служебное сообщение-подтверждение о готовности к приему основного сообщения. После получения такого подтверждения ОС на компьютере-отправителе разблокирует процесс-отправитель и тот немедленно после этого пошлет сообщение по сети. Процесс-получатель после обращения к примитиву `receive` также переводится своей ОС в состояние ожидания, из которого он выходит при поступлении сообщения по сети. Сообщение немедленно копируется ОС в память процесса-получателя, не требуя буферизации, так как процесс ожидает его прихода и готов к его обработке.

Буферизация не требуется и при другом варианте обмена сообщениями, когда процесс-отправитель посылает сообщение в сеть, не дожидаясь прихода от получателя подтверждения о готовности к приему. Затем процесс-отправитель блокируется либо до прихода такого подтверждения (в этом случае никакой дополнительной работы с данным сообщением не выполняется), либо до истечения тайм-аута, после которого сообщение посылается вновь, причем в случае многократных повторных неудачных попыток сообщение отбрасывается.

В обоих случаях сообщение непосредственно из памяти процесса-отправителя попадает в сеть, а после прихода из сети — в память процесса-получателя, минуя буфер, поддерживаемый системой. Однако такая организация в сетевых операционных системах на практике не применяется, так как в первом варианте процесс-получатель может достаточно долго ждать, пока сообщение будет передано по сети (в большой составной сети, например в Интернете, задержки могут достигать нескольких секунд), а во втором — из-за неготовности процесса-

получателя сообщение может многократно бесполезно передаваться по сети, засоряя каналы связи.

Именно поэтому даже при использовании синхронных примитивов предусматривают буферизацию. В этом случае буфер, как правило, выбирают размером в одно сообщение, так как процесс-отправитель не может послать следующее сообщение, не получив подтверждения о приеме предыдущего. Сообщение помещается в буфер, поддерживаемый операционной системой компьютера-получателя, если в момент его прихода процесс-получатель не может обработать сообщение немедленно, например, из-за того, что процесс либо не является текущим, либо не готов к приему сообщения, так как не обратился к примитиву `receive`. Буфер может располагаться как в системной области памяти, так и в области памяти пользовательского процесса, в любом случае буфером управляет операционная система, модули которой получают сообщения по сети.

Для всех вариантов обмена сообщениями с помощью асинхронных примитивов необходима буферизация. Поскольку при асинхронном обмене процесс-отправитель может посылать сообщение всегда, когда ему это требуется, не дожидаясь подтверждения от процесса-получателя, для исключения потерь сообщений требуется буфер неограниченной длины. Так как буфер в реальной системе всегда имеет ограниченный размер, то могут возникать ситуации с переполнением буфера, и на них нужно каким-то образом реагировать. Для снижения вероятности потерь сообщений степень асинхронности процесса обмена сообщениями обычно ограничивается механизмом управления потоком сообщений. Управление потоком заключается в том, что при заполнении буфера на принимающей стороне до некоторого опасного порога процесс-передатчик блокируется до тех пор, пока процесс-приемник не обработает часть принятых сообщений и не разгрузит буфер до безопасной величины. Конечно, вероятность потерь сообщений из-за переполнения буфера все равно сохраняется, например, потому, что служебное сообщение о необходимости приостановки передачи сообщений может быть потеряно сетью. Асинхронный обмен с управлением потоком — это наиболее сложный способ организации обмена сообщениями, так как для повышения эффективности, то есть максимизации скорости обмена и минимизации потерь, он требует применения сложных алгоритмов приостановки и возобновления процесса передачи, например, таких, которые применяются в протоколе TCP.

Обычно операционная система предоставляет прикладным процессам специальный примитив для создания буферов сообщений. Такого рода примитив, назовем его, например, `create_buffer` (создать буфер), процесс должен использовать перед тем, как отправлять или получать сообщения с помощью примитивов `send` и `receive`. При создании буфера его размер может либо устанавливаться по умолчанию, либо выбираться прикладным процессом. Часто такой буфер носит название **порта** (port), или **почтового ящика** (mailbox).

При реализации схем буферизации сообщений необходимо также решить вопрос о том, что должна делать операционная система с поступившими сообщениями, для которых буфер не создан. Такая ситуация может возникнуть

в том случае, если примитив `send` на одном компьютере выполнится раньше, чем примитив `create_buffer` на другом. Каким образом ядро на компьютере получателя сможет узнать, какому процессу адресовано вновь поступившее сообщение, если имеется несколько активных процессов? И как оно узнает, куда его скопировать?

Один из вариантов — просто отказаться от сообщения в расчете на то, что отправитель после тайм-аута передаст сообщение повторно, а получатель к этому времени уже создаст буфер. Этот подход не сложен в реализации, но, к сожалению, отправитель (или, скорее, ядро его компьютера) может сделать несколько таких безуспешных попыток. Еще хуже то, что после достаточно большого числа безуспешных попыток ядро отправителя может сделать неправильный вывод об аварии на машине получателя или о неправильности его адреса.

Второй подход к этой проблеме заключается в том, чтобы хранить хотя бы некоторое время поступающие сообщения в ядре получателя в расчете на то, что вскоре будет выполнен соответствующий примитив `create_buffer`. Каждый раз, когда поступает такое «неожидаемое» сообщение, включается таймер. Если заданный временной интервал истекает раньше, чем происходит создание буфера, то сообщение теряется.

Хотя этот метод и снижает вероятность потери сообщений, он порождает проблему хранения и управления преждевременно поступившими сообщениями. Необходимы буферы, которые следует где-то размещать, освобождать, то есть которыми нужно как-то управлять, что создает дополнительную нагрузку на операционную систему.

Способы адресации

Для того чтобы послать сообщение, необходимо указать адрес получателя. В очень простой сети адрес может задаваться в виде константы, но в сложных сетях нужен более гибкий способ адресации.

Одним из вариантов адресации является использование *аппаратных адресов сетевых адаптеров*. Если в компьютере-получателе выполняется только один процесс, то ядро ОС будет знать, что делать с поступившим сообщением — передать его этому процессу. Однако если на машине выполняется несколько процессов, то ядру неизвестно, какому из них предназначено сообщение, поэтому использование сетевого адреса адаптера в качестве адреса получателя приводит к очень серьезному ограничению — на каждой машине должен выполняться только один процесс. Кроме того, на основе аппаратного адреса сетевого адаптера сообщения можно передавать только в пределах одной локальной сети, в более сложных сетях, состоящих из нескольких подсетей, в том числе глобальных, для передачи данных между узлами требуются числовые адреса, несущие информацию как о номере узла, так и о номере подсети, например IP-адреса.

Наибольшее распространение получила система адресации, в которой *адрес состоит из двух частей, определяющих компьютер и процесс*, которому предназначено сообщение, то есть адрес имеет вид пары числовых идентификаторов:

machine_id@local_id. В качестве идентификатора компьютера machine_id наиболее употребительным на сегодня является использование IP-адреса, который представляет собой 32-разрядное число, условно записываемое в виде четырех десятичных чисел, разделенных точками, например 185.23.123.26. Идентификатором компьютера может служить любой другой тип адреса узла, который воспринимается транспортными средствами сети, например IPX-адрес, ATM-адрес или уже упоминавшийся аппаратный адрес сетевого адаптера, если система передачи сообщений ОС работает только в пределах одной локальной сети.

Для адресации процесса в этом случае применяется числовой идентификатор local_id, имеющий уникальное в пределах узла machine_id значение. Этот идентификатор может однозначно указывать на конкретный процесс, работающий на данном компьютере, то есть являться идентификатором типа process_id. Однако существует и другой подход, функциональный, когда используется *адрес службы*, которой пересылается сообщение, при этом идентификатор принимает вид service_id. Последний вариант удобнее для отправителя, так как службы, поддерживаемые сетевыми операционными системами, представляют собой достаточно устойчивый набор (в него входят, как правило, наиболее популярные службы FTP, SMB, NFS, SMTP, HTTP, SNMP), и этим службам можно дать вполне определенные адреса, заранее известные всем отправителям. Такие адреса называют *хорошо известными* (well-known). Примерами хорошо известных адресов служб являются номера портов в протоколах TCP и UDP. Отправитель всегда знает, что, посылая с помощью этих протоколов сообщение на порт 21 некоторого компьютера, он посылает его службе FTP, то есть службе передачи файлов. При этом отправителя не интересует, какой именно процесс (с каким локальным идентификатором) реализует в настоящий момент времени услуги FTP на данном компьютере.

Ввиду повсеместного применения стека протоколов TCP/IP, номера портов являются на сегодня наиболее популярными адресами служб в системах обмена сообщениями сетевых ОС. Порт TCP/UDP является не только абстрактным адресом службы, но и представляет собой нечто более конкретное — для каждого порта операционная система поддерживает буфер в системной памяти, куда помещаются отправляемые и получаемые сообщения, адресуемые данному порту. Порт задается в протоколах TCP/UDP двухбайтным адресом, поэтому ОС может поддерживать до 65 535 портов. Помимо *хорошо известных номеров портов*, которым отводится диапазон от 1 до 1023, существуют и динамически используемые порты с большими номерами. Значения этих портов не закрепляются за определенными службами, поэтому они часто дополняют хорошо известные порты для обмена в рамках обслуживания некоторой службы сообщениями специфического назначения. Например, FTP-клиент всегда начинает взаимодействие с FTP-сервером отправкой сообщения на порт 21, а после установления сеанса обмен данными между клиентом и сервером выполняется уже по порту, номер которого динамически выбирается в процессе установления сеанса.

Описанная схема адресации типа «машина-процесс» или «машина-служба» хорошо зарекомендовала себя, работая уже на протяжении многих лет в Интернете, а также в корпоративных IP- и IPX-сетях (в этих сетях также используется адресация службы, а не процесса). Однако эта схема имеет один существенный недостаток — она *не гибка и не прозрачна*, так как пользователь должен явно указывать адрес машины-получателя. В этом случае если в один прекрасный день машина, на которой работает некоторая служба, откажет, то программа, выполняющая все обращения к данной службе по жестко заданному адресу, не сможет использовать аналогичную службу, установленную на другой машине.

Основным способом повышения степени прозрачности адресации является использование *символьных имен* вместо числовых. Примером такого подхода является характерная для сегодняшнего Интернета нотация URL (Universal Resource Locator — универсальный указатель ресурса), в соответствии с которой адрес состоит из символьного имени узла и символьного имени службы. Например, если в сообщении указан адрес ftp://arc.bestcompany.ru, это означает, что оно отправлено службе FTP, работающей на компьютере arc.bestcompany.ru.

Использование символьных имен требует создания в сети службы оперативного отображения символьных имен на числовые идентификаторы, поскольку именно в таком виде адреса распознаются сетевым оборудованием. Применение символьного имени позволяет разорвать жесткую связь адреса с единственным компьютером, так как символьное имя перед отправкой сообщения в сеть заменяется числовым, например, IP-адресом. Этап замены позволяет сопоставлять с символьным именем различные числовые адреса и выбирать тот компьютер, который в данный момент в наибольшей степени подходит для выполнения запроса, содержащегося в сообщении. Например, отправляя запрос на получение услуг службы WWW от компании Microsoft по адресу http://www.microsoft.com, вы точно не знаете, какой из нескольких серверов этой компании, предоставляющих данный вид услуг и обслуживающих один и тот же символьный адрес, вам ответит.

Для замены символьных адресов числовыми применяются две схемы:

- широковещание;
- централизованная служба имен.

Широковещание удобно в локальных сетях, в которых все сетевые технологии нижнего уровня, такие как Ethernet, Token Ring, FDDI, поддерживают широковещательные адреса в пределах всей сети, а пропускной способности каналов связи достаточно при обслуживании таких запросов для сравнительно небольшого количества клиентов и серверов. На широковещании были построены все службы ОС NetWare (до версии 4), ставшие в свое время эталоном прозрачности для пользователей. В этой схеме сервер периодически широковещательно рассылает по сети сообщения о соответствии числовым адресам его имени и имен служб, которые он поддерживает. Клиент также может сделать широковещательный запрос о наличии в сети сервера, поддерживающего определенную службу, и если такой сервер в сети есть, то он ответит на запрос

своим числовым адресом. После обмена подобными сообщениями пользователь должен явно указать в своем запросе имя сервера, к ресурсам которого он обращается, а клиентская ОС заменит это имя числовым адресом в соответствии с информацией, широковещательно распространенной сервером.

Однако широковещательный механизм разрешения адресов плохо работает в территориальных сетях, так как наличие большого числа клиентов и серверов, а также использование менее скоростных по сравнению с локальными сетями каналов делают широковещательный трафик слишком интенсивным, практически не оставляющим пропускной способности для передачи пользовательских данных. В территориальных сетях для разрешения символьных имен компьютеров применяется другой подход, основанный на специализированных серверах, хранящих базу данных соответствия между символьными именами и числовыми адресами. Эти серверы образуют распределенную службу имен, обрабатывающую запросы многочисленных клиентов. Хорошо известным примером такой службы является **система доменных имен Интернета** (Domain Name System, DNS). Эта служба позволяет обрабатывать в реальном масштабе времени многочисленные запросы пользователей Интернета, обращающихся к ресурсам серверов по составным именам, таким как www.microsoft.com или www.gazeta.ru. Другими примерами могут служить справочные службы **NDS** (NetWare Directory Sevices) компании Novell и Activ Directory компании Microsoft, которые выполняют в крупных корпоративных сетях более общие функции, предоставляя справочную информацию по любым сетевым ресурсам, но в том числе по соответствию символьных имен компьютеров их числовым адресам.

Централизованная служба имен на сегодня считается наиболее перспективным средством повышения прозрачности услуг для пользователей сетей. С такой службой связывают перспективы дальнейшего повышения прозрачности адресации сетевых ресурсов, когда имя ресурса будет полностью независимо от компьютера, предоставляющего этот ресурс в общее пользование. Современные справочные службы уже сегодня позволяют пользователям обращаться к любым сетевым ресурсам по условным именам или атрибутам, никак не связанным с местом их расположения. Пользователь может задействовать, например имена томов, не указывая их точного расположения на том или ином компьютере. При перемещении тома с одного компьютера на другой изменение связи тома с компьютером регистрируется в базе данных справочной службы, так что все обращения к тому после его перемещения разрешаются корректно путем замены имени адресом нового компьютера.

По пути применения централизованной службы-посредника между клиентами и ресурсами идут и разработчики распределенных приложений, например разработчики технологии CORBA, в которой запросы к программным модулям приложений обрабатывает специальный элемент — брокер запросов.

Использование символьных имен вместо числовых адресов несколько повышает прозрачность, но не до той степени, которой хотелось бы достичь приверженцам идеи распределенных операционных систем, главным отличием которых

от сетевых ОС является именно полная прозрачность адресации разделяемых ресурсов. Тем не менее символьные имена — это значительный шаг вперед по сравнению с числовыми.

Надежные и ненадежные примитивы

Ранее подразумевалось, что когда отправитель посылает сообщение, адресат его обязательно получает. Но на практике сообщения могут теряться. Предположим, что для обмена сообщениями используются блокирующие примитивы. Когда отправитель посылает сообщение, то он приостанавливает свою работу до тех пор, пока сообщение не будет послано. Однако нет никаких гарантий, что после того, как он возобновит свою работу, сообщение будет доставлено адресату.

Для решения этой проблемы существует три подхода.

- Первый заключается в том, что система не берет на себя никаких обязательств по поводу доставки сообщений. Такой способ доставки сообщений обычно называют **дейтаграммным** (datagram). Реализация надежного взаимодействия при его применении становится целиком заботой прикладного программиста.
- Второй подход заключается в том, что ядро принимающей машины посылает **квитанцию-подтверждение** ядру отправляющей машины на каждое сообщение или на группу последовательных сообщений. Посылающее ядро разблокирует пользовательский процесс только после получения такого подтверждения. Обработкой подтверждений занимается подсистема обмена сообщениями ОС, ни процесс-отправитель, ни процесс-получатель их не видят.
- Третий подход заключается в использовании **ответа в качестве подтверждения** в тех системах, в которых запрос всегда сопровождается ответом, что характерно для клиент-серверных служб. В этом случае служебные сообщения-подтверждения не посылаются, так как в их роли выступают пользовательские сообщения-ответы. Процесс-отправитель остается заблокированным до получения ответа. Если же ответа нет слишком долго, то после истечения тайм-аута ОС отправителя повторно посылает запрос.

Надежная передача сообщений может подразумевать не только гарантию доставки отдельных сообщений, но и *упорядоченность* этих сообщений, то есть ситуацию, когда процесс-получатель извлекает из системного буфера сообщения в том же порядке, в котором они были отправлены. Для надежной и упорядоченной доставки чаще всего выполняется обмен с предварительным установлением **соединения**, причем на стадии установления соединения (называемого также **сеансом**) стороны обмениваются начальными номерами сообщений, чтобы можно было в процессе обмена отслеживать как факт доставки отдельных сообщений последовательности, так и их упорядочивать (сами сетевые технологии не всегда гарантируют, что порядок доставки сообщений будет совпадать с порядком их отправки, например, из-за того, что разные сообщения могут доставляться адресату по разным маршрутам).

В хорошей системе передачи сообщений должны поддерживаться как ненадежные примитивы, так и надежные. Это позволяет прикладному программисту использовать тот тип примитивов, который в наибольшей степени подходит для организации взаимодействия в той или иной ситуации. Например, для передачи данных большого объема, транспортируемых по сети в нескольких сообщениях (в сетях обычно существует ограничение на максимальный размер поля данных, из-за чего данные приходится пересылать в нескольких сообщениях), больше подходит надежный вид обмена с упорядочиванием сообщений. А вот для взаимодействия типа «короткий запрос — короткий ответ» предпочтительны ненадежные примитивы. Действительно, вероятность потери отдельного сообщения не так уж велика, а скорость такого обмена будет выше, чем при применении надежных примитивов, поскольку на установление необходимого в этом случае соединения тратится дополнительное время.

Для реализации примитивов, обладающих различной степенью надежности, система передачи сообщений ОС использует различные коммуникационные протоколы. Так, если сообщения передаются через IP-сеть, то для надежной передачи сообщений применяется протокол TCP транспортного уровня, работающий с установлением соединений, обеспечивающий гарантированную и упорядоченную доставку и управляющий потоком данных при обмене. Если же надежность при передаче сообщений не требуется, применяется протокол UDP, обеспечивающий быструю доставку небольших сообщений без всяких гарантий. Аналогично, при работе через сети Novell для надежной доставки сообщений используется протокол SPX, а для дейтаграммной — IPX. В стеке OSI существует один транспортный протокол, но он поддерживает несколько режимов, отличающихся степенью надежности.

Механизм Sockets ОС Unix

Механизм сокетов (sockets) впервые появился в BSD Unix (Berkeley Software Distribution Unix — ветвь Unix, начавшая развиваться в калифорнийском университете Беркли) версии 4.3. Позже он превратился в одну из самых популярных систем сетевого обмена сообщениями. Сегодня этот механизм реализован во многих операционных системах, иногда его по-прежнему называют Berkeley Sockets, отдавая дань уважения его создателям, хотя существует большое количество его реализаций как для различных ОС семейства Unix, так и для других ОС, например для ОС семейства Windows, где он носит название Windows Sockets (WinSock).

Механизм сокетов предоставляет удобный и достаточно универсальный интерфейс обмена сообщениями, предназначенный для разработки сетевых распределенных приложений. Его универсальность обеспечивают следующие концепции.

- Независимость от нижележащих сетевых протоколов и технологий. Для этого используется понятие **коммуникационный домен** (communication domain). Коммуникационный домен обладает некоторым набором коммуникационных свойств, определяющих способ именования сетевых узлов и ресурсов, характеристики сетевых соединений (надежные, дейтаграммные, упорядо-

ченные), способы синхронизации процессов и т. п. Одним из наиболее популярных доменов является домен Интернета с протоколами стека TCP/IP.

- Использование абстрактной конечной точки соединения, получившей название **сокета** (socket — гнездо). Сокет — это точка, через которую сообщения уходят в сеть или принимаются из сети. Сетевое соединение между двумя процессами осуществляется через пару сокетов. Каждый процесс пользуется своим сокетом, при этом сокеты могут находиться как на разных компьютерах, так и на одном (в этом случае сетевое взаимодействие между процессами сводится к локальному).
- Сокет может иметь как высокоуровневое символьное имя (адрес), так и низкоуровневое, отражающее специфику адресации определенного коммуникационного домена. Например, в домене Интернета низкоуровневое имя представлено парой IP-адрес, порт.
- Для каждого коммуникационного домена могут существовать сокеты различных типов. С помощью сокета можно задать определенный вид взаимодействия, имеющий смысл для домена. Так, во многих доменах, помимо дейтаграммных (datagram), существуют потоковые (stream) соединения, гарантирующие надежную упорядоченную доставку.

Для обмена сообщениями механизм сокетов предлагает следующие примитивы, реализованные как системные вызовы.

- *Создание сокета*:

```
s = socket(domain, type, protocol)
```

Процесс должен создать сокет перед началом его использования. Системный вызов `socket` создает новый сокет с параметрами, определяющими коммуникационный домен (`domain`), тип соединения, поддерживаемого сокетом (`type`), и транспортный протокол (например, TCP или UDP), который будет обслуживать это соединение. Если транспортный протокол не задан, то система сама выбирает протокол, соответствующий типу сокета. Указание домена определяет возможные значения остальных двух параметров. Системный вызов `socket` возвращает дескриптор созданного сокета, который используется как идентификатор сокета в последующих операциях.

- *Связывание сокета с адресом*:

```
bind(s, addr, addrlen)
```

Системный вызов `bind` связывает созданный сокет с его высокоуровневым именем или с низкоуровневым адресом. Адрес `addr` относится к тому узлу, на котором расположен сокет. Для низкоуровневого адреса домена Интернета адресом будет пара IP-адрес, порт. Третий параметр делает адрес независимым от домена, позволяя задавать адреса различных типов, в том числе символьные. Связывать сокет с адресом необходимо только в том случае, если на данный сокет будут приниматься сообщения.

- *Запрос на установление соединения с удаленным сокетом*:

```
connect(s, server_addr, server_addrlrn)
```

Системный вызов connect используется только в том случае, если предполагается передавать сообщения в потоковом режиме, который требует установления соединения. Процедура установления несимметрична: один процесс (процесс-сервер) ждет запроса на установление соединения, а второй (процесс-клиент) инициирует соединение, посылая такой запрос. Системный вызов connect является запросом клиента на установление соединения с сервером. Второй и третий аргументы вызова указывают адрес сокета сервера, с которым устанавливается соединение. После установления соединения сообщения по нему могут передаваться в дуплексном режиме, то есть в любом направлении. Системный вызов write, используемый для передачи сообщений в рамках установленного соединения, не требует указания адреса сокета получателя, так как локальный сокет, через который сообщение отправляется, уже соединен с определенным удаленным сокетом. Способ, с помощью которого клиенты узнают адрес сокета сервера, не стандартизован.

- *Ожидание запроса на установление соединения*:

```
listen (s, backlog)
```

Системный вызов listen используется для организации режима ожидания сервером запросов на установление соединения. Система обмена сообщениями после отработки данного системного вызова будет принимать запросы на установление соединения, имеющие адрес сокета s, и передавать их на обработку другому системному вызову — accept, который решает, принимать их или отвергать. Аргумент backlog оговаривает максимальное число хранимых системой запросов на установление соединения, ожидающих принятия.

- *Принятие запроса на установление соединения*:

```
snew = accept(s, client_addr, client_addrlen)
```

Системный вызов accept используется сервером для приема запроса на установление соединения, поступившего от системного вызова listen через сокет s от клиента с адресом client_addr (если этот аргумент опущен, то принимается запрос от любого клиента). При этом создается новый сокет snew, через который и устанавливается соединение с данным клиентом. Таким образом, сокет s используется сервером для приема запросов на установление соединения от клиентов, а сокеты snew — для обмена сообщениями с клиентами по индивидуальным соединениям.

- *Отправка сообщения по установленному соединению*:

```
write(s, message, msg_len)
```

Сообщение длиной msg_len, хранящееся в буфере message, отправляется получателю, с которым предварительно соединяется сокет s.

- *Прием сообщения по установленному соединению*:

```
nbytes =read(snew, buffer, amount)
```

Сообщение, поступившее через сокет snew, с которым предварительно соединяется отправитель, принимается в буфер buffer размером amount. Если сообщений нет, то процесс-получатель блокируется.

- *Отправка сообщения без установления соединения*:

```
sendto(s, message, receiver_address)
```

Так как сообщение отправляется без предварительного установления соединения, то в каждом системном вызове `sendto` необходимо указывать адрес сокета получателя.

- *Прием сообщения без установления соединения*:

```
amount = recvfrom(s, message, sender_address)
```

Аналогично предыдущему вызову при приеме без установленного соединения в каждом вызове `recvfrom` указывается адрес сокета отправителя, от которого нужно принять сообщение. Если сообщений нет, то процесс-получатель блокируется.

Рассмотрим использование системных вызовов механизма сокетов для организации обмена сообщениями между двумя узлами.

Для обмена короткими сообщениями, не требующими надежной доставки и упорядоченности, целесообразно воспользоваться системными вызовами, не нуждающимися в установлении соединения. Фрагмент программы процесса-отправителя может выглядеть так:

```
.
s = socket(AF_INET, SOCK_DGRAM,0);
.
bind(s, sender_addr, sender_addrlen);
.
sendto(s, message, receiver_addr);
.
close(s);
```

Соответственно, фрагмент программы для процесса-получателя:

```
.
s = socket(AF_INET, SOCK_DGRAM,0);
.
bind(s, receiver_addr, receiver_addrlen);
.
amount = recvfrom(s, message, sender_addr);
.
close(s);
```

Константа `AF_INET` определяет, что обмен ведется в коммуникационном домене Интернета, а константа `SOCK_DGRAM` задает дейтаграммный режим обмена без установления соединения. Выбор транспортного протокола оставлен на усмотрение системы.

Если же необходимо организовать обмен сообщениями надежным способом с упорядочением, то фрагменты программ будут выглядеть следующим образом.

Для процесса-клиента:

```
.
s = socket(AF_INET, SOCK_STREAM,0);
.
connect(s, server_addr, server_addrlen);
```

продолжение ⇩

```
.
write(s, message, msg_len);
.
write(s, message, msg_len);
.
close(s);
```

Для процесса-сервера:

```
.
s = socket(AF_INET, SOCK_STREAM,0);
.
bind(s, server_addr, server_addrlen);
listen(s, backlog);
.
snew = accept(s, client_addr, client_addrlen);
.
nbytes = read(snew, buffer, amount);
.
nbytes = read(snew, buffer, amount);
.
close(s);
```

Вызов удаленных процедур

Еще одним удобным механизмом, облегчающим взаимодействие операционных систем и приложений по сети, является механизм **вызова удаленных процедур** (Remote Procedure Call, RPC). Этот механизм представляет собой надстройку над системой обмена сообщениями ОС, поэтому в ряде случаев он позволяет более удобно и прозрачно организовать взаимодействие программ по сети, однако его полезность не универсальна.

Концепция удаленного вызова процедур

Идея вызова удаленных процедур состоит в расширении хорошо известного и понятного механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. Средства удаленного вызова процедур предназначены для облегчения организации распределенных вычислений. Впервые механизм RPC реализовала компания Sun Microsystems, и он хорошо соответствует девизу «Сеть — это компьютер», взятому этой компанией на вооружение, так как приближает сетевое программирование к локальному. Наибольшая эффективность RPC достигается в тех приложениях, в которых существует интерактивная связь между удаленными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Такие приложения называются *RPC-ориентированными*.

Характерными чертами вызова локальных процедур являются:

- *асимметричность* — одна из взаимодействующих сторон является инициатором взаимодействия;

- *синхронность* — выполнение вызывающей процедуры блокируется с момента выдачи запроса и возобновляется только после возврата из вызываемой процедуры.

Реализация удаленных вызовов существенно сложнее реализации локальных. Начнем с того, что поскольку вызывающая и вызываемая процедуры выполняются на разных машинах, то они имеют разные адресные пространства, и это создает проблемы при передаче параметров и результатов, особенно если машины и их операционные системы не идентичны. Так как RPC не может рассчитывать на разделяемую память, это означает, что параметры RPC не должны содержать указателей на ячейки памяти и что значения параметров должны как-то копироваться с одного компьютера на другой.

Следующим отличием удаленного вызова от локального является то, что хотя он обязательно использует нижележащую систему обмена сообщениями, это не должно быть явно видно ни в определении процедур, ни в самих процедурах. Удаленность вносит дополнительные проблемы. Выполнение вызывающей программы и вызываемой локальной процедуры на одной машине реализуется в рамках единого процесса. Но в реализации RPC участвуют как минимум два процесса — по одному в каждой машине. В случае если один из них аварийно завершится, могут возникнуть следующие ситуации:

- при аварии вызывающей процедуры удаленно вызванные процедуры становятся «осиротевшими»;
- при аварийном завершении удаленных процедур становятся «обездоленными родителями» вызывающие процедуры, которые безрезультатно ожидают ответа от удаленных процедур.

Кроме того, существует ряд проблем, связанных с неоднородностью языков программирования и операционных сред: структуры данных и структуры вызова процедур, поддерживаемые в каком-либо одном языке программирования, не поддерживаются точно таким же способом в других языках.

Рассмотрим, каким образом технология RPC, лежащая в основе многих распределенных операционных систем, решает эти проблемы.

Чтобы понять, как работает RPC, рассмотрим сначала выполнение вызова локальной процедуры на автономном компьютере. Пусть это, например, будет процедура записи данных в файл:

```
m = my_write(fd,buf,length);
```

Здесь `fd` — дескриптор файла, целое число, `buf` — указатель на массив символов, `length` — длина массива, целое число.

Чтобы осуществить вызов, вызывающая процедура помещает указанные параметры в стек в обратном порядке и передает управление вызываемой процедуре `my_write`. Эта пользовательская процедура после некоторых манипуляций с данными символьного массива `buf` выполняет системный вызов `write` для записи данных в файл, передавая ему параметры тем же способом, то есть помещая их в стек (при реализации системного вызова они копируются в стек системы, а при возврате из него результат помещается в пользовательский

стек). После того как процедура `my_write` выполнена, она помещает возвращаемое значение `m` в регистр, перемещает адрес возврата и возвращает управление вызывающей процедуре, которая выбирает параметры из стека, возвращая его в исходное состояние.

Заметим, что в языке C параметры могут вызываться *по ссылке* (by name), представляющей собой адрес глобальной области памяти, в которой хранится параметр, или *по значению* (by value), в этом случае параметр копируется из исходной области памяти в локальную память процедуры, располагаемую обычно в стековом сегменте. В первом случае вызываемая процедура работает с оригинальными значениями параметров, и их изменения сразу же видны вызывающей процедуре. Во втором случае вызываемая процедура работает с копиями значений параметров, и их изменения никак не влияют на значение оригиналов этих переменных в вызывающей процедуре. Эти обстоятельства весьма существенны для RPC.

Решение о том, какой механизм передачи параметров использовать, принимается разработчиками языка. Иногда это зависит от типа передаваемых данных. В языке C, например, целые и другие скалярные данные всегда передаются по значению, а массивы — по ссылке.

Рисунок иллюстрирует передачу параметров вызываемой процедуре: стек до выполнения вызова `write` (рис. 10.6, *а*), стек во время выполнения процедуры (рис. 10.6, *б*), стек после возврата в вызывающую программу (рис. 10.6, *в*).

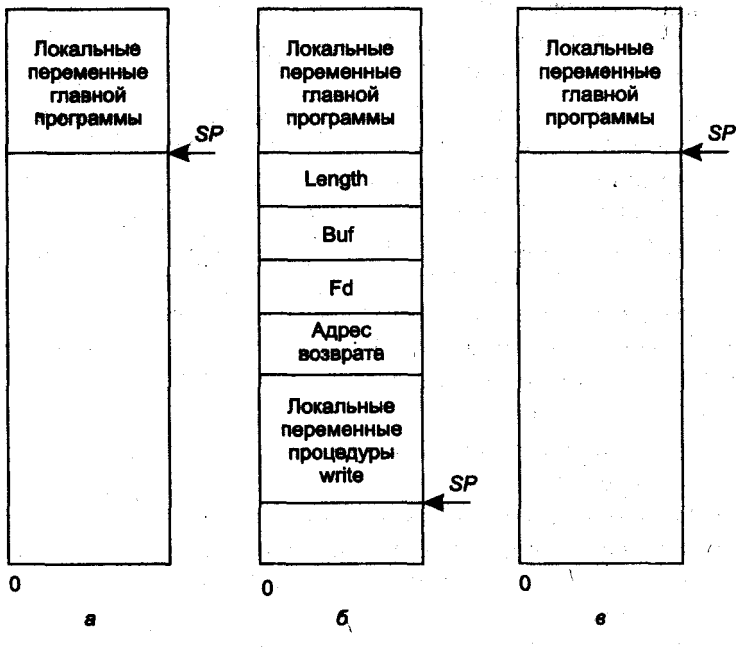


Рис. 10.6. Передача параметров вызываемой процедуре

Идея, положенная в основу RPC, состоит в том, чтобы вызов удаленной процедуры по возможности выглядел так же, как и вызов локальной процедуры. Другими словами, необходимо сделать механизм RPC *прозрачным* для программиста: вызывающей процедуре не требуется знать, что вызываемая процедура находится на другой машине, и наоборот.

Механизм RPC достигает прозрачности следующим образом. Когда вызываемая процедура действительно является удаленной, в библиотеку процедур вместо локальной реализации оригинального кода процедуры помещается другая версия процедуры, называемая **клиентским стабом** (stub — заглушка). На удаленный компьютер, который исполняет роль сервера процедур, помещается оригинальный код вызываемой процедуры, а также еще один стаб, называемый **серверным стабом**. Назначение клиентского и серверного стабов — организовать передачу параметров вызываемой процедуры и возврат значения процедуры через сеть так, чтобы код оригинальной процедуры, помещенной на сервер, был полностью сохранен. Стабы используют для передачи данных через сеть средства подсистемы обмена сообщениями, то есть существующие в ОС примитивы `send` и `receive`. Иногда в подсистеме обмена сообщениями выделяется

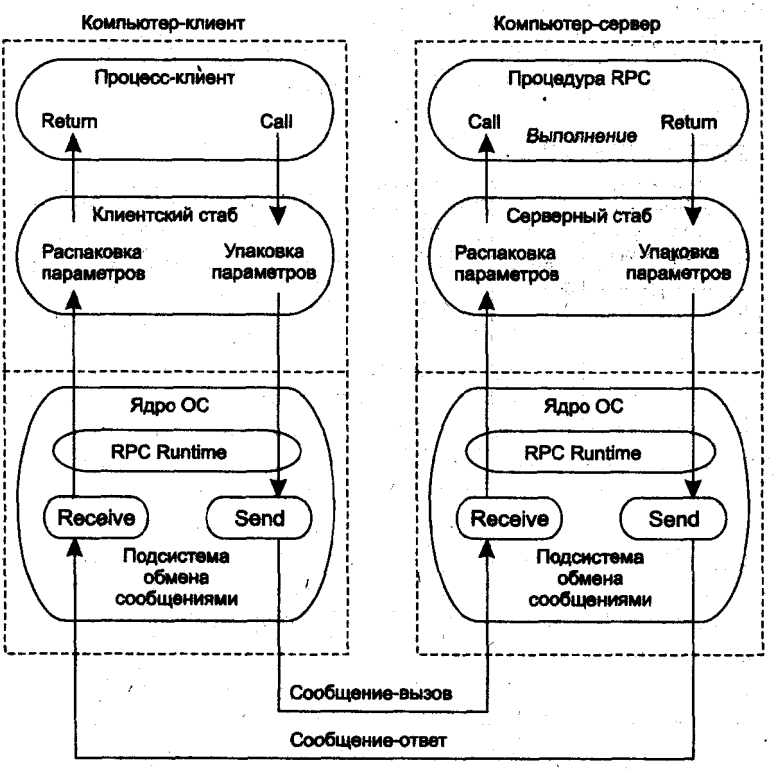


Рис. 10.7. Выполнение удаленного вызова процедуры

программный модуль, организующий связь стабов с примитивами передачи сообщений, называемый модулем RPC Runtime.

Подобно оригинальной процедуре, клиентский стаб вызывается путем обычной передачи параметров через стек (как показано на рис. 10.6), однако затем вместо выполнения системного вызова, работающего с локальным ресурсом, происходит формирование сообщения, содержащего имя вызываемой процедуры и ее параметры (рис. 10.7).

Эта операция называется *упаковкой параметров*. После упаковки клиентский стаб обращается к примитиву `send` для передачи сформированного сообщения удаленному компьютеру, на который помещена реализация оригинальной процедуры. Получив из сети сообщение, ядро ОС удаленного компьютера вызывает серверный стаб. Для получения сообщения серверный стаб должен предварительно вызвать примитив `receive`, чтобы ядро знало, для кого пришло сообщение. Серверный стаб распаковывает параметры вызова, имеющиеся в сообщении, и обычным образом вызывает оригинальную процедуру, передавая ей параметры через стек. После окончания работы процедуры серверный стаб упаковывает результат ее работы в новое сообщение и с помощью примитива `send` передает сообщение по сети клиентскому стабу, и тот возвращает обычным образом результат и управление вызывающей процедуре. Ни вызывающая процедура, ни оригинальная вызываемая процедура не изменяются оттого, что выполняются на разных компьютерах.

Генерация стабов

Стабы могут генерироваться либо вручную, либо автоматически. В первом случае программист использует для генерации ряд вспомогательных функций, которые ему предоставляет разработчик средств RPC. Программист при этом способе получает большую свободу в выборе способа передачи параметров вызова и применении тех или иных примитивов передачи сообщений, однако этот способ связан с большим объемом ручного труда.

Автоматический способ основан на применении специального **языка определения интерфейса** (Interface Definition Language, IDL). С помощью этого языка программист описывает интерфейс между RPC-клиентом и RPC-сервером. Описание включает список имен процедур, выполнение которых клиент может запросить у сервера, а также список типов аргументов и результатов этих процедур. Информация, содержащаяся в описании интерфейса, достаточна для выполнения стабами проверки типов аргументов и генерации вызывающей последовательности. Кроме того, описание интерфейса содержит некоторую дополнительную информацию, полезную для оптимизации взаимодействия стабов, например, каждый аргумент помечается как входной, как выходной или как играющий обе роли (входной аргумент передается от клиента серверу, а выходной — в обратном направлении). Интерфейс может включать также описание общих для клиента и сервера констант. Необходимо подчеркнуть, что обычно интерфейс RPC включает не одну, а некоторый набор процедур, выполняющих взаимосвязанные функции, например функции доступа к файлам,

функции удаленной печати и т. п. Поэтому при вызове удаленной процедуры обычно необходимо каким-то образом задать нужный интерфейс, а также конкретную процедуру, поддерживаемую этим интерфейсом. Часто интерфейс также называют RPC-сервером, например, файловым сервером, сервером печати.

После того как описание интерфейса составлено программистом, оно компилируется специальным IDL-компилятором, который вырабатывает исходные модули клиентских и серверных стабов для указанных в описании процедур, а также генерирует специальные файлы-заголовки с описанием типов процедур и их аргументов. Генерация исходных модулей и файлов-заголовков стабов выполняется для конкретного языка программирования, например для языка C. После этого исходные модули интерфейса могут включаться в любое приложение наряду с любыми другими модулями, как написанными программистом, так и библиотечными, компилироваться и связываться в исполняемую программу стандартными средствами инструментальной системы программирования.

Формат RPC-сообщений

Механизм RPC оперирует двумя типами сообщений:

- *сообщениями-вызовами*, с помощью которых клиент запрашивает у сервера выполнение определенной удаленной процедуры и передает ее аргументы;
- *сообщениями-ответами*, с помощью которых сервер возвращает результат работы удаленной процедуры клиенту, например сообщением об ошибке.

С помощью этих сообщений реализуется протокол RPC, определяющий способ взаимодействия клиента с сервером. Протокол RPC предусматривает установление сеанса связи на период выполнения удаленной процедуры, то есть этот протокол может быть отнесен *к сеансовому уровню* модели OSI.

Реализации RPC обычно не связаны жестко с каким-либо из протоколов транспортного уровня, с помощью которых RPC-сообщения должны доставляться по сети от клиента к серверу и обратно. При использовании в сети стека протоколов TCP/IP это может быть протокол TCP или UDP, в локальных сетях часто используется также протокол NetBEUI/NetBIOS или IPX/SPX.

Типичный формат двух типов сообщений, используемых RPC, показан на рис. 10.8.

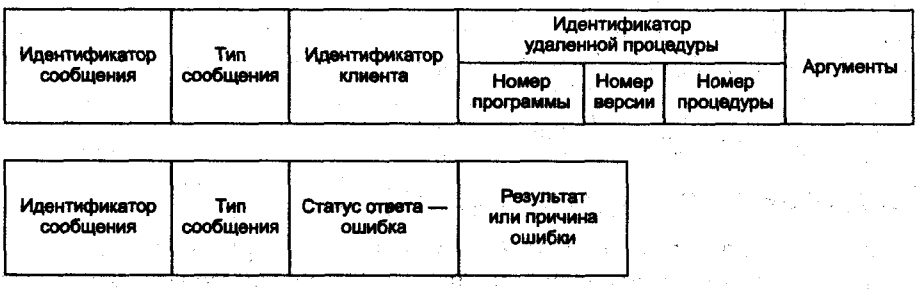


Рис. 10.8. Формат RPC-сообщений

Тип сообщения позволяет отличить сообщения-вызовы от сообщений-ответов. Поле идентификатора удаленной процедуры в сообщении-вызове дает серверу понять, вызов какой процедуры запрашивает в сообщении клиент (процедуры идентифицируются не именами, а номерами, которые при автоматической генерации стабов присваивает им IDL-компилятор, а при ручной — программист). Поле аргументов имеет переменную длину, определяемую количеством и типом аргументов вызываемой процедуры. В поле идентификатора сообщения помещается порядковый номер сообщения, который полезен для обнаружения фактов потерь сообщений или прихода дубликатов сообщений. Кроме того, этот номер позволяет клиенту правильно сопоставить полученный от сервера ответ со своим вызовом в том случае, когда ответы приходят не в том порядке, в котором посылались вызовы. Идентификатор клиента нужен серверу для того, чтобы знать, какому клиенту нужно отправить результат работы вызываемой процедуры. Это поле может также использоваться в процедурах аутентификации клиента, если эти процедуры предусмотрены протоколом RPC.

Поля статуса и результата в сообщениях-ответах позволяют серверу сообщить клиенту об успешном выполнении удаленной процедуры или же о том, что при попытке выполнения процедуры была зафиксирована ошибка определенного типа. Сервер может столкнуться с различными ситуациями, препятствующими нормальному выполнению процедуры. Такие ситуации могут возникнуть еще до выполнения, так как клиенту может быть не разрешено выполнять данную процедуру, клиент может неправильно сформировать аргументы процедуры, клиентом могут использоваться версии интерфейса, не удовлетворяющие сервер и т. п. Выполнение процедуры также может привести к ошибке, например, связанной с делением на ноль. Все эти ситуации должны корректно обрабатываться клиентским стабом, и соответствующие коды ошибок должны передаваться в вызывающую процедуру.

Для устойчивой работы RPC-серверов и RPC-клиентов необходимо каким-то образом обрабатывать ситуации, связанные с потерями сообщений, которые происходят из-за ошибок сети (эти ошибки транспортные протоколы пытаются компенсировать, но все равно некоторая вероятность потерь остается) или же по причине краха операционной системы и перезагрузки компьютера (здесь транспортные протоколы исправить ситуацию не могут). Протокол RPC использует в таких случаях механизм тайм-аутов с повторной передачей сообщений. Для того чтобы сервер мог повторно переслать клиенту потерянный результат без необходимости передачи от клиента повторного вызова, в протокол RPC иногда добавляется специальное сообщение — подтверждение клиента, которое тот посылает при получении ответа от сервера.

Связывание клиента с сервером

Рассмотрим вопрос о том, как клиент узнает место расположения сервера, которому необходимо послать сообщение-вызов. Процедура, устанавливающая соответствие между RPC-клиентом и RPC-сервером, носит название **связывания**

(binding). Методы связывания, применяемые в различных реализациях RPC, различаются:

- способом задания сервера, с которым хотел бы быть связанным клиент;
- способом обнаружения процессом связывания сетевого адреса (места расположения) требуемого сервера;
- стадией, на которой происходит связывание.

Метод связывания тесно связан с принятым методом именования сервера. В наиболее простом случае *имя или адрес RPC-сервера задается в явной форме*, в качестве аргумента клиентского стаба или программы-сервера, реализующей интерфейс определенного типа. Например, можно использовать в качестве такого аргумента IP-адрес компьютера, на котором работает некоторый RPC-сервер, а также номер порта TCP/UDP, через который он принимает сообщения-вызовы своих процедур. Основной недостаток такого подхода — *отсутствие гибкости* и *прозрачности*. При перемещении сервера или при существовании нескольких серверов клиентская программа не сможет автоматически выбрать новый сервер или тот сервер, который в данный момент наименее загружен. Тем не менее во многих случаях такой способ вполне приемлем и ввиду своей простоты часто используется на практике. Необходимый сервер обычно выбирает пользователь, например, путем просмотра списка или графического представления имеющихся в сети разделяемых файловых систем (набор этих файловых систем может быть собран операционной системой клиентского компьютера путем прослушивания широковещательных объявлений, которые периодически делают серверы). Кроме того, пользователь может задать имя требуемого сервера на основании заранее известной ему информации об адресе или имени сервера.

Подобный метод связывания можно назвать *статическим*. Существуют и другие методы, не требующие от клиента точного задания адреса RPC-сервера, вплоть до указания номера порта. Эти методы *динамически* находят нужный клиенту сервер.

Динамическое связывание требует изменения способа именования сервера. Наиболее гибким является использование для этой цели имени RPC-интерфейса, состоящего из двух частей:

- типа интерфейса;
- экземпляра интерфейса.

Тип интерфейса определяет все характеристики интерфейса, кроме его месторасположения. Это те же характеристики, что имеются в описании для IDL-компилятора, например, файловая служба определенной версии, включающая процедуры `open`, `close`, `read`, `write` и т. п. Часть, описывающая *экземпляр интерфейса*, должна точно задать сетевой адрес сервера, который поддерживает данный интерфейс. Если клиенту безразлично, какой сервер его будет обслуживать, то вторая часть имени интерфейса опускается.

Динамическое связывание иногда называют импортом/экспортом интерфейса: клиент импортирует интерфейс, а сервер его экспортирует.

В том случае, когда для клиента важен только тип интерфейса, процесс обнаружения требуемого сервера в сети с экземпляром интерфейса определенного типа может быть реализован двумя способами:

- с использованием широковещания;
- с использованием централизованного агента связывания.

Эти два способа обеспечивают поиск сетевого ресурса любого типа по его имени, они уже рассматривались в общем виде в разделе «Способы адресации». Первый способ основан на широковещательном распространении по сети RPC-серверами имени своего интерфейса, включая адрес экземпляра. Применение этого способа позволяет автоматически балансировать нагрузку на несколько серверов, поддерживающих один и тот же интерфейс, — клиент просто выбирает первое из подходящих ему объявлений.

Схема с централизованным агентом связывания предполагает наличие в сети сервера имен, который связывает тип интерфейса с адресом сервера, поддерживающего такой интерфейс. Для реализации этой схемы каждый RPC-сервер должен предварительно зарегистрировать тип своего интерфейса и сетевой адрес у агента связывания, работающего на сервере имен. Сетевой адрес агента связывания (в формате, принятом в данной сети) должен быть известным адресом как для RPC-серверов, так и для RPC-клиентов. Если сервер по каким-то причинам не может больше поддерживать определенный RPC-интерфейс, то он должен обратиться к агенту и отменить свою регистрацию. Агент связывания на основании запросов регистрации ведет таблицу текущего наличия в сети серверов и поддерживаемых ими RPC-интерфейсов.

RPC-клиент для определения адреса сервера, обслуживающего требуемый интерфейс, обращается к агенту связывания с указанием характеристик, задающих тип интерфейса. Если в таблице агента связывания имеются сведения о сервере, поддерживающем такой тип интерфейса, то он возвращает клиенту сетевой адрес этого сервера. Клиент затем кэширует эту информацию для того, чтобы при последующих обращениях к процедурам данного интерфейса не тратить время на обращения к агенту связывания.

Агент связывания может работать в составе общей централизованной справочной службы сети, такой как NDS, X.500 или LDAP (справочные службы подробно рассматриваются в следующей главе).

Описанный метод, заключающийся в импорте/экспорте интерфейсов, обладает большой *гибкостью*. Например, может быть несколько серверов, поддерживающих один и тот же интерфейс, и клиенты распределяются по серверам случайным образом. В рамках этого метода становится возможным периодический опрос серверов, анализ их работоспособности и, в случае отказа, автоматическое отключение, что повышает общую отказоустойчивость системы. Этот метод может также поддерживать аутентификацию клиента. Например, сервер может определить, что доступ к нему разрешается только клиентам из определенного списка.

Однако у динамического связывания имеются недостатки, например *дополнительные накладные расходы* (временные затраты) на экспорт и импорт интерфейсов. Величина этих затрат может быть значительной, так как многие клиентские процессы существуют короткое время, а при каждом старте процесса процедура импорта интерфейса должна выполняться заново. Кроме того, в больших распределенных системах узким местом может стать агент связывания, и тогда необходимо использовать распределенную систему агентов, что можно сделать стандартным способом с помощью распределенной справочной службы (таким свойством обладают службы NDS, X.500 и LDAP).

Необходимо отметить, что и в тех случаях, когда используется статическое связывание, такая часть адреса, как порт сервера интерфейса (то есть идентификатор процесса, обслуживающего данный интерфейс), определяется клиентом динамически. Эту процедуру поддерживает специальный модуль RPC Runtime, называемый в ОС Unix **модулем отображения портов** (portmapper), а в ОС семейства Windows — **RPC-локатором** (RPC Locator). Этот модуль работает на каждом сетевом узле, поддерживающем механизм RPC, и доступен по хорошо известному порту TCP/UDP. Каждый RPC-сервер, обслуживающий определенный интерфейс, при старте обращается к такому модулю с запросом о выделении ему для работы номера порта из динамически распределяемой области (то есть с номером, большим 1023). Модуль отображения портов выделяет серверу некоторый свободный номер порта и запоминает это отображение в своей таблице, связывая порт с типом интерфейса, поддерживаемым сервером. RPC-клиент, выяснив каким-либо образом сетевой адрес узла, на котором имеется RPC-сервер с нужным интерфейсом, предварительно соединяется с модулем отображения портов по *хорошо известному порту* и запрашивает номер порта искомого сервера. Получив ответ, клиент использует данный номер для отправки сообщений-вызовов удаленных процедур. Механизм очень напоминает механизм, лежащий в основе работы агента связывания, но только область его действия ограничивается портом одного компьютера.

Особенности реализации RPC на примере систем Sun RPC и DCE RPC

Рассмотрим особенности реализации RPC на примере двух широко распространенных систем удаленного вызова процедур: Sun RPC и DCE RPC. Система Sun RPC является продуктом компании Sun Microsystems и работает во всех сетевых операционных системах этой компании — SunOS, Solaris, а система DCE RPC — это стандарт консорциума Open Software Foundation для распределенной вычислительной среды (Distributed Computing Environment, DCE). Реализации DCE RPC доступны сегодня для многих сетевых ОС, кроме того, на основе стандарта DCE RPC разработана система Microsoft RPC, применяющаяся

в популярных ОС семейства Windows. К сожалению, реализации Sun RPC и DCE RPC несовместимы друг с другом, более того, нет гарантий, что различные реализации RPC, в основе которых лежит стандарт DCE RPC, смогут совместно работать в гетерогенной сети, так как стандарт DCE определяет только базовые свойства механизма удаленного вызова процедур, а каждая реализация добавляет к стандарту большое количество собственных дополнительных функций.

Sun RPC

Система Sun RPC позволяет *автоматически* генерировать клиентский и серверный стабы — в том случае, если RPC-интерфейс описан на языке IDL, называемом RPC Language (RPCL). Язык RPCL является расширением языка Sun XDR (eXternal Data Representation), который был разработан для системно-независимого представления внешних данных в гетерогенной среде. XDR-представление данных по умолчанию используется в Sun RPC при передаче аргументов и результатов между RPC-клиентом и RPC-сервером.

Механизм Sun RPC обладает некоторыми достаточно жесткими *ограничениями.*

Одним из них является ограничение на аргументы и результаты удаленных процедур — процедура может иметь только *один аргумент* и вырабатывать только *один результат.* Для преодоления этого ограничения в качестве аргумента и результата обычно используется структура данных.

Следующий пример описания интерфейса файловой службы иллюстрирует эту особенность:

```
/* Определение интерфейса для файловой службы с именем FILE_SERVICE_2,
включающей процедуры чтения READ и записи WRITE */
const FILE_NAME_SIZE = 16
const BUFFER_SIZE = 1024

typedef string FileName<FILE_NAME_SIZE>
typedef long Position;
typedef long Nbytes;

struct Data {
        long n;
        char buffer[BUFFER_SIZE];
    };
struct readargs {
        FileName    filename;
        Position    position;
        Nbytes        n;
    };
struct writeargs {
        FileName    filename;
        Position    position;
        Data        data;
    };
```

```
program FILE_SERVICE_2 {
        version FILE_SERVICE_VERS {
            Data        READ (readargs) = 1;
            Nbytes        WRITE (writeargs) = 2;
        }= 1;
} = 0x20000000;
```

Интерфейс однозначно идентифицируется номером программы (`FILE_SERVICE_2 = 0x20000000`) и номером версии (`FILE_SERVICE_VERS = 1`), а процедуры внутри интерфейса — номерами процедур, `READ` — номером 1 и `WRITE` — номером 2.

Структура `readargs` позволяет передать процедуре `READ` три аргумента — имя файла, позицию (смещение) в файле, с которой нужно начать чтение данных, и количество считываемых байтов. Структура `Data` позволяет вызывающей процедуре получить результат чтения в массиве `buffer` и узнать количество реально считанных байтов с помощью переменной `n`. Аналогично используются структуры `writeargs` и `Data` в процедуре записи `WRITE`.

Результатом обработки описания интерфейса `FILE_SERVICE_2` RPCL-компилятором являются несколько файлов, в том числе файлов, описывающих клиентские и серверные стабы. По умолчанию имена стабов процедур образуются путем преобразования имен процедур, указанных в описании интерфейса, в нижний регистр, а также добавлением после нижнего подчеркивания номера версии интерфейса, то есть стабы в нашем примере будут иметь имена `read_1` и `write_1`.

Механизм Sun RPC *не поддерживает динамического связывания* с сервером с помощью агента связывания. Клиент должен явно указывать сетевой адрес сервера, а для получения номера порта он должен обратиться к модулю отображения портов, работающему на узле сервера. Для взаимодействия с этим модулем система Sun RPC предоставляет в распоряжение клиента системный вызов `clnt_create`, имеющий следующий синтаксис:

```
client_handle = clnt_create(server_host_name, interface_name,
interface_version, protocol)
```

Аргумент `server_host_name` представляет собой имя (символьное DNS-имя или IP-адрес узла, на котором работает RPC-сервер), аргументы `interface_name` и `interface_version` задают номер и версию интерфейса, а аргумент `protocol` указывает на один из двух транспортных протоколов стека TCP/IP — TCP или UDP. Жесткая ориентация только на один стек коммуникационных протоколов, а именно *стек TCP/IP*, — еще одно ограничение Sun RPC. У компании Sun существует также протокольно-независимая версия системы удаленного вызова процедур — TI-RPC (Transport-Independent RPC), но она менее распространена.

Вызов `clnt_create` возвращает указатель, который необходимо далее использовать вместо адреса RPC-сервера при последовательных обращениях к удаленным процедурам, обслуживаемым данным сервером. В процессе своего

выполнения `clnt_create` создает сокет, который связывает с адресом сервера, включающим и неявно полученный от модуля отображения портов (службы portmapper) номер порта. В конце сеанса работы с сервером необходимо выполнить системный вызов `clnt_destroy`, который закрывает созданный сокет.

Еще одним ограничением Sun RPC является *максимальный размер сообщения в 8 Кбайт* при использовании протокола UDP (применение протокола TCP не накладывает таких ограничений в силу особенности его интерфейса с вышележащими протоколами, который позволяет передавать непрерывный поток байтов в течение периода существования TCP-соединения).

DCE RPC

Служба DCE RPC обладает рядом функциональных преимуществ по сравнению с Sun RPC. Она поддерживает динамическое связывание клиентов и серверов, для чего используется справочная служба среды DCE — служба Cell Directory Service (CDS). Каждый RPC-сервер при старте регистрирует свой сетевой адрес и уникальный идентификатор интерфейса на CDS-сервере. Кроме того, на каждом узле, поддерживающем RPC, работает процесс `rpcd`, который выполняет функции по отображению RPC-сервера на подходящий локальный адрес процесса (например, порт TCP/UDP, если сервер работает на компьютере, поддерживающем стек TCP/IP). Служба DCE RPC является транспортно-независимой, что позволяет ей работать на разных платформах и в различных сетях.

При описании интерфейса используется язык IDL. Процедуры DCE RPC могут иметь произвольное число аргументов, которые описываются как входные, выходные или входные/выходные.

Выводы

- Распределенные приложения обладают рядом потенциальных преимуществ по сравнению с локальными, такими как более высокая производительность, отказоустойчивость, масштабируемость.
- Основными схемами разделения приложений на функциональные части являются двухзвенная и трехзвенная модели, в которых вычислительная нагрузка распределяется между двумя или тремя компьютерами соответственно.
- Единственным по-настоящему важным отличием распределенных систем от централизованных является используемый ими способ взаимодействия между процессами. В централизованных системах связь между процессами, как правило, предполагает наличие разделяемой памяти. В распределенных системах взаимодействие процессов может осуществляться только путем передачи сообщений через сеть.
- Сообщение — это блок информации, отформатированный процессом-отправителем таким образом, чтобы он был понятен процессу-получателю. Сооб-

щение состоит из заголовка, обычно фиксированной длины, и набора данных определенного типа переменной длины.

- Все сетевые службы, предоставляющие пользователям сети разнообразные услуги (доступ к удаленным файлам, принтерам, почтовым ящикам и т. п.), работают на основе двух основных коммуникационных примитивов — `send` (отправить) и `receive` (получить).
- Основными характеристиками коммуникационных примитивов являются:
  - способ адресации;
  - наличие синхронизации;
  - способ буферизации;
  - степень надежности доставки.
- Механизм сокетов предоставляет удобный и универсальный интерфейс обмена сообщениями, предназначенный для разработки сетевых распределенных приложений. Сокет — это абстрактная конечная точка, через которую сообщения уходят в сеть или принимаются из сети. Сетевое соединение между двумя процессами осуществляется через пару сокетов, каждый процесс пользуется собственным сокетом.
- Еще одним удобным механизмом, облегчающим взаимодействие операционных систем и приложений по сети, является механизм вызова удаленных процедур (RPC). Идея, положенная в основу RPC, состоит в том, чтобы вызов удаленной процедуры по возможности выглядел так же, как и вызов локальной.

Задачи и упражнения

1. Приведите пример сетевого приложения. Каким образом в этом приложении могут быть распределены функции между клиентской и серверной частями? Каким могло бы быть распределение функций, если бы приложение имело трехзвенную структуру?
2. Чем отличаются взаимодействия процессов в рамках одного компьютера от их взаимодействия по сети?
3. В каком случае работа с удаленной базой данных порождает более интенсивный трафик: при использовании модели файлового сервера или сервера базы данных?
4. В каком случае прикладному программисту проще писать программу: с использованием синхронных или асинхронных примитивов передачи сообщений?
5. Какая структура операционной системы соответствует понятию «порт», используемому в протоколах TCP/UDP?

6. В каких случаях целесообразно применять ненадежные примитивы передачи сообщений?
7. Используя интерфейс сокетов, опишите взаимодействие двух процессов, один из которых является сервером, а другой — клиентом файловой службы.
8. В чем состоит основное назначение механизма RPC?
9. Почему в RPC-процедурах не используются глобальные переменные?
10. Почему в системных вызовах RPC аргументы передаются по значению (by value), а не по ссылке (by name)?
11. Опишите процедуру автоматической генерации стабов.

Глава 11
Сетевые службы

Для работы в сети операционные системы должны быть дополнены специальными сетевыми службами, которые позволяют пользователям получать доступ не только к локальным ресурсам их собственных компьютеров, но и к ресурсам других компьютеров, подключенных к сети (конечно, только в том случае, если удаленные ресурсы объявлены разделяемыми и у пользователя есть соответствующие права на доступ к ним). Эти службы реализуют специфические функции по организации совместной работы пользователей сети, опираясь на функции ОС по управлению локальными ресурсами. В этой главе рассматриваются принципы построения и реализации двух очень важных и тесно взаимосвязанных сетевых служб — сетевой файловой системы и справочной службы. Завершает главу обзор подходов и методов, связанных с интеграцией сетевых служб в неоднородных компьютерных сетях.

Сетевая файловая система

Модель клиент-сервер сетевой файловой системы

Сетевая файловая система — это сетевая служба, предоставляющая пользователям сети услуги по совместному использованию файлов, хранящихся на компьютерах сети.

Как и любая сетевая служба, сетевая файловая система имеет клиент-серверную природу (подробнее об этом см. раздел «Сетевые службы и сетевые сервисы» в главе 2).

Сетевая файловая система (ФС) в общем случае включает следующие элементы (рис. 11.1):

- клиент сетевой файловой системы;
- сервер сетевой файловой системы;
- интерфейс сетевой файловой системы;
- локальная файловая система;

- интерфейс локальной файловой системы;
- протокол клиент-сервер сетевой файловой системы.

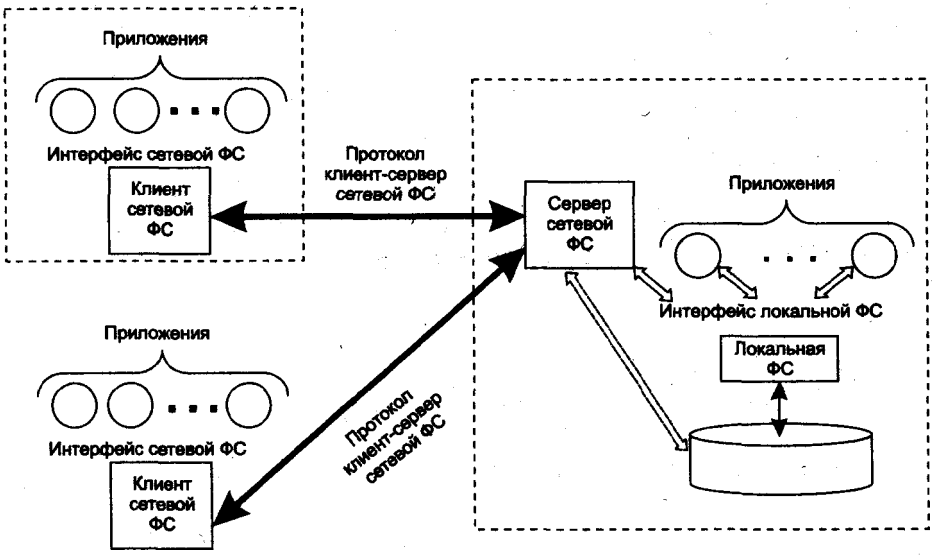


Рис. 11.1. *Модель сетевой файловой системы*

Клиенты сетевой ФС — это программы, которые работают на многочисленных компьютерах, подключенных к сети. Эти программы обслуживают запросы приложений на доступ к файлам, хранящимся на удаленном компьютере. В качестве таких приложений часто выступают графические или символьные оболочки ОС, такие как Проводник Windows (Windows Explorer) или Unix shell, а также любые другие пользовательские программы.

Клиент сетевой ФС передает по сети запросы другому программному компоненту — **серверу сетевой ФС**, работающему на удаленном компьютере. Сервер, получив запрос, может выполнить его либо самостоятельно, либо, что является более распространенным вариантом, передать запрос локальной файловой системе для обслуживания. После получения ответа от локальной файловой системы сервер передает его по сети клиенту, а тот, в свою очередь, — приложению, обратившемуся с запросом.

ПРИМЕЧАНИЕ Файловым сервером называют как программу, которая предоставляет совокупность услуг по доступу через сеть к файлам и каталогам, так и компьютер, на котором работает эта программа и на котором хранятся предоставляемые в совместный доступ файлы. Соответственно, файловым клиентом называют как программу, обращающуюся к файловому серверу с запросами, так и компьютер, на котором она работает. Такая неоднозначность терминов обычно не вызывает затруднений, поскольку из контекста обычно понятно, о каком, программном или аппаратном, компоненте сети идет речь.

Приложения обращаются к клиенту сетевой ФС, используя определенный программный интерфейс, который в данном случае является **интерфейсом сетевой файловой системы**. Этот интерфейс стараются сделать как можно более похожим на **интерфейс локальной файловой системы**, чтобы соблюсти принцип прозрачности. В идеале сетевая файловая система должна выглядеть для пользователя так же, как и его локальная файловая система. При полном совпадении интерфейсов приложение может обращаться к локальным и удаленным файлам и каталогам с помощью одних и тех же системных вызовов, совершенно не принимая во внимание места хранения данных. Такой эффект достигается, если, например, при использовании на сервере сети локальной файловой системы FAT интерфейс сетевой файловой системы повторяет системные вызовы FAT.

Клиент и сервер взаимодействуют друг с другом через сеть по **протоколу сетевой файловой системы**. Если интерфейсы локальной и сетевой файловых систем совпадают, этот протокол может быть достаточно простым — в его функции должна входить ретрансляция серверу запросов, принятых клиентом от приложений, с которыми тот затем будет обращаться к локальной ФС. Одним из механизмов, используемых для этой цели, может быть механизм RPC. Протокол сетевой файловой системы, помимо простой ретрансляции системных файловых вызовов от клиента серверу, может выполнять и более сложные функции, учитывающие природу сетевого взаимодействия, например, то, что клиент и сервер работают на разных компьютерах, которые могут отказать, или что сообщения передаются по ненадежной и вносящей порой большие задержки сетевой среде.

В качестве иллюстрации приведенной модели давайте рассмотрим сетевую файловую систему, построенную на базе локальной файловой системы FAT и использующую в качестве протокола клиент-сервер протокол **SMB** (Server Message Block). Этот протокол был совместно разработан компаниями Microsoft, Intel и IBM. Расширенные версии протокола SMB получили название CIFS (Common Internet File System). Протокол SMB/CIFS является основой сетевой файловой службы в операционных системах семейства Windows компании Microsoft.

Как и все протоколы сетевых служб, протокол SMB работает на прикладном уровне модели OSI. Для передачи по сети своих сообщений протокол SMB использует различные транспортные протоколы более низких уровней. SMB относится к классу протоколов, ориентированных на соединение. Работа протокола начинается с того, что клиент отправляет серверу специальное сообщение с запросом на установление соединения. В процессе установления соединения SMB-клиент и SMB-сервер обмениваются информацией о себе: они сообщают друг другу, какой диалект протокола SMB они будут использовать в этом соединении (диалект здесь — определенное подмножество функций протокола, так как помимо файловых функций SMB поддерживает доступ к принтерам, управление внешними устройствами и некоторые другие). Если сервер готов к установлению соединения, он отвечает сообщением-подтверждением.

После установления соединения клиент может обращаться к серверу, передавая ему в SMB-сообщениях команды манипулирования файлами и каталогами. Клиент может попросить сервер создать и удалить каталог, предоставить содержимое каталога, создать и удалить файл, прочитать и записать содержимое файла, установить атрибуты файла и т. п.

Другим примером популярной сетевой файловой системы является ФС, которую компания Novell реализовала в ОС NetWare, на протяжении многих лет доминировавшей в локальных сетях. Эта ОС использовала в качестве локальной файловой системы на файловых серверах собственную оригинальную ФС, также носящую имя NetWare, а для доступа к файлам по сети — протокол **NCP** (NetWare Control Protocol — основной протокол доступа к файлам и принтерам компании Novell).

В среде операционной системы Unix наибольшее распространение получили две сетевые файловые системы и соответственно два протокола клиент-сервер — **FTP** (File Transfer Protocol) и **NFS** (Network File System). Они первоначально разрабатывались для доступа к локальной файловой системе s5/ufs, являющейся основной для многих ОС семейства Unix.

Модель неоднородной сетевой файловой системы

Со временем в крупных сетях стали одновременно применяться несколько сетевых файловых систем разных типов, например NFS и SMB. Это часто происходило при объединении нескольких сетей в одну. Для пользователей каждой из сетей, привыкших к определенному интерфейсу и работающих с приложениями, рассчитанными на интерфейс FAT или s5/ufs, требовалось сохранить удобную среду.

В результате в сети появились различные файловые серверы, поддерживающие разные локальные файловые системы и протоколы клиент-сервер, а также клиенты, предлагающие приложениям и пользователям разные интерфейсы. Например, в распределенной системе может быть два сервера, которые предоставляют файловые услуги систем Unix и Windows соответственно, в результате пользовательские процессы могут обращаться к подходящему серверу.

Возникла проблема — как обеспечить доступ клиента любого типа к файловому серверу любого типа? Рассмотренные ранее принципы организации сетевой файловой системы и ее основных компонентов подсказывают, что существует несколько вариантов решения этой проблемы, основанные на комбинировании локальных файловых систем, протоколов клиент-сервер и интерфейсов, поддерживаемых клиентами файловой системы.

В общем случае для доступа по сети к одной и той же локальной ФС может использоваться несколько различных протоколов сетевой файловой системы. Так, к файловой системе NTFS сегодня можно получить доступ с помощью различных протоколов (рис. 11.2), в том числе SMB, NCP и NFS.

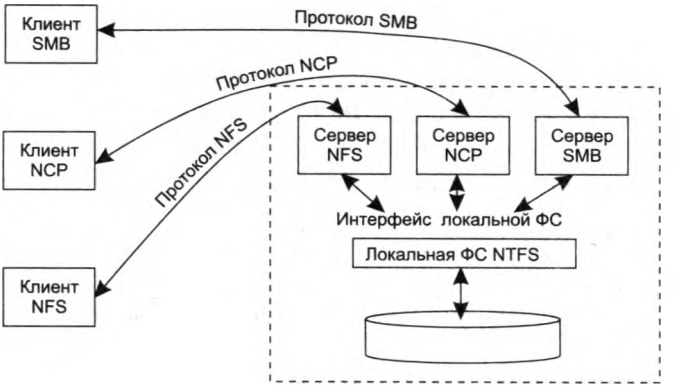


Рис. 11.2. Доступ к одной локальной файловой системе с помощью нескольких протоколов клиент-сервер

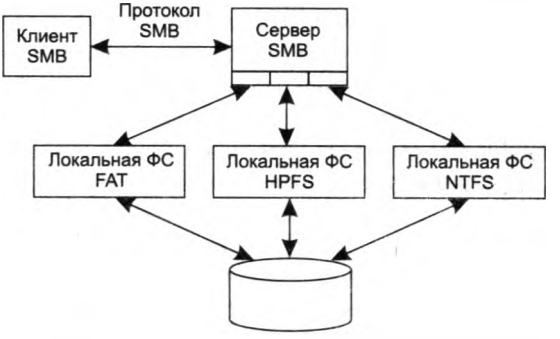


Рис. 11.3. Доступ к локальным файловым системам различного типа с помощью одного протокола клиент-сервер

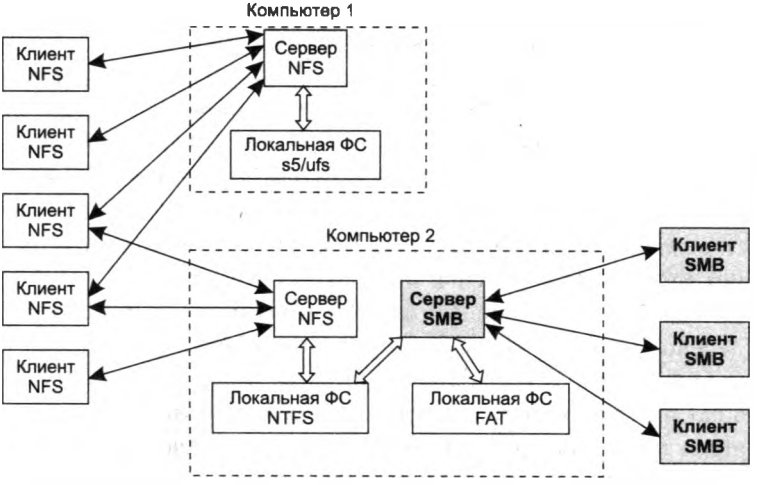


Рис. 11.4. Неоднородная сетевая файловая система

В то же время удаленный доступ к локальным ФС разного типа может реализовываться с помощью одного и того же протокола сетевой ФС. Например, протокол SMB используется для доступа не только к файловой системе FAT, но и к файловым системам NTFS и HPFS (рис. 11.3). Эти файловые системы могут располагаться как на разных, так и на одном компьютере.

На рис. 11.4 показан вариант организации неоднородной сетевой файловой системы, в которой на компьютере 2 установлены две локальные файловые системы — NTFS и FAT. С локальной файловой системой NTFS работает два файловых сервера — NFS и SMB. Оба сервера обеспечивают доступ своим клиентам к одним и тем же данным, хранящимся на локальном диске компьютера 2.

Модель загрузки-выгрузки и модель удаленного доступа

Сетевая файловая система может быть отнесена к одному из двух типов в зависимости от принятой в этой системе *единицы перемещения данных*. В первом случае, называемом **моделью загрузки-выгрузки**, пользователю предлагаются средства чтения или записи удаленного файла *целиком*. Эта модель предполагает следующую схему обработки файла: чтение файла с сервера на машину клиента, обработка файла на машине клиента и запись обновленного файла на сервер. Типичным представителем этого вида файлового интерфейса является служба FTP, пользователь которой должен применить команду `get file_name` для перемещения файла с сервера на клиентский компьютер и команду `put file_name` для возвращения файла на сервер. Таким образом, сетевой интерфейс ФС такого типа включает в себя только команды хранения и перемещения файла, а все операции с файлом выполняются на основе интерфейса локальной файловой системы.

Другой тип сетевой файловой системы соответствует **модели удаленного доступа**, которая предполагает поддержку большого количества операций с удаленными файлами: открытие и закрытие файлов, чтение и запись *частей* файла, позиционирование в файле, проверка и изменение атрибутов файла и т. д. В отличие от модели загрузки-выгрузки, все файловые операции в данном случае выполняются на серверах, а клиенты только генерируют запросы на их отработку. Преимуществом такого подхода являются низкие требования к дисковому пространству на клиентских машинах, а также исключение необходимости передачи целого файла, когда нужна только его часть. Модель удаленного доступа часто используется в прозрачных реализациях файлового интерфейса, когда удаленные файловые системы монтируются в общее с локальной системой дерево (или его часть, что происходит при отображении удаленной системы на букву логического диска). Модели удаленного доступа могут использовать различные наименьшие единицы перемещения части файла. Наиболее популярны такие единицы, как байт, блок или запись. Последний вид единицы может применяться только в том случае, когда локальная файловая система поддерживает структурированные файлы.

Архитектурные решения

Архитектурные решения программного комплекса, реализующего сетевую ФС, во многом определяют ее эффективность, производительность, отказоустойчивость и масштабируемость. Именно к вопросам архитектуры относятся решения о том, оформлять ли клиентские и серверные компоненты сетевой файловой системы в виде приложений или включать их в число модулей ОС, в каком режиме — пользовательском или привилегированном — должны выполняться эти программы, совмещать ли в единой программе функции клиента и сервера или это должны быть отдельные программы.

В некоторых файловых системах (например, NFS) *на всех компьютерах сети работает одно и то же базовое программное обеспечение*, включающее как клиентскую, так и серверную части, так что любой компьютер, который захочет предложить услуги файловой службы, может это сделать. Для этого администратору ОС достаточно объявить имена выбранных каталогов разделяемыми (экспортируемыми в терминах NFS), чтобы другие машины могли иметь к ним доступ.

В других системах *файловый сервер — это специализированный компонент серверной ОС*, отсутствующий в клиентских компьютерах. По такому пути пошли разработчики сетевой ОС NetWare, создав операционную систему, оптимизированную для работы в качестве файлового сервера, но не поддерживающую работу в качестве клиентской ОС.

Для повышения эффективности работы файловый сервер и клиент обычно включают в число модулей ядра ОС, работающих *в привилегированном режиме*. В современных ОС эти компоненты чаще всего оформляются как высокоуровневые драйверы, функционирующие в составе подсистемы ввода-вывода. Эффективность работы при этом повышается за счет прямого доступа ко всем внутренним модулям ОС, в том числе и к дисковому кэшу, без выполнения дополнительных операций и смены пользовательского режима на привилегированный. Прямой доступ к содержимому дискового кэша существенно повышает скорость доступа к данным файлов по сети, поэтому для повышения производительности файлового сервера ОС должна быть сконфигурирована для поддержки дискового кэша большого объема.

Файловый сервер и клиенты могут в некоторых случаях оформляться и как модули, работающие *в пользовательском режиме*. Такое построение было характерно для ранних сетевых файловых систем персональных компьютеров (например, IBM LAN Program), от которых не требовалась высокая скорость доступа, а объемы хранимых данных были весьма скромными. Используется такой режим работы и в файловых серверах ОС, основанных на микроядерной архитектуре, например, ОС, построенных на основе микроядра Mach. Такие файловые серверы предназначены для выполнения самой ответственной работы (в отличие от серверов ранних систем для персональных компьютеров), и перенесение сервера в пользовательский режим обусловлено общим подходом к архитектуре ОС, преследующим, как это уже было отмечено в главе 3, такие цели, как повышение надежности и улучшение возможностей модификации ОС.

Однако работа файлового сервера в пользовательском режиме снижает его производительность, из-за чего на практике такая архитектура применяется пока редко.

Производительность, надежность и безопасность сетевой файловой системы

Задачей сетевой файловой службы является предоставление всем сетевым пользователям надежного, производительного и безопасного доступа к разделяемым файлам.

Производительность сетевой файловой системы характеризуется скоростью доступа к файлам и временем выполнения требуемых операций с ними. Этот показатель тесно связан с числом пользователей, которые работают в системе: чем их больше, тем вероятнее возникновение больших задержек обслуживания из-за заторов в сети и перегрузки файлового сервера.

Для решения этой проблемы может выполняться **кэширование файлов** целиком или частично на стороне клиента. При этом необходимо учитывать то обстоятельство, что в сети может одновременно появиться большое количество копий одного и того же файла, которые независимо могут модифицироваться разными пользователями. То есть протокол сетевой ФС должен каким-то образом обеспечивать согласованность копий файлов, имеющихся на разных компьютерах. При применении модели удаленного доступа для повышения производительности может быть использована также **буферизация команд**, задающих операции с файлом. Команды группируются и передаются для выполнения на сервер в виде одного пакета, тем самым снижается сетевой трафик, а значит, повышается производительность сети. Производительность также может быть повышена за счет репликации файлов, о чем рассказывается далее.

Надежность сетевой файловой системы в той или иной степени зависит от следующих событий:

- отказы компьютеров, на которых работают клиентские программы,
- разрывы сетевых связей между клиентами и серверами;
- отказы компьютеров, на которых работают серверные программы.

Отказ клиентского компьютера не ведет к полному отказу файловой службы, а лишь исключает этот компьютер из зоны предоставления услуг. Другими словами, происходит деградация системы.

Последствия, к которым может привести разрыв связи, могут быть разными в зависимости от топологии сети и локализации повреждения. Степень деградации может варьироваться в широких пределах. Это может быть один или несколько «потерянных» клиентов, в случае нарушения линий связи присоединяющих группу клиентских компьютеров.

Однако возможен и полный отказ системы, если при отсутствии резервирования произошел разрыв связи файлового сервера с остальной частью сети. К таким же последствиям приводят отказы компьютеров, выступающих в роли файловых серверов. Чтобы избежать потери данных и разрушения целостности

файловой системы при сбоях и отказах серверов, а также при разрыве критически важных линий связи, прибегают к резервированию.

Частным случаем резервирования является **репликация** — технология поддержания нескольких копий данных. Для каждого файла (или целиком для локальной файловой системы) в сети поддерживается, по крайней мере, две копии, причем каждая копия хранится на отдельном компьютере. Такие копии файлов называются **репликами** (replica). Протокол сетевого доступа к файлам должен учитывать такую организацию файловой службы, например, обращаясь в случае отказа одного файлового сервера к другому, работоспособному и поддерживающему реплику требуемого файла.

Репликация файлов не только повышает отказоустойчивость, но и решает проблему перегрузки файловых серверов, так как запросы к файлам распределяются между несколькими серверами и повышают производительность сетевой файловой системы. Репликация в некоторых аспектах похожа на кэширование — в том и другом случаях в сети создается несколько копий одного и того же файла, при этом повышается скорость доступа к данным. Одним из отличий репликации от кэширования является то, что реплики хранятся на файловых серверах, а кэшированные файлы — на клиентах.

Рассмотрим частный случай отказа файлового сервера, когда этот отказ происходит *во время сеанса связи с клиентом*. Ранее уже отмечалось (см. раздел «Открытие файла» в главе 7), что локальная файловая система запоминает состояние последовательных операций, которые приложение выполняет с одним и тем же файлом, путем ведения внутренней системной таблицы открытых файлов (системные вызовы `open`, `read`, `write` изменяют состояние этой таблицы). Если таблица открытых файлов хранится на серверном компьютере, то после его перезагрузки, вызванной крахом системы, содержимое этой таблицы теряется, так что приложение, работающее на клиентском компьютере, не может продолжить нормальную работу с открытыми до краха файлами. Протокол должен позволять приложениям выйти из такой ситуации с наименьшими потерями. Одно из решений основано на передаче функции ведения и хранения таблицы открытых файлов от сервера клиенту. Файловый сервер в этом варианте получил название **stateless**, то есть «не запоминающий состояния». Протокол клиент-сервер при такой организации упрощается, так как перезагрузка сервера приводит только к паузе в обслуживании, но работа с файлами может быть после этого продолжена безболезненно для клиента. Очевидно, что сетевая файловая система принципиально является менее надежной, чем локальная. Также обстоит дело и с защитой данных.

Передача данных по сети делает обеспечение всех аспектов *безопасности*: аутентификации, авторизации, секретности и пр. — более сложным, чем в случае автономно работающего компьютера.

Например, при обращении пользователя к локальным файлам проверка его прав доступа (то есть авторизация) происходит на том же компьютере, на котором была выполнена проверка его легальности (то есть аутентификация). А при работе в сети возможна ситуация, когда аутентификация выполняется на

одном компьютере, а авторизация на другом. Пространственное разнесение этих двух процедур на клиентский и серверный компьютеры создает дополнительные уязвимые этапы в общем процессе обработки данных, которые должны каким-то образом учитываться протоколом взаимодействия клиентов и серверов файловой службы. Мы еще вернемся к обстоятельному обсуждению этих вопросов в следующей главе, а сейчас коротко рассмотрим варианты возможного взаимодействия сетевой и локальной файловых систем при реализации контроля доступа к удаленным файлам и каталогам.

Во многих ФС с каждым разделяемым файлом связывается **список управления доступом** (Access Control List, ACL), обеспечивающий защиту данных от несанкционированного доступа по сети. В том случае, когда локальная файловая система поддерживает механизм ACL для файлов и каталогов при локальном доступе, сетевая файловая система использует этот механизм и при доступе по сети. Если же механизм защиты в локальной файловой системе отсутствует, то сетевой файловой системе приходится поддерживать его самостоятельно, иногда — упрощенным способом, защищая разделяемый каталог и входящие в него файлы и подкаталоги как единое целое.

Рассмотрим, например, как решила задачу контроля сетевого доступа к файлам *компания Novell* в ОС NetWare. Как уже было сказано, компания Novell вынуждена была разработать собственную версию локальной файловой системы, так как наиболее популярная файловая система для персональных компьютеров тех лет — FAT — не хранила в своих служебных структурах данных о правах пользователей. В то же время протокол взаимодействия с локальной файловой системой NetWare, названный NCP, сохранил привычный для приложений персональных компьютеров интерфейс доступа к FAT, расширив его передачей атрибутов, необходимых для удаленной проверки прав доступа. Протокол NCP является хорошим примером зависимости между свойствами интерфейса, предоставляемого приложениям на клиентских машинах, и свойствами локальной ФС сервера.

Совершенно другой прием для разграничения прав доступа для удаленных пользователей был применен в файловых системах *компаний Microsoft и IBM*, построенных на основе протокола SMB. В качестве локальной ФС на сервере была оставлена неизмененная файловая система FAT, но сами серверы стали хранить в ней дополнительные служебные файлы с указанием прав пользователей на доступ к разделяемым каталогам. Эти права проверялись сервером при поступлении запроса из сети, локальные же запросы обслуживались в FAT по-прежнему без проверки прав доступа. Естественно, средства защиты каталогов нашли отражение в командах и ответах протокола SMB, а в качестве сетевого интерфейса на стороне клиентов стал использоваться расширенный интерфейс FAT. Позже протокол SMB был применен и для доступа к локальным файловым системам HPFS и NTFS.

В ОС семейства Windows NT существует два механизма защиты — на уровне разделяемых каталогов и на уровне локальных каталогов и файлов. Последний работает только в том случае, когда в качестве локальной файловой системы используется система NTFS, поддерживающая механизм ACL. Механизм защиты

разделяемых каталогов нужен для того, чтобы защищать данные, хранящиеся в локальной файловой системе FAT, не имеющей механизмов защиты. В том случае, когда работают оба уровня защиты, у пользователей и администраторов могут иногда возникать некоторые логические сложности, связанные с определением реальных прав доступа как комбинации нескольких правил.

В современных ОС контроль доступа ко всем типам разделяемых ресурсов — файлам, каталогам, принтерам, секциям памяти и др. — осуществляется с единых позиций. Основную роль в решении этой задачи играет сетевая справочная служба, которая рассматривается в этой главе далее.

Семантика разделения файлов

Организация сетевой файловой системы во многом определяется принятой в этой системе *семантикой разделения файлов*. Когда два или более пользователей совместно работают с одним и тем же файлом, необходимо совершенно точно определить, что понимается под операциями чтения, записи и сохранения файла, иначе могут возникнуть проблемы с интерпретацией результирующих данных файла. Например, если два пользователя одновременно работают с файлом и один из пользователей сделал несколько записей в файл, то должны ли они немедленно появиться на экране компьютера другого пользователя? Или они должны накапливаться и вноситься в файл только во время его закрытия? А не будут ли при этом потеряны изменения, сделанные в тот же период другим пользователем? Такого рода вопросы допускают несколько вариантов ответа, соответствующих разным семантикам.

- *Семантика Unix.* В централизованных многопользовательских операционных системах, разрешающих разделение файлов, таких как Unix (имеется в виду локальная версия этой ОС), обычно полагается, что когда операция чтения следует за операцией записи, читается уже обновленный файл. Соответственно, когда операция чтения следует за двумя операциями записи, то читается файл, измененный последней операцией записи. Тем самым система придерживается абсолютного временного упорядочивания всех операций и всегда возвращает самое последнее значение данных. Если запись осуществляется в файл открытый несколькими пользователями, то все пользователи немедленно видят результат изменения данных файла. Обычно такая модель называется семантикой Unix. В централизованной системе, где файлы хранятся в единственном экземпляре, ее легко и понять, и реализовать.

  Семантика Unix может поддерживаться и в распределенных системах, но только, если в ней имеется лишь один файловый сервер и клиенты не кэшируют файлы. Для этого все операции чтения и записи направляются на файловый сервер, который обрабатывает их строго последовательно. На практике, однако, производительность распределенной системы, в которой все запросы к файлам идут на один сервер, часто оказывается неудовлетворительной. Эта проблема иногда решается за счет разрешения клиентам обрабатывать локальные копии часто используемых файлов в своих личных кэшах. Если клиент сделает локальную копию файла в своем локальном кэше и начнет ее модифицировать, а вскоре после этого другой клиент прочитает этот файл с сервера, то он получит неверную копию файла. Одним из способов устра-

нения этого недостатка является немедленный возврат всех изменений в кэшированном файле на сервер. Такой подход хотя и концептуально прост, но не слишком эффективен. Распределенные файловые системы обычно используют более свободную семантику разделения файлов.

- *Сеансовая семантика.* В соответствии с этой моделью изменения в открытом файле сначала виды только процессу, который модифицирует файл, и только после закрытия файла эти изменения могут видеть другие процессы. В случае сеансовой семантики возникает проблема одновременного использования одного и того же файла двумя или более клиентами. Одним из решений этой проблемы является принятие правила, в соответствии с которым окончательным является тот вариант, который был закрыт последним. Однако из-за задержек в сети часто оказывается трудным определить, какая из копий файла была закрыта последней. Менее эффективным, но гораздо более простым в реализации является вариант, в котором окончательным результирующим файлом на сервере считается любой из этих файлов, то есть результат операций с файлом не детерминирован.
- *Семантика неизменяемых файлов.* Следующий подход к разделению файлов заключается в том, чтобы сделать все файлы неизменяемыми. Тогда файл нельзя открыть для записи, а можно выполнять только операции `create` (создать) и `read` (читать). В результате для изменения файла остается единственная возможность — создать полностью новый файл и поместить его в каталог под новым именем. Следовательно, хотя файл и нельзя модифицировать, его можно заменить (автоматически) новым файлом. Для неизменяемых файлов намного проще осуществить кэширование файла и его репликацию (тиражирование), так как исключаются проблемы, связанные с обновлением всех копий файла при его изменении.

  Таким образом, для файловых систем, работающих с немодифицируемыми файлами, исключаются все проблемы, связанные с одновременным доступом к файлам нескольких пользователей. Однако решение этих проблем перекладывается на плечи пользователей, которые в такой системе вынуждены вести учет многочисленных версий файла, возникающих при его модификации. Пользователи должны поддерживать систему именования файлов, отражающую тот факт, что все множество порожденных файлов имеет близкое содержание, и в то же время позволяющую различать версии файлов.
- *Транзакционная семантика.* Четвертый способ работы с разделяемыми файлами в распределенных системах — это использование механизма неделимых транзакций, описание которого можно найти в главе 8.

Файловые stateful- и stateless-серверы

Сетевой файловый сервер может быть реализован по одной из двух схем:

- с запоминанием данных о последовательности файловых операций клиента, то есть по схеме **stateful**;
- без запоминания таких данных, то есть по схеме **stateless**.

Схема stateful — это обычная для любой локальной файловой службы схема. Stateful-сервер поддерживает тот же набор вызовов, что и локальная система, то есть вызовы `open`, `read`, `write`, `seek` и `close`, рассмотренные в главе 7. Открывая файлы по вызову `open`, переданному по сети клиентом, stateful-сервер в своей внутренней системной таблице должен запоминать, какие файлы открыл каждый пользователь (рис. 11.5). Обычно при открытии файла клиентскому приложению возвращается по сети дескриптор файла `fd` или другое число, которое используется при последующих вызовах для идентификации файла. При поступлении вызова `read`, `write` или `seek` сервер с помощью дескриптора файла определяет, над каким файлом требуется выполнить требуемое действие. В этой таблице хранится также значение указателя на текущую позицию в файле, относительно которой выполняется операция чтения или записи. Таким образом, таблица открытых файлов является хранилищем информации о состоянии клиентов.

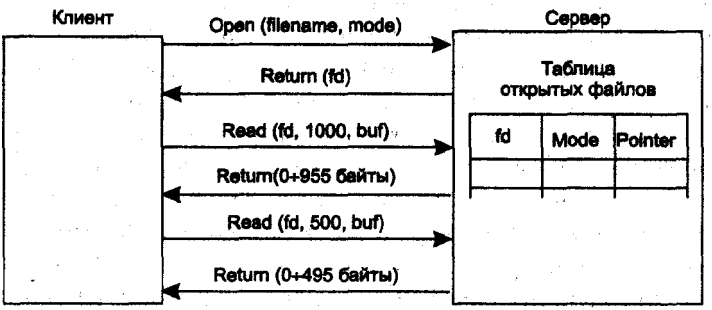


Рис. 11.5. Сервер с сохранением информации о состоянии (stateful)

Файловый сервер, работающий по схеме stateless, можно назвать «устройством без памяти»: когда клиент посылает запрос на сервер, сервер его выполняет, отсылает ответ, а затем удаляет из своих внутренних таблиц всю информацию о запросе. Между запросами на сервере не хранится никакой текущей информации о состоянии клиента. Для stateless-сервера каждый запрос должен содержать исчерпывающую информацию (полное имя файла, смещение в файле и т. п.), необходимую серверу для выполнения требуемой операции. Очевидно, что эта информация увеличивает длину сообщения и время, которое тратит сервер на локальное открытие файла каждый раз, когда над ним производится очередная операция чтения или записи. Серверы, работающее по схеме stateless, не поддерживают в протоколе обмена с сетевыми клиентами таких операций как открытие (`open`) и закрытие (`close`) файлов. Принципиально набор команд, которые stateless-сервер способен предоставлять клиенту, может состоять только из двух команд: `read` и `write` (рис. 11.6). Эти команды должны передавать в своих аргументах всю необходимую для сервера информацию — имя файла, смещение от начала файла и количество читаемых или записываемых байтов данных.

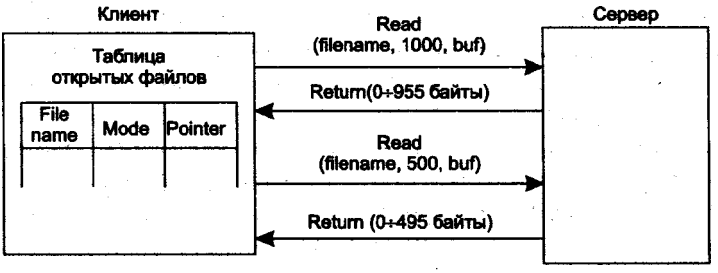


Рис. 11.6. Сервер без сохранения информации о состоянии (stateless)

Для того чтобы предоставить приложениям, работающим на клиентских машинах, привычный файловый интерфейс, включающий вызовы открытия и закрытия файлов, клиент файловой службы должен самостоятельно поддерживать таблицы открытых его приложениями файлов.

Основным последствием размещения таблиц открытых файлов на клиентах или на сервере является *реакция файловой службы на отказ сервера*. При отказе stateful-сервера теряются все его таблицы, и после перезагрузки неизвестно, какие файлы открыты каждым из пользбвателей. Последовательные попытки провести операции чтения или записи с открытыми файлами будут безуспешными. Stateless-серверы в этом плане являются более отказоустойчивыми, позволяя клиентам продолжать операции с открытыми файлами, и это является основным аргументом в пользу их применения.

Платой за отказоустойчивость stateless-серверов является снижение скорости их работы, так как при любой операции с файлом им приходится выполнять больше действий, чем stateful-серверам. Кроме того, применение stateless-серверов затрудняет реализацию блокировок файлов, так как информацию о блокировке файла одним из пользователей необходимо запоминать на всех клиентах файлового сервера.

Преимущества каждого из подходов сведены в табл. 11.1.

Таблица 11.1. Сравнительные характеристики stateful- и stateless-серверов

Stateless-серверы	Stateful-серверы
Отказоустойчивы	Более короткие сообщения при запросах
Не нужны вызовы open и close	Выше производительность
Меньше памяти сервера расходуется на таблицы	Возможно опережающее чтение
Нет ограничений на число открытых файлов	Возможна блокировка файлов
Отказ клиента не создает проблем для сервера	

Кэширование

Кэширование данных в оперативной памяти, как было показано в главе 8, может существенно повысить скорость доступа к файлам, хранящимся на дисках, независимо от того, является файловая система локальной или сетевой. В сетевых файловых системах кэширование позволяет не только повысить скорость доступа к удаленным данным (это по-прежнему является основной целью кэширования), но и улучшить масштабируемость и повысить надежность файловой системы.

Схемы кэширования, применяемые в сетевых файловых системах, отличаются решениями по трем ключевым вопросам:

- месту расположения кэша;
- способу распространения модификаций;
- проверке достоверности кэша.

Кроме того, на схему кэширования влияет выбранная в файловой системе модель переноса файлов между сервером и клиентами: это может быть модель загрузки-выгрузки, в которой файл переносится целиком, или модель удаленного доступа, позволяющая переносить файл по частям. Соответственно, в первом случае файл кэшируется целиком, а во втором кэшируются только те части файла, к которым выполняется обращение.

Место расположения кэша

В системах, состоящих из клиентов и серверов, потенциально имеются три различных места для хранения кэшируемых файлов и их частей:

- память сервера;
- диск клиента (если имеется);
- память клиента.

Память сервера практически всегда применяется для кэширования файлов, к которым обращаются по сети пользователи и приложения. *Кэширование в памяти сервера* сокращает время доступа по сети за счет исключения времени обмена с диском сервера. При этом файловый сервер может использовать для кэширования своих файлов существующий механизм локального кэша операционной системы, не применяя никакого дополнительного программного кода. Однако кэширование только в памяти сервера не решает всех проблем — скорость доступа по-прежнему снижают задержки, вносимые сетью, а также перегруженность процессора сервера при одновременном обслуживании большого потока сетевых запросов от многочисленных клиентов.

Кэширование на стороне клиента, которое подразделяется на кэширование на диске клиента и в памяти клиента, исключает обмен по сети и освобождает сервер от работы после переноса файла на клиентский компьютер. *Использование диска* как места временного хранения данных на стороне клиента позволяет

кэшировать большие файлы, что особенно важно при применении модели загрузки-выгрузки, в соответствие с которой файлы переносятся целиком. Диск также является более надежным устройством хранения информации по сравнению с оперативной памятью. Однако такой способ кэширования вносит дополнительные задержки доступа, связанные с чтением данных с клиентского диска, кроме того, он неприменим на бездисковых компьютерах.

Кэширование в оперативной памяти клиента позволяет ускорить доступ к данным, но ограничивает размер кэшируемых данных объемом памяти, выделяемой под кэш, что может стать существенным ограничением при применении модели загрузки-выгрузки.

Способы распространения модификаций

Существование в одно и то же время в сети нескольких копий одного и того же файла, хранящихся в кэшах клиентов, порождает *проблему согласования копий*, которая состоит в том, что модификации данных одной из копий должны быть своевременно распространены на все остальные копии. Существует несколько вариантов распространения модификаций. От выбранного варианта в значительной степени зависит семантика разделения файлов.

Одним из путей решения проблемы согласования копий является использование **алгоритма сквозной записи**. Когда кэшируемый элемент (файл или блок) модифицируется, новое значение записывается в кэш и одновременно посылается на сервер для обновления главной копии файла. В результате другой процесс, читающий этот файл, получает самую последнюю версию данных. Данный вариант распространения модификаций обеспечивает семантику разделения файлов в стиле Unix.

Один из недостатков алгоритма сквозной записи состоит в том, что он уменьшает интенсивность сетевого обмена *только при чтении*, при записи интенсивность сетевого обмена та же самая, что и без кэширования. Многие разработчики систем находят это неприемлемым и предлагают алгоритм отложенной записи, основанный на буферизации изменений: вместо того чтобы выполнять запись на сервер, клиент просто делает пометку, что файл изменен. Примерно каждые 30 секунд все изменения в файлах собираются вместе и отсылаются на сервер за один прием. Одна длинная запись для сетевого обмена обычно более эффективна, чем множество коротких.

Следующим шагом в этом направлении является принятие сеансовой семантики, в соответствии с которой запись файла на сервер производится только после закрытия файла. Этот алгоритм, называемый **записью по закрытию**, приводит к тому, что если две копии одного файла кэшируются на разных машинах и последовательно записываются на сервер, то второй записывается поверх первого. Данная схема не может снизить сетевой трафик, если объем изменений за сеанс редактирования файла невелик.

Для всех схем, связанных с задержкой записи, характерна низкая надежность, так как все модификации, не отправленные на сервер на момент краха системы, теряются. Кроме того, задержка делает семантику совместного ис-

пользования файлов не очень ясной, поскольку данные, которые считывает процесс из файла, зависят от соотношения момента чтения с моментом очередной записи модификаций.

Проверка достоверности кэша

Распространение модификаций решает только проблему согласования с клиентскими копиями главной копии файла, хранящейся на сервере. В то же время этот прием не дает никакой информации о том, когда должны обновляться данные, находящиеся в кэшах клиентов. Очевидно, что данные в кэше одного клиента становятся недостоверными, когда данные, модифицированные другим клиентом, переносятся в главную копию файла. Следовательно, необходимо каким-то образом проверять, являются ли данные в кэше клиента достоверными. В противном случае данные кэша должны быть повторно считаны с сервера. Существует два подхода к решению этой проблемы — инициирование проверки клиентом и инициирование проверки сервером.

В первом случае, когда *клиент является инициатором проверки*, он связывается с сервером и проверяет, соответствуют ли данные в его кэше данным главной копии файла на сервере. Клиент может выполнять такую проверку одним из трех способов.

- Перед каждым доступом к файлу. Это способ дискредитирует саму идею кэширования, так как каждое обращение к файлу вызывает обмен по сети с сервером, зато он обеспечивает семантику разделения файлов Unix.
- Периодические проверки повышают производительность, но делают семантику разделения неясной, зависящей от временных соотношений.
- Проверка при открытии файла подходит для сеансовой семантики. Отметим, что сеансовая семантика требует одновременного выполнения трех условий: во-первых, файловая система должна соответствовать модели загрузки-выгрузки, во-вторых, модификации должны распространяться в соответствии с алгоритмом записи по закрытию, в-третьих, проверка достоверности кэша должна происходить при открытии файла.

Совершенно другой подход к проблеме проверки достоверности кэша реализуется при *инициировании проверки сервером*, который также можно назвать методом централизованного управления. Когда файл открывается, то клиент, выполняющий это действие, посылает соответствующее сообщение файловому серверу, в котором указывает режим открытия файла — чтение или запись. Файловый сервер сохраняет информацию о том, кто и какой файл открыл, а также о том, открыт файл для чтения, для записи или для того и другого. Если файл открыт для чтения, то нет никаких препятствий для разрешения другим процессам открыть его для чтения, но открытие его для записи должно быть запрещено. Аналогично, если некоторый процесс открыл файл для записи, то все другие виды доступа должны быть запрещены. При закрытии файла также необходимо оповестить файловый сервер для того, чтобы он обновил свои таблицы, содержащие данные об открытых файлах. Модифицированный файл также может быть выгружен на сервер в такой момент.

Когда новый клиент делает запрос на открытие уже открытого файла и сервер обнаруживает, что режим нового открытия входит в противоречие с режимом текущего открытия, то сервер может ответить на такой запрос следующими способами:

- отвергнуть запрос;
- поместить запрос в очередь;
- запретить кэширование для данного конкретного файла, потребовав от всех клиентов, открывших этот файл, удалить его из кэша.

Подход, основанный на централизованном управлении, весьма эффективен, обеспечивает семантику разделения Unix, но обладает несколькими недостатками.

- Он отклоняется от традиционной модели взаимодействия клиента и сервера, в которой сервер только отвечает на запросы, инициированные клиентами. Это делает код сервера нестандартным и достаточно сложным.
- Сервер обязательно должен хранить информацию о состоянии сеансов клиентов, то есть работать по схеме stateful.
- По-прежнему должен использоваться механизм инициирования проверки достоверности клиентами при открытии файлов.

Репликация файлов

Сетевая файловая система может поддерживать репликацию (тиражирование) файлов в качестве одной из услуг, предоставляемых клиентам. Иногда репликацией занимается отдельная служба ОС.

> **Репликация** (replication) — это метод поддержания нескольких копий одного и того же файла, каждая из которых хранится на отдельном файловом сервере, при этом обеспечивается автоматическое согласование данных в копиях файла.

Имеется несколько причин для применения репликации, главными из которых являются две.

- *Повышение надежности* за счет наличия независимых копий каждого файла, хранящихся на разных файловых серверах. При отказе одного из них файл остается доступным.
- *Повышение производительности* за счет распределения нагрузки между несколькими серверами. Клиенты могут обращаться к данным реплицированного файла на ближайший файловый сервер, хранящий копию этого файла. Ближайшим может считаться сервер, находящийся в той же подсети, что и клиент, или же первый откликнувшийся, или же выбранный некоторым сетевым устройством, балансирующим нагрузки серверов, например коммутатором 4-го уровня.

Репликация похожа на кэширование файлов на стороне клиентов тем, что в системе создается несколько копий одного файла. Однако существуют и принципи-

альные отличия репликации от кэширования, прежде всего, в преследуемых целях. Если кэширование предназначено для обеспечения локального доступа к файлу одному клиенту и повышения за счет этого скорости работы этого клиента, то репликация нужна для повышения надежности хранения файлов и снижения нагрузки на файловые серверы. Реплики файла доступны всем клиентам, так как хранятся на файловых серверах, а не на клиентских компьютерах, и о существовании реплик известно всем компьютерам сети. Файловая система обеспечивает достоверность данных реплики и защиту ее данных.

Прозрачность репликации

Ключевым вопросом, связанным с репликацией, является прозрачность. До какой степени пользователи должны быть в курсе того, что некоторые файлы реплицируются? Должны они играть какую-либо роль в процессе репликации, или репликация должна выполняться полностью автоматически? В одних системах пользователи полностью вовлечены в этот процесс, в других система все делается без их ведома. В последнем случае говорят, что система репликационно прозрачна.

Прозрачность репликации зависит от двух факторов: используемой схемы именования реплик и степени вовлеченности пользователя в управление репликацией.

Именование реплик. Прозрачность доступа к файлу, существующему в виде нескольких реплик, может поддерживаться системой именования, которая отображает имя файла на его сетевой идентификатор, однозначно определяющий место хранения файла. Если в сети используется справочная служба (см. далее раздел «Справочная сетевая служба»), которая позволяет хранить отображения имен объектов на их сетевые идентификаторы (например, IP-адреса серверов), то для реплицированных файлов допустима прозрачная схема именования. В этом случае файлу присваивается имя, не содержащее старшей части, соответствующей имени компьютера. В справочной службе этому имени соответствует несколько идентификаторов, указывающих на серверы, хранящие реплики файла. При обращении к такому файлу приложение использует имя, а справочная служба возвращает ему один из идентификаторов, указывающий на сервер, хранящий реплику.

Наиболее просто такую схему реализовать для неизменяемых файлов, все реплики которых всегда (или почти всегда, если файлы редко, но все же изменяются) идентичны. В случае когда реплицируются изменяемые файлы, полностью прозрачный доступ требует ведения некоторой базы, хранящей сведения о том, какие из реплик содержат последнюю версию данных, а какие еще не обновлены. Для поддержания полностью прозрачной системы репликации необходимо также постоянное использование в файловом интерфейсе имен, не зависящих от местоположения, то есть не содержащих старшей части с указанием имени сервера. В более распространенной на сегодня схеме именования требуется явное указание имени сервера при обращении к файлу, что приводит к не полностью прозрачной системе репликации.

Управление репликацией подразумевает определение количества реплик и выбор серверов для хранения каждой реплики. В прозрачной системе репликации такие решения принимаются автоматически при создании файла на основе правил стратегии репликации, определенных заранее администратором системы. В непрозрачной системе репликации решения о количестве реплик и их размещении принимаются с участием пользователя, который создает файлы, или разработчика приложения, если файлы создаются в автоматическом режиме. В результате существует два режима управления репликацией.

- При *явной репликации* (explicit replication) пользователю (или прикладному программисту) предоставляется возможность управления процессом репликации. При создании файла сначала создается первая реплика с явным указанием сервера, на котором размещается файл, а затем создается несколько реплик, причем для каждой реплики сервер также указывается явно. Пользователь при желании может впоследствии удалить одну или несколько реплик. Явная репликация не исключает автоматического режима поддержания согласованности реплик, которые создал и разместил на серверах пользователь.
- При *неявной репликации* (implicit replication) выбор количества и места размещения реплик производится в автоматическом режиме без участия пользователя. Приложение не должно указывать место размещения файла при запросе на его создание. Файловая система самостоятельно выбирает сервер, на который помещает первую реплику файла. Затем в фоновом режиме система создает несколько реплик этого файла, выбирая их количество и серверы для их размещения. Если надобность в некоторых репликах исчезает (это также определяется автоматически), то система их удаляет.

Согласование реплик

Вопросы согласования реплик, в результате которого в каждой реплике сохраняется последняя версия данных файла, являются одними из наиболее важных при разработке системы репликации. Так как изменяемые файлы являются самым распространенным типом файлов, то в какой-то момент времени данные в одной из реплик модифицируются, после чего требуется предпринять усилия для распространения модификаций на все остальные реплики. Существует несколько способов обеспечения согласованности реплик.

Чтение любого — запись во все (Read-Any — Write-All). При необходимости записи в файл все реплики файла блокируются, затем выполняется запись в каждую копию, после чего блокировка снимается и файл становится доступным для чтения. Чтение может выполняться из любой копии. Этот способ обеспечивает семантику разделения файлов в стиле Unix. Недостатком является то, что запись не всегда можно осуществить, так как некоторые серверы, хранящие реплики файла, могут в момент записи быть неработоспособными.

Запись в доступные (Available-Copies). Этот метод снимает ограничение предыдущего, так как запись выполняется только в те копии, серверы которых доступны на момент записи. Чтение осуществляется из любой реплики файла, как

и в предыдущем методе. Любой сервер, хранящий реплику файла, после перезагрузки должен соединиться с другим сервером и получить от него обновленную копию файла и только потом начать обслуживать запросы на чтение из файла. Для обнаружения отказавших серверов в системе должен работать специальный процесс, постоянно опрашивающий состояние серверов. Недостатком метода является возможность появления несогласованных копий файла из-за коммуникационных проблем в сети, когда невозможно выявить отказавший сервер.

Первичная реплика (Primary-Copy). В этом методе запись разрешается только в одну реплику файла, называемую первичной (primary). Все остальные реплики файла являются вторичными (secondary), из них можно только читать данные. После модификации первичной реплики все серверы, хранящие вторичные реплики, должны связаться с сервером, поддерживающим первичную реплику, и получить от него обновления. Этот процесс может инициироваться как первичным сервером, так и вторичными (периодически проверяющими состоянии первичной реплики). Это метод является одной из реализаций метода «чтение любой — запись во все», в которой процесс записи реализован иерархическим способом. Для аккумулирования нескольких модификаций и сокращении сетевого трафика распространение модификаций может быть выполнено с запаздыванием, но в этом случае возникает проблема согласованности копий. Недостатком метода является его низкая надежность — при отказе первичного сервера модификации файла становятся невозможными (для решения этой проблемы необходимо повысить статус некоторого вторичного сервера до первичного).

Кворум (Quorum). Этот метод обобщает подходы, реализованные в предыдущих методах. Пусть в сети существует n реплик некоторого файла. Алгоритм основан на том, что при модификации файла изменения записываются в w реплик, а при чтении файла клиент обязательно производит обращение к r репликам. Значения w и r выбираются так, что $w + r > n$. При чтении клиент должен иметь возможность сначала проверить версию каждой реплики, а затем выбрать старшую и читать данные уже из нее. Очевидно, что при модификации файла номер версии должен наращиваться, а если при записи в w реплик они имеют разные версии, то выбирается максимальное значение версии для наращивания и присваивания всем репликам.

При выполнении условия $w + r > n$ среди любых выбранных произвольным образом r реплик всегда найдется хотя бы одна из w реплик, в которую были записаны последние обновления. Так, если реплик восемь ($n = 8$), можно выбрать в качестве r значение 4, а в качестве w — значение 5 (рис. 11.7). Условие $w + r > n$ при этом удовлетворяется.

Предположим, что запись последних изменений была выполнена в реплики с номерами 3, 4, 5, 6, 8. Если при чтении выбраны реплики 1, 2, 3, 7, то реплика 3 окажется общей для операций записи и чтения, поэтому прочитаны будут корректные данные (рис. 11.7, *а*). В другом примере, который иллюстрирует

рис. 11.7, *б*, значение r выбрано равным 2, а w — 7. Результат также получен корректный.

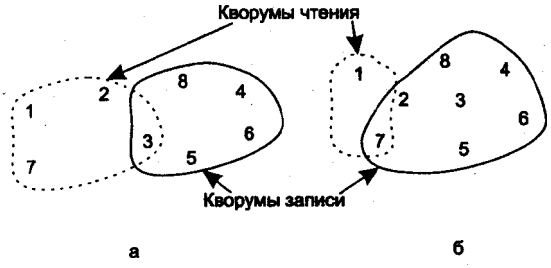


Рис. 11.7. Примеры работы метода кворума: а) $n = 8$; $r = 4$; $w = 7$; б) $n = 8$; $r = 2$; $w = 7$

У метода кворума имеются частные случаи. Так, если $r = 1$, а $w = n$, то получаем метод «чтение любого — запись во все».

Еще одним немаловажным параметром процесса согласования копий является величина временной задержки между внесением изменений в одной из реплик и отражением этих модификаций на все остальные реплики. Существует два подхода к распространению модификаций — **строгая целостность** (tight consistency) и **нестрогая целостность** (loose consistency). В первом случае идентичность реплик гарантируется в любой момент времени. Такой результат достигается, например, при использовании механизма неделимых транзакций. При этом сеть должна обладать высокими производительностью и надежностью, чтобы обеспечить постоянную доступность всех узлов. Во втором случае между внесением изменений и их отображением в остальных репликах допускается наличие временной задержки. Этот вариант более приемлем для современных файловых систем. Обычно допустимые временные задержки здесь измеряются значениями от нескольких секунд до нескольких минут.

Пример. Протокол передачи файлов FTP

Сетевая файловая служба на основе протокола **FTP** (File Transfer Protocol) представляет собой одну из наиболее ранних служб, используемых для доступа к удаленным файлам. До появления службы WWW это была самая популярная служба доступа к удаленным данным в Интернете и корпоративных IP-сетях. Первые спецификации FTP относятся к 1971 году. FTP-серверы и FTP-клиенты имеются практически в каждой ОС семейства Unix, а также во многих других сетевых ОС. FTP-клиенты встроены сегодня в программы просмотра (браузеры) Интернета, так как архивы файлов на основе протокола FTP по-прежнему популярны, и для доступа к таким архивам браузером используется протокол FTP.

Протокол FTP позволяет целиком переместить файл с удаленного компьютера на локальный, и наоборот, то есть работает по схеме загрузки-выгрузки. Кроме того, он поддерживает несколько команд просмотра удаленного каталога и перемещения по каталогам удаленной файловой системы. Поэтому FTP осо-

бенно удобно использовать для доступа к тем файлам, данные которых нет смысла просматривать удаленно, а гораздо эффективней целиком переместить на клиентский компьютер (например, файлы исполняемых модулей приложений).

В протокол FTP встроены примитивные средства аутентификации удаленных пользователей на основе передачи по сети пароля в открытом виде. Кроме того, поддерживается анонимный доступ, не требующий указания имени пользователя и пароля, который является более безопасным, так как не подвергает пароли пользователей угрозе перехвата.

Протокол FTP выполнен по схеме клиент-сервер. FTP-клиент состоит из нескольких функциональных модулей.

- User Interface — пользовательский интерфейс, принимающий от пользователя символьные команды и отображающий состояние FTP-сеанса на символьном экране.
- User-PI — интерпретатор команд пользователя. Этот модуль взаимодействует с соответствующим модулем FTP-сервера.
- User-DTP — модуль, осуществляющий передачу данных файла по командам, получаемым от модуля User-PI по протоколу клиент-сервер. Этот модуль взаимодействует с локальной файловой системой клиента.

FTP-сервер включает следующие модули.

- Server-PI — модуль, который принимает и интерпретирует команды, передаваемые по сети модулем User-PI.
- Server-DTP — модуль, управляющий передачей данных файла по командам от модуля Server-PI. Взаимодействует с локальной файловой системой сервера.

FTP-клиент и FTP-сервер поддерживают параллельно два сеанса — управляющий сеанс и сеанс передачи данных. Управляющий сеанс открывается при установлении первоначального FTP-соединения клиента с сервером, причем в течение одного управляющего сеанса может последовательно выполняться несколько сеансов передачи данных, в рамках которых передаются или принимаются несколько файлов.

Общая схема взаимодействия клиента и сервера выглядит следующим образом.

1. FTP-сервер всегда открывает управляющий TCP-порт 21 для прослушивания, ожидая приход запроса на установление управляющего FTP-сеанса от удаленного клиента.
2. После установления управляющего соединения клиент отправляет на сервер команды, которые уточняют параметры соединения:
   - имя и пароль клиента;
   - роль участников соединения (активная или пассивная);
   - порт передачи данных;

- тип передачи;
- тип передаваемых данных (двоичные данные или ASCII-код);
- директивы на выполнение действий (читать файл, писать файл, удалить файл и т. п.).

3. После согласования параметров пассивный участник соединения переходит в режим ожидания открытия соединения на порт передачи данных. Активный участник инициирует это соединение и начинает передачу данных.
4. После окончания передачи данных соединение по портам данных закрывается, а управляющее соединение остается открытым. Пользователь может по управляющему соединению активизировать новый сеанс передачи данных.

Порты передачи данных выбирает FTP-клиент (по умолчанию клиент может использовать для передачи данных порт управляющего сеанса), а сервер должен использовать порт, на единицу меньший порта клиента.

В протоколе FTP при взаимодействии клиента с сервером применяются несколько команд (не следует их путать с командами пользовательского интерфейса клиента, которые использует человек).

Эти команды делятся на три группы:

- команды управления доступом к системе;
- команды управления потоком данных;
- команды службы FTP.

В набор команд управления доступом входят следующие команды.

- `USER` — доставляет серверу имя клиента. Эта команда открывает управляющий сеанс и может также передаваться при открытом управляющем сеансе для смены имени пользователя.
- `PASS` — передает в открытом виде пароль пользователя.
- `CWD` — изменяет текущий каталог на сервере.
- `REIN` — повторно инициализирует управляющий сеанс.
- `QUIT` — завершает управляющий сеанс.

Команды управления потоком устанавливают параметры передачи данных.

- `PORT` — определяет адрес и порт хоста, который будет активным участником соединения при передаче данных. Например, команда `PORT 194,85,135,126,7,205` назначает активным участником хост 194.85.135.126 и порт 1997 (вычисление номера порта нетривиально, но вполне однозначно).
- `PASV` — назначает хост пассивным участником соединения по передаче данных. В ответ на эту команду должна быть передана команда `PORT` с указанием адреса и порта, находящегося в режиме ожидания.
- `TYPE` — задает тип передаваемых данных (ASCII-код или двоичные данные).
- `STRU` — определяет структуру передаваемых данных (файл, запись, страница).
- `MODE` — задает режим передачи (потоком, блоками и т. п.).

Как видно из описания, служба FTP может применяться для работы как со структурированными файлами, разделенными на записи или страницы, так и с неструктурированными.

Команды службы FTP инициируют действия по передаче файлов или просмотру удаленного каталога.

- `RETR` — запрашивает передачу файла от сервера на клиентский хост. Параметрами команды является имя файла. Может быть задано также смещение от начала файла — это позволяет начать передачу файла с определенного места при непредвиденном разрыве соединения (этот параметр указывается в команде `reget` пользовательского интерфейса).
- `STOR` — инициирует передачу файла от клиента на сервер. Параметры аналогичны команде `RETR`.
- `RNFR` и `RNTO` — команды переименования удаленного файла. Первая в качестве аргумента получает старое имя файла, а вторая — новое.
- `DELE`, `MKD`, `RMD`, `LIST` — эти команды соответственно удаляют файл, создают каталог, удаляют каталог и передают список файлов текущего каталога.

Каждая команда протокола FTP передается в текстовом виде по одной команде в строке. Строка заканчивается символами `CR` и `LF` ASCII-кода.

Пользовательский интерфейс FTP-клиента зависит от его программной реализации. Наряду с традиционными клиентами, работающими в символьном режиме, имеются графические оболочки, не требующие от пользователя знания символьных команд.

Символьные клиенты обычно поддерживают следующий основной набор команд.

- `open` *имя_хоста* — открытие сеанса с удаленным сервером.
- `bye` — завершение сеанса с удаленным хостом и завершение работы утилиты ftp.
- `close` — завершение сеанса с удаленным хостом, утилита ftp продолжает работать.
- `ls` (`dir`) — печать содержимого текущего удаленного каталога.
- `get` *имя_файла* — копирование удаленного файла на локальный хост.
- `put` *имя_файла* — копирование удаленного файла на удаленный сервер.

Пример. Файловая система NFS

Сетевая файловая система (Network File System, **NFS**) создана в середине 80-х годов компанией Sun Microsystems. Являясь фактическим стандартом для ОС семейства Unix, эта система реализована и для многих других ОС. Так, NFS широко используется в сетях Microsoft и Novell, а также в таких решениях компании IBM, как AS400 и OS/390. NFS, пожалуй, самая распространенная платформенно-независимая сетевая файловая система. Принципы ее организации на сегодня стандартизованы сообществом Интернета, последняя версия NFS

v.4 описывается спецификацией RFC 3530 (http://www.ietf.org/rfc/ rfc3530. txt), выпущенной в апреле 2003 года.

На рис. 11.8 показана многоуровневая структура программных средств реализации файловой системы NFS. Программный код NFS разделен на две части: клиентскую и серверную. Когда приложению требуется выполнить какие-либо действия с файлом, оно посылает запрос NFS-клиенту.

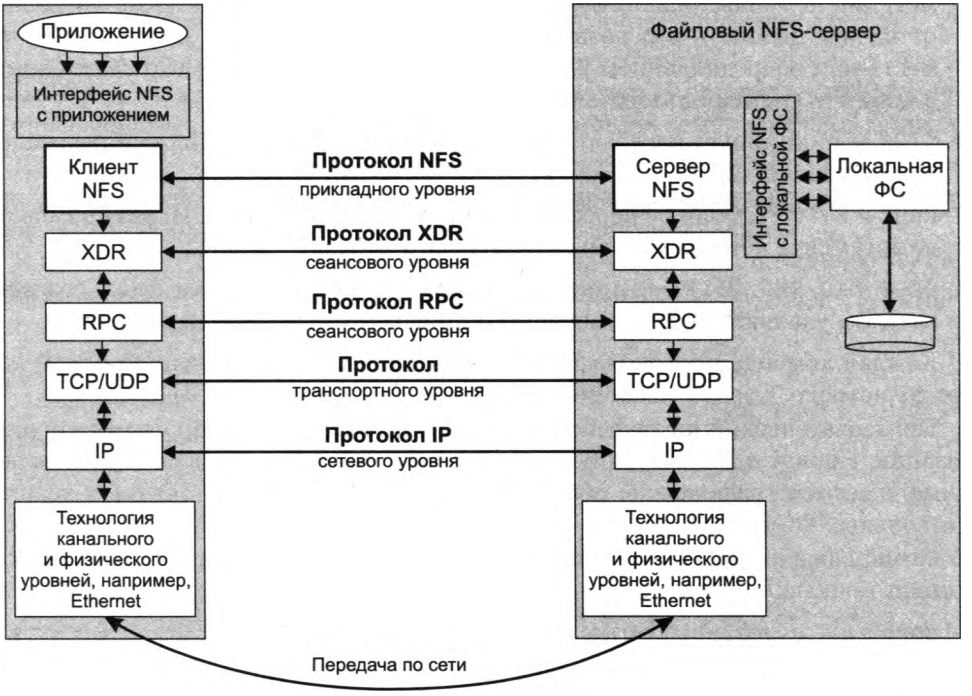


Рис. 11.8. Многоуровневая структура сетевой файловой системы NFS

Программный код NFS-клиента реализует все обращения клиентской системы к удаленным файлам путем посылки NFS-серверу одного или нескольких сообщений *протокола RPC*. Как рассказано в предыдущей главе, протокол RPC предназначен для того, чтобы сделать более простой разработку сетевых приложений, а именно дать возможность приложению вызывать программную процедуру на удаленном компьютере точно таким же образом, каким оно обращается к локальной подпрограмме. Другими словами протокол RPC, наряду с другими средствами, такими, например, как монтирование файловых систем, направлен на обеспечение прозрачности сети, в том числе при работе с файлами. При наличии механизма RPC приложение может обращаться к удаленным файлам, используя тот же набор системных вызовов («создание файла», «запись блока», «чтение блока»), что и для доступа к локальным файлам. Понятно, что в действительности, чтобы запустить процедуру на другом компьютере, требуется выполнить много дополнительной работы. Именно с помощью RPC

и происходит подмена этого «простого» вызова процедуры реально необходимым комплексом действий.

На следующем уровне модели OSI располагаются средства XDR (eXternal Data Representation — внешнее представление данных). Поскольку одной из целей разработчиков NFS была поддержка неоднородных систем с клиентами и серверами, работающими под управлением различных ОС на различных аппаратных платформах, имеющих разный порядок представления байтов в машинном слове, разное представление чисел с плавающей точкой и др., в реализациях NFS на основе механизма RPC по умолчанию поддерживаются средства для унифицированного представления аргументов удаленных процедур.

Далее для передачи сообщений используются транспортные средства стека TCP/IP.

В файловой системе NFS можно определить два типа процедур:

- процедуры монтирования удаленной файловой системы;
- процедуры доступа к удаленным файлам.

Процедуры монтирования дают возможность NFS-клиентам получать доступ к разделяемым каталогам, которые предоставляются NFS-сервером. Для выполнения монтирования NFS-клиент посылает серверу полное имя каталога и запрашивает разрешение на монтирование этого каталога в какой-либо точке собственного дерева каталогов. При этом серверу не указывается, в какое место будет монтироваться каталог сервера. Получив имя, сервер проверяет законность этого запроса и возвращает клиенту дескриптор файла, являющегося удаленной точкой монтирования. Дескриптор включает описатель типа файловой системы, номер диска, номер индексного дескриптора (inode) каталога, который является удаленной точкой монтирования, информацию безопасности. Операции чтения и записи файлов из монтируемых файловых систем используют дескрипторы файлов вместо символьного имени.

Монтирование может выполняться автоматически с помощью командных файлов при загрузке. Существует другой вариант автоматического монтирования: при загрузке ОС на рабочей станции удаленная файловая система не монтируется, но при первом открытии удаленного файла ОС посылает запросы каждому серверу и после обнаружения этого файла монтирует каталог того сервера, на котором расположен найденный файл. Программа автоматического монтирования регулярно (по умолчанию каждые 5 минут) проверяет, выполняются ли какие-либо работы с файлами и подкаталогами смонтированного дерева каталогов. Если нет, то происходит автоматическое размонтирование, а если да, то система продолжает функционировать в прежней конфигурации.

Работа пользователя с удаленными файлами после выполнения операции монтирования оказывается полностью прозрачной — поддерево файловой системы NFS-сервера становится поддеревом локальной файловой системы. Выполнение программ почти не зависит от того, где расположен файл: локально или на удаленном диске. Если два или более клиента одновременно смонтировали один и тот же каталог, то они могут связываться путем разделения файла.

Основными функциями NFS являются функции доступа к удаленным файлам и каталогам. NFS поддерживает *модель удаленного доступа*. В этом случае все файловые операции выполняются на серверах, а клиенты только генерируют запросы на выполнение какого-либо действия над каталогом или операции чтения или записи файла. Кроме того, они могут запросить атрибуты файла, такие как тип, размер, время создания и модификации.

В NFS принята схема *stateless*, то есть серверы при работе с файлами не хранят данные об открытых клиентами файлах. Поэтому в NFS поддерживается большая часть системных вызовов Unix, за исключением `open` и `close`. Для обеспечения устойчивости клиентов к отказам серверов системные вызовы `open` и `close` исключены. Вместо операции открытия удаленного файла клиент посылает серверу сообщение, содержащее имя файла, с запросом отыскать его (`lookup`) и вернуть дескриптор файла. В отличие от вызова `open` вызов `lookup` не копирует никакой информации во внутренние системные таблицы. Вызов `read` содержит дескриптор того файла, который нужно считать, смещение в уже читаемом файле и количество считываемых байтов. Подчеркнем, что в схеме stateless каждый вызов является самодостаточным. Он несет всю необходимую информацию для выполнения заданной операции, никак не зависит от предыдущих операций и никак не влияет на последующие.

Таким образом, если сервер выйдет из строя, а затем будет восстановлен, информация об открытых файлах не потеряется, потому что она не поддерживается. При отказе сервера клиент просто продолжает посылать на него команды чтения или записи в файлы, однако не получив ответа и исчерпав тайм-аут, клиент повторяет свои запросы. После перезагрузки сервер получает очередной повторный запрос клиента и отвечает на него. Таким образом, крах сервера вызывает только некоторую паузу в обслуживании клиентов, но никаких дополнительных действий по восстановлению соединений и повторному открытию файлов от клиентов не требуется.

К недостаткам NFS относится *сложность блокировки файлов*. Во многих ОС файл может быть открыт и заблокирован так, чтобы другие процессы не имели к нему доступа. Когда файл закрывается, блокировка снимается. В stateless-системах, подобных NFS, блокирование не может быть связано с открытием файла, так как сервер не знает, какой файл открыт. Для выполнения блокировки в сетевой файловой системе NFS используются специальные, формально не связанные с NFS средства управления блокированием — менеджер блокировок сети (Network Lock Manager, NLM) и монитор состояния сети (Network Status Manager, NSM), который ведет статистику блокировок. Последнее необходимо для того, чтобы при отказе NFS можно было автоматически восстановить блокировки.

Для повышения производительности в NFS используется кэширование на стороне как сервера, так и клиента, причем в последнем случае кэширование выполняется не только в оперативной памяти, но и на локальном диске.

На стороне клиента поддерживаются кэши для содержимого файлов и атрибутов файлов. Содержимое файла переносится в кэш поблочно, при этом применяется **упреждающее чтение**. При выполнении этой операции чтение блока

в кэш по требованию приложения всегда сопровождается чтением следующего блока по инициативе системы. Кэширование выполняется и при записи. В этом случае данные, генерируемые приложением для записи в удаленный файл, накапливаются в кэше содержимого и лишь затем производится их пересылка на сервер.

Кэширование на клиентском компьютере разделяемого файла может привести к тому, что в какой-то момент содержимое этого файла в кэше одного клиента не будет совпадать с содержимым этого же файла в кэше другого клиента. Для того чтобы минимизировать возможный период такого рассогласования, используются атрибуты файла. Они помещаются в кэш атрибутов и регулярно (каждые несколько секунд) автоматически обновляются. Если, например, один из двух клиентов, совместно использующих файл, выполнил запись, то это немедленно вызовет изменение значений атрибутов этого файла, таких как «время обновления» и/или «размер файла». В этот момент происходит рассогласование копий файла. Однако благодаря регулярному обновлению кэша атрибутов на обеих клиентских машинах изменения атрибутов очень быстро становятся видны второму NFS-клиенту, который во избежание рассогласования автоматически выполняет обновление кэша содержимого файла.

На стороне сервера кэшируются записываемые данные, а также информация о дублируемых запросах. Кэширование записываемых данных осуществляется на основе так называемой технологии **безопасной асинхронной записи**. После того как данные пересланы в кэш (в оперативной памяти) сервера, от клиента поступает сообщение, которое инициирует операцию записи на диск сервера. Если операция записи данных на диск сервера проходит успешно, сервер отправляет подтверждение об этом клиенту. Такая процедура позволяет гарантировать целостность данных после перезагрузки сервера в случае его отказа.

Кэширование дублированных запросов направлено не столько на повышение производительности, сколько на повышение устойчивости и корректности работы NFS. Если при выполнении запроса возникает задержка, клиент может повторить свой запрос, причем не один раз. Это является штатной ситуацией для NFS, где сервер функционирует по схеме stateless и повтор запроса клиентом является частью восстановительной процедуры. Однако клиент не может знать точно, выполнен ли его предыдущий запрос, и может послать избыточный запрос. Для одних операций, таких как запись данных на сервер, такая избыточность не несет никакой опасности — сервер просто повторяет чтение блока. Повторение же других операций, таких, например, как «удалить каталог», может привести к серьезной ошибке. Для избежания этого NFS-сервер поддерживает кэширование дублированных запросов, и все повторные запросы, поступившие в течение короткого периода, игнорируются.

Метод кэширования NFS не сохраняет семантику Unix для разделения файлов. Вместо этого используется не раз подвергавшаяся критике *сеансовая семантика*, в которой изменения данных в кэшируемом клиентом файле видны другому клиенту по-разному, в зависимости от временных соотношений. Клиент при очередном открытии файла, имеющегося в его кэше, проверяет у сервера, когда файл был в последний раз модифицирован. Если это произошло после того, как файл был помещен в кэш, клиент удаляет его из кэша и получает от

сервера новую копию этого файла. Клиенты распространяют модификации, сделанные в кэше, с периодом в 30 секунд, так что сервер может получить обновления с большой задержкой. В результате работы механизмов удаления данных из кэша и распространения модификаций данные, получаемые каким-либо клиентом, не всегда являются самыми последними.

Контроль доступа в NFS выполняется путем совместного применения правил, определенных в локальной файловой системе NFS-сервера, и правил, заданных при выполнении так называемой процедуры экспорта.

Эта процедура заключается в следующем. Для того чтобы клиент смог выполнить монтирование удаленной файловой системы (или ее части), на сервере необходимо предварительно явно объявить, какие ресурсы файловой системы сервер может предоставить клиентам для совместного использования. Кроме того, процедура экспорта включает определение прав доступа к разделяемым файлам и каталогам на сервере для сетевых клиентов, которые в этом случае идентифицируются IP-адресами или доменными именами соответствующих компьютеров.

Сервер выдает разрешение клиенту на выполнение тех или иных действий в виде числового кода *magic cookie*. Это число служит своего рода паролем, который клиент должен включать в каждый RPC-запрос в качестве доказательства своих полномочий. Кроме того, RPC-запрос содержит идентификаторы пользователя uid, от имени которого выступает приложение, и список идентификаторов групп gid, в которые входит данный пользователь. Заметим, что администратор сервера задает права доступа к разделяемым файловым ресурсам на основе идентификаторов uid и gid клиентов. А это значит, что имеется потенциальная возможность коллизии. Обязанностью администратора является обеспечение согласованной идентификации пользователей на сервере и клиенте. В большой сети эта задача становится нетривиальной, поэтому для этих целей используется сетевая информационная служба **NIS** (Network Information System), представляющая собой достаточно простой вариант сетевой справочной службы. Функциями NIS является централизованное хранение и обработка системных файлов.

Понятно, что описанный механизм контроля доступа не является вполне надежным. Злоумышленнику достаточно перехватить одно RPC-сообщение, чтобы получить несанкционированный доступ. Для совершенствования безопасности NFS разработаны дополнительные программные средства, в частности протокол SecureRPC, который делает более защищенными не только контроль доступа, но и процедуры аутентификации.

Справочная сетевая служба

Назначение справочной службы

Подобно большой организации, большая компьютерная сеть нуждается в хранении и удобном доступе к как можно более полной справочной информации о самой себе. Решение многих задач в сети опирается на информацию о пользо-

вателях сети — их именах, применяемых для логического входа в систему, паролях, правах доступа к ресурсам сети, а также на информацию о ресурсах и компонентах сети — серверах, клиентских компьютерах, маршрутизаторах, шлюзах, томах файловых систем, принтерах и т. п.

Одной из наиболее часто решаемых в системе задач, базирующихся на справочной информации о пользователях, является проверка их легальности (аутентификация), на основе которой затем выполняется проверка прав доступа пользователей к ресурсам системы (авторизация). Для этого в сети должны храниться учетные записи пользователей, содержащие их имена и пароли.

Электронная почта является еще одним популярным примером службы, которой требуется справочная информация о почтовых именах и других характеристиках абонентов этой службы.

В последнее время в сетях все чаще стали применяться средства управления качеством обслуживания (Quality of Service, QoS) трафика, которым также необходимы сведения о пользователях и приложениях системы, о требованиях, предъявляемых ими к параметрам качества обслуживания трафика, о сетевых устройствах, позволяющих управлять трафиком (маршрутизаторах, коммутаторах, шлюзах и т. п.).

Чтобы обеспечить прозрачность доступа пользователей и приложений к сетевым ресурсам, таким как серверы, тома файловой системы, интерфейсы RPC-процедур и другим, требуется поддержка базы данных, в которой должны храниться отображения символьных имен ресурсов, указанных в запросах пользователей и приложений, на их числовые идентификаторы (например, IP-адреса). Эти отображения позволяют автоматически и прозрачно для пользователя находить эти ресурсы в сети.

Организация распределенных приложений может существенно упроститься, если в сети имеется база данных, хранящая информацию об имеющихся программных модулях-объектах и их расположении на серверах сети. Приложение, которому необходимо выполнить некоторое стандартное действие, обращается с запросом к такой базе и получает адрес программного объекта, имеющего возможность выполнить требуемое действие.

Система автоматизированного управления сетью должна располагать информацией о топологии сети и характеристиках всех ее аппаратных и программных элементов, таких как маршрутизаторы, коммутаторы, серверы и клиентские компьютеры, установленные на них операционные системы и приложения. Только обладая этой информацией, система управления способна правильно идентифицировать сообщения об аварийных событиях и находить их первопричину.

Кроме того, упорядоченная по подразделениям предприятия информация об имеющимся сетевом оборудовании и установленном программном обеспечении полезна сама по себе, так как помогает администраторам составить достоверную картину состояния сети и разработать планы по ее развитию.

Результатом развития систем хранения справочной информации стало появление в сетевых ОС специальной подсистемы — справочной службы.

Справочная служба (directory[1] services), называемая также **службой каталогов**, имеет основной целью хранение информации, относящейся к сети, в которой эта служба установлена, с тем чтобы предоставлять эту информацию по запросам всем пользователям и приложениям, имеющим права на доступ к этим данным.

ПРИМЕЧАНИЕ В некоторых случаях справочная служба может быть использована и для хранения информации, не связанной с функционированием сети. Например, она может включать персональные данные служащих, такие как фамилия, имя, отчество, должность, заработная плата, домашний адрес, телефон, дата рождения и т. п. Или содержать данные о наличии и характеристиках оборудования (не обязательно сетевого) в разных подразделениях предприятия. Важно, чтобы характер этих данных совпадал с характером справочной информации, для хранения которой предназначена справочная служба, а именно данные должны быть потенциально полезными для потребителей в пределах всей сети, меняться относительно редко и в небольших масштабах.

Справочная служба *хранит информацию обо всех пользователях и ресурсах сети* в виде унифицированных объектов, снабженных определенными атрибутами, а также отражает взаимосвязи между хранимыми объектами, такие как принадлежность пользователей к определенной группе, права доступа пользователей к компьютерам и разделяемым ресурсам, вхождение нескольких узлов в одну подсеть, коммуникационные связи между подсетями, производственная принадлежность серверов и т. д.

Справочная служба *позволяет выполнять над хранимыми объектами набор некоторых базовых операций*, таких как добавление и удаление объекта, изменение значений атрибута объекта, чтение атрибутов и некоторые другие. Объекты справочной службы могут быть организованы в иерархические структуры, что делает возможным выполнение групповых операций над объектами. Например, администратор может определять права доступа сразу для группы пользователей или единовременно выполнять переименование/удаление сразу группы ресурсов.

Сетевая служба регулирует взаимодействие между сетевыми объектами, *предоставляя контроллируемый доступ к информации* в соответствии с заданными в ее базе данных правами доступа для разных типов клиентов этой службы.

Клиентами справочной службы являются администраторы, пользователи, приложения, сетевые службы и сетевые устройства.

- *Администраторы* создают учетные записи для каждого пользователя, определяют права доступа к разделяемым ресурсам, конфигурируют сетевые устройства на основе имеющейся в справочной службе информации, делегируют право администрирования различных частей сети другим специалистам. Справочная служба позволяет рационализировать работу администратора,

[1] Directory (англ.) — справочник, каталог.

предоставляя ему средства выполнения операций над группами объектов сети, так что рутинные действия не приходится повторять для многочисленных объектов сети.

- *Пользователи* в ходе работы получают доступ к ресурсам после прохождения процедур проверки доступа, выполняемых ОС на основе информации, хранимой справочной службой. При наличии соответствующего разрешения пользователи могут читать из хранилища и записывать в хранилище данные справочной службы.
- *Приложения, сетевые службы.* Поверх справочной службы работают другие сетевые службы, такие как служба аутентификации или почтовая служба, а также различные приложения. Они запрашивают исходные данные для решения своих специфических задач, например, клиенту почтовой службы может потребоваться полное имя адресата, а приложению, решающему задачу инвентаризации, — перечень характеристик программного и аппаратного обеспечения сети.
- *Сетевые устройства* обращаются к справочной службе с запросами о конфигурационной информации.
- Для выполнения всех операций, связанных с доступом к базе данных справочной службы, администраторы, пользователи, приложения, службы и устройства должны быть наделены соответствующими правами доступа.

Альтернативой единой справочной службе сети является применение нескольких автономных справочных служб узкого назначения: одной — для аутентификации, другой — для управления сетью, третей — для разрешения имен компьютеров и т. д. Однако в крупной сети такой подход оказывается неэффективным. Даже если каждая из таких служб хорошо организована и сочетает централизованный интерфейс с распределенной базой данных, большое число справочных служб приводит к дублированию информации, усложняет администрирование и управление сетью. Например, до 2000 года в операционной системе Windows NT компании Microsoft имелось, по крайней мере, пять различных типов справочных баз данных. Главный справочник домена (NT Domain Directory Service) хранил информацию о пользователях, требуемую для их логического входа в сеть. Данные о тех же пользователях могли содержаться и в другом справочнике, используемом электронной почтой Microsoft Mail. Еще три базы данных поддерживали разрешение адресов: служба WINS устанавливала соответствие Netbios-имен IP-адресам, справочник DNS — соответствие доменных имен IP-адресам, справочник протокола DHCP служил для автоматического назначения IP-адресов компьютерам сети. Очевидно, что такое разнообразие справочных служб усложняло жизнь администратора и приводило к дополнительным ошибкам, например, когда учетные данные одного и того же пользователя нужно было ввести в несколько баз данных. Поэтому в сменивших Windows NT операционных системах, начиная с Windows 2000, на смену всем этим разрозненным справочным службам пришла интегрированная с системой DNS распределенная справочная служба Active Directory, способная хра-

нить и поддерживать всю справочную информацию о системе. Далее мы подробно рассмотрим работу этой справочной службы.

Помимо Active Directory компании Microsoft, существуют и другие реализации справочных служб для сетевых ОС, в частности справочная служба NDS компании Novell, а также сетевые информационные службы NIS, NIS+ и Sun компании Sun, Java System Directory Server компании Sun Microsystems, применяемые в основном в сетях Unix.

Как и все современные программные системы, справочные службы придерживаются общепринятых стандартов. Основными процедурами, подлежащими стандартизации в данном случае, являются хранение данных и доставка этих данных потребителям. Наиболее популярным стандартом по хранению справочной информации является стандарт **X.500**, первоначально разработанный ITU-T для использования вместе с почтовой службой X.400. Другой широко распространенный в справочных службах стандарт – протокол **LDAP** (Lightweight Directory Access Protocol). Этот стандарт, разработанный сообществом Интернета, определяет процедуру доставки данных из хранилища справочной службы ее клиентам.

Архитектура справочной службы

Для типичной справочной службы характерно использование *модели клиент-сервер*: выделенные серверы хранят базу справочной информации, которой пользуются клиенты справочной службы, передавая серверам по сети соответствующие запросы.

В соответствии с выбранной архитектурой различают следующие типы справочной службы:

- централизованная;
- распределенная;
- децентрализованная.

Централизованная модель, при которой вся справочная информация о сети и пользователях хранится и обрабатывается на одном компьютере, достаточно эффективна только в небольших сетях.

Для крупных сетей более подходящим вариантом является *распределенная* структура сетевой справочной службы, которая предполагает наличие нескольких физически разделенных баз данных, виртуально представленных для клиентов в виде единой центральной базы данных.

Первые реализации справочных служб (тогда для их обозначения еще не использовался данный термин) работали локально, то есть каждый компьютер был оснащен собственной справочной службой, которая работала независимо от остальных компьютеров сети; информационные запросы, порожденные на каком-либо компьютере, обрабатывались на нем же и касались ресурсов и пользователей, связанных с этим компьютером. Мы будем говорить, что в этом случае справочная служба сети имеет *децентрализованную* архитектуру.

Еще одним параметром архитектуры справочной службы является наличие или отсутствие *резервирования* базы данных. Резервирование повышает производительность и надежность справочной службы и может иметь место как в централизованных, так и распределенных структурных решениях.

Децентрализованная модель

Для **децентрализованной модели** характерно наличие нескольких логически не связанных между собой баз справочных данных, расположенных на разных компьютерах (рис. 11.9). Долгое время такого рода организация справочной информации была наиболее распространенной. Существовало и до сих пор существует много частных решений, позволяющих достаточно эффективно организовать работу сети на основе частных баз справочной информации, которые могут быть представлены обычными текстовыми файлами или таблицами, хранящимися в теле приложения. Например, в ОС Unix для хранения данных об именах и паролях пользователей традиционно служит файл passwd, который охватывает пользователей только одного компьютера. Имена адресатов электронной почты также можно хранить в локальном файле клиентского компьютера.

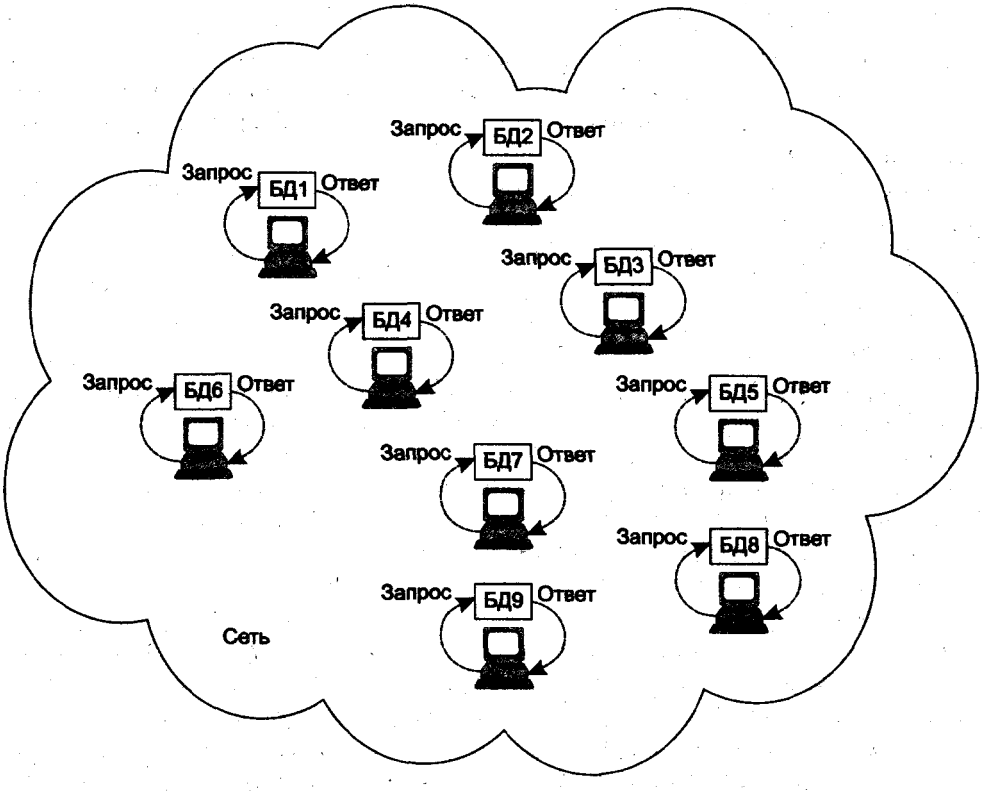


Рис. 11.9. Схема децентрализованной справочной службы

Однако децентрализованный подход в построении справочной службы входит в противоречие с главной целью сети — совместной работой пользователей. Поясним это на примере задачи аутентификации. В соответствии с децентрализованным подходом для каждого компьютера администратор поддерживает набор справочной информации о пользователях данного компьютера. Этот набор включает имя, пароль, права доступа и другую информацию, которая требуется при логическом входе пользователя в *данный* компьютер, а также при попытке получить доступ к ресурсам *данного* компьютера. Однако сетевой пользователь в общем случае должен иметь возможность доступа к ресурсам не только собственного, но и других компьютеров сети, более того, очень желательно, чтобы он мог работать не только за одним, закрепленным за ним компьютером, а за любым другим компьютером сети. Для того чтобы предоставить ему такую возможность, при децентрализованном подходе администратор должен предварительно выполнить достаточно трудоемкую процедуру: занести записи об этом пользователе в базы данных всех компьютеров сети, на которых пользователь предполагает работать. Более того, на каждом из этих компьютеров администратор должен обеспечить воспроизведение одной и той же привычной пользователю конфигурации рабочего экрана, индивидуальную настройку прикладных программ и т. п.

В целом такой децентрализованный подход в большой сети приводит к дублированию значительного объема справочных данных, а также к недопустимо большим затратам на администрирование. Последний недостаток отчасти смягчается тем обстоятельством, что децентрализованная справочная служба позволяет легко разделить работу по администрированию между несколькими специалистами.

Централизованная модель

Естественной альтернативой децентрализованной модели служит **централизованная модель**, в соответствии с которой все справочные данные о ресурсах и пользователях сети хранятся и обрабатываются централизовано на выделенном компьютере. На рис. 11.10 показан сервер справочной службы, обслуживающий центральную базу данных, которая объединяет справочную информацию БД1 – БД9, относящуюся к каждому из компьютеров сети. Клиентские компоненты справочной службы установлены на всех остальных компьютерах. Используя клиентскую программу, каждый пользователь и приложение, работающие на некотором компьютере сети, могут сделать запрос и получить данные о ресурсах всех других компьютеров. При этом нет необходимости в дублировании информации.

Рассмотрим, как, например, изменяется решение задачи администрирования пользователей при централизованном подходе. В этом случае, вместо того чтобы заводить для пользователей учетные записи на тех компьютерах, на которых каждый из них работает, администратор создает и поддерживает *единую базу данных* для всех пользователей сети. Эту базу данных задействует серверный компонент подсистемы аутентификации. Работа пользователя по-прежнему

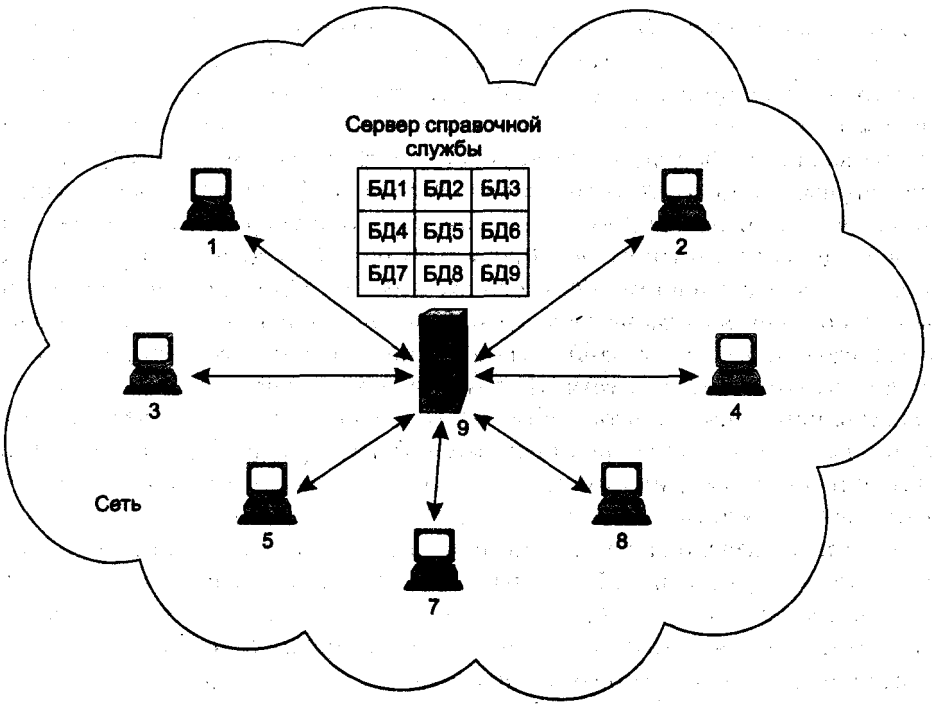


Рис. 11.10. Схема централизованной справочной службы

начинается с того, что он логически входит в сеть, вводя свои имя-идентификатор и пароль. Клиентский компонент передает эти данные на сервер[1], на котором происходит сравнение введенного пользователем пароля с паролем, занесенным в базу данных администратором. Если эти пароли совпадают, считается, что аутентификация прошла успешно, о чем и отправляется сообщение на клиентский компьютер. Такая централизованная процедура не «привязывает» пользователя к определенному компьютеру (или нескольким компьютерам) и резко снижает избыточность и сложность ведения учетной информации. Аналогичные преимущества характерны и для решения других задач на основе централизованной модели справочной службы.

Однако эта модель хорошо работает только *в небольшой сети*. Единая база данных, хранящая справочную информацию большого объема, порождает все то же множество проблем, что и любая другая крупная база данных. Реализация справочной службы как локальной базы данных, хранящейся в виде одной копии на одном из серверов сети, не подходит для большой системы в первую очередь вследствие низкой производительности и низкой надежности такого решения.

[1] В действительности из соображений безопасности диалог клиента с сервером происходит в более сложной форме, не требующей передачи пароля в открытом виде по сети.

Производительность оказывается низкой из-за того, что запросы к справочной службе от всех пользователей и приложений сети будут поступать на единственный сервер, который обязательно перестанет справляться с их обработкой при превышении определенного порога количества запросов. Кроме того, централизация приводит к росту внутрисетевого трафика. Ситуация усугубляется, если сеть включает медленные глобальные связи. Процедура выполнения запроса к серверу может стать неприемлемо длительной из-за задержки передачи запроса, времени пребывания запроса в очереди к серверу и времени, затраченного на поиск информации в базе данных, которое может оказаться значительным, если центральная БД имеет большой объем. Другими словами, такое решение плохо масштабируется в отношении количества обслуживаемых пользователей и разделяемых ресурсов.

Надежность также не может быть высокой в системе с единственной копией данных. Отказ аппаратуры или программного обеспечения сервера, на котором поддерживается эта база данных, приведет к параличу справочной службы в масштабах всей сети.

Помимо снятия ограничений по производительности и надежности, желательно, чтобы структура базы данных позволяла производить логическое группирование ресурсов и пользователей по структурным подразделениям предприятия и назначать для каждой такой группы своего администратора. Централизованный подход не удовлетворяет этому требованию.

Централизованная модель с резервированием

Смягчить недостатки централизованной модели можно путем *резервирования*, то есть поддержания нескольких копий базы данных на разных компьютерах.

На рис. 11.11 база данных справочной службы представлена двумя идентичными экземплярами, что дает возможность повысить как производительность, так и надежность. Справочная служба в такой схеме будет работать быстрее, так как поток запросов может быть разделен между двумя центрами обработки. Наличие избыточного центра обработки запросов повышает и надежность, так как в случае отказа аппаратуры или программного обеспечения компьютера, на котором хранится одна из копий БД, справочная служба полностью сохраняет свою работоспособность.

На первый взгляд, наличие резервных копий может повысить производительность справочной службы еще и за счет возможности приближения справочной информации к источникам запросов. Так, например, если в сети есть медленная глобальная связь, то размещение каждого из двух экземпляров базы данных по обе стороны этой связи исключило бы передачу клиентских запросов через глобальную связь. Однако возможная выгода в значительной степени нивелируется из-за необходимости поддержания идентичности обеих баз данных. При выполнении каждого запроса содержимое справочной базы данных может быть изменено. Для того чтобы обеспечить динамическое совпадение копий, хранящихся на разных компьютерах сети, все изменения, выполненные на одном компьютере, должны дублироваться на другие. Другими словами,

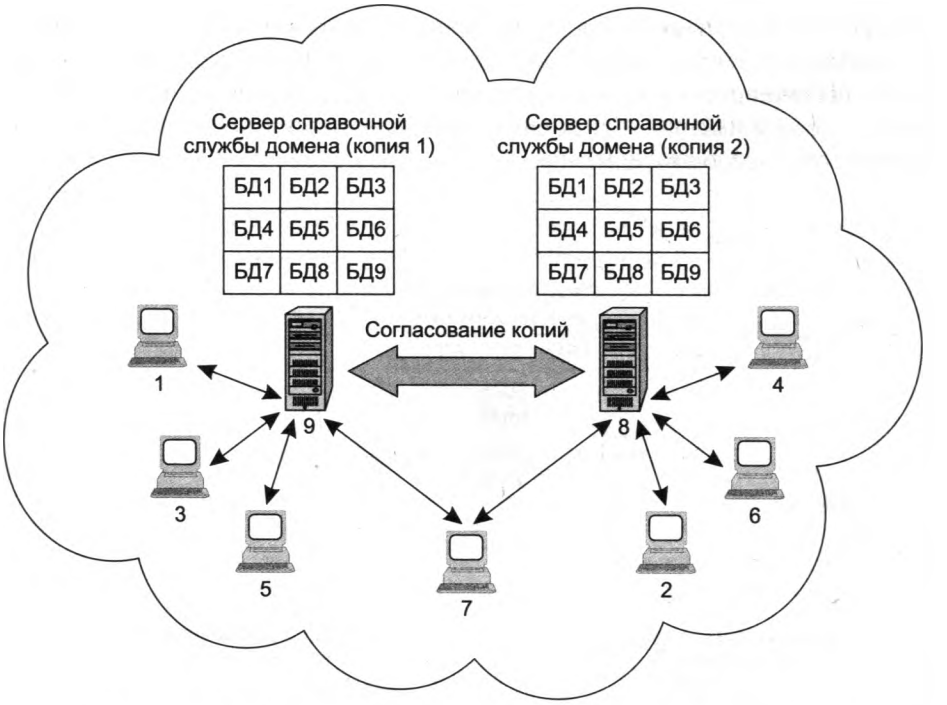


Рис. 11.11. Схема централизованной справочной службы с резервированием

информация сохраняется обновленной и синхронной путем периодического копирования изменений в каждой копии БД на все компьютеры, хранящие остальные копии БД. Таким образом, размещая базу данных ближе к пользователям и ликвидируя барьер глобальной связи между пользователями и сервером справочной службы, мы одновременно устанавливаем этот барьер между двумя копиями БД, что замедляет процедуру согласования этих копий.

За повышение надежности и производительности централизованная система с резервированием расплачивается избыточностью и сложностью поддержания нескольких копий. Кроме того, она не решает проблему плохой масштабируемости, характерную для любой централизованной модели: в крупных сетях, имеющих тысячи пользователей и простирающихся на тысячи километров, единая БД может стать настолько объемной, что время выполнения запроса окажется недопустимо большим, а задача администрирования хранилища данных — трудноразрешимой.

Декомпозиция справочной службы на домены

Исходя из того, что в небольшой сети централизованная схема работает эффективно, одним из возможных решений могло бы быть разделение большой сети на части (будем называть их **доменами**) и реализация во всех доменах отдельных, не связанных между собой централизованных справочных служб. На рис. 11.12

показана сеть, разделенная на три домена, в каждом из которых работает собственная централизованная справочная служба. Базы данных, размещенные на серверах справочной службы в каждом домене, содержат лишь часть справочных данных сети, а именно — те данные, которые относятся к ресурсам и пользователям соответствующих доменов.

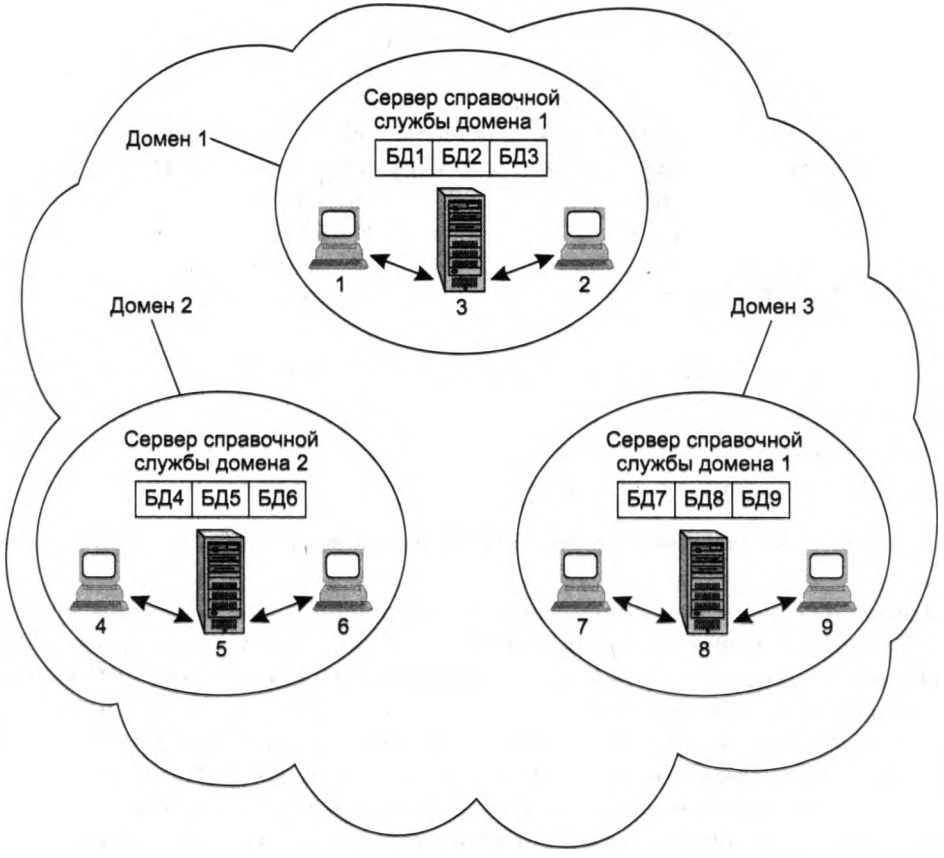


Рис. 11.12. Декомпозиция справочной службы на не связанные между собой справочные службы доменов

Справочные службы доменов обладают всеми приемуществами, свойственными централизованным системам, а главный недостаток централизованных систем — плохая масштабируемость — преодолевается структуризацией сети. Действительно, хотя увеличение размера сети и ведет к росту общего объема справочных данных, размер каждой отдельной БД может поддерживаться в разумных границах за счет образования новых доменов и связанных с ними новых баз данных. При правильном выборе размера домена справочная служба может успешно справляться с потоком запросов своих клиентов. Кроме того, при необходимости повышения производительности и надежности в каждом

домене может быть применено резервирование. Повышению производительности может также способствовать приближение баз данных к источникам запросов путем рационального разбиения сети на домены. Так, если минимизировать использование глобальных связей в пределах домена, то обмен сообщениями между клиентами и сервером справочной службы домена будет осуществляться быстрее.

Коренным недостатком декомпозиционного подхода является то, что из-за изолированности справочных служб доменов пользователи и приложения получают удобный доступ к справочной информации *только в пределах своего домена*. Доступ к информации других доменов чрезвычайно затруднен. Действительно, представим, что произойдет, если пользователю некоторого домена потребуется найти информацию о других (принадлежащих другим доменам) пользователях или ресурсах, таких, например, как принтеры. В отсутствие какого-либо механизма объединения доменов пользователю придется самому решать, где может находиться искомая информация и по какому адресу следует посылать запрос. Возможно, для этого ему потребуется действовать наугад, перебирая все возможные адреса сети. Понятно, что такой способ организации справочной службы в виде нескольких не связанных между собой справочных служб отдельных доменов нельзя признать эффективным.

Распределенная модель

Для того чтобы обеспечить эффективную работу справочной службы в пределах всей сети, необходимо средство, которое позволило бы *объединить справочные службы отдельных доменов в единую справочную службу*. По сути, мы здесь сталкиваемся с задачей виртуализации: на основе физически распределенного хранилища справочной информации, образованного БД доменов, требуется построить виртуальную централизованную справочную службу, с помощью которой пользователь любого домена может получить информацию о любом объекте сети вне зависимости от того, с какой рабочей станции поступил запрос и где находится требуемая информация. Такая справочная служба должна скрывать от пользователей различные физические параметры сети: местонахождение серверов, применяемые коммуникационные протоколы, маршруты перемещения запросов, характеристики коммуникационного оборудования и др.

Логическое связывание доменных справочных служб в единую для всей сети виртуальную централизованную службу означает создание **распределенной системы**. Прекрасным примером распределенной сетевой службы является рассмотренная нами в предыдущей главе система доменных имен (DNS). Как уже было сказано, первоначально эта узкоспециализированная справочная служба имела децентрализованную структуру: на каждом компьютере хранился файл с информацией о соответствии имен и IP-адресов компьютеров. Как только размеры Интернета превысили определенный предел, хранить справочную информацию в локальных текстовых файлах стало неэффективно. Потребовалось создать распределенную базу данных, поддерживаемую иерархически свя-

занными серверами имен, чтобы процедуры разрешения символьных имен в Интернете стали выполняться быстро и эффективно.

Существуют различные механизмы связывания доменных справочных служб в единую службу сети. Например, в справочной службе Active Directory компании Microsoft таким механизмом является **глобальный каталог** (global catalog), схему применения которого иллюстрирует рис. 11.13.

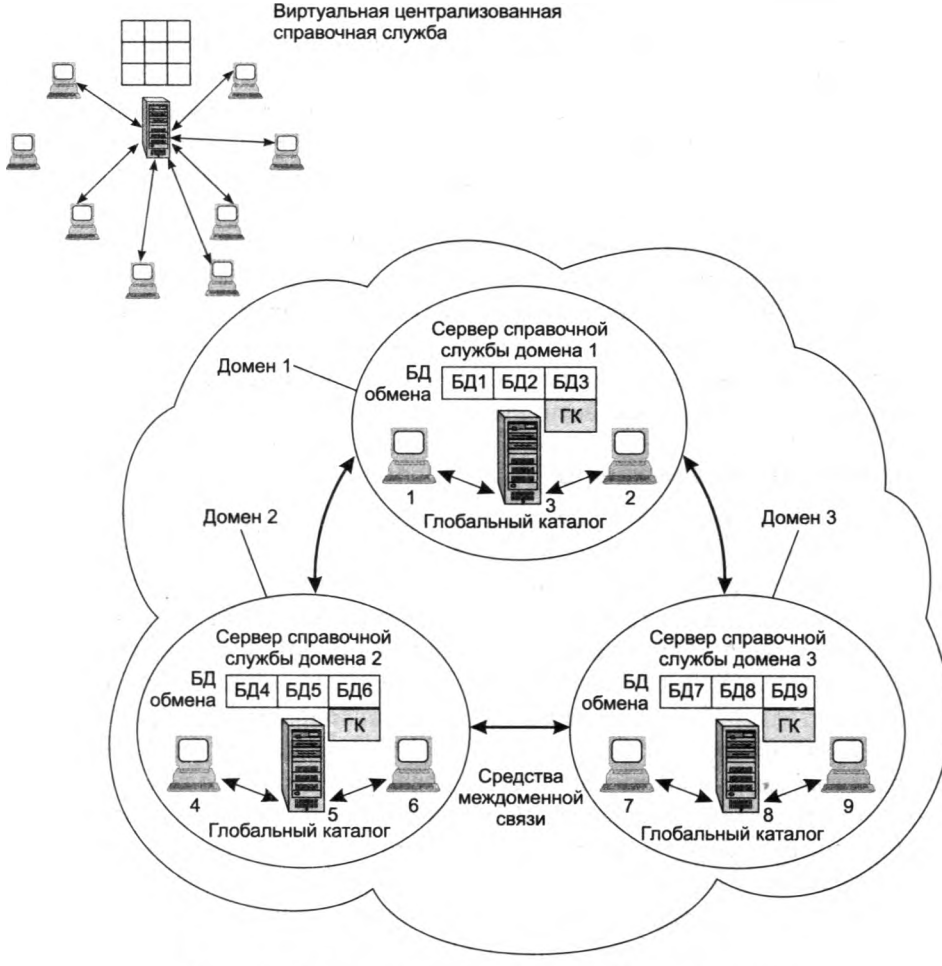


Рис. 11.13. Схема распределенной справочной службы

В то время как доменные базы данных содержат *полную* информацию об объектах соответствующего *домена*, в глобальном каталоге представлена *частичная* информация обо всех объектах *сети*. В качестве обязательной информации глобальный каталог хранит для каждого объекта сети атрибуты, которые могут быть использованы для определения местонахождения полной информации о данном объекте.

Копии глобального каталога размещают на сервере справочной службы в каждом домене. При поступлении запроса пользователя к ресурсам, находящимся вне его домена, справочная служба, пользуясь информацией из локальной копии глобального каталога, переадресует запрос к базе данных того домена, где находится интересующий пользователя объект. Все эти действия скрыты от пользователей справочной службы и выполняются автоматически.

Распределенная организация справочной службы является наиболее эффективной для крупных сетей. К числу ее достоинств можно отнести следующие.

- *Удобство доступа пользователей* к справочной информации. В распределенной системе для пользователя поддерживается иллюзия единого централизованного хранилища всей информации, когда степень сложности доступа к любому объекту сети не зависит от того, с какого компьютера поступил запрос.
- *Удобство администрирования.* Для каждой части распределенной базы данных, например домена, можно назначить отдельного администратора и наделить его правами доступа только к части информации обо всей системе.
- *Надежность.* Распределенная система по определению имеет несколько хранилищ и центров обработки информации, а, значит, при отказе одного из них система может продолжать функционирование, возможно, в ограниченном объеме. Кроме того, надежность может быть повышена за счет поддержания в каждом домене нескольких копий баз данных этого домена. Необходимые для этого процедуры согласования копий требуют значительно меньших затрат, чем в централизованных системах, так как проводятся в пределах домена, а не всей сети.
- *Высокая производительность.* Разделение данных между несколькими серверами снижает нагрузку на каждый сервер. Количество серверов не ограничивается числом доменов, так как в каждом домене могут быть установлены серверы, поддерживающие копии доменных БД. Повышению производительности может также способствовать приближение баз данных к источникам запросов путем рационального разбиения сети на домены.
- *Хорошая масштабируемость.* Распределенная служба продолжает эффективно функционировать даже в очень крупных сетях за счет возможности логической декомпозиции сети на домены. Это, в частности, позволяет ограничить объем БД, снизить вычислительные затраты на поддержание копий БД, приблизить серверы к клиентам, уменьшить сетевой трафик, ускорить время выполнения запросов.

Основные концепции справочной службы Active Directory

Домены, контроллеры доменов

Справочная служба **Active Directory** компании Microsoft построена на базе доменов. Домены являются основными элементами логической структуры Active Directory. В каждом из доменов реализуется централизованная справочная

служба. Будучи связаны логически, службы доменов образуют распределенную справочную службу сети.

Справочная служба построена по архитектуре клиент-сервер. Клиентские компоненты, устанавливаемые на всех компьютерах сети, передают запросы к серверам справочной службы, которые в Active Directory называются **контроллерами домена** (Domain Controllers, DC). В каждом домене должен присутствовать хотя бы один контроллер домена. Он поддерживает доменную базу данных, то есть БД о пользователях и ресурсах того домена, в котором установлен этот контроллер. Контроллер домена также занимается аутентификацией пользователей при их регистрации в сети.

Местоположение каждого контроллера домена администратор выбирает в ходе проектирования сети. Географическое разнесение серверов позволяет обеспечить приближенность информации к источникам запросов. Любой компьютор в сети, будь то контроллер домена, рядовой сервер или рабочая станция, может быть присоединен только к одному домену.

Возможность декомпозиции сети на домены является важнейшим условием масштабируемости службы Active Directory. Решение о том, сколько и каких иметь доменов, принимается с учетом различных факторов.

Прежде всего, учитывается, что домен Active Directory является *единицей администрирования*. По умолчанию при создании домена права и разрешения, которыми наделяются администраторы и пользователи, распространяются только на этот домен. А для того чтобы пользователь (в том числе администратор) одного домена мог получить доступ к ресурсам другого домена, должны быть выполнены дополнительные административные действия. Таким образом, разделение сети на домены является наиболее надежным и наиболее естественным способом разделения зон ответственности между несколькими администраторами.

Домен выступает также как *граница безопасности*, так что некоторые параметры политики безопасности действуют только в пределах этого домена. Таким образом, в домен следует объединять компьютеры и пользователей, относительно которых должна проводиться более-менее сходная политика администрирования, а это характерно для компьютеров и пользователей, связанных между собой организационно, то есть принадлежащих одному подразделению предприятия. При этом вовсе не обязательно, чтобы все они были расположены близко друг от друга. Например, домен может быть определен для всех компьютеров и пользователей, входящих в подразделение маркетинга, несмотря на то, что это подразделение состоит из нескольких филиалов, находящихся в разных странах.

Для повышения надежности и производительности в домене устанавливается несколько равнозначных контроллеров домена, которые поддерживают несколько идентичных копий доменной базы данных. Процедура динамического копирования всех изменений, происходящих на каждом из контроллеров, на все оставшиеся контроллеры домена называется **репликацией**. Таким образом,

домен является *единицей репликации*, то есть определяет часть сети, в пределах которой происходит репликация базы данных домена.

Также следует принять во внимание, что каждый домен Active Directory в то же время является *доменом DNS-имен*.

Сайты

В некоторых случаях справочная служба для оптимизации своей работы обращается к *информации о физической структуре сети*, а именно о том, какие связи — медленные или быстрые — используются в той или иной ее части.

Это, например, происходит при обращении клиента Active Directory к контроллеру домена. Поскольку обычно в домене имеется несколько идентичных контроллеров, клиент Active Directory должен выбрать тот из них, время выполнения запроса к которому является минимальным. Очевидно, что решение этой задачи основывается на информации о пропускной способности связей, соединяющих клиентский компьютер с каждым из контроллеров.

Другим примером использования справочной службой Active Directory информации о характере связей в сети является оптимизация процедур репликации. Для медленных связей контроллеры домена выполняют сжатие репликационных данных, а для быстрых связей они экономят процессорную мощность и отправляют данные несжатыми.

Необходимая информация о типе связей между клиентами и серверами поступает в Active Directory от администратора, который во время установки справочной службы определяет сайты

Сайт (site) — это элемент физической структуры компьютерной сети, представляющий собой ее часть, которая характеризуется «хорошей связностью». В типичном случае сайтом является набор IP-подсетей, связанных друг с другом высокоскоростными соединениями на основе локальной сетевой технологии, такой как Ethernet, а для соединения сайтов используются глобальные связи с гораздо более низкой пропускной способностью, например каналы PDH со скоростью 2 Мбит/с.

Имея информацию о том, какие серверы и клиенты относятся к одним и тем же сайтам, Active Directory легко может определить, какими связями, быстрыми или медленными, соединена любая пара компьютеров. Информацию о сайтах могут использовать в своих целях другие службы ОС, а также приложения, работающие с Active Directory.

Домены и сайты имеют различную природу. Домены являются элементами *логической*, а сайты — *физической* структуры сети. Поэтому деление сети на сайты никак не связано с делением сети на домены. В один сайт могут входить компьютеры, принадлежащие нескольким разным доменам, а в одном домене могут находиться компьютеры, относящиеся к нескольким разным сайтам. Вместе с тем, при установке контроллеров домена часто оказывается полезным учитывать физическую структуру сети. Очевидно, что более-менее равномерное

распределение контроллеров по сайтам может способствовать более эффективной работе справочной службы.

Объекты

Информация в справочной базе данных Active Directory представлена в виде иерархически организованного набора объектов, которые соответствуют отдельным пользователям, группам пользователей, компьютерам, принтерам, разделяемым папкам, элементам структуры (доменам и организационным единицам[1]), конфигурационным параметрам и другим сетевым ресурсам. Объекты могут создаваться как «вручную» администратором, который использует для этой цели диалоговые средства Active Directory, так и автоматически службой Active Directory и другим программным обеспечением.

Объект уникально идентифицируется своим именем и представляет собой набор значений атрибутов, которые свойственны данному классу объектов.

Класс объектов — это формальное описание множества объектов, имеющих сходную природу и вследствие этого характеризуемых одним и тем же набором обязательных и необязательных атрибутов.

Соотношение между классом объектов и объектом примерно такое же, как между переменной и ее значением. Атрибуты, определенные для класса объектов, принимают разные значения для разных объектов.

Из определения класса объектов следует, что объекты, содержащие информацию о компьютерах, относятся к одному классу объектов, а объекты, представляющие принтеры, — к другому. Рассмотрим, например, стандартный для Active Directory класс объектов `user`. Этот класс задает множество атрибутов, которыми может быть представлена информация о пользователях. Как и другие классы объектов, класс `user` определяет набор обязательных и необязательных атрибутов. В число семи обязательных атрибутов объектов этого класса входит, в частности, *каноническое имя пользователя*, а *номер телефона* является примером одного из 250 возможных необязательных атрибутов.

Когда администратор регистрирует нового пользователя, в базе данных Active Directory для этого пользователя создается учетная запись, которая и представляет собой объект. Для всех обязательных и для некоторых необязательных атрибутов данного объекта устанавливаются вполне определенные значения. Например, регистрируя в качестве пользователя некую Полину, администратор определяет значение канонического имени пользователя (обязательного атрибута) — Polina и значение номера телефона (необязательного атрибута) — 22345777. Таким образом, появляется новый объект класса `user`.

Глобальный каталог

Задача распределенной справочной службы состоит в предоставлении клиентам контроллируемого доступа ко *всем* объектам сети, даже если источник запроса и запрашиваемый ресурс находятся в разных доменах.

---

[1] Об организационных единицах см. далее в разделе «Иерархия организационных единиц».

Для решения этой задачи используется **глобальный каталог**. В отличие от доменных баз данных, которые хранят объекты, относящиеся только к собственным доменам, в базе данных глобального каталога хранится информация обо всех объектах сети. Однако в отличие от доменных баз данных, хранящих объект со всеми его атрибутами, в глобальном каталоге каждый объект представлен в виде «усеченной» версии, которая, как минимум, должна содержать атрибут[1], указывающий на местонахождение полной версии объекта. Такого рода атрибутом для большинства объектов является **отличительное имя** (Distinguished Name, DN), которое однозначно в пределах всей сети идентифицирует объект.

Глобальный каталог легко достижим для запросов всех клиентов, так как в каждом домене хранится одна или несколько его копий. Обычно администратор выделяет для хранения глобального каталога, по меньшей мере, по одному контроллеру на каждый сайт, такие контроллеры называют также **серверами глобального каталога**.

Для *поиска объектов* пользователь может направлять запрос к Active Directory, указывая отличительное имя объекта. Это имя подобно полному составному символьному имени файла в иерархической файловой системе, только в нем вместо имен каталогов указываются имена доменов и других узлов иерархической структуры базы данных объектов. Средства пользовательского интерфейса позволяют пользователю обращаться к объекту по его краткому имени — **относительному отличительному имени**. В этом случае служба Active Directory сама дополняет его до полного имени, используя контекстную информацию. Аналогично поиску файлов в файловой системе, поиск объекта может осуществляться и по значению какого-либо его атрибута. Например, чтобы найти объект класса `user`, клиент справочной службы может указать имя пользователя или адрес его электронной почты, а при поиске принтера — его тип. В этом случае Active Directory может вернуть клиенту информацию о нескольких объектах, каждый из которых удовлетворяет запросу, давая возможность пользователю самостоятельно выбрать интересующий его объект. Информация в главном каталоге позволяет определить DNS-имя контроллера, в котором хранится объект. На основании этого имени система DNS определяет IP-адрес контроллера, после чего задачу доступа к требуемому объекту можно считать решенной.

Наряду с поиском объектов на основе глобального каталога решается еще одна важнейшая задача справочной службы — *глобальная аутентификация пользователей*. Слово «глобальная» в данном случае означает, что пользователь при определенных условиях может выполнять логический вход в сеть с любого компьютера любого домена сети. Такая принципиальная возможность появляется благодаря тому, что в глобальном каталоге хранится информация об универсальных группах пользователей, в которые могут включаться члены разных доменов. А это значит, что для процедуры аутентификации пользователя, вхо-

[1] По желанию администратора в глобальный каталог могут быть добавлены дополнительные атрибуты.

дящего в одну из таких групп, достаточно обращения к ближайшему контроллеру локального домена, который хранит копию глобального каталога. Например, если сотруднику некоторого предприятия в Томске, оказавшемуся в командировке в барнаульском отделении, потребовалось выполнить некоторую работу на компьютере, то он может войти в сеть с любого компьютера сети данного предприятия в Барнауле независимо от того, относятся ли сети в Томске и Барнауле к одному и тому же домену или нет. Набрав идентификатор и пароль, полученные при регистрации от администратора в томском отделении, командированный сотрудник получает доступ к ресурсам сети в соответствии с теми же правами и разрешениями, которые он имел, входя в сеть со своего рабочего компьютера в Томске. Важно, что для аутентификации этого сотрудника вместо сравнительно медленной процедуры получения информации от удаленного сервера томского фрагмента сети здесь выполняется быстрый доступ к локальному серверу, хранящему копию глобального каталога.

Active Directory позволяет также выполнять *глобальную авторизацию*: при обращении к какому-либо ресурсу, расположенному в удаленном домене, приложению или пользователю не требуется взаимодействовать для проверки правомочности доступа с контроллером этого домена, достаточно обратиться к ближайшему серверу глобального каталога, в котором, помимо атрибутов о местонахождении каждого из объектов сети, могут храниться атрибуты, описывающие, какой вид доступа к этим объектам разрешен.

Возможность глобальной аутентификации и авторизации пользователей снижает трафик и нагрузку на сеть.

Иерархическая структура Active Directory

Как и все современные справочные службы, Active Directory является *иерархически организованной системой*. Иерархия логически упорядочивает информацию о многочисленных объектах, дает возможность «увидеть» всю систему в целом, упрощает процесс именования объектов и делает более эффективным выполнение индивидуальных и групповых действий над ними, таких как поиск, удаление, изменение свойств и т. д.

В зависимости от размеров сети администратор создает более или менее сложную иерархическую структуру в Active Directory. Основными строительными блоками иерархической структуры Active Directory являются домены и организационные единицы.

Иерархия организационных единиц

В пределах каждого домена может существовать иерархия организационных единиц.

Организационная единица (OU) — это специфический объект Active Directory, который представляет группу объектов, объединенных в соответствии с теми или иными их свойствами.

Если таким свойством является, например, тип объектов, то организационные единицы могут представлять собой группы пользователей, компьютеров или принтеров. Территориальная общность или принадлежность к одному и тому же подразделению также может быть использована для группирования объектов в организационные единицы, например: OU филиала предприятия в Москве и OU филиала в Перми, OU рекламного отдела, OU отдела перспективных разработок.

Как следует из определения, организационные единицы являются объектами. Такого рода объекты, используемые исключительно с целью группирования других объектов, называются также **контейнерами** (container). В этом случае оказывается полезной аналогия с файловой системой, в которой каталоги могут рассматриваться как контейнеры, содержащие внутри себя логически сгруппированные файлы. Контейнер, как и любой объект, имеет атрибуты и относится к определенному классу объектов.

OU не только являются логической группой объектов, но и сами могут быть включены в другие организационные единицы. Группируя таким образом OU, можно образовывать древовидные структуры (**OU-деревья**), подобные древовидным структурам файловой системы. В Active Directory для каждого из доменов строится отдельное OU-дерево. В качестве корня в OU-дереве выступает не организационная единица, а объект, соответствующий домену, для которого построено это дерево. OU-дерево, показанное на рис. 11.14, включает два уровня организационных еденицы. На верхнем уровне расположены OU офиса филиала предприятия в Москве и OU офиса центрального отделения предприятия в Барнауле. В каждую из этих организационных единиц входит еще по две организационные единицы: computers и users для московского офиса, user и hardware для офиса в Барнауле.

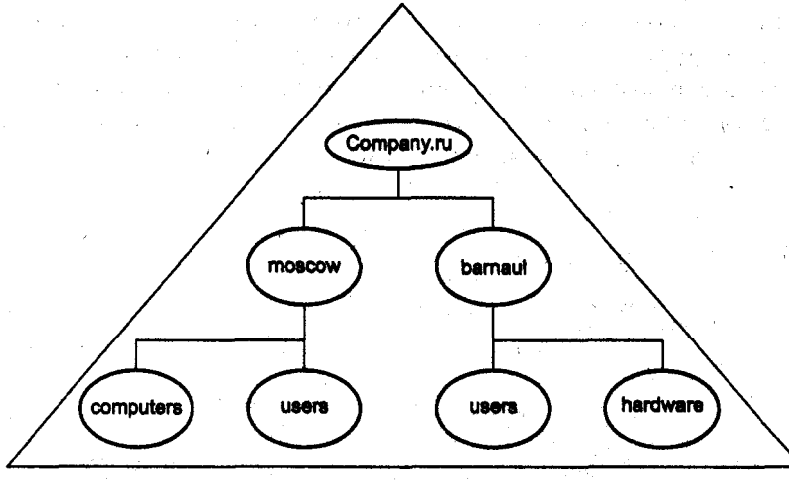


Рис. 11.14. OU-дерево

Иерархическое структурирование объектов позволяет эффективно выполнять *групповые операции* (переименование, уничтожение, определение правил доступа и т. п.) сразу для нескольких объектов, представленных как отдельной организационной единицей, так и ветвью OU-дерева, а также *делегировать администрирование* отдельными объектами и ветвями OU-дерева другим администраторам. В примере на рисунке для каждого из двух подразделений предприятия администратором была создана отдельная ветвь OU-дерева, что дает возможность назначить для этих двух подразделений, находящихся в разных часовых поясах, двух разных администраторов, которые смогут более эффективно управлять пользователями и ресурсами, относящимися к этим структурным единицам. В частности, локальным администраторам может быть передано право создавать пользовательские учетные записи, переустанавливать пароли и выполнять другие действия, которые проще выполнять в непосредственной близости к пользователям.

Иерархия доменов. Доверительные отношения

Домены являются более независимыми структурными единицами, чем организационные единицы внутри домена. В частности, именно пределами домена, а не OU ограничиваются действие политики паролей (password policies), устанавливающей длину и другие параметры паролей пользователей, и действие политики блокировки учетных записей (account lockout policies), определяющей функции администратора в тех случаях, когда пользователь забыл или не смог правильно набрать свой пароль, и его учетная запись была заблокирована. Разграничение функций по администрированию на уровне доменов происходит очень естественно благодаря тому, что каждый домен имеет встроенные группы Administrators и Server Operators, членам которых разрешено совместно использовать папки и форматировать диски только на контроллерах своего домена.

Такие свойства доменов делают привлекательной идею представления сети в виде нескольких доменов. Наиболее часто используемой многодоменной структурой является **дерево доменов** (рис. 11.15).

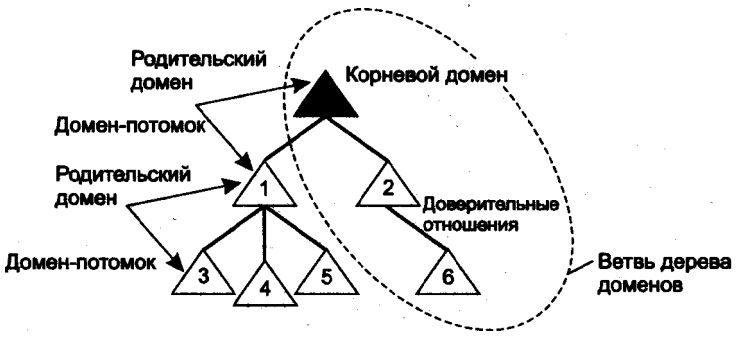


Рис. 11.15. Дерево доменов

На рисунке показано трехуровневое дерево доменов. Каждый домен условно изображен в виде треугольника. Упорядочение доменов по уровням иерархии происходит аналогично упорядочиванию каталогов файловой системы. Первый по времени создания домен становится **корнем дерева**. Для следующего создаваемого домена, называемого **потомком**, корневой домен является **родительским**. Далее при создании каждого последующего домена необходимо выбрать, какой из существующих доменов будет его родительским доменом. Корневой домен на рисунке имеет двух потомков: домен 1 и домен 2. Каждый из них в свою очередь является родительским доменом для доменов третьего уровня. Домены 3, 4 и 5 — потомки домена 1, а домен 6 — потомок домена 2. Дерево доменов, показанное на рисунке, состоит из двух **ветвей**.

Иерархические отношения между доменами выражаются в том числе и в их именовании. Корневому домену при его создании присваивается DNS-имя, пусть это будет, например, company.ru (рис. 11.16). Все домены-потомки получают «в наследство» имя родительского домена, которое добавляется к их собственным именам, образуя DNS-имена следующего уровня. В нашем примере домен production имеет полное имя production.company.ru, а домен academic — имя academic.research.company.ru. Множество полученных таким образом имен доменов (и объектов) образует **пространство имен дерева доменов**.

Между каждым новым доменом-потомком и его родительским доменом автоматически устанавливаются так называемые **доверительные отношения**. Установление доверительных отношений включает передачу части доменного справочника присоединяемого домена в глобальный каталог родительского домена, после чего он становится общим для всего дерева доменов. Глобальный каталог создается автоматически во время установки первого контроллера в первом домене сети. Первоначально в главном каталоге размещается полная копия всех объектов, относящихся к этому первому домену. В результате создания новых доменов к главному каталогу прибавляются частичные копии баз данных каждого из этих доменов. Эти копии содержат атрибуты объектов, позволяющие определить местонахождение соответствующих объектов, а значит, дающие пользователям возможность получать доступ к объектам не только своих, но и других доменов.

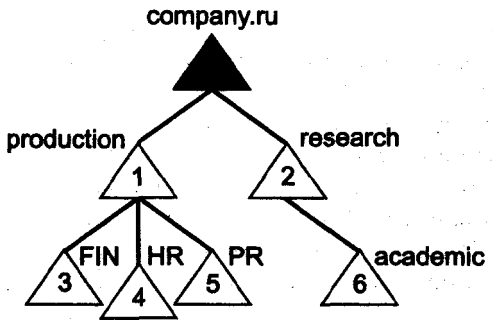


Рис. 11.16. Пространство имен дерева доменов

Для обеспечения безопасности в процедуру установления доверительных отношений вовлечена система аутентификации Kerberos[1].

Установление доверительных отношений между двумя доменами дает возможность:

- пользователю (или группе пользователей) одного домена *получать доступ к ресурсам другого домена* (это означает, что нет необходимости создавать две учетные записи в этих двух доменах для одного и того же пользователя);
- *администрировать один домен из другого домена*, для чего необходимо включить в группу пользователей, наделенную правами администрирования домена, пользователей из другого домена.

В Active Directory доверительные отношения, устанавливаемые по умолчанию между родительским доменом и доменом-потомком, являются транзитивными и двусторонними.

Доверительные отношения **транзитивны**, если из того, что *A* доверяет *B*, а *B* доверяет *C*, автоматически следует, что *A* доверяет *C*. Отношения являются **двусторонними**, если из того, что *A* доверяет *B*, следует, что *B* доверяет *A*.

Отсюда следует, что все домены дерева связаны друг с другом двусторонними транзитивными доверительными отношениями.

ПРИМЕЧАНИЕ В Active Directory OU-деревья разных доменов никак не связаны между собой в отличие, например, от справочной службы NDS компании Novell, в которой OU-деревья всех доменов сети образуют единое дерево.

Active Directory может иметь и более сложную доменную структуру, состоящую из нескольких деревьев, называемую **лесом** (рис. 11.17). Все домены леса, так же как и домены дерева, связаны двусторонними транзитивными доверительными отношениями, то есть имеют общий глобальный каталог. Следовательно, пользователи сети, имеющей доменную структуру в виде леса, могут быть наделены правом доступа к любым ресурсам любого домена.

Корневые домены каждого дерева, входящего в лес, получают независимые друг от друга DNS-имена, порождающие непересекающиеся пространства имен. То есть в отношении именования доменов (и объектов) деревья леса равноправны между собой.

В то же время корневые домены деревьев, составляющих лес, относятся к разным уровням иерархии, а именно, домен, который по времени был создан раньше (пусть это домен comp.ru), становится **корневым доменом леса** и находится на более высоком уровне иерархии, нежели созданный позже домен branch.ru. Корневой домен леса отличается от всех корневых доменов деревьев, составляющих лес, тем, что только в корневом домене леса имеются встроенные группы Enterprise Admins и Schema Admins, члены которых наделены правом администрирования в пределах леса. Однако, несмотря на более низкое поло-

[1] Схема работы Kerberos описана в главе 12.

жение в иерархии, было бы ошибкой интерпретировать фрагмент леса, порождаемый доменом branch.ru, как ветвь дерева comp.ru. Между доменами branch.ru и comp.ru нет отношений родитель-потомок и, как следствие, у них разные пространства имен.

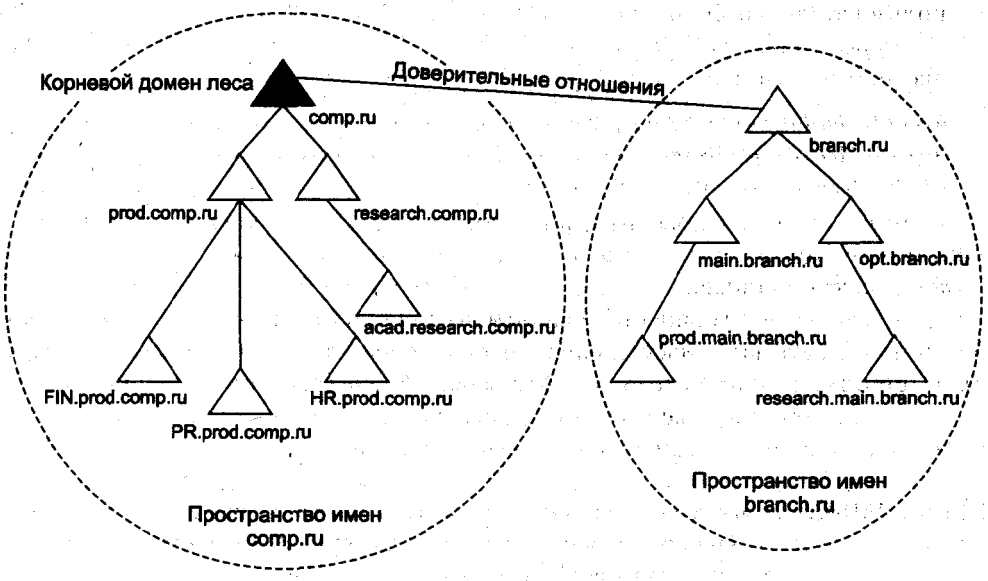


Рис. 11.17. Лес доменов

Еще одной доменной структурой, используемой в Active Directory, является модель **множественных лесов**. Эта модель может оказаться полезной для крупных транснациональных корпораций, образованных в результате слияния нескольких предприятий, а также в тех случаях, когда организация представляет собой объединение нескольких крупных и достаточно независимых подразделений. Являясь наиболее децентрализованной структурой, модель множественных лесов состоит из нескольких изначально совершенно не связанных между собой лесов. Между доменами разных лесов могут быть установлены контролируемые администратором доверительные отношения, однако это происходит не автоматически, как при построении деревьев и лесов, а в результате некоторых дополнительных действий администратора.

Пространство имен

Пространство имен (namespace) справочной службы — это множество имен, однозначно соответствующих всем объектам и их атрибутам, хранящимся в данной справочной службе.

Пространство имен является необходимым компонентом любой справочной системы, позволяя по имени объекта (здесь мы используем слово «объект»

в широком смысле) находить информацию, связанную с этим объектом. Например, в адресных или телефонных справочниках, в которых информацию об адресе или телефоне человека можно найти, зная имя этого человека, пространством имен является множество фамилий. Этот пример является не совсем точным, так как в реальной жизни существуют однофамильцы, а, значит, соответствие между фамилиями и персонами, информация о которых представлена в справочнике, является неоднозначной. В компьютерных системах, в которых поиск информации выполняется автоматически, однозначность необходима. Так, множество имен файлов, сгенерированных на основе принятых в файловой системе правил, однозначно определяет существующие в системе файлы. Другим примером является система DNS, используемая для именования узлов в IP-сети, она также включает в себя пространство DNS-имен, однозначно соответствующих IP-адресам.

Рассмотрим, как формируется пространство имен Active Directory. В Active Directory поддерживается несколько форм записи имени объекта. Мы остановимся на системе записи имен, принятой в *стандарте LDAP*. Согласно требованиям протокола LDAP, имя объекта должно представлять собой последовательность имен всех компонентов иерархии, лежащих на пути от объекта до корня дерева.

В Active Directory, как и в любой иерархической системе, имеющей древовидную структуру, имеется *единственный* путь от корня дерева до его конечного элемента (листа). Уникальность путей ко всем листьям в древовидных структурах делает уникальными имена, полученные путем перечисления имен транзитных узлов, лежащих на этих путях. Такие имена, однозначно определяющие объекты в пределах всего дерева, часто называют полными (составными) именами. Ааналогом полного имени в Active Directory служит уже упоминавшееся отличительное имя[1] объекта, которое описывает путь от данного объекта до корневого домена.

Другой тип имени, используемый в Active Directory, называется **относительным отличительным именем**[2] (Relative Distinguished Name, RDN). Это имя представляет собой компонент отличительного имени объекта, однозначно определяющий объект в пределах контейнера. Это имя подобно краткому имени каталога или файла и представляет собой значение одного атрибута (очень редко нескольких атрибутов) данного объекта, обеспечивающего уникальность этого объекта в контейнере. Из всего этого следует, что отличительное имя объекта есть цепочка относительных отличительных имен объектов, находящихся на пути от объекта до корня.

Компактность относительного имени делает его удобным для применения в пользовательском интерфейсе, хотя в LDAP-пакетах всегда указывается отличительное имя.

---

[1] Вы можете встретить и другой вариант перевода — «различающееся имя».

[2] Не путать с относительным именем файла, которое определяет положение файла в дереве относительно текущего каталога. В Active Directory понятие текущего каталога (домена, организационной единицы) при именовании объектов не используется.

Особенность именования объектов в Active Directory, отличающая ее, например, от именования файлов, состоит в неоднородности узлов дерева объектов. Действительно, в то время как в дереве файловой системы все транзитные узлы на пути от корневого каталога к файлу имеют единую природу — все они являются каталогами, в Active Directory путь, соединяющий объект с корневым каталогом, делится на две части: одна часть пролегает по дереву доменов, а другая проходит по дереву организационных единиц, находящемуся «внутри домена» (рис. 11.18). То есть компоненты полного имени объекта имеют разную природу. Этот факт отражается в синтаксисе имен Active Directory — имя строится путем приписывания соответствующих префиксов к компонентам имени. К числу основных префиксов[1] относятся:

- *DC* (*Domain Component*) — компонент является относительным именем домена;
- *OU* (*Organizational Unit*) — компонент является относительным именем организационной единицы;
- *CN* (*Common Name*) — компонент относится к объекту любого типа, не являющемуся ни организационной единицей, ни доменом.

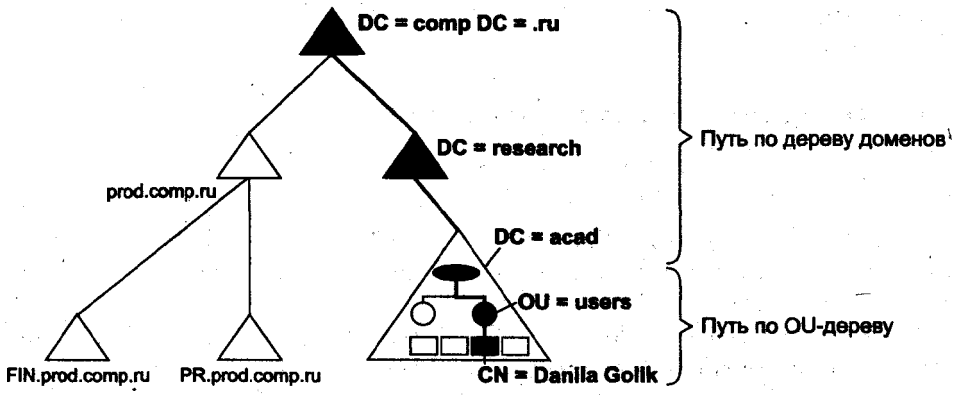


Рис. 11.18. Именование объектов в стандарте LDAP

Посмотрим, как будет выглядеть отличительное имя в стандарте LDAP для объекта, представляющего, например, пользователя Данилу Голика, который был помещен администратором в организационную единицу, имеющую относительное имя users. Эта организационная единица относится к домену, DNS-имя которого acad.research.comp.ru. В соответствии с принятым синтаксисом перечисление компонентов начинается от листа OU-дерева, которым является объект-пользователь, до корня дерева доменов. Каждое относительное имя предваряется соответствующим префиксом. В результате получаем:

```
CN = Danila Golik, OU = users, DC = acad, DC = research, DC = comp, DC = ru
```

[1] Стандарт LDAP предусматривает и другие префиксы: C (страна), O (организация) и L (местонахождение), однако в Active Directory они не используются.

Как уже было сказано, в Active Directory поддерживается несколько форм записи имени объекта, например, для приведенного имени альтернативой может служить одно из следующих имен:

```
acad.research.comp.ru/users/Danila Golik
DC=ru/DC=comp/DC=research/DC=acad/OU=users/CN=Danila Golik
```

Получив от клиента отличительное имя, справочная служба должна установить контакт с контроллером того домена, где находится интересующий клиента объект. Из имени объекта легко определяется DNS-имя контроллера, а для определения IP-адреса Active Directory использует систему доменных имен (DNS).

Контроллеры доменов Active Directory могут отличаться друг от друга набором предоставляемых услуг. Для того чтобы пользователи и приложения могли находить именно те контроллеры, на которых установлены необходимые им сервисные программные модули, контроллеры регистрируют имена своих сервисов в системе DNS, которая по запросам клиентов сообщает им о местонахождении (IP-адресах) серверов и сервисных программ Active Directory. В некоторых случаях система DNS может вернуть IP-адреса нескольких контроллеров, предоставляющих эту услугу. Используя дополнительную информацию о месте расположения контроллеров, клиентская станция выбирает ближайший к ней контроллер.

Репликация в Active Directory

В Active Directory база данных каждого домена, как правило, существует в виде нескольких копий, что дает возможность повысить надежность и производительность справочной службы. Однако это порождает *проблему поддержания идентичности копий* БД на всех контроллерах в пределах домена. Любое изменение, внесенное в одну из копий БД клиентом справочной службы (приложением, администратором или пользователем), немедленно приводит к нежелательному рассогласованию баз данных.

Для того чтобы обеспечить динамическое совпадение копий, хранящихся на разных контроллерах домена, Active Directory использует механизм **репликации**, заключающийся в периодическом переносе всех изменений, произведенных на одном контроллере, на все остальные контроллеры этого домена.

Active Directory использует механизм репликации также для поддержания идентичности всех реплик глобального каталога в пределах сети.

В зависимости от того, насколько долго сохраняется в системе возникшее после внесения изменений рассогласование копий, различают два подхода к репликации — синхронный и асинхронный.

- *Синхронная* репликация обеспечивает строгую целостность данных. Это означает, что в любой момент гарантируется совпадение всех копий БД.
- *Асинхронная* репликации обеспечивает нестрогую целостность данных, которая допускает наличие временно́й задержки между внесением изменений в одну из реплик и их отражением в остальных репликах.

В Active Directory используется асинхронный подход. Изменения, внесенные в базу данных какого-либо контроллера домена, передаются в базы данных других контроллеров этого домена спустя некоторое время. Такой подход дает возможность «накопить» изменения одной копии, прежде чем передавать произошедшие изменения всем контроллерам сайта. Благодаря этому уменьшается сетевой трафик и загрузка контроллеров, однако в течение периода задержки реплики базы данных, хранящиеся на разных контроллерах, будут отличаться друг от друга.

Доменная репликация Active Directory может выполняться с учетом физических характеристик сети. Для этого администратор во время установки Active Directory должен представить сеть в виде совокупности нескольких связанных между собой сайтов. Как было ранее сказано, сайт представляет собой часть сети, узлы которой связаны между собой высокоскоростными LAN-соединениями. Между собой сайты связаны сравнительно медленными WAN-соединениями. Отсюда ясно, что процесс перенесения изменений с одного контроллера домена на другой отличается в зависимости от того, происходит этот процесс внутри или между сайтами. В первом случае мы имеем дело с *внутрисайтовой* репликацией, во втором — *межсайтовой*. Внутридоменная репликация является частным случаем внутрисайтовой репликации, когда в домене определен только один сайт.

Внутрисайтовая репликация выполняется, как правило, значительно чаще, чем межсайтовая. Так, по умолчанию репликация в пределах сайта выполняется каждые 15 секунд, а репликация между сайтами — каждые 3 часа. Администратор имеет возможность управлять периодичностью репликации. Причем параметры репликации для различных линий связи сети могут быть выбраны различными.

Внутри сайта Active Directory использует *распределенную* репликацию. Это означает, что при возникновении изменений на любом из контроллеров эти изменения будут скопированы на все остальные. То есть все контроллеры сайта при таком подходе равноправны[1].

ПРИМЕЧАНИЕ В справочной службе Windows NT был использован другой подход, в соответствии с которым один из контроллеров домена назначался главным и играл роль посредника: все изменения, произошедшие на каком-либо контроллере, копировались на главный контроллер и только затем переносились с главного контроллера на все остальные контроллеры домена.

При передаче реплицируемых данных между сайтами Active Directory использует *сжатие*.

Для выполнения репликации между сайтами система автоматически выбирает в каждом сайте по одному контроллеру, который будет играть особую роль

[1] Строго говоря, контроллеры домена не являются функционально эквивалентными: в некоторых случаях в процессе репликации для выполнения особых операций требуется их специализация.

в этом процессе. Эти выделенные контроллеры, называемые **серверами-плацдармами** (bridgeheads), используются для создания двухточечных связей между сайтами. Каждый сайт может быть соединен с одним или несколькими другими сайтами путем установления связи между соответствующими парами выделенных контроллеров. Связи устанавливаются администратором в целях создания путей, по которым будет происходить репликация.

На рис. 11.19 показаны пять сайтов, два из которых, *A* и *B*, имеют по одной связи, сайты *D* и *E* — по две связи, а сайт *C* — четыре связи. Передача обновлений может проходить по любому возможному маршруту, например, сайт *D* может связываться с сайтом *C* как непосредственно по связи *D-C*, так через транзитный узел *E* по маршруту *D-E-C*.

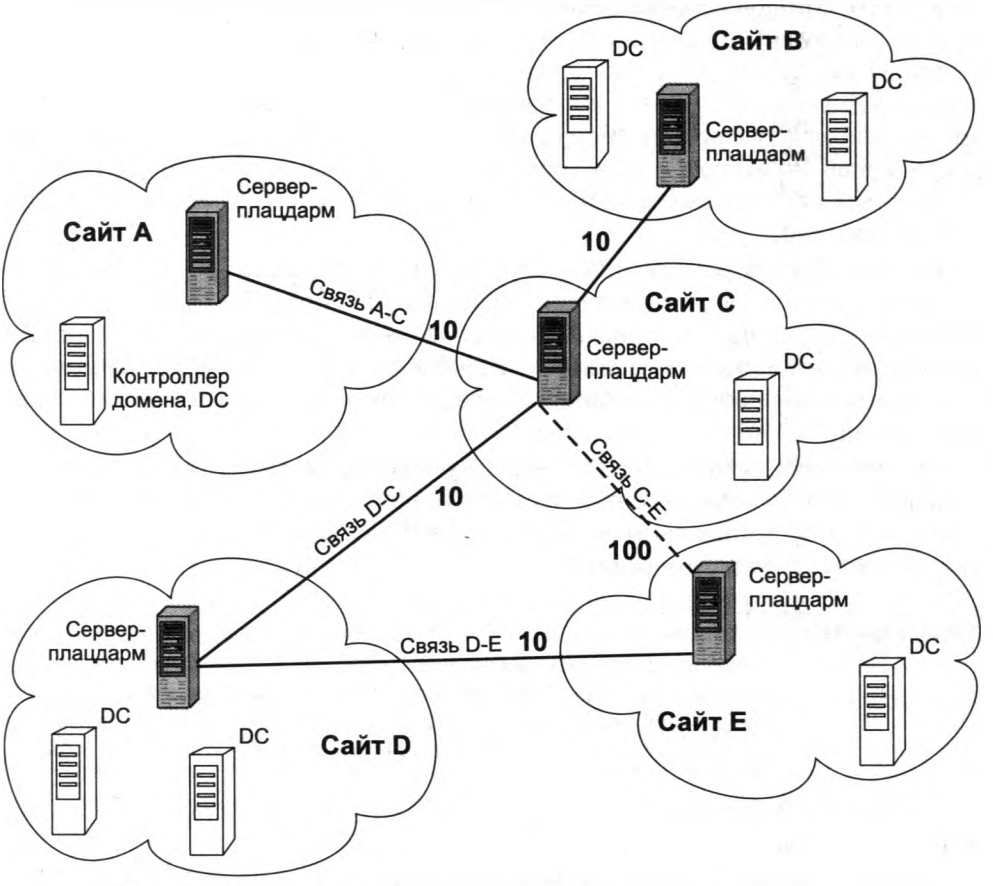


Рис. 11.19. Топология межсайтовой репликации

Для каждой установленной связи администратор может сообщить Active Directory дополнительную информацию, которая в случае нескольких возможных

путей репликации помогает системе выбрать наиболее подходящий маршрут. К числу характеристик связей относятся расписание, интервал и стоимость.

- *Расписание* (schedule) — определяет период времени, когда связь доступна для репликации. По умолчанию задается значение always (всегда), однако администратор, принимая во внимание загруженность сети или другие обстоятельства, может разрешить использование этой связи только в определенные часы.
- *Интервал* (interval) — задает периодичность межсайтовой репликации. По умолчанию интервал равен 3 часам, администратор может изменять его в большую и меньшую сторону.
- *Стоимость* (cost) — это характеристика, которая выражается в условных единицах и отражает скорость передачи данных по связи. Стоимость является аналогом такой характеристики линии связи, как расстояние, используемой протоколами маршрутизации, рассмотренными в главе 9. Стоимость каждой связи входит в качестве слагаемого при вычислении стоимости пути репликации. В автоматической процедуре выбора пути Active Directory всегда предпочитает пути с низкой стоимостью путям с высокой стоимостью. Исходя из этого, администратор приписывает некие условные числа связям. Пусть, например, в сети на рис. 11.19 все связи между сайтами, кроме связи *C-E*, являются высокоскоростными каналами T1, а связь *C-E* представляет собой медленный канал 56K. Администратор хотел бы использовать низкоскоростную связь *C-E* для репликации только в качестве резервной связи, если откажет какая-либо из двух скоростных связей *C-D* или *D-E*. Для реализации этого своего решения администратор должен назначить основным связям меньшую стоимость, скажем 10, а резервной — большую, скажем 100. Стоимостные коэффициенты выбираются достаточно произвольно, в данном случае необходимо обеспечить, чтобы стоимость пути *E-D-C*, равная сумме стоимостей связей *C-D* (10) или *D-E* (10), была меньше стоимости резервного, хотя и прямого, пути *E-C* (100), что и было сделано. Единственное формальное ограничение состоит в том, что числа должны выбираться из диапазона от 1 до 32 767.

Помимо определения стоимости связей, администратор может директивно определить предпочтительные пути для выполнения репликации. Все это дает возможность администратору эффективно управлять процессом репликации в Active Directory.

Межсетевое взаимодействие

Только небольшое количество сетей обладает однородностью (гомогенностью) программного и аппаратного обеспечения. Однородными чаще всего являются сети, которые состоят из небольшого количества компонентов одного производителя.

Более общим случаем являются **неоднородные (гетерогенные)** сети, состоящие из разнотипных рабочих станций, операционных систем и приложений, в которых для взаимодействия между компьютерами используются различные типы коммуникационного оборудования — коммутаторов и маршрутизаторов, а также разные коммуникационные протоколы.

Одной из причин неоднородности является эволюционный характер развития любой большой сети. Сеть, как правило, не строится «с нуля» и не возникает в одно мгновение. Поскольку жизнь не стоит на месте, за время существования сети появляются привлекательные технологические новшества, и создатели сети вынуждены вносить в нее изменения, которые часто требуют согласования новых технологий с уже имеющимися старыми.

Неоднородность — это также естественное следствие того, что люди, ответственные за функционирование сети, стремятся выбрать программные и аппаратные средства, которые наилучшим образом отвечают поставленным целям. Часто оказывается, что средство, которое хорошо подходит для решения одной задачи, совсем не обязательно также хорошо работает для решения другой. Поэтому и появляются в сети коммуникационное оборудование разных технологий и разных производителей, Ethernet-сегменты соседствуют с PPP-сегментами, серверы работают под управлением Windows, Unix или Mac OS, а в системах IP-телефонии применяются конкурирующие протоколы H.323 и SIP.

Отсюда следует важное требование, предъявляемое к современных сетевым ОС, — *способность к интеграции с другими ОС* и разнообразным сетевым оборудованием.

Основные подходы к организации межсетевого взаимодействия

На первый взгляд выражение «организация межсетевого взаимодействия» может показаться парадоксальным. Действительно, если сети взаимодействуют, значит, их компьютеры связаны между собой, и следовательно, они все вместе образуют сеть. Тогда что же понимается под сетями, взаимодействие которых надо организовать? Понимается ли под этим совокупность компьютеров, которые работают под управлением одной и той же сетевой операционной системы, например Microsoft Windows XP или Unix? Или это те компьютеры, которые связаны между собой средствами одной и той же базовой сетевой технологии, например Ethernet или Token Ring? И наконец, может быть, здесь имеются в виду сети в терминах протокола сетевого уровня, то есть части большой сети, разделенные между собой маршрутизаторами?

Ни то, ни другое, ни третье. В контексте межсетевого взаимодействия под термином «сеть» понимается *совокупность компьютеров, общающихся друг с другом с помощью единого стека протоколов*. Средства взаимодействия компьютеров в сети организованы в виде многоуровневой структуры — стека протоколов. В однородной сети все компьютеры используют один и тот же стек. Проблема возникает тогда, когда требуется организовать взаимодействие компьютеров, принадлежащих разным сетям в указанном смысле, то есть организо-

вать взаимодействие компьютеров, на которых поддерживаются отличающиеся стеки коммуникационных протоколов.

Проблема межсетевого взаимодействия может иметь разные внешние проявления, но суть ее одна — *несовпадение коммуникационных протоколов*. В худшем случае не совпадают протоколы всех уровней, это как раз и происходит, когда в сетях используются различные стеки протоколов. Однако проблема межсетевого взаимодействия возникает и в том случае, когда в сетях не совпадает только протокол одного из уровней стека. Например, если в одной сети на канальном уровне работает технология Ethernrt, а в другой применяется протокол PPP, то непосредственно эти протоколы общаться и передавать между сетями кадры не могут. Как мы знаем, именно для решения такого рода проблем и были в свое время созданы протоколы сетевого уровня — IPX, IP и другие. Основная функция этих протоколов состоит в организации взаимодействия канальных протоколов различных сетей, как построенных на основе одной технологии, так и различных (более подробно об этом рассказывалось в главе 9). Однако идея использования протокола сетевого уровня перестает работать, когда в сети на этом уровне работают разные протоколы, например, одна сеть построена на основе протокола IP, а другая — IPX.

Равным образом проблема межсетевого взаимодействия может возникнуть и при объединении сетей, в которых используется один и тот же протокол сетевого уровня, и транспортные проблемы передачи пакетов успешно им решаются. Однако если компьютеры этих сетей применяют различные протоколы прикладного уровня, то для пользователей этих приложений проблема остается. Например, если в одной сети работает протокол пакетной телефонии стандарта H.323, а в другой — протокол SIP того же уровня и назначения, то пакетные телефоны этих сетей взаимодействовать не будут, даже если обе сети используют один и тот же стек протоколов TCP/IP, так как протоколы прикладного уровня в них отличаются[1].

Аналогичная ситуация возникает, когда клиентские и серверные компоненты какой-либо службы операционных систем разных сетей функционируют на базе разных протоколов прикладного уровня. Например, компьютеры, работающие под управлением ОС Microsoft Windows, по умолчанию используют для доступа к файлам протокол SMB, а компьютеры, работающие под управлением Unix, — протокол NFS. Конечно, эти сети могут сосуществовать, передавая данные через общие транспортные средства, но не предоставляя пользователям сервис общей разделяемой файловой системы. В том случае, когда потребуется обеспечить доступ к данным файлового сервера Windows 2003 клиентам Unix, администратор сети столкнется с необходимостью согласования сетевых служб.

[1] Нужно заметить, что на прикладном уровне в сети обычно работают несколько типов протоколов, так как каждый тип приложений использует собственный протокол, например, распределенная база данных — протокол SQL, служба Web — протокол HTTP и т. д. Поэтому несовпадение в описанном примере прикладного протокола пакетной телефонии не означает, что и другие типы приложений этих сетей должны решать проблему межсетевого взаимодействия. Если они построены на основе одних и тех же прикладных протоколов, то такой проблемы просто не существует.

Задачи устранения неоднородности имеют некоторую специфику и даже разные названия в зависимости от того, к какому уровню модели OSI они относятся. Задача объединения транспортных подсистем, отвечающих только за передачу сообщений, обычно называется задачей **межсетевого взаимодействия** (internetworking). Классическим подходом для ее решения является использование единого сетевого протокола, такого, например, как IP или IPX. Однако существуют ситуации, когда этот подход неприменим или нежелателен (эти ситуации далее рассмотрены).

Другая задача, называемая **операционной совместимостью** (interoperability), возникает при объединении сетей, использующих разные протоколы более высоких уровней — в основном прикладного и представительного. Будем называть ее задачей согласования сетевых служб операционных систем, так как протоколы прикладного и представительного уровней реализуются именно этими сетевыми компонентами.

Кардинальным решением проблемы межсетевого взаимодействия могло бы стать повсеместное *использование единого стека протоколов*. И такая попытка введения единого стека коммуникационных протоколов была сделана в 1990 году правительством США, обнародовавшем программу GOSIP (Government OSI Profile), в соответствие с которой стек протоколов OSI предполагалось сделать общим для всех сетей, устанавливаемых в правительственных организациях США. Однако массового перехода на стек OSI не произошло. В то же время в связи со стремительным ростом популярности Интернета стандартом де-факто стал стек протоколов TCP/IP, который сегодня установлен практически на любом компьютере. В результате степень неоднородности сетей, начиная с конца 90-х годов, существенно снизилась. Помимо появления единого протокола сетевого уровня, этому способствовали еще две тенденции: доминирование на канальном уровне технологии Ethernet и сокращение перечня популярных сетевых ОС до двух семейств — Microsoft Windows и Unix (Linux также принадлежат к последнему семейству, а последняя версия Mac OS X построена с использованием большого количества элементов кода Unix). Третий крупный игрок 90-х, Novell NetWare, хотя все еще и поставляется, но уже не оказывает существенного влияния на сетевой мир.

Технология Ethernet сегодня практически вытеснила из локальных сетей все остальные технологии канального уровня, такие как FDDI и Token Ring, которые были весьма популярны в 90-е годы. Сегодня экспансия Ethernet распространилась на глобальные сети, где она стала вытеснять такие традиционные протоколы, как HDLC и PPP. Анализ причин доминирования Ethernet выходит за рамки темы данной книги, однако результат такого доминирования очевиден — однородность сетей возросла, и транспортные проблемы построения составных сетей упростились. Нужно подчеркнуть, что необходимость в существовании сетевого протокола не отпадет даже в том случае, если все подсети составной сети будут построены на базе технологии Ethernet — разделение сети на подсети происходит не только из-за существования разных технологий канального уровня в каждой из подсетей. Такое разделение имеет смысл и в однородной в отношении канальной технологии среде, так как создает иерархию

и тем самым улучшает управляемость сети — общая теория систем и просто здравый смысл говорят, что большие однородные системы не эффективны, ими невозможно управлять, будь то государство или же компьютерная сеть.

Сокращение типов применяемых сетевых ОС сократило число применяемых прикладных протоколов и, соответственно, повысило степень однородности сетей на прикладном уровне.

Таким образом, тенденция к унификации сетевого мира налицо, однако означает ли это, что проблема межсетевого взаимодействия почти решена и на нее не стоит обращать внимания?

Правильный ответ — нет. Во-первых, для решения одной и той же задачи по-прежнему могут использоваться разные протоколы. Наиболее выпукло это проявляется на прикладном уровне, как это видно из приведенных ранее примеров. К тому же и на более низких уровнях иерархии протоколов не все так просто. Даже в условиях доминирования стека TCP/IP сегодня существует две его версии — IP v4 и IP v6, которые не могут непосредственно взаимодействовать друг с другом. Во-вторых, разнообразие является общим законом природы, так что никогда нельзя исключать вероятности появления новых технологий, протоколов и их стеков.

Поэтому можно сделать вывод, что средства межсетевого взаимодействия по-прежнему актуальны и таковыми, по всей вероятности, останутся.

Для решения проблем межсетевого взаимодействия были разработаны несколько общих подходов, к которым относятся:

- трансляция;
- мультиплексирование;
- инкапсуляция (туннелирование).

Трансляция

Трансляция обеспечивает согласование стеков протоколов путем преобразования сообщений, поступающих от одной сети, в формат сообщений другой сети. Транслирующий элемент, в качестве которого могут выступать, например, программный или аппаратный шлюз, мост, коммутатор или маршрутизатор, размещается между взаимодействующими сетями и служит посредником в их «диалоге». Термин **шлюз** в данном контексте подразумевает средство, выполняющее трансляцию протоколов верхних уровней, хотя в традиционной терминологии Интернета шлюзом (gateway) называют маршрутизатор.

В зависимости от типа транслируемых протоколов процедура трансляции может иметь разную степень сложности. Так, преобразование протокола Ethernet в протокол Token Ring сводится к нескольким несложным действиям, главным образом благодаря тому, что оба протокола ориентированы на единую схему адресации узлов. А вот трансляция протоколов сетевого уровня IP и IPX представляет собой гораздо более сложный интеллектуальный процесс, включающий не только преобразование форматов сообщений, но и отображение адресов сетей и узлов, различным образом трактуемых в этих протоколах.

Трансляция протоколов прикладного уровня включает отображение инструкций одного протокола на инструкции другого, что представляет собой сложную, логически неоднозначную интеллектуальную процедуру, сравнимую с работой переводчика с одного языка на другой. Например, в файловых службах операционных систем Microsoft Windows (протокол SMB) и Unix (протокол NFS) существуют как совпадающие по назначению операции, например `open` и `close`, открывающие и закрывающие файл для доступа, так и операции, уникальные для каждой ОС, например, для операции `SMB_COM_OPEN_PRINT_FILE` ОС Windows, создающей файл принт-спулинга на сервере, нет прямого аналога в протоколе NFS ОС Unix. В то же время в протоколе NCP отсутствует обычное для протокола NFS понятие монтирования файловой системы.

На рис. 11.20 показан размещенный на компьютере 2 шлюз, который согласовывает протоколы клиентского компьютера 1 в сети *A* с протоколами компь-

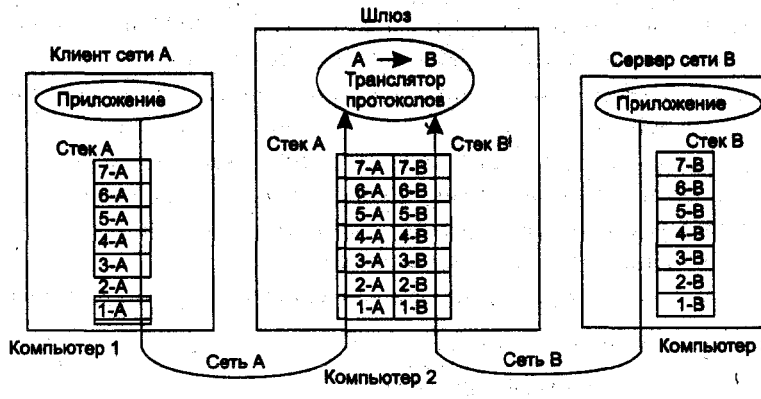


Рис. 11.20. Принципы функционирования шлюза

ютера 3 в сети *B*. Допустим, что стеки протоколов в сетях *A* и *B* различаются на всех уровнях. В шлюзе установлены оба стека протоколов.

Запрос от прикладного процесса клиентского компьютера сети *A* поступает на прикладной уровень его стека протоколов. В соответствие с этим протоколом на прикладном уровне формируются соответствующий пакет (или несколько пакетов), в которых передается запрос на предоставление услуг некоторому серверу сети *B*. Пакет прикладного уровня перемещается вниз по стеку компьютера сети *A*, обрастая заголовками нижележащих протоколов, а затем передается по линиям связи в компьютер 2, то есть в шлюз.

В шлюзе обработка поступивших данных идет в обратном порядке, от протокола самого нижнего к протоколу самого верхнего уровня стека *A*. Затем пакет прикладного уровня стека сети *A* преобразуется (транслируется) в пакет прикладного уровня серверного стека сети *B*. Алгоритм преобразования пакетов зависит от конкретных протоколов и, как уже было сказано, может быть достаточно сложным. В качестве общей информации, позволяющей корректно провести трансляцию, может использоваться, например, информация о символьных

именах сервера и запрашиваемого у сервера ресурса (в частности, это может быть имя каталога файловой системы). Преобразованный пакет от верхнего уровня стека сети *B* передается к нижним уровням в соответствии с правилами этого стека, а затем по физическим линиям связи в соответствии с протоколами физического и канального уровней сети *B* поступает в другую сеть к нужному серверу. Ответ сервера преобразуется шлюзом аналогично.

Примером шлюза, транслирующего протоколы прикладного уровня, является компонент Gateway for NFS, который обеспечивает клиентам Windows прозрачный доступ к каталогам и файлам серверов Unix, поддерживающий сетевую файловую систему NFS (Network File System). Шлюз устанавливается на том же компьютере, на котором установлен сервер Windows 2000 или 2003. Между шлюзом и серверами Unix реализуется связь по протоколу NFS.

Для доступа к файлам NFS клиенты Windows, используя свой «родной» протокол SMB, обращаются к серверу Windows 2000/2003, на котором работает шлюз NSF. Шлюз виртуализирует разделяемые каталоги серверов Unix: они выглядят для SMB-клиентов точно так же, как и разделяемые каталоги сервера Windows 2000/2003 (рис. 11.21). Если запрос, поступивший на сервер Windows 2000/2003, относится к ресурсам серверов Unix, то он переадресуется шлюзу, который транслирует его в понятный для сервера Unix вид и передает по протоколу NFS соответствующему серверу. При этом шлюз выступает по отношению к серверам Unix как клиент-пользователь.

Таким образом, шлюз реализует взаимодействие «многие ко многим».

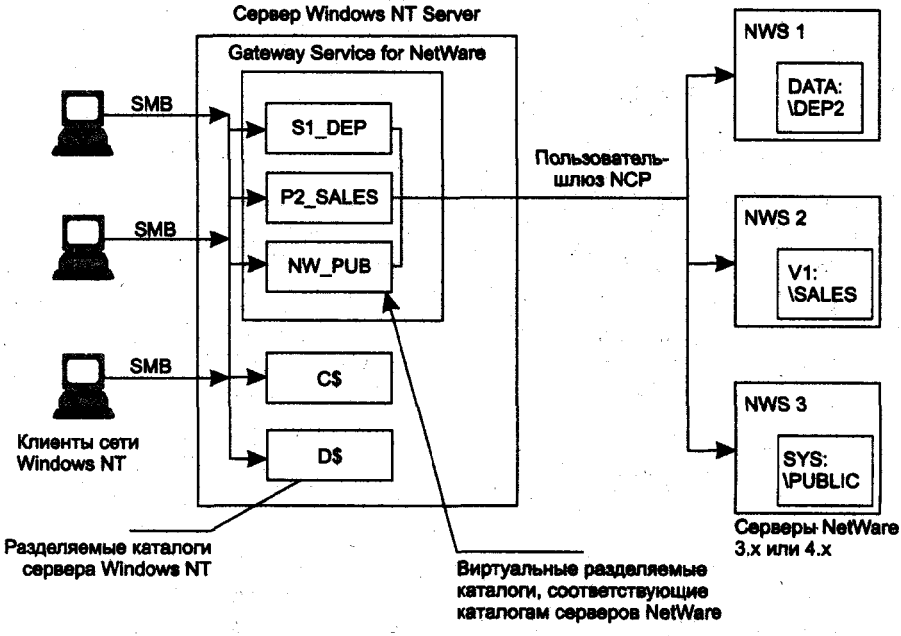


Рис. 11.21. Шлюз Gateway for NSF

Достоинство шлюзов состоит в том, что они сохраняют в неизменном виде программное обеспечение на клиентских компьютерах. Пользователи работают в привычной среде и могут даже не заметить, что получают доступ к ресурсам другой сети. Однако как и всякий централизованный ресурс, шлюз снижает надежность сети. Кроме того, при обработке запросов в шлюзе возможны относительно большие временные задержки, во-первых, из-за затрат времени на собственно процедуру трансляции, во-вторых, из-за задержек запросов в очереди к разделяемому всеми клиентами шлюзу, особенно если запросы поступают с большой интенсивностью. Это делает шлюз плохо масштабируемым решением. Правда, ничто не мешает установить в сети несколько параллельно работающих шлюзов.

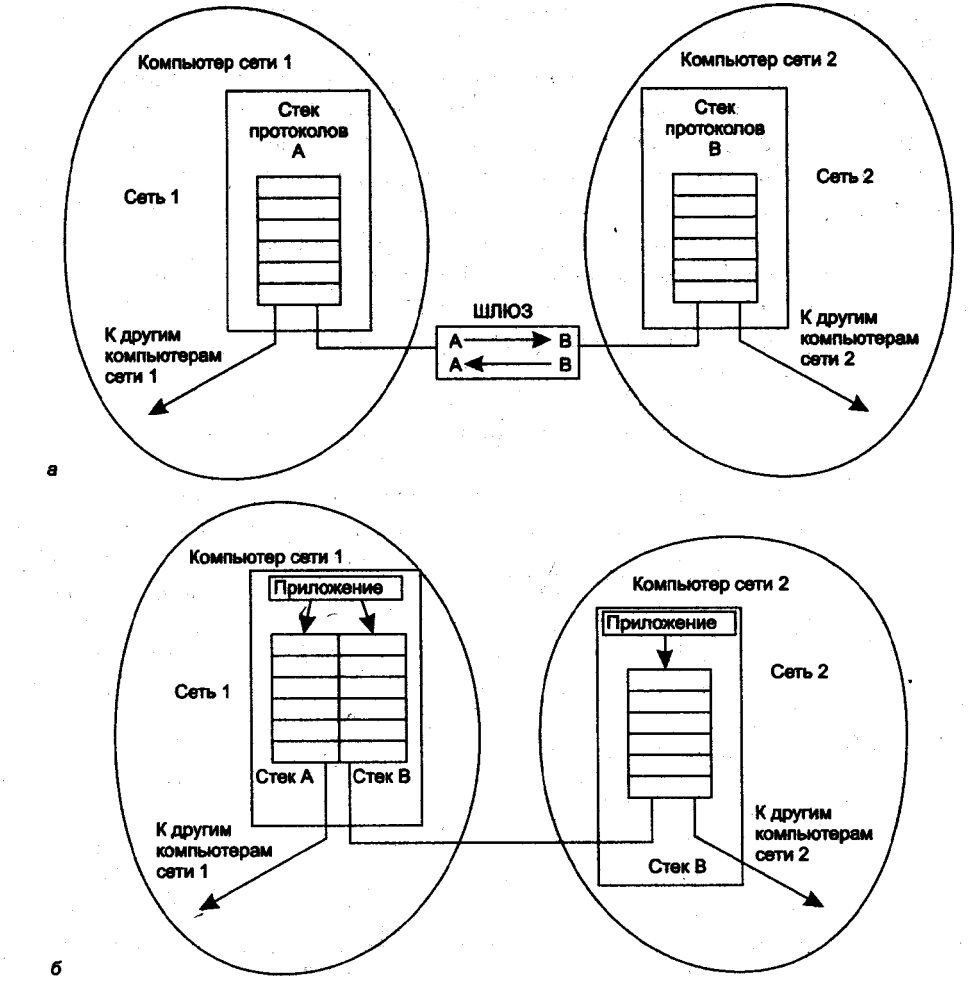


Рис. 11.22. Два варианта согласования протоколов: трансляция протоколов (*а*), мультиплексирование стеков протоколов (*б*)

Мультиплексирование стеков протоколов

Другой подход к согласованию протоколов получил название **мультиплексирования стеков протоколов**. Он заключается в том, что в сетевое оборудование или в операционные системы серверов и рабочих станций встраиваются несколько стеков протоколов. Это позволяет клиентам и серверам выбирать для взаимодействия тот протокол, который является для них общим.

Сравнивая мультиплексирование с уже рассмотренной трансляцией протоколов, можно заметить, что взаимодействие компьютеров, принадлежащих разным сетям, напоминает общение людей, говорящих на разных языках (рис. 11.22). Для достижения взаимопонимания они также могут использовать два подхода: пригласить переводчика (аналог транслирующего устройства) или перейти на язык собеседника, если они им владеют (аналог мультиплексирования стеков протоколов).

При мультиплексировании стеков протоколов на один из двух взаимодействующих компьютеров с различными стеками протоколов помещается коммуникационный стек другого компьютера. На рис. 11.23 приведен пример взаимодействия клиентского компьютера сети *B* с сервером в своей сети и сервером сети *A*, работающей со стеком протоколов, полностью отличающимся от стека сети *B*. В клиентском компьютере реализованы оба стека. Для того чтобы запрос от прикладного процесса был правильно обработан и направлен через соответствующий стек, необходимо наличие специального программного элемента — **мультиплексора протоколов**, называемого также **менеджером** протоколов. Менеджер должен уметь определять, к какой сети направляется запрос клиента. Для этого может использоваться служба имен сети, в которой отмечается принадлежность того или иного ресурса определенной сети с соответствующим стеком протоколов.

При использовании технологии мультиплексирования структура коммуникационных средств операционной системы может быть и более сложной. В общем случае на каждом уровне вместо одного протокола появляется целый набор протоколов, и может существовать несколько мультиплексоров, выполняющих коммутацию между протоколами разных уровней. Например, рабочая станция, стек протоколов которой показан на рис. 11.24, может через один сетевой адаптер получить доступ к сетям, работающим по протоколам NetBIOS, IP, IPX. Данная рабочая станция может быть клиентом сразу нескольких файловых серверов: NetWare (NCP), Windows NT (SMB) и Sun (NFS).

Предпосылкой для развития технологии мультиплексирования стеков протоколов стало появление строгих открытых описаний протоколов различных уровней и межуровневых интерфейсов, так что фирма-производитель при реализации «чужого» протокола может быть уверена, что ее продукт будет корректно взаимодействовать с продуктами других фирм по данному протоколу, этот протокол корректно впишется в стек и с ним будут нормально взаимодействовать протоколы соседних уровней.

Мультиплексирование протоколов реализует отношение «один ко многим», то есть один клиент с дополнительным стеком может обращаться ко всем серверам,

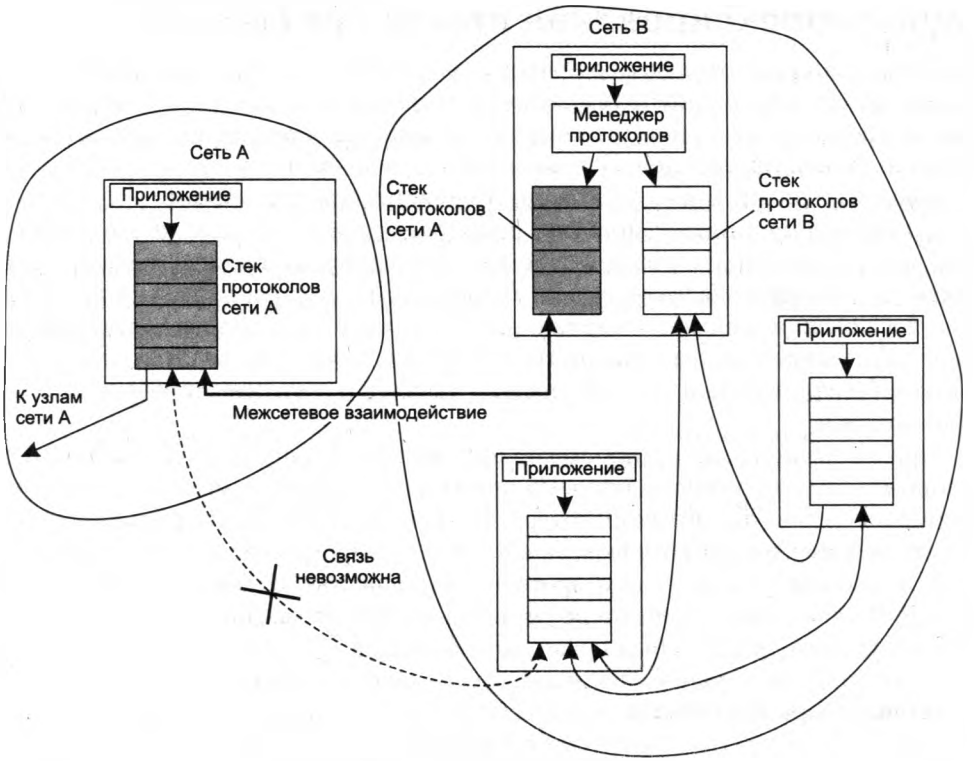


Рис. 11.23. Мультиплексирование стеков

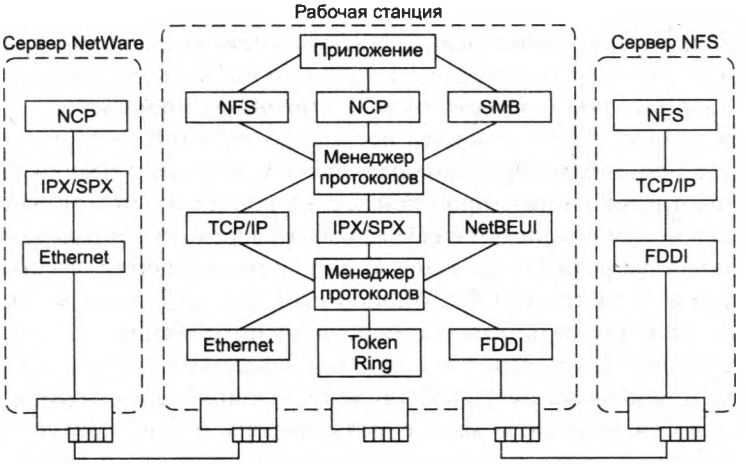


Рис. 11.24. Мультиплексирование протоколов

поддерживающим этот стек, или один сервер с дополнительным стеком может предоставлять услуги многим клиентам.

При использовании мультиплексоров протоколов существует два варианта размещения дополнительного стека протоколов — на одном или на другом взаимодействующем компьютере. Если дополнительный стек устанавливается на сервере, то этот сервер становится доступным для всех клиентов с этим стеком. При этом нужно тщательно оценивать влияние установки дополнительного продукта на производительность сервера.

Аналогично, дополнительный стек на клиенте дает ему возможность устанавливать связи с другими серверами, использующими этот стек протоколов. При размещении дополнительного стека на клиентах вопросы производительности не так важны. Здесь более важными являются ограничения таких ресурсов, как память и дисковое пространство клиентских машин, а также затраты труда администратора на установку и поддержание дополнительных стеков в работоспособном состоянии на большом числе компьютеров.

Заметим, что при организации взаимодействия двух разнородных сетей в общем случае нужно решать две задачи согласования служб (рис. 11.25):

- обеспечение доступа клиентам сети *A* к ресурсам сети *B*;
- обеспечение доступа клиентам сети *B* к ресурсам сети *A*.

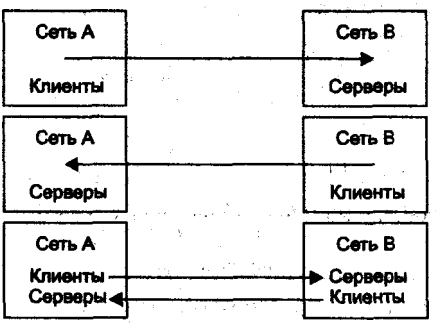


Рис. 11.25. Направления согласования служб

Эти задачи независимы и их можно решать отдельно. В некоторых случаях требуется полное решение, например, чтобы пользователи Unix-машин имели доступ к ресурсам серверов сети NetWare, а пользователи персональных машин имели доступ к ресурсам Unix-хостов; в других же случаях достаточно обеспечить доступ клиентам сети NetWare к ресурсам сети Unix. Большинство имеющихся на рынке продуктов обеспечивает только однонаправленное согласование прикладных служб.

Рассмотрим возможные варианты размещения программных средств, реализующих взаимодействие двух сетей, основанных на мультиплексировании протоколов. Введем обозначения: *C* — сервер, *K* — клиент, Δ — дополнительный протокол (или протоколы), предоставляющий возможности межсетевого взаимодействия. Оба возможных варианта *однонаправленного* взаимодействия *A→B* реализуются либо путем добавления нового стека к клиентам сети *A* (рис. 11.26 *а*), либо путем присоединения «добавки» к серверам сети *B* (рис. 11.26, *б*).

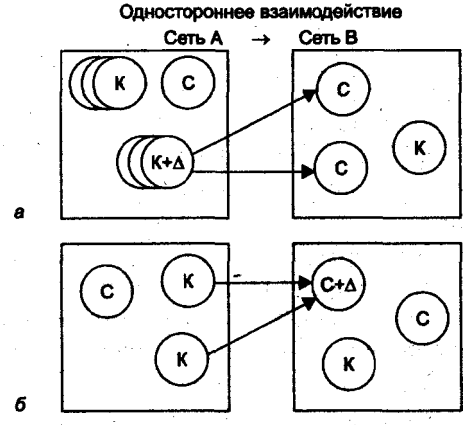


Рис. 11.26. Варианты размещения средств межсетевого взаимодействия

В первом случае, когда средства мультиплексирования протоколов располагаются на клиентских частях, только клиенты, снабженные такими средствами, могут обращаться к серверам сети *B*, причем к любому из них. Во втором случае, когда набор дополнительных стеков расположен на каком-либо сервере сети *B*, данный сервер может обслуживать всех клиентов сети *A*. Очевидно, что серверы сети *B* без средств мультиплексирования не могут быть использованы клиентами сети *A*.

Примером «добавки», модифицирующей клиентскую часть, может служить популярное программное средство SAMBA, которое добавляет серверную и клиентские части протокола SMB компьютерам, работающим под управлением Unix. Это позволяет клиентам Microsoft Windows обращаться к файлам и принтерам серверов Unix так, как будто это серверы Microsoft Windows.

При мультиплексировании протоколов дополнительное программное обеспечение — соответствующие стеки протоколов — должно быть установлено на каждый компьютер, которому может потребоваться доступ к разнородным сетям. В некоторых операционных системах имеются средства борьбы с избыточностью, свойственной этому подходу. Операционная система может быть сконфигурирована для работы с несколькими стеками протоколов, но динамически загружаются только нужные.

В то же время избыточность повышает надежность системы в целом, поскольку отказ компьютера с установленным дополнительным стеком не ведет к потере возможности межсетевого взаимодействия для других пользователей сети.

Важным преимуществом мультиплексирования является меньшее время выполнения запроса, чем при использовании шлюза. Это связано, во-первых, с отсутствием временных затрат на процедуру трансляции, а во-вторых, с тем, что при мультиплексировании на каждый запрос требуется только одна сетевая передача, в то время как при трансляции — две: запрос сначала передается в шлюз, а затем из шлюза на ресурсный сервер.

В принципе, при применении нескольких стеков протоколов у пользователя может возникнуть проблема работы в незнакомой среде с незнакомыми коман-

дами, правилами и методами адресации. Чаще всего разработчики операционных систем стремятся в какой-то степени облегчить жизнь пользователю в этой ситуации. Независимо от применяемого протокола прикладного уровня (например, SMB или NFS), ему предоставляется один и тот же интуитивно понятный графический интерфейс, с помощью которого он просматривает и выбирает нужные удаленные ресурсы.

В табл. 11.2 приведены сравнительные характеристики двух подходов к реализации межсетевого взаимодействия.

Таблица 11.2. Сравнение методов трансляции и мультиплексирования протоколов

Метод	Достоинства	Недостатки
Мультиплексирование протоколов	Более быстрый доступ; повышение надежности взаимодействия за счет установки стека на нескольких узлах сети; хорошо масштабируемое средство	Усложнение администрирования и контроля доступа; большая избыточность, требующая дополнительных ресурсов от рабочих станций; менее удобная среда для пользователя, чем шлюзы
Трансляция протоколов (шлюзы, маршрутизаторы, коммутаторы)	Сохранение привычной среды пользователей; отсутствие необходимости в дополнительном программном обеспечении на рабочих станциях; локализация всех проблем межсетевого взаимодействия; обеспечение возможности доступа к «чужим» ресурсам сразу для нескольких клиентов	Замедление работы; снижение надежности; плохая масштабируемость; необходимость в двух сетевых передачах для выполнения одного запроса

Инкапсуляция протоколов

Инкапсуляция (encapsulation), или **туннелирование** (tunneling), — это еще один метод решения задачи согласования сетей, который, однако, применим только для согласования транспортных протоколов и только при определенных ограничениях. Инкапсуляция может быть применена, когда две сети, построенные на базе одной и той же транспортной технологии, необходимо соединить через транзитную сеть, построенную на базе другой транспортной технологии.

В процессе инкапсуляции принимают участие три типа протоколов:

- протокол-«пассажир»;
- несущий протокол;
- протокол инкапсуляции.

Транспортный протокол объединяемых сетей является *протоколом-пассажиром*, а протокол транзитной сети — *несущим протоколом*. Пакеты протокола-пассажира помещаются в поле данных пакетов несущего протокола и извлекаются оттуда с помощью *протокола инкапсуляции*.

Пакеты протокола-пассажира никаким образом не обрабатываются при транспортировке их по транзитной сети. Инкапсуляцию выполняет пограничное устройство (обычно маршрутизатор или шлюз), которое располагается на границе между исходной и транзитной сетями. Извлечение пакетов-пассажиров из несущих пакетов выполняет второе пограничное устройство, которое находится на границе между транзитной сетью и сетью назначения. Пограничные устройства указывают в несущих пакетах свои адреса, а не адреса узлов назначения.

В связи с большой популярностью Интернета и стека TCP/IP несущим протоколом транзитной сети обычно выступает протокол IP, а в качестве протоколов-пассажиров — протоколы локальных сетей. Вместе с тем, применяются и другие схемы инкапсуляции, такие как инкапсуляция IP в IP, Ethernet в MPLS, Ethernet в Ethernet. Подобные схемы инкапсуляции служат не только для того, чтобы согласовать транспортные протоколы, но и для других целей, например, для шифрования исходного трафика или для изоляции адресного пространства транзитной сети провайдера от адресного пространства пользовательских сетей.

В приведенном на рис. 11.27 примере две сети, использующие протокол IPX, нужно соединить через транзитную сеть TCP/IP. Необходимо обеспечить только взаимодействие узлов двух сетей IPX, а взаимодействие между IPX-узлами и узлами сети TCP/IP не предусматривается, поэтому для соединения сетей IPX можно применить метод инкапсуляции.

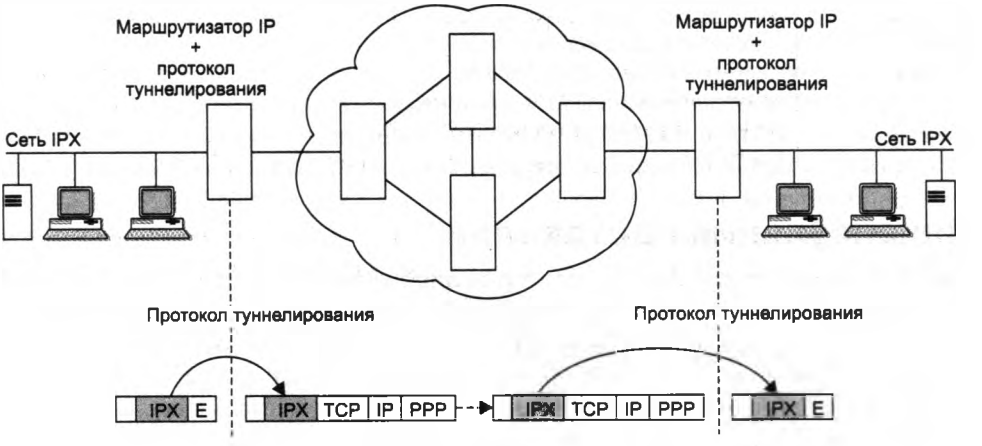


Рис. 11.27. Туннелирование протоколов сетевого уровня

В пограничных маршрутизаторах, соединяющих сети IPX с транзитной сетью IP, работают протоколы IPX, IP и дополнительный протокол — протокол инкапсуляции IPX в IP. Этот протокол извлекает IPX-пакеты из Ethernet-кадров и помещает их в UDP- или TCP-дейтаграммы (на рисунке выбран вариант с TCP). Затем несущие IP-пакеты направляются другому пограничному маршрутизатору. Протокол инкапсуляции должен иметь информацию о соответствии IPX-адреса удаленной сети IP-адресу пограничного маршрутизатора, об-

служивающего эту сеть. Если через сеть IP объединяется несколько сетей IPX, то должна быть таблица соответствия всех IPX-адресов IP-адресам пограничных маршрутизаторов.

Инкапсуляция может быть использована для транспортных протоколов разного уровня. Например, протокол сетевого уровня X.25 может быть инкапсулирован в протокол транспортного уровня TCP, или же протокол сетевого уровня IP может быть инкапсулирован в протокол сетевого уровня X.25. Существуют протоколы инкапсуляции трафика протокола PPP через сети IP.

Обычно инкапсуляция ведет к более простым и быстрым решениям по сравнению с трансляцией, так как решает более частную задачу, не обеспечивая взаимодействия с узлами транзитной сети. Помимо согласования транспортных технологий, инкапсуляция обеспечивает секретность передаваемых данных. При этом исходные пакеты-пассажиры шифруются и передаются по транзитной сети с помощью пакетов несущего протокола.

Выводы

- Файловая служба включает программы-серверы и программы-клиенты, взаимодействующие с помощью определенного протокола по сети между собой.
- Один компьютер может в одно и то же время предоставлять пользователям сети услуги различных файловых служб.
- В сетевой файловой службе в общем случае можно выделить следующие основные компоненты: локальную файловую систему, интерфейс локальной файловой системы, сервер сетевой файловой системы, клиент сетевой файловой системы, интерфейс сетевой файловой системы, протокол клиент-сервер сетевой файловой системы.
- В сетевых файловых системах используется различная семантика чтения и записи разделяемых данных, позволяющая избежать проблем с интерпретацией результирующих данных файла.
- Файловый интерфейс может быть отнесен к одному из двух типов в зависимости от того, поддерживает ли он модель загрузки-выгрузки или модель удаленного доступа.
- Файловый сервер может быть реализован по одной из двух схем: с запоминанием данных о последовательности файловых операций клиента, то есть по схеме stateful, и без запоминания таких данных, то есть по схеме stateless.
- Кэширование в сетевых файловых системах позволяет повысить скорость доступа к удаленным данным, улучшить масштабируемость и повысить надежность файловой системы.
- Репликация подразумевает существование нескольких копий одного и того же файла, каждая из которых хранится на отдельном файловом сервере, при этом обеспечивается автоматическое согласование данных в копиях файла.

- Существует несколько способов обеспечения согласованности реплик, которые общаются в методе кворума.
- Справочная служба хранит информацию обо всех пользователях и ресурсах сети в виде унифицированных объектов с определенными атрибутами, а также позволяет отражать взаимосвязи между хранимыми объектами.
- Справочная служба упрощает работу распределенных приложений и повышает управляемость сети.
- Справочная служба обычно строится на основе модели клиент-сервер: серверы хранят базу справочной информации, которой пользуются клиенты, передавая серверам по сети соответствующие запросы.
- Наиболее перспективным стандартом доступа к службе каталогов является стандарт LDAP (Light-weight Directory Access Protocol), разработанный сообществом Интернета.
- В контексте межсетевого взаимодействия понятие «сеть» можно определить как совокупность компьютеров, общающихся друг с другом с помощью единого стека протоколов. Проблема организации межсетевого взаимодействия возникает тогда, когда компьютеры принадлежат разным сетям, но поддерживают стеки протоколов, отличающиеся на одном или более уровнях.
- Средства, позволяющие организовать взаимодействие на нижних уровнях стека протоколов, называются средствами межсетевого взаимодействия, а средства согласования протоколов и служб верхних уровней — средствами операционной совместимости.
- Существует три основных способа согласования протоколов: трансляция, мультиплексирование и инкапсуляция (туннелирование).
- Трансляция заключается в преобразовании сообщений, поступающих от одного протокола, в сообщения другого протокола. Трансляция бывает одноуровневая, когда для выполнения преобразования используется информация только данного протокола, и двухуровневая, когда для преобразования привлекается информация протокола верхнего уровня.
- Мультиплексирование заключается в установке нескольких стеков протоколов на клиентах или серверах согласуемых сетей.
- К достоинствам метода трансляции протоколов относятся: сохранение привычной среды пользователя, локализация функций согласования сетей в одном месте, отсутствие необходимости в установке дополнительного программного обеспечения на многих компьютерах. Недостатки — низкая скорость и ненадежность. Достоинствами метода мультиплексирования стеков протоколов являются высокая скорость и надежность, недостатками — избыточность и большой объем администрирования.
- Инкапсуляция применяется для транзитной передачи одного транспортного протокола через сеть с другим стеком транспортных протоколов.
- Способы согласования протоколов прикладного уровня — сетевых служб — имеют свою специфику, связанную с несимметричностью этих протоколов,

реализуемых в архитектуре клиент-сервер, а также наличием в каждой операционной системе большого количества разнообразных сетевых служб (файловой службы, службы печати, электронной почты, справочной службы и т. д.).

Задачи и упражнения

1. Какие из следующих протоколов являются протоколами взаимодействия клиентской и серверной частей файлового сервиса: SMTP, NFS, SMB, SNMP, UDP, NLSP, FTP, TFTP, NCP.
2. Какая модель файлового сервера (stateful или stateless) обеспечивает большую степень устойчивости к отказам сервера?
3. Поскольку репликация и кэширование файлов преследуют близкие цели, то стоит ли реализовывать эти два механизма в одной файловой системе?
4. Заполните таблицу, отметив наличие или отсутствие соответствующих свойств у механизмов репликации и кэширования файлов.

	Повышение производительности	**Повышение отказоустойчивости**
Репликация		
Кэширование		

5. Можно ли с помощью одного прикладного протокола осуществлять доступ по сети к различным локальным файловым системам?
6. Может ли несколько пользователей одновременно модифицировать один и тот же файл в ОС Windows NT? А в ОС Unix?
7. Какой результат видят на экране два пользователя ОС Unix, набирающие текст в одном и том же файле?
8. Какая модель сетевого файлового сервиса более прозрачна для пользователя: загрузки-выгрузки или удаленного доступа?
9. Сравните два метода кэширования, используемые в сетевой файловой службе, — на стороне клиента и на стороне сервера. Приведите достоинства и недостатки каждого метода.
10. Какими свойствами должна обладать база данных справочной службы?
11. Вставьте один из двух терминов — «реплицируемость» или «распределенность» — вместо пропущенных слов в следующем утверждении: «База данных службы каталогов должна обладать... для обеспечения масштабируемости службы и... для обеспечения ее отказоустойчивости».
12. Поясните разницу в терминах «межсетевое взаимодействие» (internetworking) и «операционная совместимость» (interoperability).
13. Если на клиентской машине установлен стек протоколов, не совпадающий со стеком протоколов, установленным на сервере, то положение можно ис-

править, дополнительно установив соответствующий стек протоколов на одной из машин. Имеет ли значение, на какой из машин (сервере или клиенте) будет установлен этот стек?

14. Возможно ли в принципе обеспечить доступ всех клиентов сети *A* к серверам сети *B* и доступ всех клиентов сети *B* к серверам сети *A* путем установки дополнительного программного обеспечения только в одной из сетей, например, в сети *A*?
15. Пусть в некоторой сети, состоящей из сервера Windows 2003 и клиентских станций Windows Vista, работают немногочисленные пользователи-непрофессионалы, выполняющие некритические приложения. Клиентские станции имеют весьма ограниченные ресурсы. Время от времени у пользователей возникает необходимость доступа к данным, находящимся на файловом сервере NetWare, который подключен к тому же сегменту Ethernet. Как вы считаете, какой вариант межсетевого взаимодействия является более предпочтительным в этой ситуации:
    1) на всех компьютерах установить клиентскую часть протокола NCP;
    2) на сервере Windows 2003 установить шлюз.
16. Пусть в сети Ethernet, в которой на всех компьютерах установлены протоколы сетевого уровня IP, драйверы сетевых адаптеров одних компьютеров выполнены в стандарте NDIS, а других — в стандарте ODI. Может ли это помешать нормальной работе сети?
17. Пусть распределенное приложение состоит из двух частей. Одна часть распределенного приложения выполняется на компьютере, на котором установлены следующие коммуникационные протоколы:
    - на прикладном уровне: SMB, SMTP;
    - на транспортных уровнях: TCP, IP, Ethernet.

    Вторая часть приложения установлена на компьютере, на котором работают такие протоколы:
    - на прикладном уровне: NFS, X.400;
    - на транспортных уровнях: TCP, IP, Ethernet.

    Может ли в таких условиях приложение работать нормально?

Глава 12
Сетевая безопасность

В области безопасности информационных систем обычно выделяют две группы проблем: безопасность компьютера и сетевая безопасность.

К *безопасности компьютера* относят все проблемы защиты данных, хранящихся и обрабатывающихся компьютером, который рассматривается как автономная система. Эти проблемы решаются средствами операционных систем и приложений, таких как, например, система управления базами данных или система управления ресурсами предприятия, а также встроенными аппаратными средствами компьютера.

Под *сетевой безопасностью* понимают все вопросы, связанные с взаимодействием устройств в сети. Это, прежде всего, защита данных в момент их передачи по линиям связи и защита от несанкционированного удаленного доступа в сеть. И хотя подчас проблемы компьютерной и сетевой безопасности трудно отделить друг от друга, настолько тесно они связаны, совершенно очевидно, что сетевая безопасность имеет свою специфику.

Автономно работающий компьютер можно эффективно защитить от внешних покушений разнообразными способами, например, просто запереть на замок клавиатуру или снять жесткий накопитель и поместить его в сейф. Компьютер, работающий в сети, по определению не может полностью отгородиться от мира, он должен общаться с другими компьютерами, возможно даже удаленными от него на большое расстояние, поэтому обеспечение безопасности в сети является задачей значительно более сложной. Логический вход чужого пользователя в ваш компьютер является штатной ситуацией, если вы работаете в сети и объявили некоторые ресурсы вашего компьютера как разделяемые. Обеспечение безопасности в такой ситуации сводится к тому, чтобы сделать это проникновение контролируемым — каждому пользователю сети должны быть четко определены его права по доступу к информации, внешним устройствам и выполнению системных действий на каждом из компьютеров сети.

Помимо проблем, порождаемых возможностью удаленного входа в сетевые компьютеры, сети по своей природе подвержены еще одному виду опасности — перехвату и анализу сообщений, передаваемых по сети, а также созданию «ложного»

трафика. Большая часть средств обеспечения сетевой безопасности направлена на предотвращение именно этого типа нарушений.

Вопросы сетевой безопасности приобретают особое значение сейчас, когда при построении корпоративных сетей чаще всего используются не выделенные каналы, а Интернет, который представляет собой всемирную многодоменную публичную сеть, населенную, к сожалению, не только добросовестными пользователями. При использовании Интернета для построения корпоративной сети пока трудно получить качественную защиту пользовательских данных, особенно когда сайты корпоративной сети подключены к различным поставщикам услуг Интернета. Поэтому в общем случае заботы по конфиденциальности, целостности и доступности данных при их передаче через сеть ложатся на плечи администраторов корпоративных сетей и самих пользователей.

Основные понятия безопасности

Конфиденциальность, целостность и доступность данных

Безопасная информационная система — это система, которая, во-первых, защищает данные от несанкционированного доступа, во-вторых, всегда готова предоставить их своим пользователям, а в-третьих, надежно хранит информацию и гарантирует неизменность данных. Таким образом, безопасная система по определению обладает свойствами конфиденциальности, доступности и целостности.

- **Конфиденциальность** (confidentiality) — это гарантия того, что секретные данные будут доступны только тем пользователям, которым этот доступ разрешен (такие пользователи называются авторизованными).
- **Доступность** (availability) — это гарантия того, что авторизованные пользователи всегда получат доступ к данным.
- **Целостность** (integrity) — это гарантия сохранности данными правильных значений, которая обеспечивается запретом для неавторизованных пользователей каким-либо образом изменять, модифицировать, разрушать или создавать данные.

Требования безопасности могут меняться в зависимости от назначения системы, характера используемых данных и типа возможных угроз. Трудно представить систему, для которой были бы не важны свойства целостности и доступности, но свойство конфиденциальности не всегда является обязательным. Например, если вы публикуете информацию в Интернете на веб-сервере и вашей целью является сделать ее доступной для самого широкого круга людей, то в данном случае конфиденциальность не требуется. Однако требования целостности и доступности остаются актуальными.

Действительно, если вы не предпримете специальных мер по обеспечению целостности данных, злоумышленник может изменить данные на вашем сервере и нанести этим ущерб вашему предприятию. Преступник может, например,

внести изменения в помещенный на веб-сервере прайс-лист, что негативно отразится на конкурентоспособности вашего предприятия, или испортить коды свободно распространяемого вашей фирмой программного продукта, что безусловно скажется на ее деловом имидже.

Не менее важным в данном примере является и обеспечение доступности данных. Затратив немалые средства на создание и поддержание сервера в Интернете, предприятие вправе рассчитывать на отдачу: увеличение числа клиентов, количества продаж и т. д. Однако существует вероятность того, что злоумышленник предпримет атаку, в результате которой помещенные на сервер данные станут недоступными для тех, кому они предназначались. Примером таких злонамеренных действий может служить «бомбардировка» сервера IP-пакетами с неправильным обратным адресом, которые в соответствии с логикой работы этого протокола могут вызывать тайм-ауты, а в конечном счете, сделать сервер недоступным для всех остальных запросов.

Понятия конфиденциальности, доступности и целостности могут быть определены не только по отношению к информации, но и к другим ресурсам вычислительной сети, например внешним устройствам или приложениям. Существует множество системных ресурсов, возможность «незаконного» использования которых может привести к нарушению безопасности системы. Например, неограниченный доступ к устройству печати позволяет злоумышленнику получать копии распечатываемых документов, изменять параметры, что может привести к изменению очередности работ и даже к выводу устройства из строя. Свойство конфиденциальности по отношению к устройству печати можно интерпретировать так, что доступ к устройству имеют те и только те пользователи, которым этот доступ разрешен, причем они могут выполнять только те операции с устройством, которые для них определены. Свойство доступности устройства означает его готовность к использованию всякий раз, когда в этом возникает необходимость. А свойство целостности может быть определено как свойство неизменности параметров данного устройства. Легальность использования сетевых устройств важна не только постольку, поскольку она влияет на безопасность данных. Устройства могут предоставлять различные услуги (распечатка текстов, отправка факсов, доступ в Интернет, электронная почта и т. п.), незаконное потребление которых, наносящее материальный ущерб предприятию, также является нарушением безопасности системы.

Любое действие, которое направлено на нарушение конфиденциальности, целостности и/или доступности информации, а также на нелегальное использование других ресурсов сети, называется **угрозой**. Реализованная угроза становится **атакой**. **Риск** — это вероятностная оценка величины возможного ущерба, который может понести владелец информационного ресурса в результате успешно проведенной атаки. *Значение риска тем выше, чем более уязвимой является существующая система безопасности и чем выше вероятность реализации атаки.*

Классификация угроз

Универсальной классификации угроз не существует, возможно, потому что нет предела творческим способностям человека, и каждый день применяются

новые способы незаконного проникновения в сеть, разрабатываются новые средства мониторинга сетевого трафика, появляются новые вирусы, находятся новые изъяны в существующих программных и аппаратных сетевых продуктах. В ответ на это разрабатываются все более изощренные средства защиты, которые ставят преграду на пути многих типов угроз, но затем сами становятся новыми объектами атак. Тем не менее попытаемся сделать некоторые обобщения. Так, прежде всего, угрозы могут быть разделены на умышленные и неумышленные.

Неумышленные угрозы вызываются ошибочными действиями лояльных сотрудников, следствием их низкой квалификации или безответственности. Кроме того, к такому роду угроз относятся последствия ненадежной работы программных и аппаратных средств системы. Так, например, из-за отказа диска, контроллера диска или всего файлового сервера могут оказаться недоступными данные, критически важные для работы предприятия. Поэтому вопросы безопасности так тесно переплетаются с вопросами надежности, отказоустойчивости технических средств. Угрозы безопасности, которые вытекают из ненадежности работы программно-аппаратных средств, предотвращаются путем их совершенствования, использования резервирования на уровне аппаратуры (RAID-массивы, многопроцессорные компьютеры, источники бесперебойного питания, кластерные архитектуры) или на уровне массивов данных (тиражирование файлов, резервное копирование).

Умышленные угрозы могут ограничиваться либо пассивным чтением данных или мониторингом системы, либо включать в себя активные действия, например, нарушение целостности и доступности информации, приведение в нерабочее состояние приложений и устройств. Так, умышленные угрозы возникают в результате деятельности хакеров и явно направлены на нанесение ущерба предприятию.

В вычислительных сетях можно выделить следующие типы умышленных угроз:

- незаконное проникновение в один из компьютеров сети под видом легального пользователя;
- разрушение системы с помощью программ-вирусов;
- нелегальные действия легального пользователя;
- «подслушивание» внутрисетевого трафика.

Незаконное проникновение может быть реализовано через уязвимые места в системе безопасности с использованием недокументированных возможностей операционной системы. Эти возможности могут позволить злоумышленнику «обойти» стандартную процедуру входа в сеть.

Другим способом незаконного проникновения в сеть является использование «чужих» паролей, полученных путем подглядывания, расшифровки файла паролей, подбора паролей или получения пароля путем анализа сетевого трафика. Особенно опасно проникновение злоумышленника под именем пользователя, наделенного большими полномочиями, например администратора сети.

Для того чтобы завладеть паролем администратора, злоумышленник может попытаться войти в сеть под именем простого пользователя. Поэтому очень важно, чтобы все пользователи сети сохраняли свои пароли в тайне, а также выбирали их так, чтобы максимально затруднить угадывание.

Подбор паролей злоумышленник выполняет с использованием специальных программ, которые работают путем перебора слов из некоторого файла, содержащего большое количество слов. Содержимое файла-словаря формируется с учетом психологических особенностей человека, которые выражаются в том, что человек выбирает в качестве пароля легко запоминаемые слова или буквенные сочетания.

Еще один способ получения пароля — это внедрение в чужой компьютер «троянского коня». Так называют резидентную программу, работающую без ведома хозяина данного компьютера и выполняющую действия, заданные злоумышленником. В частности, такого рода программа может считывать коды пароля, вводимого пользователем во время логического входа в систему.

Программа-*троянский конь* всегда маскируется под какую-нибудь полезную утилиту или игру, но при этом производит действия, разрушающие систему. По такому принципу действуют и *программы-вирусы*, отличительной особенностью которых является способность «заражать» другие файлы, внедряя в них свои собственные копии. Чаще всего вирусы поражают исполняемые файлы. Когда такой исполняемый код загружается в оперативную память для выполнения, вместе с ним получает возможность исполнить свои вредительские действия вирус. Вирусы могут привести к повреждению или даже полной утрате информации.

Нелегальные действия легального пользователя — этот тип угроз исходит от легальных пользователей сети, которые, используя свои полномочия, пытаются выполнять действия, выходящие за рамки их должностных обязанностей. Например, администратор сети имеет практически неограниченные права на доступ ко всем сетевым ресурсам. Однако на предприятии может быть информация, доступ к которой администратору сети запрещен. Для реализации этих ограничений могут быть предприняты специальные меры, такие, например, как шифрование данных, но и в этом случае администратор может попытаться получить доступ к ключу. Нелегальные действия может попытаться предпринять и обычный пользователь сети. Существующая статистика говорит о том, что едва ли не половина всех попыток нарушения безопасности системы исходит от сотрудников предприятия, которые как раз и являются легальными пользователями сети.

«Подслушивание» внутрисетевого трафика — это незаконный мониторинг сети, захват и анализ сетевых сообщений. Существует много доступных программных и аппаратных анализаторов трафика, которые делают эту задачу достаточно тривиальной. Еще более усложняется защита от этого типа угроз в сетях с глобальными связями. Глобальные связи, простирающиеся на десятки и тысячи километров, по своей природе являются менее защищенными, чем локальные связи (больше возможностей для прослушивания трафика, более удобная для

злоумышленника позиция при проведении процедур аутентификации). Такая опасность одинаково присуща всем видам территориальных каналов связи и никак не зависит от того, используются собственные, арендуемые каналы или услуги общедоступных территориальных сетей, подобных Интернету.

Однако использование общественных сетей (речь в основном идет об Интернете) еще более усугубляет ситуацию. Действительно, Интернет добавляет к опасности перехвата данных, передаваемых по линиям связи, опасность несанкционированного входа в узлы сети, поскольку наличие огромного числа хакеров в Интернете повышает вероятность попыток незаконного проникновения в компьютер. Это представляет постоянную угрозу для сетей, подсоединенных к Интернету.

Интернет сам является целью для разного рода злоумышленников. Поскольку Интернет создавался как открытая система, предназначенная для свободного обмена информацией, совсем не удивительно, что практически все протоколы стека TCP/IP имеют «врожденные» недостатки защиты. Используя эти недостатки, злоумышленники все чаще предпринимают попытки несанкционированного доступа к информации, хранящейся на узлах Интернета.

Системный подход к обеспечению безопасности

Построение и поддержка безопасной системы требует системного подхода. В соответствии с этим подходом, прежде всего, необходимо осознать весь спектр возможных угроз для конкретной сети и для каждой из этих угроз продумать тактику ее отражения. В этой борьбе можно и нужно использовать самые разные средства и приемы — морально-этические и законодательные, административные и психологические, а также задействовать защитные возможности программных и аппаратных средств сети.

К *морально-этическим* средствам защиты можно отнести всевозможные нормы, которые сложились по мере распространения вычислительных средств в той или иной стране. Например, подобно тому, как в борьбе против пиратского копирования программ в настоящее время в основном используются меры воспитательного плана, необходимо внедрять в сознание людей аморальность всяческих покушений на нарушение конфиденциальности, целостности и доступности чужих информационных ресурсов.

Законодательные средства защиты — это законы, постановления правительства и указы президента, нормативные акты и стандарты, которыми регламентируются правила использования и обработки информации ограниченного доступа, а также вводятся меры ответственности за нарушения этих правил. Правовая регламентация деятельности в области безопасности имеет целью защиту информации, составляющей государственную тайну, обеспечение прав потребителей на получение качественных продуктов, защиту конституционных прав граждан на сохранение личной тайны, борьбу с организованной преступностью.

Административные меры — это действия, предпринимаемые руководством предприятия или организации для обеспечения информационной безопасности. К таким мерам относятся конкретные правила работы сотрудников пред-

приятия, например, режим работы сотрудников, их должностные инструкции, строго определяющие порядок использования конфиденциальной информации на компьютере. К административным мерам также относятся правила приобретения предприятием средств безопасности. Представители администрации, которые несут ответственность за защиту информации, должны выяснить, насколько безопасным является использование продуктов, приобретенных у зарубежных поставщиков. Особенно это касается продуктов, связанных с шифрованием. В таких случаях желательно проверить наличие у продукта сертификата, выданного российскими тестирующими организациями.

Психологические меры безопасности могут играть значительную роль в укреплении безопасности системы. Пренебрежение учетом психологических моментов в неформальных процедурах, связанных с безопасностью, может привести к нарушениям защиты. Рассмотрим, например, сеть предприятия, в которой работает много удаленных пользователей. Время от времени пользователи должны менять пароли (обычная практика для предотвращения их подбора). В данной системе выбор паролей осуществляет администратор. В таких условиях злоумышленник может позвонить администратору по телефону и от имени легального пользователя попробовать получить пароль. При большом количестве удаленных пользователей не исключено, что такой простой психологический прием может сработать.

К *физическим* средствам защиты относятся экранирование помещений для защиты от излучения, проверка поставляемой аппаратуры на соответствие ее спецификациям и отсутствие аппаратных «жучков», средства наружного наблюдения, устройства, блокирующие физический доступ к отдельным блокам компьютера, различные замки и другое оборудование, защищающие помещения, где находятся носители информации, от незаконного проникновения и т. д., и т. п.

Технические средства информационной безопасности реализуются программным и аппаратным обеспечением вычислительных сетей. Такие средства, называемые также службами сетевой безопасности, решают самые разнообразные задачи по защите системы, например, контроль доступа, включающий процедуры аутентификации и авторизации, аудит, шифрование информации, антивирусную защиту, контроль сетевого трафика и много других задач. Технические средства безопасности могут быть либо встроены в программное (операционные системы и приложения) и аппаратное (компьютеры и коммуникационное оборудование) обеспечение сети, либо реализованы в виде отдельных продуктов, созданных специально для решения проблем безопасности.

Политика безопасности

Важность и сложность проблемы обеспечения безопасности требует выработки **политики информационной безопасности**, которая подразумевает ответы на следующие вопросы.

- Какую информацию защищать?
- Какой ущерб понесет предприятие при потере или при раскрытии тех или иных данных?

- Кто или что является возможным источником угрозы, какого рода атаки на безопасность системы могут быть предприняты?
- Какие средства использовать для защиты каждого вида информации?

Специалисты, ответственные за безопасность системы, формируя политику безопасности, должны учитывать несколько базовых принципов. Одним из таких принципов является предоставление каждому сотруднику предприятия того *минимально уровня привилегий* на доступ к данным, который необходим ему для выполнения его должностных обязанностей. Учитывая, что большая часть нарушений в области безопасности предприятий исходит именно от собственных сотрудников, важно ввести четкие ограничения для всех пользователей сети, не наделяя их излишними возможностями.

Следующий принцип — *комплексный подход* к обеспечению безопасности. Чтобы затруднить злоумышленнику доступ к данным, необходимо предусмотреть самые разные средства безопасности, начиная от организационно-административных запретов и заканчивая встроенными средствами сетевой аппаратуры. Административный запрет на работу в воскресные дни ставят потенциального нарушителя под визуальный контроль администратора и других пользователей, физические средства защиты (закрытые помещения, блокировочные ключи) ограничивают непосредственный контакт пользователя только приписанным ему компьютером, встроенные средства сетевой ОС (система аутентификации и авторизации) предотвращают вход в сеть нелегальных пользователей, а для легального пользователя ограничивают возможности только разрешенными для него операциями (подсистема аудита фиксирует его действия). Такая система защиты с многократным резервированием средств безопасности повышает вероятность сохранности данных.

Используя многоуровневую систему защиты, важно поддерживать *баланс надежности защиты всех уровней*. Если в сети все сообщения шифруются, но ключи легкодоступны, то эффект от шифрования нулевой. Или если на компьютерах установлена файловая система, поддерживающая избирательный доступ на уровне отдельных файлов, но имеется возможность получить жесткий диск и установить его на другой машине, то все достоинства средств защиты файловой системы сводятся «на нет». Если внешний трафик сети, подключенной к Интернету, проходит через мощный брандмауэр (межсетевой экран), но пользователи имеют возможность связываться с узлами Интернета по коммутируемым линиям, используя локально установленные модемы, то деньги (как правило, немалые), потраченные на брандмауэр, можно считать «выброшенными на ветер».

Следующим универсальным принципом является использование средств, которые при отказе переходят в состояние *максимальной защиты*. Это касается самых различных средств безопасности. Если, например, автоматический пропускной пункт в какое-либо помещение ломается, то он должен фиксироваться в таком положении, чтобы ни один человек не мог пройти на защищаемую территорию. А если в сети имеется устройство, которое анализирует весь входной трафик и отбрасывает кадры с определенным заранее заданным обратным адре-

сом, то при отказе оно должно полностью блокировать вход в сеть. Неприемлемым следовало бы признать устройство, которое бы при отказе пропускало в сеть весь внешний трафик.

Принцип единого контрольно-пропускного пункта — весь входящий во внутреннюю сеть и выходящий во внешнюю сеть трафик должен проходить через единственный узел сети, например, через межсетевой экран (firewall). Только это позволяет в достаточной степени контролировать трафик. В противном случае, когда в сети имеется множество пользовательских станций, имеющих независимый выход во внешнюю сеть, очень трудно скоординировать правила, ограничивающие права пользователей внутренней сети по доступу к серверам внешней сети и обратно — права внешних клиентов по доступу к ресурсам внутренней сети.

Принцип баланса возможного ущерба от реализации угрозы и затрат на ее предотвращение. Ни одна система безопасности не гарантирует защиту данных на уровне 100 %, поскольку является результатом компромисса между возможными рисками и возможными затратами. Определяя политику безопасности, администратор должен взвесить величину ущерба, которую может понести предприятие в результате нарушения защиты данных, и соотнести ее с величиной затрат, требуемых на обеспечение безопасности этих данных. Так, в некоторых случаях можно отказаться от дорогостоящего межсетевого экрана в пользу стандартных средств фильтрации обычного маршрутизатора, в других же можно пойти на беспрецедентные затраты. Главное, чтобы принятое решение было обосновано экономически.

При определении политики безопасности для сети, имеющей выход в Интернет, специалисты рекомендуют разделить задачу на две части: выработать политику доступа к сетевым службам Интернета и выработать политику доступа к ресурсам внутренней сети компании.

Политика доступа к сетевым службам Интернета включает следующие пункты.

- Определение списка служб Интернета, к которым пользователи внутренней сети должны иметь ограниченный доступ.
- Определение ограничений на методы доступа, например на использование протоколов SLIP (Serial Line Internet Protocol) и PPP (Point-to-Point Protocol). Ограничения методов доступа необходимы для того, чтобы пользователи не могли обращаться к «запрещенным» службам Интернета обходными путями. Например, если для ограничения доступа в Интернет установить в сети специальный шлюз, который не дает возможности пользователям работать в системе WWW, они могут устанавливать с веб-серверами PPP-соединения по коммутируемой линии. Во избежание этого надо просто запретить применение протокола PPP.
- Принятие решения о том, разрешен ли доступ внешних пользователей из Интернета во внутреннюю сеть. Если да, то кому. Часто доступ разрешают только некоторым абсолютно необходимым для работы предприятия службам, например электронной почте.

Политика доступа к ресурсам внутренней сети компании может быть выражена в одном из двух принципов:

- запрещать все, что не разрешено в явной форме;
- разрешать все, что не запрещено в явной форме.

В соответствии с выбранным принципом определяются правила обработки внешнего трафика межсетевыми экранами и маршрутизаторами. Реализация защиты на основе первого принципа дает более высокую степень безопасности, однако при этом могут возникать большие неудобства у пользователей, а кроме того, такой способ защиты обойдется значительно дороже. При реализации второго принципа сеть окажется менее защищенной, однако пользоваться ей будет удобнее и потребуется меньше затрат.

Базовые технологии безопасности

В разных программных и аппаратных продуктах, предназначенных для защиты данных, часто используются одинаковые подходы, приемы и технические решения. К таким базовым технологиям безопасности относятся аутентификация, авторизация и аудит, а также технология защищенного канала.

Шифрование

Шифрование — это краеугольный камень всех служб информационной безопасности, будь то система аутентификации или авторизации, защищенный канал или средства безопасного хранения данных.

Любая процедура шифрования, превращающая информацию из обычного «понятного» вида в «нечитабельный» зашифрованный вид, естественно должна быть дополнена процедурой **дешифрирования**, которая, будучи примененной к зашифрованному тексту, снова приводит его в понятный вид.

Пара процедур — шифрование и дешифрирование — называется **криптосистемой**.

Информацию, над которой выполняются функции шифрования и дешифрирования, будем условно называть *текстом*, учитывая, что это может быть также числовой массив или графические данные.

В современных алгоритмах шифрования предусматривается наличие параметра — **секретного ключа**. В криптографии принято правило Керкхоффа: *«Стойкость шифра должна определяться только секретностью ключа»*. Так, все стандартные алгоритмы шифрования (например, AES, DES, PGP) широко известны[1], их детальное описание содержится в легко доступных документах, но от этого их эффективность не снижается. Система остается защищенной, если злоумышленнику известно все об алгоритме шифрования, но он не знает секретный ключ.

[1] Следует, однако, отметить, что существует немало фирменных алгоритмов, описание которых не публикуется.

Алгоритм шифрования считается *раскрытым*, если найдена процедура, позволяющая подобрать ключ за реальное время. Сложность алгоритма раскрытия является одной из важных характеристик криптосистемы и называется **криптостойкостью**.

Существует два класса криптосистем — симметричные и асимметричные. В симметричных схемах шифрования (классическая криптография) секретный ключ шифрования совпадает с секретным ключом дешифрирования. В асимметричных схемах шифрования (криптография с открытым ключом) открытый ключ шифрования не совпадает с секретным ключом дешифрирования.

Симметричные алгоритмы шифрования

На рис. 12.1 приведена классическая модель **симметричной криптосистемы**, теоретические основы которой впервые были изложены в 1949 году в работе Клода Шеннона. В данной модели три участника: отправитель, получатель, злоумышленник. Задача отправителя заключается в том, чтобы по открытому каналу передать некоторое сообщение в защищенном виде. Для этого он зашифровывает открытый текст *X* ключом *k* и передает шифрованный текст *Y*. Задача получателя заключается в том, чтобы расшифровать *Y* и прочитать сообщение *X*. Предполагается, что отправитель имеет свой источник ключа. Сгенерированный ключ заранее по надежному каналу передается получателю. Задача злоумышленника заключается в перехвате и чтении передаваемых сообщений, а также в имитации ложных сообщений.

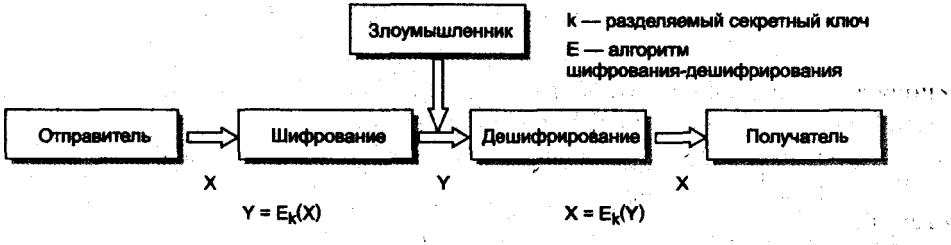


Рис. 12.1. Модель симметричного шифрования

Модель является универсальной — если зашифрованные данные хранятся в компьютере и никуда не передаются, отправитель и получатель совмещаются в одном лице, а в роли злоумышленника выступает некто, имеющий доступ к компьютеру в ваше отсутствие.

Наиболее популярным стандартным симметричным алгоритмом шифрования данных является **DES** (Data Encryption Standard). Алгоритм разработан фирмой IBM и в 1976 году был рекомендован Национальным бюро стандартов к использованию в открытых секторах экономики. Суть этого алгоритма заключается в следующем (рис. 12.2).

Данные шифруются *поблочно*. Перед шифрованием любая форма представления данных преобразуется в числовую. Числа получают путем применения любой открытой процедуры преобразования блока текста в число. Например,

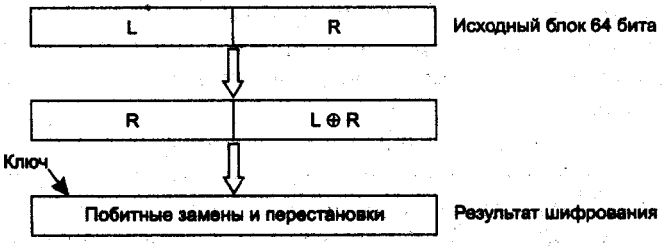


Рис. 12.2. Схема шифрования по алгоритму DES

ими могли бы быть значения двоичных чисел, полученных слиянием ASCII-кодов последовательных символов соответствующего блока текста. На вход шифрующей функции поступает блок данных, размером 64 бита, он делится пополам на левую (L) и правую (R) части. На первом этапе на место левой части результирующего блока помещается правая часть исходного блока. Правая часть результирующего блока вычисляется как сумма по модулю 2 (операция XOR) левой и правой частей исходного блока. Затем на основе случайной двоичной последовательности по определенной схеме в полученном результате выполняются побитные замены и перестановки. Используемая двоичная последовательность, представляющая собой ключ данного алгоритма, имеет длину 64 бита, из которых 56 действительно случайны, а 8 предназначены для контроля ключа.

Вот уже в течение трех десятков лет алгоритм DES испытывается на стойкость. И хотя существуют примеры успешных попыток «взлома» данного алгоритма, в целом можно считать, что он выдержал испытания. Алгоритм DES широко используется в различных технологиях и продуктах, связанных с безопасностью информационных систем. Для того чтобы повысить криптостойкость алгоритма DES, иногда применяют его усиленный вариант, называемый «тройным алгоритмом DES», который включает троекратное шифрование с использованием двух разных ключей. При этом можно считать, что длина ключа увеличивается с 56 до 112 бит, а значит, криптостойкость алгоритма существенно повышается. Но за это приходится платить производительностью — «тройной алгоритм DES» требует в три раза больше времени, чем «обычный».

В 2001 году Национальное бюро стандартов США приняло новый стандарт симметричного шифрования, который получил название **AES** (Advanced Encryption Standard). AES был разработан в результате проведения конкурса на лучший симметричный алгоритм шифрования, обладающий лучшим, чем у DES, соотношением показателей безопасности и скорости. Победителем был признан алгоритм Rijndael, который и стал основой AES. В результате AES обеспечивает лучшую защиту, так как использует 128-разрядные ключи (а также может работать со 192- и 256-разрядными ключами), и более высокую скорость, кодируя за один цикл 128-разрядный блок в отличие от 64-разрядного блока DES. В настоящее время AES является наиболее используемым симметричным алгоритмом шифрования.

В симметричных алгоритмах главную проблему представляют ключи. Во-первых, криптостойкость многих симметричных алгоритмов зависит от качества ключа, это предъявляет повышенные требования к службе генерации ключей. Во-вторых, принципиальной является надежность канала передачи ключа второму участнику секретных переговоров. Проблема с ключами возникает даже в системе с двумя абонентами, а в системе с n абонентами, желающими обмениваться секретными данными по принципу «каждый с каждым», потребуется $n \times (n - 1)/2$ ключей, которые должны быть сгенерированы и распределены надежным образом. То есть количество ключей пропорционально квадрату количества абонентов, что при большом числе абонентов делает задачу чрезвычайно сложной. Несимметричные алгоритмы, основанные на использовании открытых ключей, снимают эту проблему.

Несимметричные алгоритмы шифрования

В середине 70-х двое ученых — Винфилд Диффи и Мартин Хеллман — описали принципы шифрования с открытыми ключами.

Особенность шифрования на основе открытых ключей состоит в том, что одновременно генерируется уникальная **пара ключей**, таких, что текст, зашифрованный одним ключом, может быть расшифрован только с использованием второго ключа, и наоборот.

В модели криптосхемы с открытым ключом также три участника: отправитель, получатель, злоумышленник (рис. 12.3). Задача отправителя заключается в том, чтобы по открытому каналу связи передать некоторое сообщение в защищенном виде. Получатель генерирует на своей стороне два ключа: открытый E и закрытый D. Закрытый ключ D (часто называемый также личным ключом) абонент должен сохранять в защищенном месте, а открытый ключ E он может передать всем, с кем хочет поддерживать защищенные отношения. Открытый ключ используется для шифрования текста, но расшифровать текст можно только с помощью закрытого ключа. Поэтому открытый ключ передается отправителю в незащищенном виде. Отправитель, используя открытый ключ получателя, шифрует сообщение X и передает его получателю. Получатель расшифровывает сообщение своим закрытым ключом D.

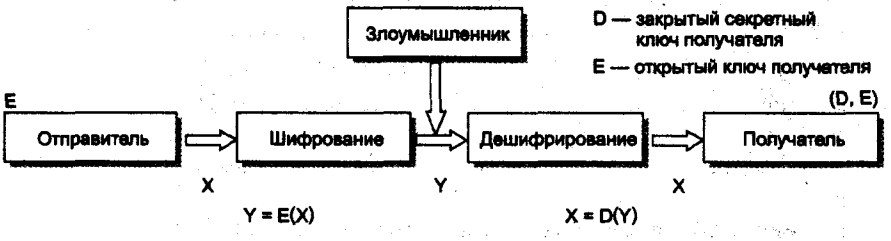


Рис. 12.3. Модель криптосхемы с открытым ключом

Очевидно, что числа, одно из которых используется для шифрования текста, а другое — для дешифрирования, не могут быть независимыми друг от друга, а значит, есть теоретическая возможность вычисления закрытого ключа по

открытому, но это связано с огромным объемом вычислений, которые требуют соответственно огромного времени. Поясним принципиальную связь между закрытым и открытым ключами следующей аналогией.

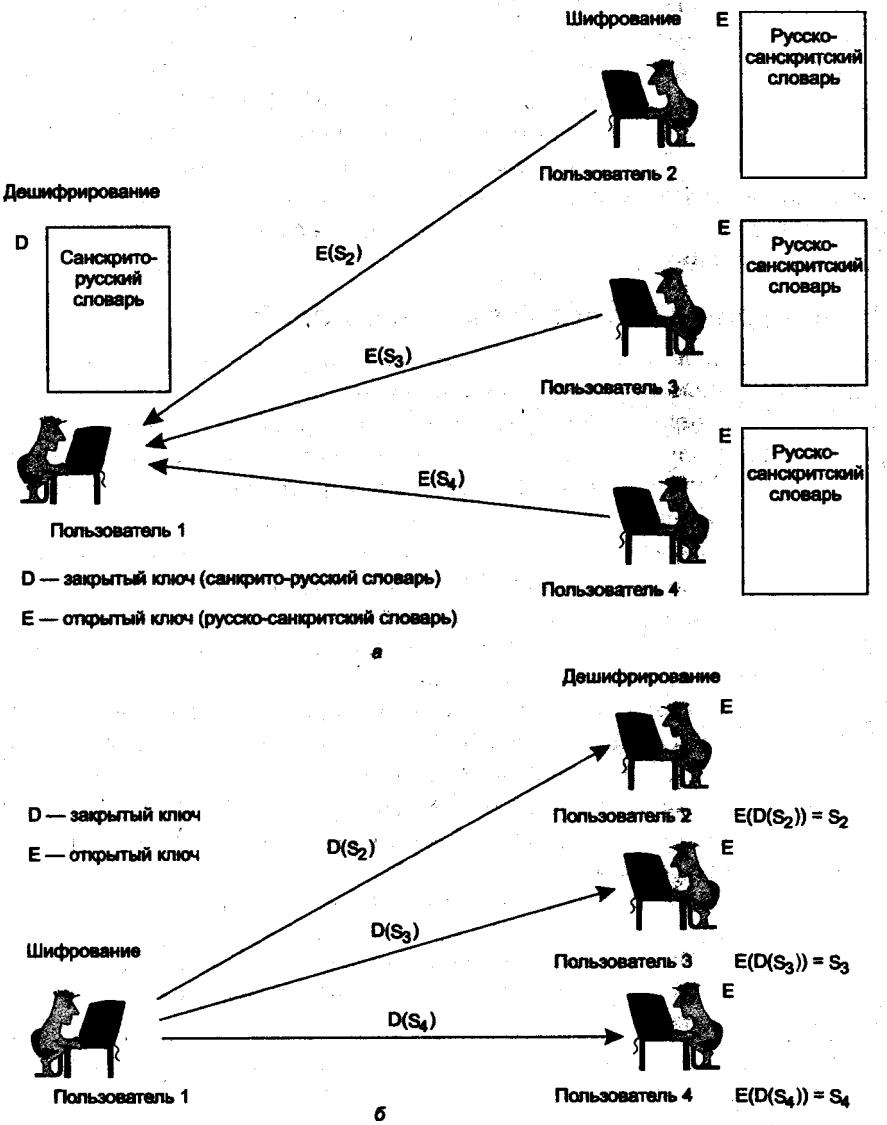


Рис. 12.4. Две схемы использования открытого и закрытого ключей

Пусть абонент 1 (рис. 12.4, *а*) решает вести секретную переписку со своими сотрудниками на малоизвестном языке, например санскрите. Для этого он обзаводится санскритско-русским словарем, а всем своим абонентам посылает русско-санскритские словари. Каждый из них, пользуясь словарем, пишет сооб-

щения на санскрите и посылает их абоненту 1, который переводит их на русский язык, пользуясь доступным только ему санскритско-русским словарем. Очевидно, что здесь роль открытого ключа E играет русско-санскритский словарь, а роль закрытого ключа D — санскритско-русский. Могут ли абоненты 2, 3 и 4 прочитать чужие сообщения S_2, S_3, S_4, которые посылает каждый из них абоненту 1? Вообще-то нет, так как для этого им нужен санскритско-русский словарь, обладателем которого является только абонент 1. Но теоретическая возможность для этого у них имеется, так как, затратив массу времени, можно прямым перебором составить санскритско-русский словарь по русско-санскритскому словарю. Такая процедура, требующая больших временных затрат, отдаленно напоминает восстановление закрытого ключа по открытому.

На рис. 12.4, *б* показана другая схема использования открытого и закрытого ключей, целью которой является подтверждение авторства (аутентификация или электронная подпись) посылаемого сообщения. В этом случае поток сообщений имеет обратное направление — от абонента 1, обладателя закрытого ключа D, к его корреспондентам, обладателям открытого ключа E. Если абонент 1 хочет аутентифицировать себя (поставить электронную подпись), то он шифрует известный текст своим закрытым ключом D и передает шифровку своим корреспондентам. Если им удается расшифровать текст открытым ключом абонента 1, то это доказывает, что текст был зашифрован его же закрытым ключом, а значит, именно он является автором этого сообщения. Заметим, что в этом случае сообщения S_2, S_3, S_4, адресованные разным абонентам, не являются секретными, так как все они — обладатели одного и того же открытого ключа, с помощью которого они могут расшифровывать все сообщения, поступающие от абонента 1.

Если же нужна взаимная аутентификация и двунаправленный секретный обмен сообщениями, то каждая из общающихся сторон генерирует собственную пару ключей и посылает открытый ключ своему корреспонденту.

Для того чтобы в сети все n абонентов имели возможность не только принимать зашифрованные сообщения, но и сами посылать таковые, каждый абонент должен обладать собственной парой ключей E и D. Всего в сети будет $2n$ ключей: n открытых ключей для шифрования и n секретных ключей для дешифрирования. Таким образом решается проблема масштабируемости — квадратичная зависимость количества ключей от числа абонентов в симметричных алгоритмах заменяется линейной зависимостью в несимметричных алгоритмах. Исчезает и задача секретной доставки ключа. Злоумышленнику нет смысла стремиться завладеть открытым ключом, поскольку это не дает возможности расшифровывать текст или вычислить закрытый ключ.

Хотя информация об открытом ключе не является секретной, ее нужно защищать от подлогов, чтобы злоумышленник под именем легального пользователя не навязал свой открытый ключ, после чего с помощью своего закрытого ключа он сможет расшифровывать все сообщения, посылаемые легальному пользователю, и отправлять свои сообщения от его имени. Проще всего было бы распространять списки, связывающие имена пользователей с их открытыми

ключами, широковещательно путем публикаций в средствах массовой информации (бюллетени, специализированные журналы и т. п.). Однако при таком подходе мы снова, как и в случае с паролями, сталкиваемся с плохой масштабируемостью. Решением этой проблемы является технология цифровых сертификатов. Сертификат — это электронный документ, который связывает конкретного пользователя с конкретным ключом.

В настоящее время одним из наиболее популярных криптоалгоритмов с открытым ключом является криптоалгоритм *RSA*.

Криптоалгоритм RSA

В 1978 году трое ученых (Ривест, Шамир и Адлеман) разработали систему шифрования с открытыми ключами **RSA** (Rivest, Shamir, Adleman), полностью отвечающую всем принципам Диффи–Хеллмана. Этот метод состоит в следующем.

1. Случайно выбираются два очень больших простых числа p и q.
2. Вычисляются два произведения $n = p \times q$ и $m = (p - 1) \times (q - 1)$.
3. Выбирается случайное целое число E, не имеющее общих сомножителей с m.
4. Находится D такое, что $DE = 1$ по модулю m.
5. Исходный текст X разбивается на блоки таким образом, чтобы $0 < X < n$.
6. Для шифрования сообщения необходимо вычислить $C = X^E$ по модулю n.
7. Для дешифрирования вычисляется $X = C^D$ по модулю n.

Таким образом, чтобы зашифровать сообщение, необходимо знать пару чисел (E, n), а чтобы расшифровать — пару чисел (D, n). Первая пара — это открытый ключ, а вторая — закрытый.

Зная открытый ключ (E, n), можно вычислить значение закрытого ключа D. Необходимым промежуточным действием в этом преобразовании является нахождение чисел p и q, для чего нужно разложить на простые множители очень большое число n, а на это требуется очень много времени. Именно с огромной вычислительной сложностью разложения большого числа на простые множители связана высокая криптостойкость алгоритма RSA. В некоторых публикациях приводятся следующие оценки: для того чтобы найти разложение 200-значного числа, понадобится 4 миллиарда лет работы компьютера с быстродействием миллион операций в секунду. Однако следует учесть, что в настоящее время активно ведутся работы по совершенствованию методов разложения больших чисел, поэтому в алгоритме RSA стараются применять числа длиной более 200 десятичных разрядов.

Программная реализация криптоалгоритмов типа RSA значительно сложнее и менее производительна реализации классических криптоалгоритмов типа DES. Вследствие сложности реализации операций модульной арифметики криптоалгоритм RSA часто используют только для шифрования небольших объемов информации, например для рассылки классических секретных ключей или в алгоритмах цифровой подписи, а основную часть пересылаемой информации шифруют с помощью симметричных алгоритмов.

В табл. 12.1 приведены некоторые сравнительные характеристики классического криптоалгоритма DES и криптоалгоритма RSA.

Таблица 12.1. Сравнительные характеристики алгоритмов шифрования

Характеристика	DES	RSA
Скорость шифрования	Высокая	Низкая
Используемая функция шифрования	Перестановка и подстановка	Возведение в степень
Длина ключа	56 бит	Более 500 бит
Наименее затратный криптоанализ (его сложность определяет стойкость алгоритма)	Перебор по всему ключевому пространству	Разложение числа на простые множители
Время генерации ключа	Миллисекунды	Минуты
Тип ключа	Симметричный	Асимметричный

Односторонние функции шифрования

Во многих базовых технологиях безопасности используется еще один прием шифрования — шифрование с помощью односторонней функции (one-way function), называемой также хэш-функцией (hash function), или дайджест-функцией (digest function).

Эта функция, примененная к шифруемым данным, дает в результате значение (дайджест), состоящее из фиксированного небольшого числа байтов (рис. 12.5, *а*). Дайджест передается вместе с исходным сообщением. Получатель сообщения, зная, какая односторонняя функция шифрования (ОФШ) была применена для получения дайджеста, заново вычисляет его, используя незашифрованную часть сообщения. Если значения полученного и вычисленного дайджестов совпадают, значит, содержимое сообщения не было подвергнуто никаким изменениям. Знание дайджеста не дает возможности восстановить исходное сообщение, зато позволяет проверить целостность данных.

Дайджест является своего рода контрольной суммой для исходного сообщения. Однако имеется и существенное отличие. Использование контрольной суммы является средством проверки целостности передаваемых сообщений по ненадежным линиям связи. Это средство не направлено на борьбу со злоумышленниками, которым в такой ситуации ничто не мешает подменить сообщение, добавив к нему новое значение контрольной суммы. Получатель в этом случае не заметит никакой подмены.

В отличие от контрольной суммы, при вычислении дайджеста требуются секретные ключи. В случае если для получения дайджеста использовалась односторонняя функция с параметром, который известен только отправителю и получателю, любая модификация исходного сообщения будет немедленно обнаружена.

На рис. 12.5, *б* показан другой вариант использования односторонней функции шифрования для обеспечения целостности данных. В данном случае одно-

сторонняя функция не имеет параметра-ключа, но зато применяется не просто к сообщению, а к сообщению, дополненному секретным ключом. Получатель, извлекая исходное сообщение, также дополняет его тем же известным ему секретным ключом, после чего применяет к полученным данным одностороннюю функцию. Результат вычислений сравнивается с полученным по сети дайджестом.

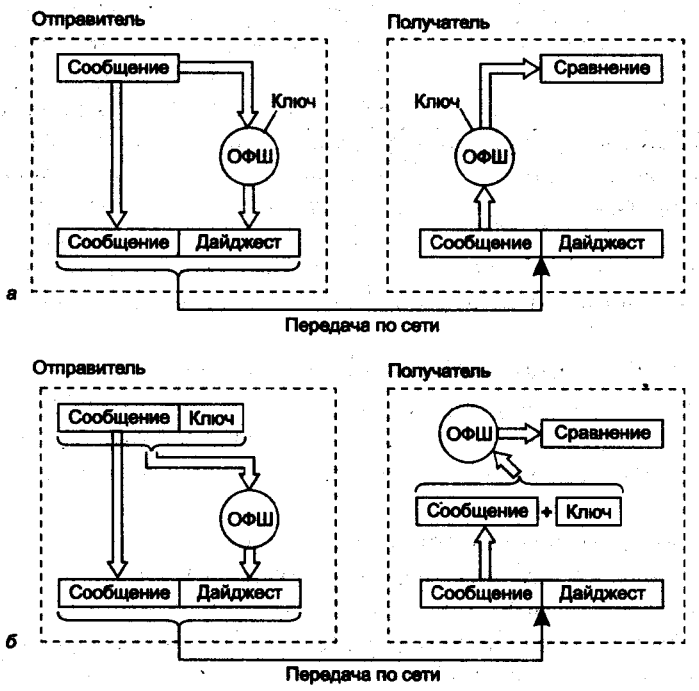


Рис. 12.5. Односторонние функции шифрования

Помимо обеспечения целостности сообщений дайджест может быть использован в качестве электронной подписи для аутентификации передаваемого документа.

Построение односторонних функций является трудной задачей. Такого рода функции должны удовлетворять двум условиям:

- по дайджесту, вычисленному с помощью данной функции, должно быть невозможно каким-либо образом вычислить исходное сообщение;
- должна отсутствовать возможность вычисления двух разных сообщений, для которых с помощью данной функции могли быть вычислены одинаковые дайджесты.

Наиболее популярными в системах безопасности в настоящее время является серия хэш-функций MD2, MD4, MD5. Все они генерируют дайджесты фиксированной длины 16 байт. Адаптированным вариантом MD4 является американский стандарт SHA, длина дайджеста в котором составляет 20 байт. Компания

IBM поддерживает односторонние функции MDC2 и MDC4, основанные на алгоритме шифрования DES.

Аутентификация, авторизация, аудит

Аутентификация

Аутентификация (authentication) предотвращает доступ к сети нежелательных лиц и разрешает вход для легальных пользователей. Термин «аутентификация» в переводе с латинского означает «установление подлинности». Аутентификацию следует отличать от идентификации. Идентификаторы пользователей применяются в системе с теми же целями, что и идентификаторы любых других объектов: файлов, процессов, структур данных, но они не связаны непосредственно с обеспечением безопасности. Идентификация заключается в сообщении пользователем системе своего идентификатора, в то время как аутентификация — это процедура доказательства пользователем того, что он есть тот, за кого себя выдает, в частности доказательство того, что именно ему принадлежит введенный им идентификатор.

В процедуре аутентификации участвует две стороны: одна сторона доказывает свою аутентичность, предъявляя некоторые доказательства, а другая сторона — аутентификатор — проверяет эти доказательства и принимает решение. В качестве доказательства аутентичности используются самые разнообразные приемы:

- аутентифицируемый может продемонстрировать знание некоего общего для обеих сторон секрета: слова (пароля) или факта (даты и места события, прозвища человека и т. п.);
- аутентифицируемый может продемонстрировать, что он владеет неким уникальным предметом (физическим ключом), в качестве которого может выступать, например, электронная магнитная карта;
- аутентифицируемый может доказать свою идентичность, используя собственные биохарактеристики: рисунок радужной оболочки глаза или отпечатки пальцев, которые предварительно были занесены в базу данных аутентификатора.

Сетевые службы аутентификации строятся на основе всех этих приемов, но чаще всего для доказательства идентичности пользователя применяют пароли. Простота и логическая ясность механизмов аутентификации на основе паролей в какой-то степени компенсирует известные слабости паролей. Это, во-первых, возможность раскрытия и разгадывания паролей, во-вторых, возможность «подслушивания» пароля путем анализа сетевого трафика. Для снижения уровня угрозы раскрытия паролей администраторы сети, как правило, применяют встроенные программные средства, служащие для формирования политики назначения и использования паролей: задание максимального и минимального сроков действия пароля, хранение списка уже использованных паролей, управление поведением системы после нескольких неудачных попыток логического входа и т. п.

Перехват паролей по сети можно предупредить путем их шифрования перед передачей в сеть.

Легальность пользователя может устанавливаться по отношению к различным системам. Так, работая в сети, пользователь может проходить процедуру аутентификации и как локальный пользователь, который претендует на ресурсы только данного компьютера, и как пользователь сети, желающий получить доступ ко всем сетевым ресурсам. При локальной аутентификации пользователь вводит свои идентификатор и пароль, которые автономно обрабатываются операционной системой, установленной на данном компьютере. При логическом входе в сеть данные о пользователе (идентификатор и пароль) передаются на сервер, который хранит учетные записи всех пользователей сети. Многие приложения имеют свои средства определения, является ли пользователь законным. И тогда пользователю приходится проходить дополнительные этапы проверки.

В качестве объектов, требующих аутентификации, могут выступать не только пользователи, но и различные устройства, приложения, текстовая и другая информация. Так, например, пользователь, обращающийся с запросом к корпоративному серверу, должен доказать ему свою легальность, но он также должен убедиться сам, что ведет диалог действительно с сервером своего предприятия. Другими словами, сервер и клиент должны пройти процедуру взаимной аутентификации. Здесь мы имеем дело с аутентификацией на уровне приложений. При установлении сеанса связи между двумя устройствами также часто предусматриваются процедуры взаимной аутентификации на более низком, канальном уровне. Примером такой процедуры является аутентификация по протоколам PAP и CHAP, входящим в семейство протоколов PPP. Аутентификация данных означает доказательство целостности этих данных, а также то, что они поступили именно от того человека, который объявил об этом. Для этого используется механизм электронной подписи.

В вычислительных сетях процедуры аутентификации часто реализуются теми же программными средствами, что и процедуры авторизации. В отличие от аутентификации, которая позволяет распознать легальных и нелегальных пользователей, система авторизации имеет дело только с легальными пользователями, успешно прошедшими процедуру аутентификации. Цель подсистем авторизации состоит в том, чтобы предоставить каждому легальному пользователю именно те виды доступа и к тем ресурсам, которые были для него определены администратором системы.

Авторизация доступа

Средства **авторизации** (authorization) контролируют доступ легальных пользователей к ресурсам системы, предоставляя каждому из них именно те права, которые ему были определены администратором. Помимо предоставления пользователям прав доступа к каталогам, файлам и принтерам, система авторизации может контролировать возможность выполнения пользователями различных

системных функций, таких как локальный доступ к серверу, установка системного времени, создание резервных копий данных, выключение сервера и т. п.

Система авторизации наделяет пользователя сети правами выполнять определенные действия над определенными ресурсами. Для этого могут быть использованы различные формы предоставления правил доступа, которые часто делят на два класса:

- избирательный доступ;
- мандатный доступ.

Избирательные права доступа реализуются в операционных системах универсального назначения. В наиболее распространенном варианте такого подхода определенные операции с определенным ресурсом разрешаются или запрещаются пользователям или группам пользователей, явно указанным своими *идентификаторами*. Например, пользователю, имеющему идентификатор User_T, может быть разрешено выполнять операции чтения и записи по отношению к файлу File1. Модификацией этого способа является идентификация пользователей по их *должностям*, по их принадлежности к персоналу того или иного производственного подразделения или еще по каким-либо другим позиционирующим характеристикам. Примером такого правила может служить следующее: файл бухгалтерской отчетности BUCH могут читать работники бухгалтерии и руководитель предприятия.

Мандатный подход к определению прав доступа заключается в том, что вся информация делится на уровни в зависимости от степени секретности, а все пользователи сети также делятся на группы, образующие иерархию в соответствии с *уровнем допуска* к этой информации. Такой подход используется в известном делении информации на информацию для служебного пользования, секретную, совершенно секретную. При этом пользователи этой информации, в зависимости от определенного для них статуса, получают разные формы допуска: первую, вторую или третью. В отличие от систем с избирательными правами доступа, в системах с мандатным подходом пользователи в принципе не имеют возможности изменить уровень доступности информации. Например, пользователь более высокого уровня не может разрешить читать данные из своего файла пользователю, относящемуся к более низкому уровню. Отсюда видно, что мандатный подход является более строгим, он в корне пресекает всякий волюнтаризм со стороны пользователя. Именно поэтому он больше характерен для систем военного назначения.

Процедуры авторизации реализуются программными средствами, которые могут встраиваться в операционную систему или приложение, а также поставляться в виде отдельных программных продуктов. При этом программные системы авторизации могут строиться на базе двух схем:

- централизованная схема авторизации, базирующаяся на сервере;
- децентрализованная схема, базирующаяся на рабочих станциях.

В первой схеме сервер управляет процессом предоставления ресурсов пользователю. Главная цель таких систем — реализовать «принцип единого входа».

В соответствии с централизованной схемой пользователь один раз логически входит в сеть и получает на все время работы некоторый набор разрешений по доступу к ресурсам сети. Система Kerberos с ее сервером безопасности и архитектурой клиент-сервер является наиболее известной системой этого типа. Системы TACACS и RADIUS, часто применяемые совместно с системами удаленного доступа, также реализуют этот подход.

При втором подходе рабочая станция сама является защищенной — средства защиты работают на каждой машине, и сервер не требуется. Рассмотрим систему, в которой процедура однократного логического входа не предусмотрена. Теоретически, доступ к каждому приложению должен контролироваться средствами безопасности самого приложения или же средствами, существующими в той операционной среде, в которой оно работает. В корпоративной сети администратору придется отслеживать работу механизмов безопасности, используемых всеми типами приложений — электронной почтой, службой каталогов локальной сети, базами данных хостов и т. п. Когда администратору приходится добавлять или удалять пользователей, то часто требуется вручную конфигурировать доступ к каждой программе или системе.

В крупных сетях часто применяется комбинированный подход предоставления пользователю прав доступа к ресурсам сети: сервер удаленного доступа ограничивает доступ пользователя к подсетям или серверам корпоративной сети, то есть к укрупненным элементам сети, а каждый отдельный сервер сети сам по себе ограничивает доступ пользователя к своим внутренним ресурсам: разделяемым каталогам, принтерам или приложениям. Сервер удаленного доступа предоставляет доступ на основании имеющегося у него списка прав доступа пользователя, или списка контроля доступа (Access Control List, ACL), а каждый отдельный сервер сети предоставляет доступ к своим ресурсам на основании хранящегося у него списка прав доступа, например ACL файловой системы.

Подчеркнем, что системы аутентификации и авторизации совместно решают одну задачу, поэтому к ним необходимо предъявлять одинаковый уровень требований. Ненадежность одного звена здесь не может быть компенсирована надежностью другого. Если при аутентификации используются пароли, то требуются чрезвычайные меры по их защите. Однажды украденный пароль открывает двери ко всем приложениям и данным, к которым пользователь с этим паролем имел легальный доступ.

Аудит

Аудитом (auditing) называют фиксацию в системном журнале событий, связанных с доступом к защищаемым системным ресурсам. Подсистема аудита современных ОС позволяет дифференцированно задавать перечень интересующих администратора событий с помощью удобного графического интерфейса. Средства учета и наблюдения обеспечивают возможность обнаружить и зафиксировать важные события, связанные с безопасностью, или любые попытки создать,

получить доступ или удалить системные ресурсы. Аудит позволяет засекать даже неудачные попытки «взлома» системы.

Учет и наблюдение означает способность системы безопасности «шпионить» за выбранными объектами и их пользователями и выдавать сообщения тревоги, когда кто-нибудь пытается читать или модифицировать системный файл. Если кто-то пытается выполнить действия, выбранные системой безопасности для мониторинга, то система аудита пишет сообщение в журнал регистрации, идентифицируя пользователя. Системный менеджер может создавать отчеты о безопасности, которые содержат информацию из журнала регистрации. Для «сверхбезопасных» систем предусматриваются аудио- и видеосигналы тревоги, устанавливаемые на машинах администраторов, отвечающих за безопасность.

Поскольку никакая система безопасности не гарантирует защиту на уровне 100 %, то последним рубежом в борьбе с нарушениями оказывается система аудита. Действительно, после того как злоумышленнику удалось провести успешную атаку, пострадавшей стороне не остается ничего другого, как обратиться к службе аудита. Если при настройке службы аудита были правильно заданы события, которые требуется отслеживать, то подробный анализ записей в журнале может дать много полезной информации. Эта информация, возможно, позволит найти злоумышленника или, по крайней мере, предотвратить повторение подобных атак путем устранения уязвимых мест в системе защиты.

Технология защищенного канала

Как уже было сказано, задачу защиты данных можно разделить на две подзадачи: защиту данных внутри компьютера и защиту данных в процессе их передачи из одного компьютера в другой. Для обеспечения безопасности данных при их передаче по публичным сетям используются различные технологии защищенного канала.

Технология защищенного канала призвана обеспечивать безопасность передачи данных по открытой транспортной сети, например по Интернету. Защищенный канал подразумевает выполнение трех основных функций:

- взаимная аутентификация абонентов при установлении соединения, которая может быть выполнена, например, путем обмена паролями;
- защита передаваемых по каналу сообщений от несанкционированного доступа, например, путем шифрования;
- подтверждение целостности поступающих по каналу сообщений, например, путем передачи одновременно с сообщением его дайджеста.

Совокупность защищенных каналов, созданных предприятием в публичной сети для объединения своих филиалов, часто называют **виртуальной частной сетью** (Virtual Private Network, VPN).

Существуют разные реализации технологии защищенного канала, которые, в частности, могут работать на разных уровнях модели OSI. Так функции популярного протокола SSL соответствуют *представительному* уровню модели OSI.

Новая версия *сетевого* протокола IP предусматривает все функции — взаимную аутентификацию, шифрование и обеспечение целостности, — которые по определению свойственны защищенному каналу, а протокол туннелирования PPTP защищает данные на *канальном* уровне.

В зависимости от места расположения программного обеспечения защищенного канала различают две схемы его образования:

- схема с конечными узлами, взаимодействующими через публичную сеть (рис. 12.6, *а*);
- схема с оборудованием поставщика услуг публичной сети, расположенным на границе между частной и публичной сетями (рис. 12.6, *б*).

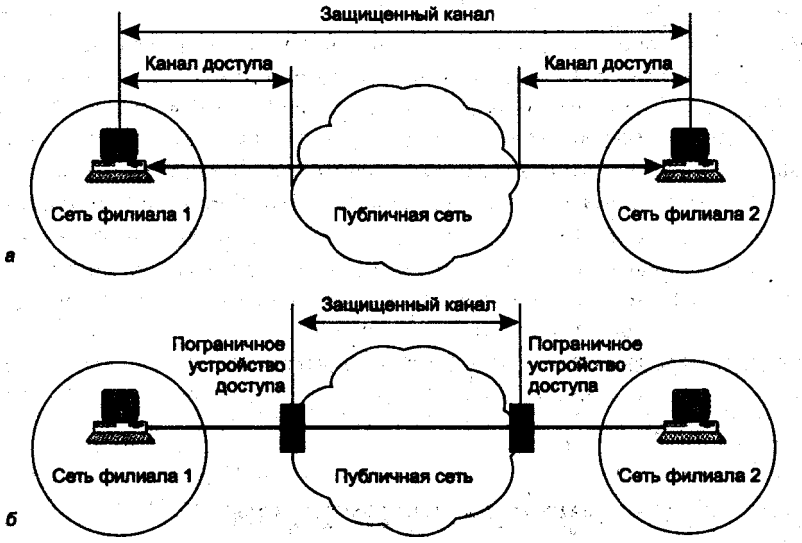


Рис. 12.6. Два подхода к образованию защищенного канала

В первом случае защищенный канал образуется программными средствами, установленными на двух удаленных компьютерах, принадлежащих двум разным локальным сетям одного предприятия и связанных между собой через публичную сеть. Преимуществом этого подхода является полная защищенность канала вдоль всего пути следования, а также возможность использования любых протоколов создания защищенных каналов, лишь бы на конечных точках канала поддерживался один и тот же протокол. Недостатки заключаются в избыточности и децентрализованности решения. Избыточность состоит в том, что вряд ли стоит создавать защищенный канал на всем пути прохождения данных: уязвимыми для злоумышленников обычно являются сети с коммутацией пакетов, а не каналы телефонной сети или выделенные каналы, через которые локальные сети подключены к территориальной сети. Поэтому защиту каналов доступа к публичной сети можно считать избыточной. Децентрализация заключается в том, что для каждого компьютера, которому требуется предоставить

услуги защищенного канала, необходимо отдельно устанавливать, конфигурировать и администрировать программные средства защиты данных. Подключение каждого нового компьютера к защищенному каналу требует выполнять эти трудоемкие операции заново.

Во втором случае клиенты и серверы не участвуют в создании защищенного канала — он прокладывается только внутри публичной сети с коммутацией пакетов, например внутри Интернета. Так, канал может быть проложен между сервером удаленного доступа поставщика услуг публичной сети и пограничным маршрутизатором корпоративной сети. Это хорошо масштабируемое решение, управляемое централизовано администраторами как корпоративной сети, так и сети поставщика услуг. Для компьютеров корпоративной сети канал прозрачен — программное обеспечение этих конечных узлов остается без изменений. Такой гибкий подход позволяет легко образовывать новые каналы защищенного взаимодействия между компьютерами независимо от места их расположения. Реализация этого подхода сложнее — нужен стандартный протокол образования защищенного канала, требуется установка у всех поставщиков услуг программного обеспечения, поддерживающего такой протокол, необходима поддержка протокола производителями пограничного коммуникационного оборудования. Однако вариант, когда все заботы по поддержанию защищенного канала берет на себя поставщик услуг публичной сети, оставляет сомнения в надежности защиты: во-первых, незащищенными оказываются каналы доступа к публичной сети, во-вторых, потребитель услуг чувствует себя в полной зависимости от надежности поставщика услуг. И тем не менее специалисты прогнозируют, что именно вторая схема в ближайшем будущем станет основной в построении защищенных каналов.

Технологии аутентификации

Сетевая аутентификация на основе многоразового пароля

В соответствии с базовым принципом «единого входа», когда пользователю достаточно один раз пройти процедуру аутентификации, чтобы получить доступ ко всем сетевым ресурсам, в современных операционных системах предусматриваются централизованные службы аутентификации. Такая служба поддерживается одним из серверов сети и использует для своей работы базу данных, в которой хранятся учетные данные (иногда называемые бюджетами) о пользователях сети. Учетные данные содержат наряду с другой информацией идентификаторы и пароли пользователей. Упрощенно схема аутентификации в сети выглядит следующим образом. Когда пользователь осуществляет логический вход в сеть, он набирает на клавиатуре своего компьютера свои идентификатор и пароль. Эти данные используются службой аутентификации — в централизованной базе данных, хранящейся на сервере, по идентификатору пользователя находится соответствующая запись, из нее извлекается пароль и сравнивается

с тем, который ввел пользователь. Если они совпадают, то аутентификация считается успешной, пользователь получает легальный статус и те права, которые определены для него системой авторизации.

Однако такая упрощенная схема имеет большой изъян. А именно — при передаче пароля с клиентского компьютера на сервер, выполняющий процедуру аутентификации, этот пароль может быть перехвачен злоумышленником. Поэтому в разных операционных системах применяются разные приемы, чтобы избежать передачи пароля по сети в незащищенном виде. Рассмотрим, как эта проблема была решена в ОС Windows NT.

В основе концепции сетевой безопасности ОС Windows NT лежит понятие домена, которое в данном случае означает совокупность пользователей, серверов и рабочих станций, учетная информация о которых централизованно хранится в общей базе данных, называемой SAM (Security Accounts Manager database). Над этой базой данных реализована служба Directory Services, которая, как и любая централизованная справочная служба, устраняет дублирование учетных данных в нескольких компьютерах и сокращает число рутинных операций по администрированию. Одной из функций службы Directory Services является аутентификация пользователей. Служба Directory Services построена в архитектуре клиент-сервер. Каждый пользователь при логическом входе в сеть вызывает клиентскую часть службы, которая передает запрос на аутентификацию и поддерживает диалог с серверной частью.

Аутентификация пользователей домена выполняется на основе их паролей, хранящихся в зашифрованном виде в базе SAM (рис. 12.7). Пароли зашифровываются с помощью односторонней функции шифрования (ОФШ) при занесении их в базу данных во время процедуры создания учетной записи для нового пользователя. Введем обозначение для этой односторонней функции — ОФШ1. Таким образом, пароль *P* хранится в базе данных SAM в виде дайджеста *d(P)*. (Напомним, что знание дайджеста не позволяет восстановить исходное сообщение.)

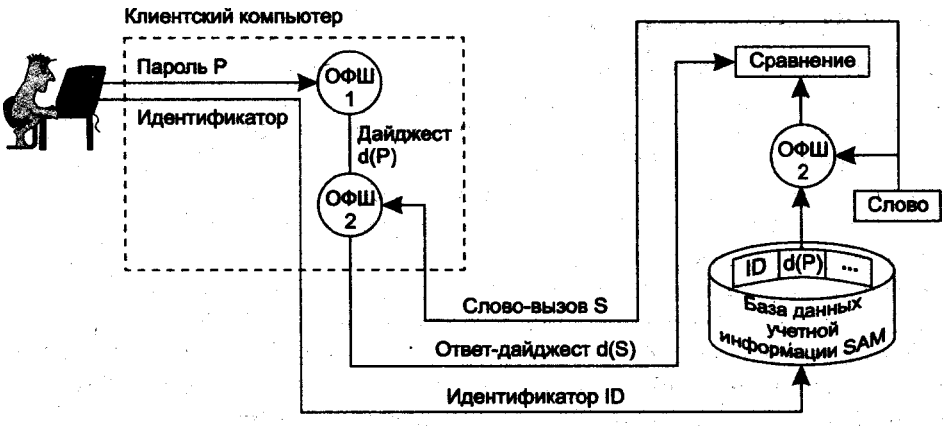


Рис. 12.7. Схема сетевой аутентификации на основе многоразового пароля

При логическом входе пользователь локально вводит в свой компьютер имя-идентификатор (ID) и пароль *P*. Клиентская часть подсистемы аутентификации, получив эти данные, передает запрос по сети на сервер, хранящий базу SAM. В этом запросе в открытом виде содержится идентификатор пользователя ID, но пароль не передается в сеть ни в каком виде.

К паролю на клиентской станции применяется та же односторонняя функция ОФШ1, которая была использована при записи пароля в базу данных SAM, то есть динамически вычисляется дайджест пароля $d(P)$.

В ответ на поступивший запрос серверная часть службы аутентификации генерирует случайное число *S* случайной длины, называемое словом-вызовом (challenge). Это слово передается по сети с сервера на клиентскую станцию пользователя. К слову-вызову на клиентской стороне применяется односторонняя функция шифрования ОФШ2. В отличие от функции ОФШ1, функция ОФШ2 является параметрической и получает в качестве параметра дайджест пароля $d(P)$. Полученный в результате ответ $d(S)$ передается по сети на сервер базы SAM.

Параллельно этому на сервере слово-вызов *S* аналогично шифруется с помощью той же односторонней функции ОФШ2 и дайджеста пароля пользователя $d(P)$, извлеченного из базы SAM, а затем сравнивается с ответом, переданным клиентской станцией. При совпадении результатов считается, что аутентификация прошла успешно. Таким образом, при логическом входе пользователя пароли в сети ОС Windows NT никогда не передаются по каналам связи.

Заметим также, что при каждом запросе на аутентификацию генерируется новое слово-вызов, так что перехват ответа $d(S)$ клиентского компьютера не может быть использован в ходе другой процедуры аутентификации.

Аутентификация с использованием одноразового пароля

Алгоритмы аутентификации, основанные на многоразовых паролях, не очень ненадежны. Пароли можно подсмотреть или просто украсть. Более надежными оказываются схемы с *одноразовыми паролями*. К тому же одноразовые пароли намного дешевле и проще биометрических систем аутентификации, таких как сканеры сетчатки глаза или отпечатков пальцев. Все это делает системы, основанные на одноразовых паролях, очень перспективными. Следует иметь в виду, что, как правило, системы аутентификации на основе одноразовых паролей рассчитаны на проверку только удаленных, а не локальных пользователей.

Генерация одноразовых паролей может выполняться либо программно, либо аппаратно. Некоторые аппаратные реализации систем доступа на основе одноразовых паролей представляют собой миниатюрные устройства со встроенным микропроцессором, похожие на обычные пластиковые карточки, используемые для доступа к банкоматам. Такие карточки, часто называемые **аппаратными ключами**, могут иметь клавиатуру и маленькое дисплейное окно. Аппаратные ключи могут быть также реализованы в виде присоединяемого к разъему

устройства, которое располагается между компьютером и модемом, или в виде карты (гибкого диска), вставляемой в дисковод компьютера.

Существуют и программные реализации средств аутентификации на основе одноразовых паролей (**программные ключи**). Программные ключи размещаются на сменном магнитном диске в виде обычной программы, важной частью которой является генератор одноразовых паролей. Применение программных ключей и присоединяемых к компьютеру карт связано с некоторым риском, так как пользователи часто забывают гибкие диски в машине или не отсоединяют карты от ноутбуков.

Независимо от того, какую реализацию системы аутентификации на основе одноразовых паролей выбирает пользователь, он, как и в системах аутентификации с применением многоразовых паролей, сообщает системе свой идентификатор, однако вместо того, чтобы вводить каждый раз один и тот же пароль, он указывает последовательность цифр, сообщаемую ему аппаратным или программным ключом. Через определенный небольшой период времени генерируется другая последовательность — новый пароль. Сервер аутентификации проверяет введенную последовательность и разрешает пользователю осуществить логический вход. Сервер аутентификации может представлять собой отдельное устройство, выделенный компьютер или же программу, выполняемую на обычном сервере.

Рассмотрим подробнее две схемы, основанные на использовании аппаратных ключей.

Синхронизация по времени

Механизм аутентификации в значительной степени зависит от производителя. Одной из наиболее популярной является схема, разработанная компанией Security Dynamics (рис. 12.8). Эта схема основана на алгоритме, который через определенный интервал времени (изменяемый при желании администратором сети) генерирует случайное число. Алгоритм использует два параметра:

- секретный ключ, представляющий собой 64-разрядное число, уникально назначаемое каждому пользователю и хранящееся и в аппаратном ключе, и в базе данных сервера аутентификации;
- значение текущего времени.

Когда удаленный пользователь пытается совершить логический вход в сеть, то ему предлагается ввести его личный персональный номер (PIN), состоящий из 4 десятичных цифр, а также 6 цифр случайного числа, отображаемого в тот момент на дисплее аппаратного ключа. На основе PIN-кода сервер извлекает из базы данных информацию о пользователе, а именно — его секретный ключ. Затем сервер выполняет алгоритм генерации случайного числа, используя в качестве параметров найденный секретный ключ и значение текущего времени, проверяя, совпадает ли сгенерированное число с числом, которое ввел пользователь. Если они совпадают, то пользователю разрешается логический вход.

Потенциальной проблемой этой схемы является временная синхронизация сервера и аппаратного ключа (ясно, что вопрос согласования часовых поясов

решается просто). Гораздо сложнее обстоит дело с постепенным рассогласованием внутренних часов сервера и аппаратного ключа, тем более что потенциально аппаратный ключ может работать несколько лет. Компания Security Dynamics решает эту проблему двумя способами. Во-первых, при производстве аппаратного ключа измеряется отклонение частоты его таймера от номинала. Далее эта величина учитывается в виде параметра алгоритма сервера. Во-вторых, сервер отслеживает коды, генерируемые конкретным аппаратным ключом, и если таймер данного ключа постоянно спешит или отстает, то сервер динамически подстраивается под него.

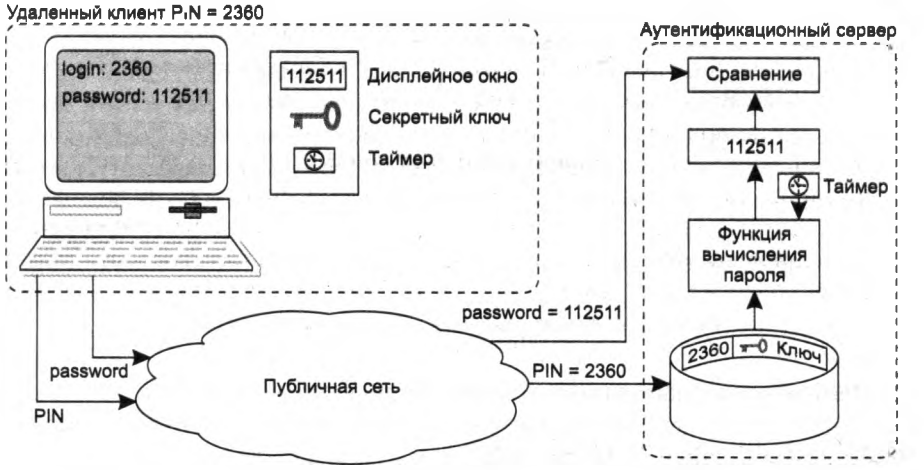


Рис. 12.8. Аутентификация, основанная на временной синхронизации

Существует еще одна проблема, связанная со схемой временной синхронизации. Случайное число, генерируемое аппаратным ключом, является достоверным паролем в течение определенного интервала времени. Теоретически возможно, что очень проворный хакер может перехватить PIN-код и случайное число и использовать их для доступа к какому-либо серверу сети.

Использование слова-вызова

Другая схема применения аппаратных ключей основана на идее, очень сходной с рассмотренной ранее идеей сетевой аутентификации. В том и другом случаях используется слово-вызов. Такая схема получила название «запрос-ответ». Когда пользователь пытается осуществить логический вход, то сервер аутентификации передает ему запрос в виде случайного числа (рис. 12.9). Аппаратный ключ пользователя зашифровывает это случайное число с помощью алгоритма DES и секретного ключа пользователя. Секретный ключ пользователя хранится в базе данных сервера и в памяти аппаратного ключа. В зашифрованном виде слово-вызов возвращается на сервер. Сервер, в свою очередь, также зашифровывает сгенерированное им самим случайное число с помощью алгоритма DES и того же секретного ключа пользователя, а затем сравнивает результат

с числом, полученным от аппаратного ключа. Как и в методе временной синхронизации, в случае совпадения этих двух чисел пользователю разрешается вход в сеть.

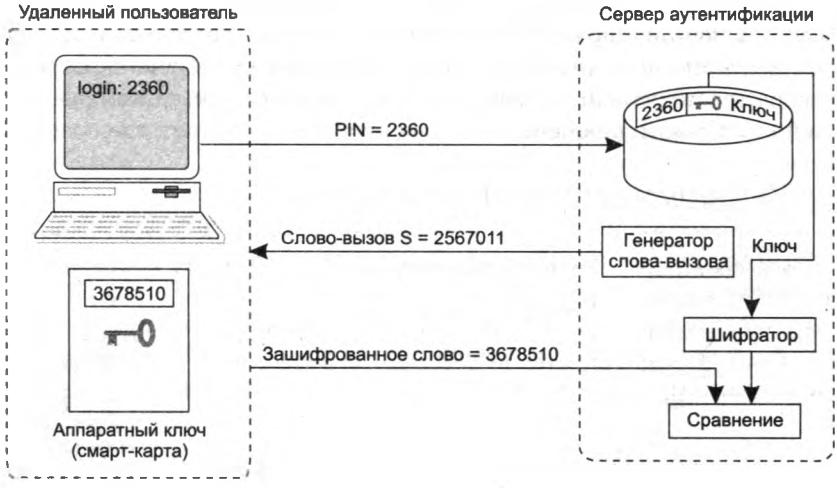


Рис. 12.9. Аутентификация по схеме «запрос-ответ»

Механизм слова-вызова имеет свои ограничения — он обычно требует наличия компьютера на каждом конце соединения, так как аппаратный ключ должен иметь возможность как получать, так и отправлять информацию. А схема временной синхронизации позволяет ограничиться простым терминалом или факсом. В этом случае пользователи могут даже вводить свой пароль с телефонной клавиатуры, когда звонят в сеть для получения голосовой почты.

Схема «запрос-ответ» уступает схеме временной синхронизации в простоте использования. Для логического входа по схеме временной синхронизации пользователю достаточно набрать 10 цифр. Схемы же «запрос-ответ» могут потребовать от пользователя выполнения большего числа ручных действий. В некоторых схемах «запрос-ответ» пользователь должен сам вводить секретный ключ, а затем набирать на клавиатуре компьютера полученное с помощью аппаратного ключа зашифрованное слово-вызов. В некоторых случаях пользователь должен вторично совершить логический вход в коммуникационный сервер уже после аутентификации.

Аутентификация на основе сертификатов

Аутентификация с применением цифровых сертификатов является альтернативой применению паролей и представляется естественным решением в условиях, когда число пользователей сети (пусть и потенциальных) измеряется миллионами. В таких обстоятельствах процедура предварительной регистрации пользователей, связанная с назначением и хранением их паролей, становится крайне обременительной, опасной, а иногда и просто нереализуемой. При нали-

чии сертификатов сеть, которая дает пользователю доступ к своим ресурсам, не хранит никакой информации о своих пользователях — они ее предоставляют сами в своих запросах в виде сертификатов, удостоверяющих личность пользователей. Сертификаты выдаются специальными уполномоченными организациями — **центрами сертификации** (Certificate Authority, CA). Поэтому задача хранения секретной информации (закрытых ключей) возлагается на самих пользователей, что делает это решение гораздо более масштабируемым, чем вариант с централизованной базой паролей.

Схема использования сертификатов

Аутентификация личности на основе сертификатов происходит примерно так же, как на проходной большого предприятия. Вахтер пропускает людей на территорию на основании пропуска, который содержит фотографию и подпись сотрудника, удостоверенных печатью предприятия и подписью лица, выдавшего пропуск. Сертификат является аналогом пропуска и выдается по запросам специальными сертифицирующими центрами при выполнении определенных условий.

Сертификат представляет собой электронную форму, в которой содержится следующая информация:

- открытый ключ владельца данного сертификата;
- сведения о владельце сертификата, такие, например, как имя, адрес электронной почты, наименование организации, в которой он работает, и т. п.;
- наименование сертифицирующей организации, выдавшей данный сертификат.

Кроме того, сертификат содержит электронную подпись сертифицирующей организации — зашифрованные закрытым ключом этой организации данные, содержащиеся в сертификате.

Использование сертификатов основано на предположении, что сертифицирующих организаций немного и их открытые ключи могут быть всеми получены каким-либо способом, например, из публикаций в журналах.

Когда пользователь хочет подтвердить свою личность, он предъявляет свой сертификат в двух формах — открытой (то есть такой, в которой он получил его в сертифицирующей организации) и зашифрованной с применением своего закрытого ключа (рис. 12.10). Сторона, проводящая аутентификацию, берет из открытого сертификата открытый ключ пользователя и расшифровывает с его помощью зашифрованный сертификат. Совпадение результата с открытым сертификатом подтверждает, что предъявитель действительно является владельцем закрытого ключа, соответствующего указанному открытому.

Затем с помощью известного открытого ключа указанной в сертификате организации проводится расшифровка подписи этой организации в сертификате. Если в результате получается тот же сертификат с тем же именем пользователя и его открытым ключом, значит, он действительно прошел регистрацию в сертификационном центре, является тем, за кого себя выдает, и указанный в сертификате открытый ключ действительно принадлежит ему.

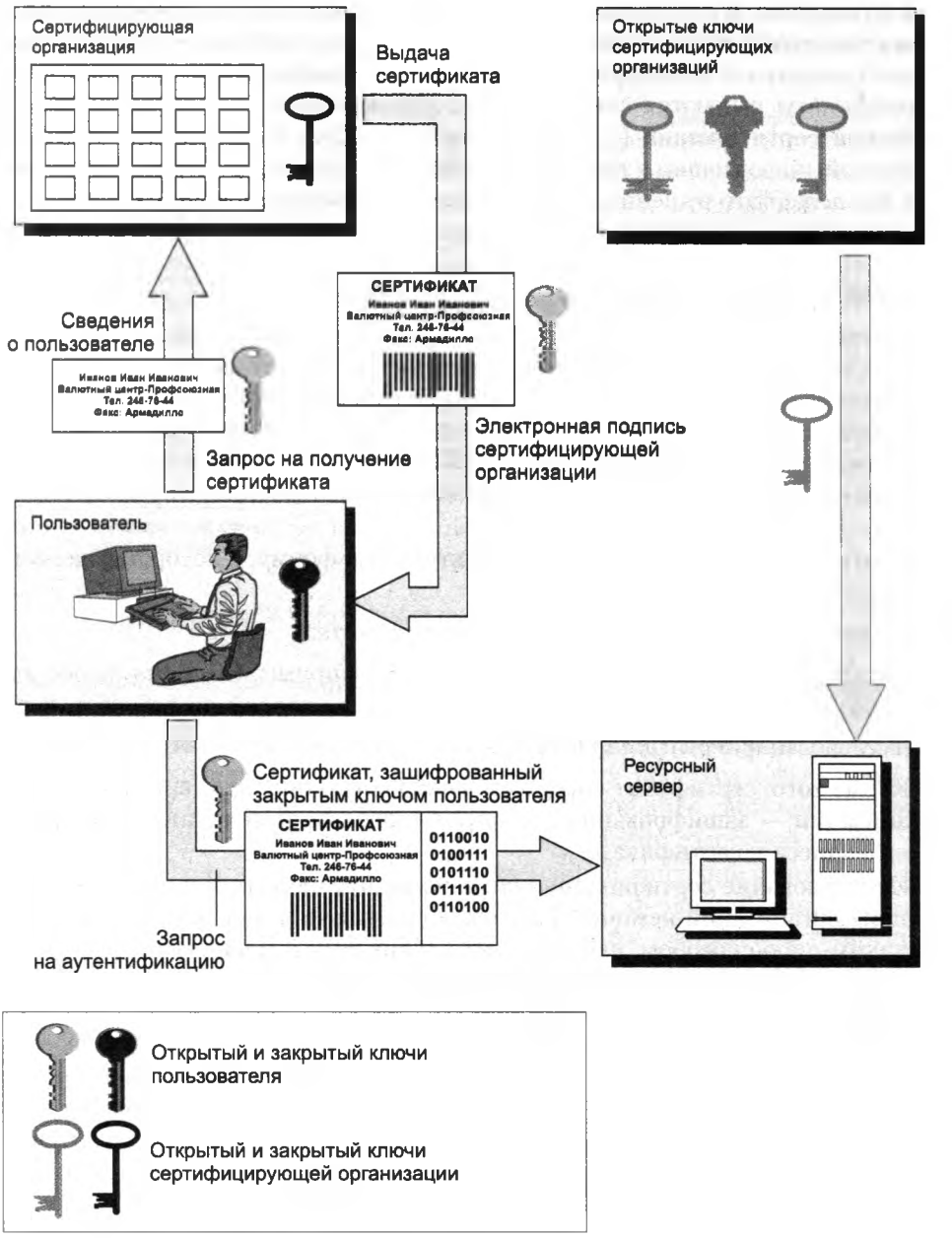


Рис. 12.10. Аутентификация пользователей на основе сертификатов

Сертификаты можно использовать не только для аутентификации, но и для предоставления избирательных прав доступа. Для этого в сертификат могут вводиться дополнительные поля, в которых указывается принадлежность его

владельцев той или иной категории пользователей. Эта категория назначается сертифицирующей организацией в зависимости от условий, на которых выдается сертификат. Например, организация, поставляющая через Интернет на коммерческой основе информацию, может выдавать сертификаты определенной категории пользователям, оплатившим годовую подписку на некоторый бюллетень, а веб-сервер будет предоставлять доступ к страницам бюллетеня только пользователям, предъявившим сертификат данной категории.

Подчеркнем тесную связь открытых ключей с сертификатами. Сертификат является удостоверением не только личности, но и принадлежности открытого ключа. Цифровой сертификат устанавливает и гарантирует соответствие между открытым ключом и его владельцем. Это предотвращает угрозу подмены открытого ключа. Если некоторому абоненту поступает открытый ключ в составе сертификата, то он может быть уверен, что этот открытый ключ гарантированно принадлежит отправителю, адрес и другие сведения о котором содержатся в этом сертификате.

При использовании сертификатов отпадает необходимость хранить на серверах корпораций списки пользователей с их паролями, вместо этого достаточно иметь на сервере список имен и открытых ключей сертифицирующих организаций. Может также понадобиться некоторый механизм отображений категорий владельцев сертификатов на традиционные группы пользователей для того, чтобы можно было в неизменном виде задействовать механизмы управления избирательным доступом большинства операционных систем или приложений.

Сертифицирующие центры

Сертификат является средством аутентификации пользователя при его обращении к сетевым ресурсам, роль аутентифицирующей стороны играют при этом информационные серверы корпоративной сети или Интернета. В то же время и сама процедура получения сертификата включает этап аутентификации, здесь аутентификатором выступает сертифицирующая организация. Для получения сертификата клиент должен сообщить сертифицирующей организации свой открытый ключ и те или иные сведения, удостоверяющие его личность. Все эти данные клиент может отправить по электронной почте или принести на гибком диске лично. Перечень необходимых данных зависит от типа получаемого сертификата. Сертифицирующая организация проверяет доказательства подлинности, помещает свою цифровую подпись в файл, содержащий открытый ключ, и посылает сертификат обратно, подтверждая факт принадлежности данного конкретного ключа конкретному лицу. После этого сертификат может быть встроен в любой запрос на использование информационных ресурсов сети.

Практически важным вопросом является вопрос о том, кто имеет право выполнять функции сертифицирующей организации. Во-первых, задачу обеспечения своих сотрудников сертификатами может взять на себя само предприятие. В этом случае упрощается процедура первичной аутентификации при выдаче сертификата. Предприятия достаточно осведомлены о своих сотрудниках, чтобы

брать на себя задачу подтверждения их личности. Для автоматизации процесса генерации, выдачи и обслуживания сертификатов предприятия могут использовать готовые программные продукты, например, компания Oracle выпускает сервер сертификатов, который организации могут у себя устанавливать для выпуска своих сертификатов.

Механизм получения пользователем сертификата хорошо автоматизируется в сети в модели клиент-сервер, когда браузер исполняет роль клиента, а в сертифицирующей организации установлен специальный сервер выдачи сертификатов. Браузер вырабатывает для пользователя пару ключей, оставляет закрытый ключ у себя и передает частично заполненную форму сертификата серверу. Для того чтобы неподписанный еще сертификат нельзя было подменить при передаче по сети, браузер ставит свою электронную подпись, зашифровывая сертификат выработанным закрытым ключом. Сервер сертификатов подписывает полученный сертификат, фиксирует его в своей базе данных и возвращает его каким-либо способом владельцу. Очевидно, что при этом может выполняться еще и неформальная процедура подтверждения пользователем своей личности и права на получение сертификата, требующая участия оператора сервера сертификатов. Это могут быть доказательства оплаты услуги, доказательства принадлежности к той или иной организации — все случаи жизни предусмотреть и автоматизировать нельзя. После получения сертификата браузер сохраняет его вместе с закрытым ключом и использует при аутентификации на тех серверах, которые поддерживают такой процесс.

В настоящее время существует уже большое количество протоколов и продуктов, использующих сертификаты. Например, компания Microsoft реализовала поддержку сертификатов и в своем браузере Internet Explorer, и в сервере Internet Information Server, разработала собственный сервер сертификатов, а также продукты, которые позволяют хранить сертификаты пользователя, его закрытые ключи и пароли защищенным образом.

Инфраструктура с открытыми ключами

Несмотря на активное использование технологии цифровых сертификатов во многих системах безопасности, эта технология еще не решила целый ряд серьезных проблем. Это, прежде всего, поддержание базы данных о выпущенных сертификатах. Сертификат выдается не навсегда, а на некоторый вполне определенный срок. По истечении срока годности сертификат должен либо обновляться, либо аннулироваться. Кроме того, необходимо предусмотреть возможность досрочного прекращения полномочий сертификата. Все заинтересованные участники информационного процесса должны быть вовремя оповещены о том, что некоторый сертификат уже недействителен. Для этого сертифицирующая организация должна оперативно поддерживать список аннулированных сертификатов.

Имеется также ряд проблем, связанных с тем, что сертифицирующие организации существуют не в единственном числе. Все они выпускают сертификаты, но даже если эти сертификаты соответствуют единому стандарту (сейчас

это, как правило, стандарт X.509), все равно остаются нерешенными многие вопросы. Все ли сертифицирующие центры заслуживают доверия? Каким образом можно проверить полномочия того или иного сертифицирующего центра? Можно ли создать иерархию сертифицирующих центров, когда сертифицирующий центр, стоящий выше, мог бы сертифицировать центры, расположенные ниже по иерархии? Как организовать совместное использование сертификатов, выпущенных разными сертифицирующими организациями?

Для решения этих и многих других проблем, возникающих в системах, использующих технологии шифрования с открытыми ключами, оказывается необходимым комплекс программных средств и методик, называемый **инфраструктурой с открытыми ключами** (Public Key Infrastructure, PKI). Информационные системы больших предприятий нуждаются в специальных средствах администрирования и управления цифровыми сертификатами, парами открытых/закрытых ключей, а также приложениями, функционирующими в среде с открытыми ключами,

В настоящее время любой пользователь имеет возможность, загрузив широко доступное программное обеспечение, абсолютно бесконтрольно сгенерировать себе пару открытый/закрытый ключ. Затем он может также совершенно независимо от администрации вести шифрованную переписку со своими внешними абонентами. Такая «свобода» пользователя часто не соответствует принятой на предприятии политике безопасности. Для более надежной защиты корпоративной информации желательно реализовать централизованную службу генерации и распределения ключей. Для администрации предприятия важно иметь возможность получить копии закрытых ключей каждого пользователя сети, чтобы в случае увольнения пользователя или потери пользователем его закрытого ключа сохранить доступ к зашифрованным данным этого пользователя. В противном случае резко ухудшается одна из трех характеристик безопасной системы — доступность данных.

Процедура, позволяющая получать копии закрытых ключей, называется **восстановлением ключей**. Вопрос, включать ли в продукты безопасности средства восстановления ключей, в последние годы приобрел политический оттенок. В Соединенных Штатах Америки прошли бурные дебаты, тему которых можно примерно сформулировать так: обладает ли правительство правом иметь доступ к любой частной информации при условии, что на это есть постановление суда?

И хотя в такой широкой постановке проблема восстановления ключей все еще не решена, необходимость наличия средств восстановления в корпоративных продуктах ни у кого сомнений не вызывает. Принцип доступности данных не должен нарушаться из-за волюнтаризма сотрудников, монопольно владеющих своими закрытыми ключами. Ключ может быть восстановлен при выполнении некоторых условий, которые должны быть четко определены в политике безопасности предприятия.

Как только принимается решение о включении в систему безопасности средств восстановления, возникает вопрос, как же быть с надежностью защиты данных,

как обеспечить пользователю уверенность в том, что его закрытый ключ не используется с какими-либо другими целями, не имеющими отношения к резервированию? Некоторую уверенность в секретности хранения закрытых ключей может дать технология **депонирования ключей**. Депонирование ключей — это предоставление закрытых ключей на хранение третьей стороне, надежность которой не вызывает сомнений. Этой третьей стороной может быть правительственная организация или группа уполномоченных на это сотрудников предприятия, которым оказывается полное доверие.

Аутентификация информации

Под аутентификацией информации в компьютерных системах понимают установление подлинности данных, полученных по сети, исключительно на основе информации, содержащейся в полученном сообщении.

Если конечной целью шифрования информации является защита от несанкционированного ознакомления с этой информацией, то конечной целью аутентификации информации является защита участников информационного обмена от навязывания ложной информации. Концепция аутентификации в широком смысле предусматривает установление подлинности информации как при наличии взаимного доверия между участниками обмена, так и при его отсутствии.

В компьютерных системах выделяют два вида аутентификации информации:

- аутентификация хранящихся массивов данных и программ — установление факта того, что данные не подвергались модификации;
- аутентификация сообщений — установление подлинности полученного сообщения, в том числе решение вопроса об авторстве этого сообщения и установление факта приема.

Цифровая подпись

Для решения задачи аутентификации информации используется концепция цифровой (или электронной) подписи. Согласно терминологии, утвержденной Международной организацией по стандартизации (ISO), под термином «цифровая подпись» понимаются методы, позволяющие устанавливать подлинность автора сообщения (документа) при возникновении спора относительно авторства. Основная область применения цифровой подписи — это финансовые документы, сопровождающие электронные сделки, документы, фиксирующие международные договоренности и т. п.

До настоящего времени наиболее часто для построения схемы цифровой подписи использовался алгоритм RSA. Как уже отмечалось (см. раздел «Криптоалгоритм RSA»), в основе этого алгоритма лежит концепция Диффи–Хеллмана. Она заключается в том, что каждый пользователь сети имеет свой закрытый ключ, необходимый для формирования подписи; соответствующий этому секретному ключу открытый ключ, предназначенный для проверки подписи, известен всем другим пользователям сети.

На рис. 12.11 показана схема формирования цифровой подписи по алгоритму RSA. Подписанное сообщение состоит из двух частей: незашифрованной части, в которой содержится исходный текст T, и зашифрованной части, представляющей собой цифровую подпись. Цифровая подпись S вычисляется с использованием закрытого ключа (D, n) по формуле: $S = T^D \bmod n$.

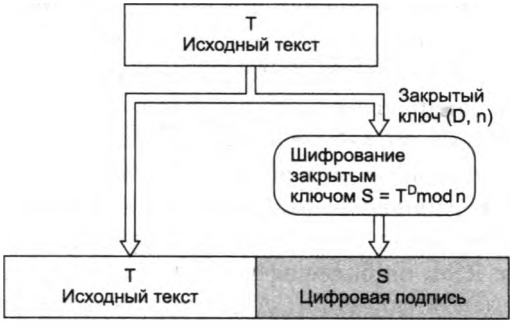


Рис. 12.11. Схема формирования цифровой подписи по алгоритму RSA

Сообщение посылается в виде пары (T, S). Каждый пользователь, имеющий соответствующий открытый ключ (E, n), получив сообщение, отделяет открытую часть T, расшифровывает цифровую подпись S и проверяет равенство: $T = S^E \bmod n$.

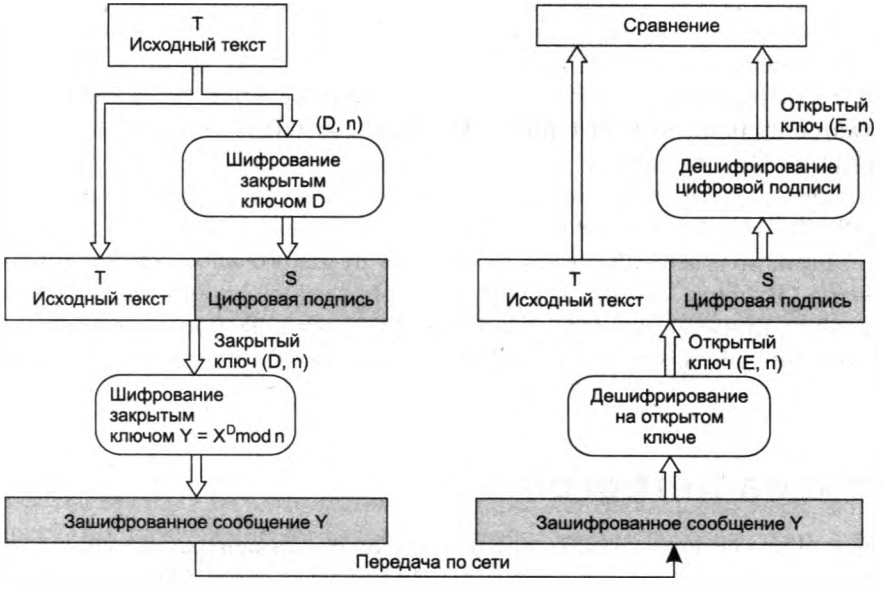


Рис. 12.12. Обеспечение конфиденциальности документа с цифровой подписью

Если результат расшифровки цифровой подписи совпадает с открытой частью сообщения, считается, что документ подлинный, не претерпел никаких

изменений в процессе передачи, а автором его является именно тот человек, который передал свой открытый ключ получателю. Если сообщение снабжено цифровой подписью, то получатель может быть уверен, что оно не было изменено или подделано по пути. Такие схемы аутентификации называются асимметричными. К недостаткам данного алгоритма можно отнести то, что длина подписи в этом случае равна длине сообщения, что не всегда удобно.

Цифровые подписи применяются к тексту до того, как он шифруется. Если помимо снабжения текста электронного документа цифровой подписью надо обеспечить его конфиденциальность, то вначале к тексту применяют цифровую подпись, а затем шифруют все вместе: и текст, и цифровую подпись (рис. 12.12).

Другие методы цифровой подписи основаны на формировании соответствующей сообщению контрольной комбинации с помощью классических алгоритмов типа DES. Учитывая более высокую производительность алгоритма DES по сравнению с RSA, он более эффективен для подтверждения аутентичности больших объемов информации. А для коротких сообщений типа платежных поручений или квитанций подтверждения приема, наверное, лучше подходит алгоритм RSA.

Аутентификация программных кодов

Компания Microsoft разработала средства для доказательства аутентичности программных кодов, распространяемых через Интернет. Пользователю важно иметь доказательства, что программа, которую он загрузил с какого-либо сервера, действительно содержит коды, разработанные определенной компанией. Протоколы защищенного канала типа SSL помочь здесь не могут, так как позволяют удостоверить только аутентичность сервера. Суть технологии аутентикода (authenticode), разработанной Microsoft, состоит в следующем.

Организация, желающая подтвердить свое авторство на программу, должна встроить в распространяемый код так называемый подписывающий блок (рис. 12.13). Этот блок состоит из двух частей. Первая часть — сертификат этой организации, полученный обычным образом от какого-либо сертифицирующего центра. Вторую часть образует зашифрованный дайджест, полученный в результате применения односторонней функции к распространяемому коду. Шифрование дайджеста выполняется с помощью закрытого ключа организации.

Система Kerberos

Kerberos — это сетевая служба, предназначенная для централизованного решения задач аутентификации и авторизации в крупных сетях. Она может работать в среде многих популярных ОС, например в ОС семейства Windows NT система Kerberos встроена как основной компонент безопасности.

В основе функционирования этой достаточно громоздкой системы лежит несколько простых принципов.

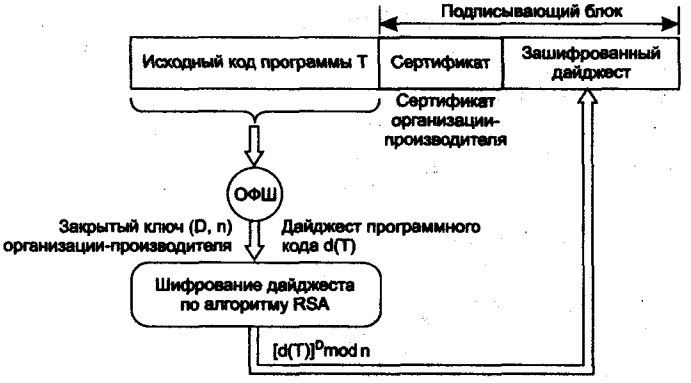


Рис. 12.13. Схема получения аутентикода

- В сетях, использующих систему безопасности Kerberos, все процедуры аутентификации между клиентами и серверами сети выполняются через посредника, которому доверяют обе стороны процесса аутентификации, причем таким авторитетным арбитром является сама система Kerberos.
- В системе Kerberos клиент должен доказывать свою аутентичность для доступа к каждой службе, услуги которой он запрашивает.
- Все обмены данными в сети выполняются в защищенном виде с использованием алгоритма шифрования DES.

Сетевая служба Kerberos построена по архитектуре клиент-сервер, что позволяет ей работать в самых сложных сетях. Kerberos-клиент устанавливается на всех компьютерах сети, которые могут обратиться к какой-либо сетевой службе. В таких случаях Kerberos-клиент от лица пользователя передает запрос на Kerberos-сервер и поддерживает с ним диалог, необходимый для выполнения функций системы Kerberos.

Итак, в системе Kerberos имеются следующие участники: *Kerberos-сервер*, *Kerberos-клиенты*, *ресурсные серверы* (рис. 12.14). Kerberos-клиенты пытаются получить доступ к сетевым ресурсам — файлам, приложениям, принтеру и т. д. Этот доступ может быть предоставлен, во-первых, только легальным пользователям, а во-вторых, при наличии у них достаточных полномочий, определяемых службами авторизации соответствующих ресурсных серверов, — файловым сервером, сервером приложений, сервером печати. Однако в системе Kerberos ресурсным серверам запрещается «напрямую» принимать запросы от клиентов, им разрешается начинать рассмотрение запроса клиента только тогда, когда на это поступает разрешение от Kerberos-сервера. Таким образом, путь клиента к ресурсу в системе Kerberos состоит из трех этапов.

1. Определение легальности клиента, логический вход в сеть, получение разрешения на продолжение процесса получения доступа к ресурсу.
2. Получение разрешения на обращение к ресурсному серверу.
3. Получение разрешения на доступ к ресурсу.

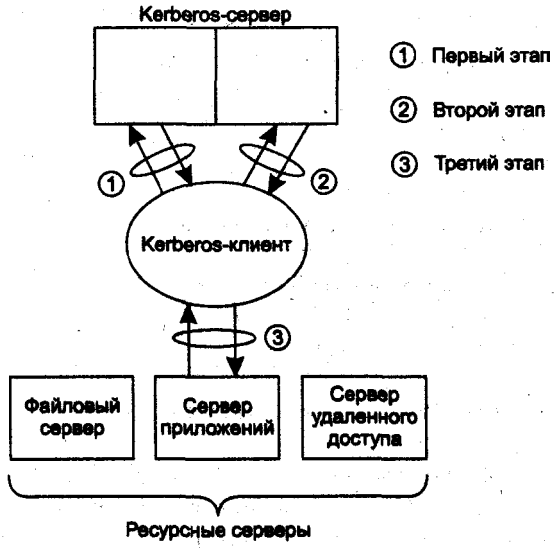


Рис. 12.14. Три этапа работы системы Kerberos

Для решения первой и второй задач клиент обращается к Kerberos-серверу. Каждая из этих двух задач решается отдельным сервером, входящим в состав Kerberos-сервера. Выполнение первичной аутентификации и выдача разрешения на продолжение процесса получения доступа к ресурсу осуществляется так называемым **сервером аутентификации** (Authentication Server, AS). Этот сервер хранит в своей базе данных информацию об идентификаторах и паролях пользователей.

Вторую задачу, связанную с получением разрешения на обращение к ресурсному серверу, решает другая часть Kerberos-сервера — **сервер квитанций** (Ticket-Granting Server, TGS). Сервер квитанций для легальных клиентов выполняет дополнительную проверку и дает клиенту разрешение на доступ к нужному ему ресурсному серверу, для чего наделяет его электронной формой-квитанцией. Для выполнения своих функций сервер квитанций использует копии секретных ключей всех ресурсных серверов, которые хранятся у него в базе данных. Помимо этих ключей TGS-сервер имеет еще один секретный DES-ключ, общий с AS-сервером.

Третья задача — получение разрешения на доступ непосредственно к ресурсу — решается на уровне ресурсного сервера.

ПРИМЕЧАНИЕ При описании протоколов взаимодействия Kerberos-клиента и Kerberos-сервера, а также Kerberos-клиента и ресурсного сервера использован термин «квитанция» (ticket), означающий в данном случае электронную форму, выдаваемую Kerberos-сервером клиенту, которая играет роль некоего удостоверения личности и разрешения на доступ к ресурсу.

Первичная аутентификация

Процесс доступа пользователя к ресурсам включает две процедуры: во-первых, пользователь должен доказать свою легальность (аутентификация), во-вторых, он должен получить разрешение на выполнение определенных операций с определенным ресурсом (авторизация). В системе Kerberos пользователь один раз аутентифицируется во время логического входа в сеть, а затем проходит процедуры аутентификации и авторизации всякий раз, когда ему требуется доступ к новому ресурсному серверу.

Выполняя логический вход в сеть, пользователь, а точнее Kerberos-клиент, установленный на его компьютере, посылает серверу аутентификации AS идентификатор пользователя ID (рис. 12.15).

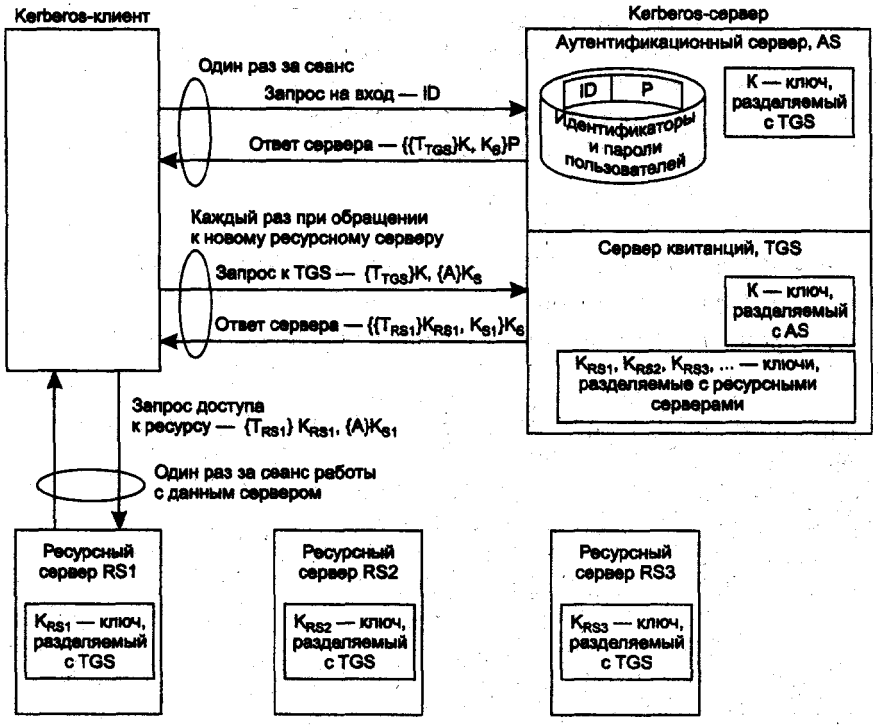


Рис. 12.15. Последовательность обмена сообщениями в системе Kerberos

Вначале сервер аутентификации проверяет в базе данных, есть ли запись о пользователе с таким идентификатором, затем, если такая запись существует, извлекает из нее пароль пользователя *p*. Данный пароль потребуется для шифрования всей информации, которую направит сервер аутентификации Kerberos-клиенту в качестве ответа. А ответ состоит из квитанции T_{TGS} на доступ к серверу квитанций Kerberos и ключа сеанса K_S. Под сеансом здесь понимается все время работы пользователя от момента логического входа в сеть до момента логического выхода. Ключ сеанса потребуется для шифрования в процедурах

аутентификации в течение всего пользовательского сеанса. Квитанция шифруется с помощью секретного DES-ключа K, который разделяют серверы аутентификации и квитанций. Все вместе — зашифрованная квитанция и ключ сеанса — еще раз шифруются с помощью пользовательского пароля p. Таким образом, квитанция шифруется дважды ключом K и паролем p. В приведенных обозначениях сообщение-ответ, которое сервер аутентификации посылает клиенту, выглядит так: $\{\{T_{TGS}\}K, K_S\}p$.

После того как такое ответное сообщение поступает на клиентскую машину, клиентская программа Kerberos просит пользователя ввести свой пароль. Когда пользователь вводит пароль, то Kerberos-клиент пробует с помощью пароля расшифровать поступившее сообщение. Если пароль верен, то из сообщения извлекаются квитанция на доступ к серверу квитанций $\{T_{TGS}\}K$ (в зашифрованном виде) и ключ сеанса K_S (в открытом виде). Успешное дешифрирование сообщения означает успешную аутентификацию. Заметим, что сервер аутентификации AS аутентифицирует пользователя без передачи пароля по сети.

Квитанция T_{TGS} на доступ к серверу квитанций TGS является удостоверением легальности пользователя и разрешением ему продолжать процесс получения доступа к ресурсу. Эта квитанция содержит:

- идентификатор пользователя;
- идентификатор сервера квитанций, на доступ к которому получена квитанция;
- отметку о текущем времени;
- период времени, в течение которого может продолжаться сеанс;
- копию ключа сеанса K_S.

Как уже было сказано, клиент обладает квитанцией в зашифрованном виде. Шифрование повышает уверенность в том, что никто, даже сам клиент — обладатель данной квитанции — не сможет квитанцию подделать, подменить или изменить. Только TGS-сервер, получив от клиента квитанцию, сможет ее расшифровать, так как в его распоряжении имеется ключ шифрования K.

Время действия квитанции ограничено длительностью сеанса. Разрешенная длительность сеанса пользователя, содержащаяся в квитанции на доступ к серверу квитанций, задается администратором и может изменяться в зависимости от требований к защищенности сети. В сетях с жесткими требованиями к безопасности время сеанса может быть ограничено 30 минутами, в других условиях это время может составить 8 часов. Информация, содержащаяся в квитанции, определяет ее срок годности. Предоставление квитанции на вполне определенное время защищает ее от неавторизованного пользователя, который мог бы ее перехватить и применить в будущем.

Получение разрешения на доступ к ресурсному серверу

Итак, следующим этапом для пользователя является получение разрешения на доступ к ресурсному серверу (например, к файловому серверу или серверу приложений). Но для этого надо обратиться к TGS-серверу, который выдает такие

разрешения (квитанции). Чтобы получить доступ к серверу квитанций, пользователь уже обзавелся квитанцией $\{T_{TGS}\}K$, выданной ему AS-сервером. Несмотря на защиту паролем и шифрование, пользователю, помимо квитанции, нужно кое-что еще, чтобы доказать серверу квитанций, что он имеет право на доступ к ресурсам сети.

Как уже упоминалось, первое сообщение от сервера аутентификации содержит не только квитанцию, но и секретный ключ сеанса K_S, который разделяется с сервером квитанций (TGS). Клиент использует этот ключ для шифрования еще одной электронной формы, называемой **аутентификатором** $\{A\}K_S$. Аутентификатор A содержит идентификатор и сетевой адрес пользователя, а также собственную временную отметку. В отличие от квитанции $\{T_{TGS}\}K$, которая в течение сеанса используется многократно, аутентификатор предназначен для одноразового использования и имеет очень короткое время жизни — обычно несколько минут. Kerberos-клиент посылает серверу квитанций сообщение-запрос, содержащее квитанцию и аутентификатор: $\{T_{TGS}\}K$, $\{A\}K_S$.

Сервер квитанций расшифровывает квитанцию имеющимся у него ключом K, проверяет, не истек ли срок действия квитанции, и извлекает из нее идентификатор пользователя.

Затем TGS-сервер расшифровывает аутентификатор, применяя ключ сеанса пользователя K_S, который он извлек из квитанции. Сервер квитанций сравнивает идентификатор пользователя и его сетевой адрес с аналогичными параметрами в квитанции и сообщении. Если они совпадают, сервер квитанций удостоверяется, что данная квитанция действительно представлена ее законным владельцем.

Заметим, что простое обладание квитанцией на получение доступа к серверу квитанций не доказывает идентичности пользователя. Так как аутентификатор действителен только в течение короткого промежутка времени, то маловероятно украсть одновременно и квитанцию, и аутентификатор и использовать их в течение этого времени. Каждый раз, когда пользователь обращается к серверу квитанций для получения новой квитанции на доступ к ресурсу, он посылает многоразовую квитанцию и новый аутентификатор.

Клиент обращается к серверу квитанций за разрешением на доступ к ресурсному серверу, который здесь обозначен как RS1. Сервер квитанций, удостоверившись в легальности запроса и личности пользователя, отсылает ему ответ, содержащий две электронных формы: многоразовую квитанцию на получение доступа к запрашиваемому ресурсному серверу T_{RS1} и новый ключ сеанса K_{S1}.

Квитанция на получение доступа шифруется секретным ключом K_{RS1}, общим только для сервера квитанций и того сервера, к которому предоставляется доступ, в данном случае — RS1. Сервер квитанций разделяет уникальные секретные ключи с каждым сервером сети. Эти ключи распределяются между серверами сети физическим способом или каким-либо иным секретным способом при установке системы Kerberos. Когда сервер квитанций передает квитанцию на доступ к какому-либо ресурсному серверу, то он шифрует ее, так что только этот сервер сможет расшифровать ее с помощью своего уникального ключа.

Новый ключ сеанса K_{S1} содержится не только в самом сообщении, посылаемом клиенту, но и внутри квитанции T_{RS1}. Все сообщение шифруется старым ключом сеанса клиента K_S, так что его может прочитать только этот клиент. Используя введенные обозначения, ответ TGS-сервера клиенту можно представить в следующем виде: $\{\{T_{RS1}\}K_{RS1}, K_{S1}\}K_S$.

Получение доступа к ресурсу

Когда клиент расшифровывает поступившее сообщение, то он отсылает серверу, к которому он хочет получить доступ, запрос, содержащий квитанцию на получение доступа и аутентификатор, зашифрованный новым ключом сеанса: $\{T_{RS1}\}K_{RS1}$, $\{A\}K_{S1}$.

Это сообщение обрабатывается аналогично тому, как обрабатывался запрос клиента TGS-сервером. Сначала расшифровывается квитанция ключом K_{RS1}, затем извлекается ключ сеанса K_{S1} и расшифровывается аутентификатор. Далее сравниваются данные о пользователе, содержащиеся в квитанции и аутентификаторе. Если проверка проходит успешно, то доступ к сетевому ресурсу разрешается.

На этом этапе клиент тоже может захотеть проверить аутентичность сервера перед тем, как начать с ним работать. Взаимная процедура аутентификации предотвращает любую возможность попытки получения неавторизованным пользователем доступа к секретной информации от клиента путем подмены сервера.

Аутентификация ресурсного сервера в системе Kerberos выполняется в соответствии со следующей процедурой. Клиент обращается к серверу с предложением, чтобы тот прислал ему сообщение, в котором повторил временную отметку из аутентификатора клиента, увеличенную на 1. Кроме того, требуется, чтобы данное сообщение было зашифровано ключом сеанса K_{S1}. Чтобы выполнить такой запрос клиента, сервер извлекает копию ключа сеанса из квитанции на доступ, использует этот ключ для расшифровки аутентификатора, наращивает значение временной отметки на 1, заново зашифровывает сообщение с помощью ключа сеанса и возвращает сообщение клиенту. Клиент расшифровывает это сообщение, чтобы получить увеличенную на единицу отметку времени.

При успешном завершении описанного процесса клиент и сервер удостоверяются в секретности своих транзакций. Кроме этого, они получают ключ сеанса, который могут использовать для шифрования будущих сообщений.

Достоинства и недостатки

Изучая довольно сложный механизм системы Kerberos, нельзя не задаться вопросом: какое влияние оказывают все эти многочисленные процедуры шифрования и обмена ключами на производительность сети, какую часть ресурсов сети они потребляют и как это сказывается на ее пропускной способности?

Ответ весьма оптимистичный — если система Kerberos реализована и сконфигурирована правильно, она лишь незначительно сказывается на производительности сети. Так как квитанции используются многократно, сетевые ресурсы,

затрачиваемые на запросы предоставления квитанций, невелики. Хотя передача квитанции при аутентификации логического входа несколько снижает пропускную способность, такой обмен требуется в любых других системах и при использовании любых методов аутентификации. Дополнительные же издержки незначительны. Опыт внедрения системы Kerberos показал, что время отклика при установленной системе Kerberos существенно не отличается от времени отклика без нее — даже в очень больших сетях с десятками тысяч узлов. Такая эффективность делает систему Kerberos весьма перспективной.

Среди уязвимых мест системы Kerberos можно назвать централизованное хранение всех секретных ключей системы. Успешная атака на Kerberos-сервер, в котором сосредоточена вся информация, критическая для системы безопасности, приводит к краху информационной защиты всей сети. Альтернативным решением могла бы быть система, построенная на использовании алгоритмов шифрования с парными ключами, для которых характерно распределенное хранение секретных ключей. Однако в настоящий момент еще не появились коммерческие продукты, построенные на базе несимметричных методов шифрования, которые бы обеспечивали комплексную защиту больших сетей.

Еще одной слабостью системы Kerberos является то, что исходные коды приложений, доступ к которым осуществляется через Kerberos, должны быть соответствующим образом модифицированы. Такая модификация называется «керберизацией» приложения. Некоторые поставщики продают «керберизаванные» версии своих приложений. Однако если такой версии нет или нет исходного текста, то Kerberos не может обслуживать доступ к такому приложению.

Выводы

- Безопасная информационная система обладает свойствами конфиденциальности, доступности и целостности. Конфиденциальность — гарантия того, что секретные данные будут доступны только авторизованным пользователям, то есть только тем пользователям, которым этот доступ разрешен. Доступность — гарантия того, что авторизованные пользователи всегда получат доступ к данным. Целостность — гарантия сохранности данными правильных значений, которая обеспечивается запретом для неавторизованных пользователей каким-либо образом изменять, модифицировать, разрушать или создавать данные.
- Любое действие, которое может быть направлено на нарушение конфиденциальности, целостности и/или доступности информации, а также на нелегальное использование других ресурсов сети называется угрозой. Реализованная угроза называется атакой. Риск — это вероятностная оценка величины возможного ущерба, который может понести владелец информационного ресурса в результате успешно проведенной атаки.
- Безопасность информационной системы складывается из компьютерной безопасности, связанной с хранением и обработкой данных в компьютере,

и сетевой безопасности, связанной с работой компьютера в сети. Сетевая безопасность, в свою очередь, базируется на двух компонентах: защите данных в момент их передачи по линиям связи и защите от несанкционированного удаленного доступа в сеть.

- Политика информационной безопасности определяет, какую информацию и от кого следует защищать, каков может быть ущерб от той или иной успешно реализованной угрозы, какими средствами вести защиту.
- Алгоритм шифрования считается раскрытым, если найдена процедура, позволяющая подобрать ключ за реальное время. Сложность алгоритма раскрытия называется криптостойкостью.
- Существует два класса криптосистем — симметричные и асимметричные. В симметричных схемах шифрования секретный ключ шифрования совпадает с секретным ключом дешифрирования. В асимметричных схемах шифрования открытый ключ шифрования не совпадает с секретным ключом дешифрирования.
- В настоящее время наиболее популярным стандартным симметричным алгоритмом шифрования является DES, а из несимметричных криптоалгоритмов с открытым ключом — RSA.
- Симметричные алгоритмы в общем случае обладают более высокой скоростью шифрования и требуют меньше времени на генерацию ключа, чем несимметричные алгоритмы с открытым ключом, но предъявляют высокие требования к надежности канала передачи секретного ключа, а также менее масштабируемы: в симметричных алгоритмах количество ключей находится в квадратичной зависимости от числа абонентов, а в несимметричных алгоритмах количество ключей равно удвоенному числу абонентов.
- Аутентификация предотвращает доступ к сети нежелательных лиц и разрешает вход для легальных пользователей. Доказательством аутентичности может служить знание аутентифицируемым некоего общего для обеих сторон слова (пароля) или факта, владение некоторым уникальным предметом или демонстрация уникальных биохарактеристик. Чаще всего доказательством идентичности пользователя служат пароли.
- Средства авторизации контролируют доступ легальных пользователей к ресурсам системы, предоставляя каждому из них именно те права, которые ему были определены администратором.
- Аудит — фиксация в системном журнале событий, связанных с доступом к защищаемым системным ресурсам.
- Технология защищенного канала призвана обеспечивать безопасность передачи данных по открытой транспортной сети, например через Интернет. Защищенный канал обеспечивает выполнение трех основных функций: взаимную аутентификацию абонентов при установлении соединения, защиту передаваемых по каналу сообщений от несанкционированного доступа, подтверждение целостности поступающих по каналу сообщений.

- Совокупность защищенных каналов, созданных предприятием в публичной сети для объединения своих филиалов, часто называют виртуальной частной сетью (VPN).
- Аутентификация с применением цифровых сертификатов является альтернативой применению паролей и особенно эффективна в сетях с очень большим числом пользователей. Цифровой сертификат устанавливает и гарантирует соответствие между открытым ключом и его владельцем.
- Централизованная система Kerberos является посредником между клиентами и серверами сети при проведении процедур аутентификации и авторизации. В системе Kerberos клиент должен доказывать свою аутентичность для доступа к каждой службе, услуги которой он запрашивает. Все обмены данными в сети выполняются в защищенном виде с использованием алгоритма шифрования DES.

Задачи и упражнения

1. Поясните значения основных свойств безопасной системы: конфиденциальности, целостности и доступности.
2. Приведите примеры средств, обеспечивающих конфиденциальность, но не гарантирующих целостность данных.
3. Приведите примеры действий воображаемого злоумышленника, направленных на нарушение доступности данных.
4. Предложите какой-нибудь способ обеспечения целостности данных.
5. Что такое политика безопасности?
6. В чем заключаются психологические меры безопасности?
7. Поясните значение терминов «идентификация», «аутентификация», «авторизация».
8. Почему удаленный доступ и другие варианты использования глобальных связей делают систему более уязвимой?
9. Какая схема шифрования — симметричная или асимметричная — является более масштабируемой?
10. В каких случаях предпочтительнее использовать симметричные алгоритмы шифрования, а в каких — алгоритмы шифрования с открытым ключом?
11. Правильно ли утверждение: «Поскольку открытый ключ не является секретным, то его не нужно защищать»?
12. Что такое электронная подпись?
13. Период генерации одноразовых паролей является одним из настраиваемых параметров аппаратного ключа. Из каких соображений администратор может выбрать и назначить значение для этого параметра?
14. Нужна ли клавиатура для набора цифр на карте — аппаратном ключе, применяемом при аутентификации на основе одноразовых паролей по схеме «запрос-ответ»? А по схеме, использующей синхронизацию по времени?

15. В чем заключается масштабируемость метода аутентификации на основе сертификатов?
16. Какая информация содержится в сертификате?
17. Как убедиться в подлинности сертификата?
18. Опишите процедуру получения сертификата. На основании каких сведений о пользователе выдается сертификат?
19. В чем заключается проблема восстановления ключей?
20. Какими средствами можно доказать пользователю, загрузившему программу из Интернета, что она действительно является продуктом компании, об авторстве которой заявляют распространители кода?
21. Администратор сети, мотивируя свой отказ от использования системы Kerberos, заявил, что эта система для своей работы требует слишком больших накладных расходов, а кроме того, защищаемые приложения придется переписывать. Согласны ли вы с его аргументами?

Ответы к задачам и упражнениям

Глава 1

2. Мониторы пакетной обработки, в отличие от системных обрабатывающих программ, начали выполнять новые задачи — задачи автоматизированной организации вычислительного процесса.
3. Да.

Глава 2

6. Пользователю *истинно-распределенной* ОС не требуется знать, на каком из компьютеров сети хранятся файлы, с которыми он работает, а пользователю *сетевой* ОС эти сведения обычно необходимы.
7. Варианты 1, 2 и 3.
10. Варианты 1 и 2.
11. Часто используются как синонимы термины *сервис* и *услуга*, а также *клиент* и *редиректор*.
12. Да, например, NetWare for Unix.
14. Варианты 2 и 3.
15. Да, если у него есть соответствующая клиентская часть.

Глава 3

1. Как синонимы могут использоваться термины *привилегированный режим*, *режим супервизора*, *режим ядра*.
2. Да, так как анализ может выявить наличие в программе привилегированных команд.
4. Стремление повысить производительность системы.
8. Да.

Глава 4

4. Вариант 1 — в очереди процессов, ожидающих ввода-вывода, и вариант 2 — в очереди готовых процессов.
5. Вариант 3.
6. Нет.
9. Нет.
11. Вариант 1 — да, вариант 2 — нет, вариант 3 — да, вариант 4 — да, вариант 5 — нет.
13. Строго говоря, нет.
14. Да.
16. Да.
17. В отношении времени выполнения отдельного приложения — первый вариант; в отношении суммарной производительности компьютера — второй.
18. Невытесняющая многозадачность.
19. Вариант 1: вытесняющий алгоритм, использует абсолютные динамические приоритеты, фиксированные кванты, мягкое реальное время; вариант 2 — невытесняющий алгоритм, использует относительные динамические приоритеты, не использует квантование, не поддерживает процессы реального времени; вариант 3 — вытесняющий алгоритм, использует абсолютные динамические приоритеты, динамические кванты, мягкое реальное время.
20. Прерывания от таймера, выполнение процессом некоторых системных вызовов, связанных с запросом и освобождением ресурсов, аппаратное прерывание, которое сигнализирует о завершении периферийным устройством операции ввода-вывода, внутреннее прерывание, сообщающее об ошибке выполнения активной задачи.
23. Векторный.
24. Нет.
28. Нет, в данном случае мы имеем дело не с клинчем, а с очередью. Действительно, студент, ожидающий в читальном зале, может быть «разблокирован» в результате освобождения какого-либо другого места в читальном зале, а затем он, выполнив свою работу и сдав книгу, «разблокирует» другого студента, ожидающего в книжном хранилище. Для того чтобы ситуация могла быть названа клинчем, следует дополнить задачу еще одним условием — в читальном зале имеется только одно рабочее место.

Глава 5

1. Характеристиками аппаратуры.
2. Разрядностью адреса в системе команд.
3. Да.

4. Могут быть верными оба варианта в зависимости от типа подсистемы управления памятью и требований приложения. Например, приложения реального времени могут загружаться в память всегда в физических адресах, а фоновые приложения — в виртуальных.
6. Варианты 3, 4 и 5.
7. Процедура сжатия не имеет смысла для страничного распределения, но применима при сегментном.
12. 163456_8.
13. В оперативной памяти.
14. Вариант 2.

Глава 6

2. GDTR и LDTR.
3. Менее привилегированные задачи могут получить доступ к данным более привилегированных задач, хранящимся в общем стеке, а это может привести к их несанкционированному использованию или разрушению.
4. Значения селекторов стека нужны в том случае, когда уровень привилегий вызываемого кода отличается от уровня привилегий вызывающего кода. Процедуру же с уровнем привилегий 3 нельзя вызвать из процедуры другого уровня привилегий, так как в процессоре Pentium запрещено вызывать процедуры с более низким уровнем привилегий.
5. В физической памяти.
6. С помощью шлюзов предоставляется возможность вызывать контролируемый набор процедур и задач, более привилегированных по сравнению с вызывающими процедурами и задачами.
7.

Соотношение уровней	Тип сегмента	Возможность доступа
CPL > DPL	C = 1	Да
CPL < DPL	C = 1	Нет
CPL = DPL	C = 1	Да
CPL > DPL	C = 0	Нет
CPL < DPL	C = 0	Нет
CPL = DPL	C = 0	Да

8. Да.
9. Страницы, хранящие разделы таблицы, выгружать можно, а страницу, содержащую таблицу разделов, — нельзя.
10. В первом случае запрет предотвращает передачу некоторой работы (функции) от более надежной процедуры менее надежной, во втором случае такой запрет не нужен, так как каждая задача выполняет собственную работу, и ее

вызов не уменьшает надежность более привилегированного вызывающего кода, который, как правило, является кодом ядра ОС. Если бы такой запрет существовал, то ОС не смогла бы выполнять свои функции по переключению задач.

11. Использование шлюза задачи вызывает переключения контекста, а шлюза прерывания — нет.

Глава 7

1. За счет контроллеров внешних устройств.
3. Варианты 1, 2, 4 и 6.
4. Диск как устройство и свободное место на диске.
5. Тем и другим.
8. С корневого каталога.
9. Вариант 3.
10. Размер кластера не может быть меньше, чем объем раздела, поделенный на максимально возможное число кластеров, которое для любой файловой системы FAT16 определяется разрядностью элемента таблицы FAT и равно 216. Таким образом, нижняя граница размера кластера равна 272 × 220/216 = 4352 (байт). Выбираем ближайшее число, превышающее 4352 и равное степени двойки, — это 8192 (8 Кбайт = 213) байт. Обозначим через X число кластеров, отведенных под область данных. Тогда справедливо следующее соотношение: объем раздела представляет собой сумму объемов, занимаемых загрузочным сектором (512 байт), двумя копиями таблицы FAT (2 × 2X байт), корневым каталогом (32 × 512 = 214 байт) и областью данных (213 × × X байт), или 272 × 220 = 512 + 2 × 2X + 214 +213 × X. Отсюда получаем X = 34 797 кластеров. Таким образом, память в разделе диска распределится следующим образом:
    - загрузочный сектор — 1 сектор;
    - FAT1 и FAT2 — по 2 × 34 797 = 69 594 (байт) или по 136 секторов;
    - корневой каталог — 32 сектора;
    - область данных — 34 797 × 16 = 556 752 сектора.
11. Во-первых, после удаления в файловую систему не были добавлены новые файлы. Во-вторых, файл занимал непрерывную последовательность кластеров или между кластерами данного файла в момент удаления не было никаких свободных кластеров.
12. Повышение производительности доступа к данным.
13. При избирательной.
15. С помощью механизма эффективных прав.
17. Разрешение *No Access*.

Глава 8

1. На стадии просмотра индексного дескриптора.
2. Правилен вариант 2. Доступ к данным кэша и внутренним переменным представляет собой в этом случае обращение к оперативной памяти, но время доступа к кэшу несколько больше, чем обращение к внутренним переменным, так как оно связано с дополнительными затратами на выполнение системного вызова, реализующего переход из пользовательского режима в привилегированный и передачу данных из системной области памяти в пользовательскую.
3. Секция `strategy`.
6. При отображении данные файла помещаются в пользовательскую часть виртуального адресного пространства процесса, причем по указанию прикладного программиста, а при кэшировании — в системную часть прозрачным для прикладного программиста образом.
7. Нет, не отображаются каталоги и символьные связи.
9. Наряду с откатом незавершенных транзакций необходимо выполнить повторение зафиксированных транзакций.
11. Нет.
13. Можно.
14. Более высокой скоростью доступа.
15. Операций чтения.
16. При динамическом восстановлении данные отказавшего диска генерируются на основании данных остальных дисков в момент поступления запроса на их чтение, при этом они не хранятся статически ни на одним из дисков.
17. В том случае, когда процессы не имеют общего прародителя.

Глава 9

1. Буферы интерфейсов маршрутизаторов и коммутаторов.
2. Нет.
6. Для выполнения основной функции — продвижения пакетов на основании сетевого адреса — нет, а для выполнения некоторых вспомогательных функций, например удаленного управления маршрутизатором или поддержки протокола маршрутизации, который использует протокол транспортного уровня, — да.
7. Пакет обычно употребляют для обозначения единицы независимо передаваемых сетью данных на сетевом уровне (например, IP-пакет), а кадр — для того же самого, но на канальном уровне (например, Ethernet-кадр).
8. За счет уникального значения идентификатора производителя сетевого оборудования (старшие три байта MAC-адреса), присваемого централизованно,

а также за счет уникальности значения младших трех байтов адреса, обеспечиваемой самим производителем.

9. Такие коммутаторы прозрачны для конечных узлов сети — компьютеров. Это означает, что появление в сети Ethernet коммутатора (а также его исчезновение) не требует изменения конфигурации компьютера.
12. Маска определяет, какую часть IP-адреса занимает номер сети, а какую — номер узла.
13. Для обеспечения независимости IP-уровня от способа адресации нижележащей технологии.
14. Широковещание.

Глава 10

2. В первом случае процессы могут обмениваться данными через общую память, а во втором случае они такой возможности лишены, поэтому единственным средством обмена является передача сообщений.
3. При использовании модели файлового сервера или сервера базы данных.
4. С использованием синхронных примитивов передачи сообщений.
5. Буфер.

Глава 11

1. Протоколы NFS, SMB, FTP, TFTP, NCP.
2. Модель stateless.
4.

	Повышение производительности	**Повышение отказоустойчивости**
Репликация	Да	Да
Кэширование	Да	Нет

5. Можно.
6. В ОС Windows NT — нельзя, а в ОС Unix — можно.
7. Каждый пользователь видит на экране не только свои исправления, но и исправления, сделанные другим пользователем.
8. Модель удаленного доступа.
11. База данных службы каталогов должна обладать *распределенностью* для обеспечения масштабируемости службы и *реплицируемостью* для обеспечения ее отказоустойчивости.
14. Возможно.
15. Вариант 2.

16. Нет.
17. Может, если оно не использует протоколы прикладного уровня.

Глава 12

9. Асимметричная.
11. Неправильно, его нужно защищать от подмены.

Рекомендуемая литература

1. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 3-е издание. Олифер В. Г, Олифер Н. А. — СПб: «Питер», 2006.
2. Очерки истории советской вычислительной техники. Наталья Дубова. — Журнал «Открытые системы» № 01/1999.
3. Операционные системы. Столлингс В. — М.: Издательский дом «Вильямс», 2004.
4. Современные операционные системы. 2-е издание. Таненбаум Э. — СПб: «Питер», 2007.
5. Операционные системы. Разработка и реализация. Таненбаум Э., Вудхалл. А. — СПб: «Питер», 2006.
6. Операционные системы. Основы и принципы. Дейтел Х. М., Дейтел П. Дж., Чофнес Д. Р. — Изд.: Бином-Пресс, 2006.
7. Операционные системы. Часть 2. Распределенные системы, сети, безопасность. Дейтел Х. М., Дейтел П. Дж., Чофнес Д. Р. — Изд.: Бином-Пресс, 2006.
8. Структура операционной системы Cisco IOS. Боллапрагада В., Мэрфи К., Уайт Р. — М.: Издательский дом «Вильямс», 2002.
9. Ядро Linux в комментариях. С. Максвелл; пер. с англ. — К.: «ДиаСофт», 2000.
10. Операционная система Unix. Андрей Робачевский. — BHV, 1999.
11. Windows для профессионалов: Программирование для Windows NT 4.0 и Windows 95 на базе Win32 API. Д. Рихтер; пер. с англ. — М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1997.
12. Основы Windows NT и NTFS. Х. Кастер. — М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1996.
13. Перспектива: Windows NT 5.0. Зубанов Ф. В. — М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1998.
14. Windows 2000 изнутри. Каплан Нильсен. — ДМК, 2000.
15. Сети NetWare 5. Руководство от Novell. Джефри Ф. Хьюз, Блейер В. Томас. — Вильямс, 2000.

16. Программирование NLM в NetWare 4.0. Дэй М., Кунц М., Маршалл Д. — М.: «ЛОРИ», 1994.
17. OS/2: Принципы построения и установка. Гранже М., Менсье Ф. — М.: Мир, 1991.
18. OS/2 Warp изнутри. Том 1, 2. Минаси М., Камарда Б. — СПб: «Питер», 1996.
19. Процессоры Pentium II, Pentium Pro и просто Pentium. Гук М. — СПб: «Питер», 1999.
20. Защищенный режим процессоров Intel 80286, 80386, 80486. Практическое руководство по использованию защищенного режима. Фролов А. В., Фролов Г. В. — М.: «Диалог-МИФИ», 1993.
21. Inside Active Directory: A System Administrator's Guide, Second Edition (Microsoft Windows Server System). Sakari Kouti, Mika Seitsonen. — Addison-Wesley Professional, 2004.
22. The Magic Garden Explained: The Internals of UNIX System V Release 4, An Open Systems Design. B. Goodheart, J. Cox. — Prentice Hall, 1994.
23. The Design of the UNIX Operating System. Maurice J. Bach. — Prentice-Hall, 1986.
24. Distributed Systems, 2/e. Edited by S. Mullender. — Addison-Wesley, 1998.
25. Internet Security protocols: Protecting IP Traffic. Uyless Black. — Prentice-Hall, 2000.

Алфавитный указатель

W

X

A

Б

В

Г

Д

Ж

З

И

К

Л

М

H

О

П

Р

C

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я

Виктор Олифер, Наталья Олифер

Сетевые операционные системы: Учебник для вузов

2-е издание

Заведующий редакцией	*А. Юрченко*
Руководитель проекта	*П. Маннинен*
Ведущий редактор	*О. Некруткина*
Литературный редактор	*А. Жданов*
Художник	*А. Татарко*
Верстка	*Л. Харитонов*
Корректор	*В. Листова*

Подписано в печать 13.08.08. Формат 70×100/16. Усл. п. л. 54,18. Доп. тираж 3500. Заказ 10763.
ООО «Питер Пресс», 198206, Санкт-Петербург, Петергофское шоссе, д. 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.