Ставропольский филиал
ГОУ ВПО «Московский государственный гуманитарный университет имени М.А. Шолохова»

С. И. МАКАРЕНКО

ОПЕРАЦИОННЫЕ СИСТЕМЫ, СРЕДЫ И ОБОЛОЧКИ

Учебное пособие

Ставрополь
СФ МГГУ им. М. А. Шолохова
2008

УДК 004.75
ББК 32.973.26-018.2

Макаренко С. И. Операционные системы, среды и оболочки: учебное пособие. – Ставрополь: СФ МГГУ им. М. А. Шолохова, 2008. – 210 с.

Рецензенты:

доцент кафедры прикладной информатики и математики Ставропольского филиала Московского государственного гуманитарного университета имени М. А. Шолохова кандидат технических наук, доцент Федосеев В. Е.,

доцент кафедры прикладной информатики и математики Ставропольского филиала Московского государственного гуманитарного университета имени М. А. Шолохова кандидат технических наук Дятлов Д. В.

Учебное пособие адресовано студентам, обучающимся по специальности 080801 (351400) «Прикладная информатика в экономике» изучающих дисциплину «Операционные системы, среды и оболочки», а также может быть использовано специалистами в области проектирования и организации вычислительных систем.

Утверждено на заседании кафедры прикладной информатики и математики Ставропольского филиала Московского государственного гуманитарного университета имени М. А. Шолохова в качестве учебно-методического пособия для студентов по специальности 080801 (351400) - «Прикладная информатика в экономике».

Оглавление

Список сокращений

ACE - Access Control Entry – система контроля доступа
ACL - Access Control List - список управления доступом
API - Application Programming Interface - интерфейс прикладного программирования
DACL - Discretionary Access Control List - список разрешений при управлении доступом
DDF - Driver Device Interface - интерфейс «драйвер-устройство
DKI - Driver Kernel Interface - интерфейс «драйвер-ядро»
DLL - Dynamical Link Library - динамически подключаемая библиотека
DNS - Domain Name System – система доменных имён
DRAM - Dynamic Random Access Memory - динамическая оперативная память с произвольным доступом
DS - Directory Services - Служба каталогов
Ext - Extended (File System) - расширенная файловая система в операционных системах Unix/Linux
FAT - File Allocation Table – файловая система, хранящая информацию о расположении файлов в таблицах
FTP - File Transfer Protocol – протокол передачи файлов
GID - Group ID - идентификатор группы пользователя
GUI - Graphical User Interface - графический интерфейс пользователя
IP - Internet Protocol Address — уникальный сетевой адрес узла в компьютерной сети
IPX - Internet Packet eXchange - межсетевой обмен пакетами
IRQ - Interrupt ReQuest - запрос на прерывание
IRQL - Interrupt Request Levels - уровень запросов прерываний
LCN - Logical Cluster Number - логический номер кластера
LSA - Local Security Authority - локальный администратор безопасности
MFT - Master File Table - главная таблица файлов в файловой системе NTFS
DOS - Disks Operation System - дисковая операционная система
NCP - NetWare Control Protocol — основной протокол доступа к файлам и принтерам сетевой ОС NetWare компании Novell
NCSC - National Computer Security Center - национальным центром компьютерной безопасности США
NFS - Network File System — протокол сетевой файловой системы
NTFS - New Technology File System – файловая система, используемая в Windows NT
OSI - Open System Interconnections – модель взаимодействия открытых систем
PID - Process IDentifier - идентификатор процесса

QoS - Quality of Service – качество обслуживания
RPC - Remote Procedure Call - вызов удаленных процедур
SACL - System Access Control List – системный список управления доступом
SAM - Security Account Manager – менеджер управления безопасностью пользователей
SD - Security Descriptor - дескриптор безопасности
SI - Standard Information - стандартная информация
SID - Security IDentifier - идентификатор безопасности
SMB - Server Message Block
SMP - Symmetric Multiprocessing - симметричная многопроцессорная архитектура
SRAM - Synchronous Dynamic Random Access Memory - синхронная память с произвольным доступом
SRM - Security Reference Monitor - диспетчер доступа
TCP - Transmission Control Protocol -протокол управления передачей
UDP - User Datagram Protocol — протокол передачи пользовательских данных
UID - User ID - идентификатор пользователя
VCN - Virtual Cluster Number - виртуальный номер кластера
ЗУ - запоминающее устройство
ОС - операционная система
ПО - программное обеспечение
ФС - файловая система

Введение

Учебное пособие написано по опыту преподавания автором дисциплины «Операционные системы, среды и оболочки» в Ставропольском филиале МГГУ имени М. А. Шолохова и в первую очередь адресовано студентам, обучающимся по специальности «Прикладная информатика в экономике». Также учебное пособие может быть использовано специалистами в области проектирования и организации вычислительных систем.

Учебное пособие учитывает требования государственного образовательного стандарта, структурно соответствует учебной программе [2] и тематическому плану изучения дисциплины «Операционные системы, среды и оболочки». Отдельные главы пособия соответствуют материалу отдельных занятий, преимущественно лекциям.

При написании пособия автор придерживался принципа необходимости дополнения общетеоретических и концептуальных основ построения вычислительных и операционных систем, изучение которых предусмотрено государственным образовательным стандартом (изложены в главах 1-10), практическими сведениями по особенностям построения и принципам функционирования реальных операционных систем, которые являются актуальными для специалистов в информационной сфере (главы 11-13). Так же, пособие может быть использовано и в качестве конспекта лекций, так как в тексте пособия материал, рекомендуемый к конспектированию на лекциях, выделен *курсивом*.

При составлении учебного пособия автор ориентировался на известные учебные материалы в предметной области, а также использовал ресурсы сети Интернет посвященные вопросам организации вычислительных и операционных систем.

В основу глав учебного пособия был положен материал следующих источников:

- главы 1-10: учебник [1], дополненный материалами работ [3, 4, 9, 10];
- глава 11: работы [3, 4, 5];
- глава 12: работы [3, 4, 6, 7];
- глава 13: работа [8].

Таким образом, учебник [1] составляет **основную литературу** по дисциплине и рекомендуется к изучению при освоении материала дисциплины. Для читателей, интересующихся отдельными вопросами изучаемого материала, автор рекомендует **дополнительную литературу** [3 -10]. Также для более полного понимания материала глав 1-10 целесообразно опираться на знание особенностей работы аппаратных средств вычислительных систем, изучаемых по дисциплине «Вычислительные системы, сети и телекоммуникации» и представленных в учебном пособии [10].

Хотелось бы отметить, что в материалах учебного пособия нашли отражение часть результатов научно-исследовательской работы автора по анализу качества функционирования информационно-вычислительных систем в составе автоматизированных систем управления и обработки информации [11-17].

Автор выражает благодарность рецензентам за кропотливый труд по поиску ошибок и неточностей, а также ценные замечания, которые помогли сделать материал пособия лучше и доступнее.

Предложения и замечания по учебному пособию автор просит направлять на email: mak-serg@yandex.ru.

1. АРХИТЕКТУРА ОПЕРАЦИОННОЙ СИСТЕМЫ

1.1 Ядро и вспомогательные модули ОС

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.

Архитектура ОС *- структурная организация 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 и 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.

Общим подходом к структуризации ОС является разделение ее модулей на две группы:

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

Модули ядра *выполняют такие базовые функции ОС, как управление процессами, памятью, устройствами ввода-вывода и т. п.*

Ядро составляет сердцевину операционной системы, без него ОС является полностью неработоспособной и не сможет выполнить ни одну из своих функций.

В состав ядра входят *функции, решающие внутрисистемные задачи организации вычислительного процесса*, такие как:

- переключение контекстов,
- загрузка/выгрузка станиц,
- обработка прерываний,
- поддержка приложений, создание для них так называемой прикладной программной среды.

Приложения могут обращаться к ядру с запросами — системными вызовами — для выполнения тех или иных действий, например для открытия и чтения файла, вывода графической информации на дисплей, получения системного времени и т. д.

Интерфейс прикладного программирования (API) *- функции ядра, которые могут вызываться приложениями.*

Скорость выполнения функций ядра определяет производительность всей 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.

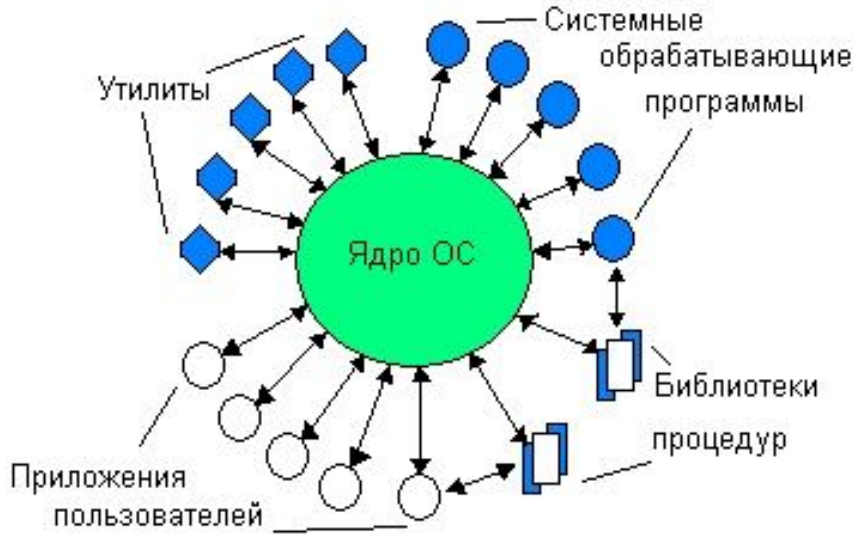


Рис. 1.1. Взаимодействие между ядром и вспомогательными модулями ОС

Вспомогательные модули ОС 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;

- *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.

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 ответственные функции, образующие ядро системы. Однако внесение изменений в функции ядра может оказаться гораздо сложнее, и сложность эта зависит от структурной организации самого ядра и может потребовать полной перекомпиляции модулей ядра.

1.2 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. 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 дополнительную область памяти, занимать процессор дольше разрешенного операционной системой периода времени, непосредственно управлять совместно используемыми внешними устройствами.

Привилегии для модулей ядра операционной системы обеспечивают специальные средства аппаратной поддержки.

Аппаратура должна поддерживать как минимум два режима работы:

- *пользовательский режим (user mode),*
- *привилегированный режим, который также называют режимом ядра (kernel mode), или режимом супервизора (supervisor mode).*

Подразумевается, что операционная система или некоторые ее части работают в привилегированном режиме, а приложения — в пользовательском режиме.

Так как ядро выполняет все основные функции ОС, то чаще всего именно ядро становится той частью ОС, которая работает в привилегированном режиме (рис. 1.2). Иногда это свойство — работа в привилегированном режиме — служит основным определением понятия «ядро».

Рис. 1.2. Архитектура операционной системы с ядром в привилегированном режиме

Приложения ставятся в подчиненное положение за счет запрета выполнения в пользовательском режиме следующих критичных команд:

- *переключение процессора с задачи на задачу,*
- *управление устройствами ввода-вывода,*
- *управления доступом к механизмам распределения и защиты памяти.*

Выполнение некоторых инструкций в пользовательском режиме запрещается безусловно (очевидно, что к таким инструкциям относится инструкция перехода в привилегированный режим), тогда как другие запрещается выполнять только при определенных условиях.

Повышение устойчивости операционной системы, обеспечиваемое переходом ядра в привилегированный режим, достигается за счет некоторого замедления выполнения системных вызовов.

Системный вызов *привилегированного ядра инициирует переключение процессора из пользовательского режима в привилегированный, а при возврате к приложению — обратно* (рис. 1.3).

Рис. 1.3. 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 многие популярные операционные системы, в том числе многочисленные версии UNIX, OS/2, и с определенными модификациями ОС семейства Windows NT.

В некоторых случаях разработчики ОС отступают от этого классического варианта архитектуры, организуя работу ядра и приложений в одном и том же режиме. Так, известная специализированная операционная система NetWare компании Novell использует привилегированный режим процессоров Intel x86/ Pentium как для работы ядра, так и для работы своих специфических приложений — загружаемых модулей NLM (рис. 1.4).

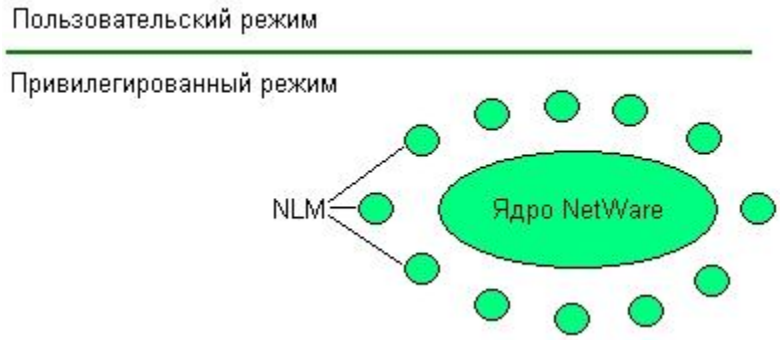


Рис. 1.4. Упрощенная архитектура операционной системы NetWare

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. Потенциальное снижение надежности ОС NetWare, компенсируется за счет тщательной отладки каждого приложения.

1.3 Многослойная структура ОС

Вычислительную систему, работающую под управлением ОС, можно рассматривать как систему, состоящую из трех иерархически расположенных слоев (рис. 1.5)***:***

- *нижний слой образует аппаратура,*
- *промежуточный — ядро ОС,*
- *утилиты, обрабатывающие программы и приложения, составляют верхний слой системы.*

При такой организации ОС приложения не могут непосредственно взаимодействовать с аппаратурой, а только через слой ядра.

Рис. 1.5. Трехслойная схема вычислительной системы

Многослойный подход является универсальным и эффективным способом декомпозиции сложных систем любого типа, в том числе и программных. В соответствии с ним система состоит из иерархии слоев. *Каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс* (рис. 1.6).

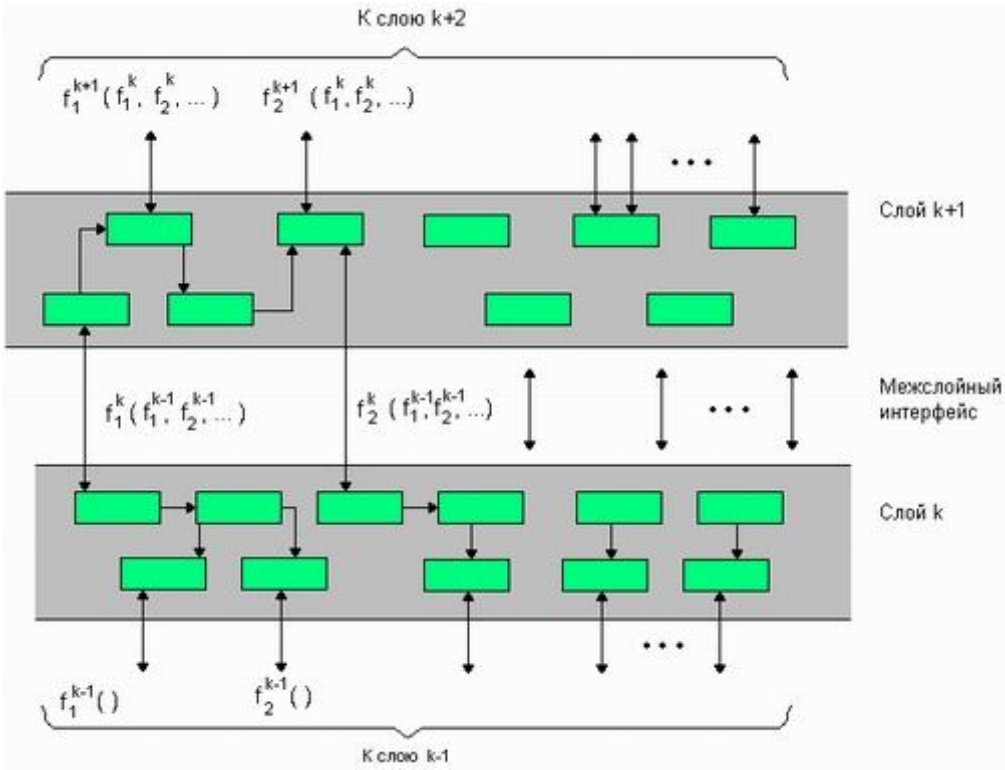


Рис. 1.6. Концепция многослойного взаимодействия

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. 1.7. Многослойная структура ядра ОС

Ядро может состоять из следующих слоев (рис. 1.7)**:**

- ***Средства аппаратной поддержки ОС***. Часть функций ОС может выполняться и аппаратными средствами. 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.
- ***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. Это позволяет разрабатывать вышележащие слои на основе машинно-независимых модулей, существующих в единственном экземпляре для всех типов аппаратных платформ, поддерживаемых данной ОС.
  - 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 «наверху» решения, что и дает повод называть их исполнительными механизмами для модулей верхних слоев.

- ***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
  - оперативной памяти.

  Каждый из менеджеров ведет учет свободных и используемых ресурсов определенного типа и планирует их распределение в соответствии с запросами приложений. Для исполнения принятых решений менеджер обращается к нижележащему слою. 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.
- ***Интерфейс системных вызовов (API).*** Этот слой является самым верхним слоем ядра и взаимодействует непосредственно с приложениями и системными утилитами, образуя прикладной программный интерфейс операционной системы. Функции API, обслуживающие системные вызовы, предоставляют доступ к ресурсам системы в удобной и компактной форме, без указания деталей их физического расположения. Для осуществления комплексных действий системные вызовы обычно обращаются за помощью к функциям слоя менеджеров ресурсов.

Приведенное разбиение ядра ОС на слои является достаточно условным. В реальной системе количество слоев и распределение функций между ними может быть и иным.

1.4. Аппаратная зависимость и переносимость ОС

1.4.1 Аппаратная зависимость 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 ОС можно выделить достаточно компактный слой машинно-зависимых компонентов ядра и сделать остальные слои ОС общими для разных аппаратных платформ.*

Четкой границы между программной и аппаратной реализацией функций ОС не существует — решение о том, какие функции ОС будут выполняться программно, а какие аппаратно, принимается разработчиками аппаратного и программного обеспечения компьютера.

Современные аппаратные платформы имеют некоторый типичный набор средств аппаратной поддержки ОС, в который входят следующие компоненты:

- *средства поддержки привилегированного режима;*
- *средства трансляции адресов;*
- *средства 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 признаки, определяющие режимы работы процессора, в том числе и 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 (например, ядро- супервизор-выполнение- пользователь у платформы что соответствует значениям 0-1-2-3 у регистра процессоров Intel x86/Pentium). В обязанности средств поддержки привилегированного режима входит выполнение проверки допустимости выполнения активной программой инструкций 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 используют 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 *нужна для того, чтобы оповестить процессор о возникновении в вычислительной системе некоторого непредсказуемого события или события, которое не синхронизировано с циклом работы процессора и позволяет компьютеру реагировать на внешние события, синхронизировать выполнение процессов и работу устройств ввода-вывода, быстро переходить с одной программы на другую.* 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. После завершения обработки прерывания обычно происходит возврат к исполнению прерванного кода.

Системный таймер*, необходим операционной системе для выдержки интервалов 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. 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.

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 передачах 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.

1.4.2 Переносимость ОС

Опыт разработки операционных систем показывает: *ядро можно спроектировать таким образом, что только часть модулей будут машинно-зависимыми, а остальные не будут зависеть 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.*

Переносимая (portable) или мобильная ОС *– система код которой может быть сравнительно легко перенесен с аппаратной платформы одного типа на аппаратную платформу другого типа.*

Для обеспечения свойства мобильности ОС, разработчики должны следовать следующим правилам:

- *Большая часть кода должна быть написана на языке, трансляторы которого имеются на всех машинах, куда предполагается переносить систему. Такими языками являются стандартизованные языки высокого уровня. Б*ольшинство переносимых ОС написано на языке С, который имеет много особенностей, полезных для разработки кодов операционной системы, и компиляторы которого широко доступны. Ассемблер используется только для тех непереносимых частей системы, которые должны непосредственно взаимодействовать с аппаратурой (например, обработчик прерываний), или для частей, которые требуют максимальной скорости (например, целочисленная арифметика повышенной точности).
- *Объем машинно-зависимых частей кода, которые непосредственно взаимодействуют с аппаратными средствами, должен быть по возможности минимизирован.* Следует всячески избегать прямого манипулирования регистрами и другими аппаратными средствами

процессора. Разработчики ОС должны также исключить возможность использования по умолчанию стандартных конфигураций аппаратуры или их характеристик. Аппаратно-зависимые параметры можно «спрятать» в программно- задаваемые данные абстрактного типа. Для осуществления всех необходимых действий по управлению аппаратурой, представленной этими параметрами, должен быть написан набор аппаратно-зависимых функций.

– *Аппаратно-зависимый код должен быть надежно изолирован в нескольких модулях, а не быть распределен по всей системе.* Изоляции подлежат все части ОС, которые отражают специфику как процессора, так и аппаратной 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, которую поддерживает данная ОС (рис. 1.8).

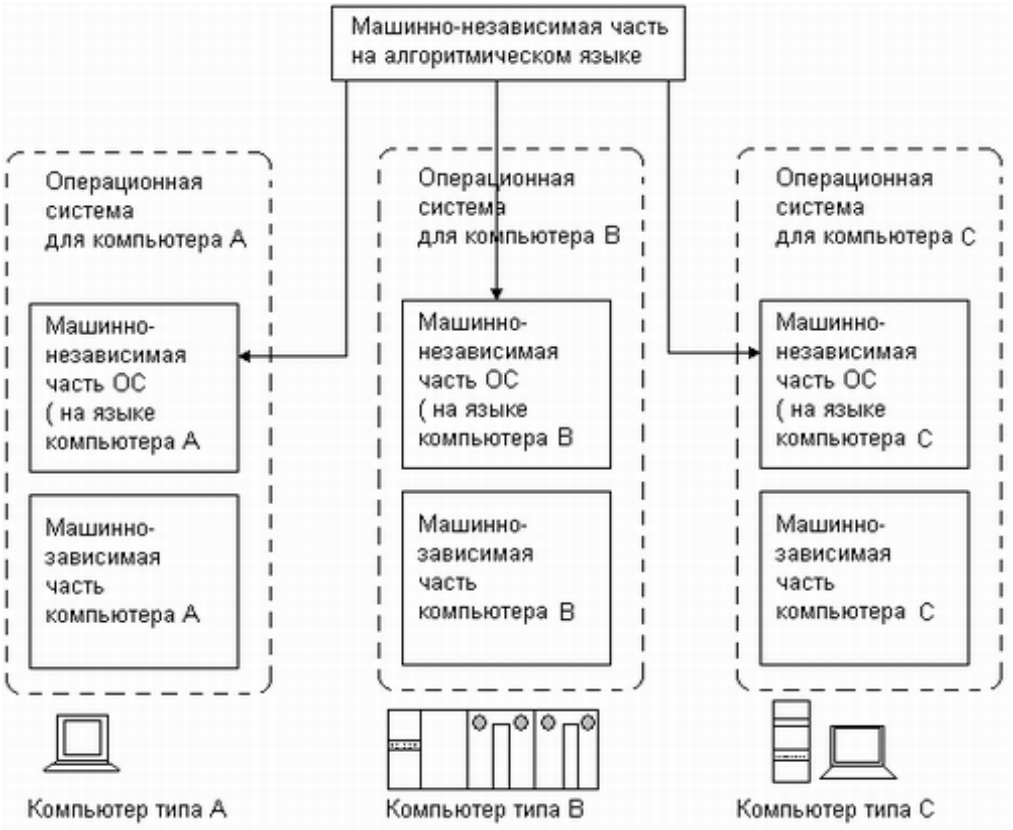


Рис. 1.8. Перенос операционной системы на разные аппаратные платформы

Для реализации свойства переносимости 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, управляют именно этой виртуальной аппаратурой. Таким образом, у разработчиков появляется возможность создавать один вариант машинно-независимой части ОС (включая компоненты ядра, утилиты, системные обрабатывающие программы) для всего набора поддерживаемых платформ.

1.5 Микроядерная архитектура

Микроядерная архитектура является альтернативой классическому способу построения операционной системы.

Под классической архитектурой в данном случае понимается рассмотренная выше 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.

Суть микроядерной архитектуры *состоит в том, что в привилегированном режиме остается работать только очень небольшая часть ОС, называемая микроядром. При этом микроядро защищено как от остальных частей ОС, так и от приложений* (рис. 1.9).

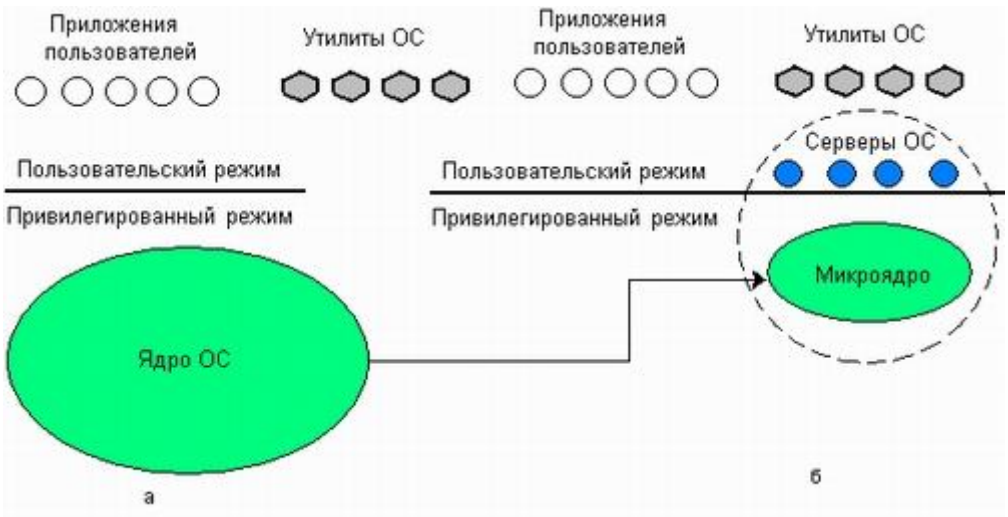


Рис. 1.9. Перенос основного объема функций ядра в пользовательское пространство

Набор функций микроядра обычно соответствует функциям слоя базовых механизмов обычного ядра. Такие функции операционной системы трудно, если не невозможно, выполнить в пространстве пользователя.

При этом в составе микроядра обычно остаются:

- машинно-зависимые модули,
- модули, выполняющие базовые (но не все!) функции ядра по управлению процессами,
- обработка прерываний,
- модули управления виртуальной памятью,
- модули управления пересылкой сообщений и управлению устройствами ввода-вывода,
- модули связанные с загрузкой или чтением регистров устройств.

Все остальные более высокоуровневые функции ядра оформляются в виде приложений, работающих в пользовательском режиме.

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.

Схематично механизм обращения к функциям ОС, оформленным в виде серверов, выглядит следующим образом (рис. 1.10).

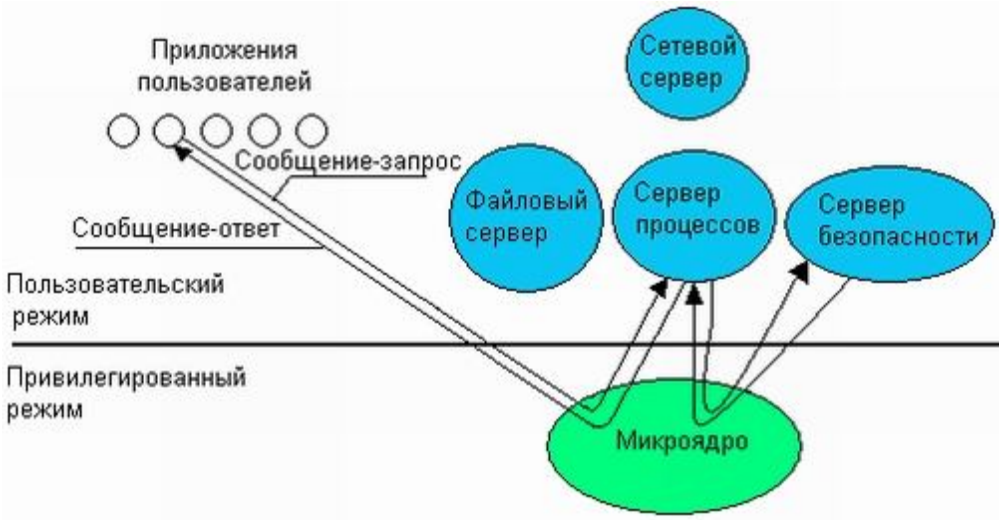


Рис. 1.10. Реализация системного вызова в микроядерной архитектуре

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 выполняет микроядро.

Операционные системы, основанные на концепции микроядра, в высокой степени удовлетворяют большинству требований, предъявляемых к современным ОС, ***обладая следующими достоинствами:***

- *высокой степенью 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.
- *Расширяемостью.* 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.
- *Надежностью.* 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. Другим потенциальным источником повышения надежности ОС является уменьшенный объем кода микроядра по сравнению с традиционным ядром — это снижает вероятность появления ошибок программирования.
- *Поддержкой распределенных приложений.* 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. Переход к распределенной обработке требует минимальных изменений в работе операционной системы — просто локальный транспорт заменяется на сетевой.

Основным недостатком микроядерной архитектуры является невысокая производительность. При классической организации ОС (рис. 1.11, а) выполнение системного вызова сопровождается двумя переключениями режимов, а при микроядерной организации (рис. 1.11, б) — четырьмя.

Рис. 1.11. Смена режимов при выполнении системного вызова

Таким образом, операционная система на основе микроядра при прочих равных условиях всегда будет менее производительной, чем ОС с классическим ядром. Именно по этой причине микроядерный подход не получил такого широкого распространения, которое ему предрекали.

Классическим примером микроядерной ОС является QNX (Neytrino).

1.6 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.

Совместимость ОС *- свойство выполнять приложения, написанные для других операционных систем.*

Рассматривают следующие уровни совместимости:

- ***Совместимость на уровне исходных текстов*** *требует наличия соответствующего компилятора в составе ОС, а также совместимости на уровне библиотек и системных вызовов.* При этом необходима перекомпиляция имеющихся исходных текстов в новый исполняемый модуль.
- ***Двоичная совместимость****, возможность использовать один и тот же программный продукт, поставляемый в виде двоичного исполняемого кода, в различных операционных средах и на различных машинах.*

Достаточными условиями двоичной совместимости является соблюдения следующих условий:

- вызовы функций API, 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 выше условий необходимо организовать эмуляцию обработки двоичного кода.

Пусть, например, требуется выполнить DOS-программу для IBM PC-совместимого компьютера на компьютере Macintosh. Компьютер Macintosh построен на основе процессора Motorola 680x0, а компьютер IBM PC — на основе процессора Intel 80x86. Процессор Motorola имеет архитектуру (систему команд, состав регистров и т. п.), отличную от архитектуры процессора Intel, поэтому ему непонятен двоичный код DOS-программы, содержащей инструкции этого процессора. Для того чтобы компьютер Macintosh смог интерпретировать машинные инструкции, которые ему изначально непонятны, на нем должно быть установлено специальное программное обеспечение — эмулятор.

Создание эмулятора 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:

1. *Реализация множественных прикладных сред основанной на стандартной многоуровневой структуре ОС.* На рис. 1.12 операционная система OS1 поддерживает кроме своих «родных» приложений приложения операционных систем OS2 и OS3. Для этого в ее составе имеются специальные приложения — прикладные программные среды, — которые транслируют интерфейсы «чужих» операционных систем API OS2 и API OS3 в интерфейс своей «родной» операционной системы — API OS1.

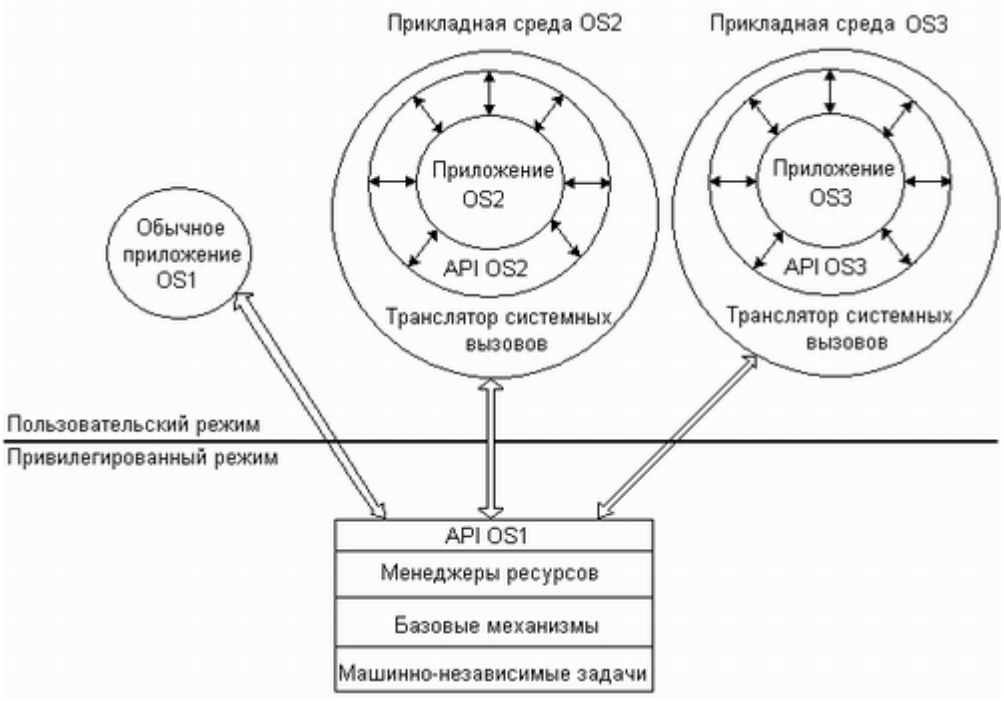


Рис. 1.12. Прикладные программные среды, транслирующие системные вызовы

2. *Реализация множественных прикладных сред операционная система через несколько равноправных прикладных программных интерфейсов.* В приведенном на рис. 1.13 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.

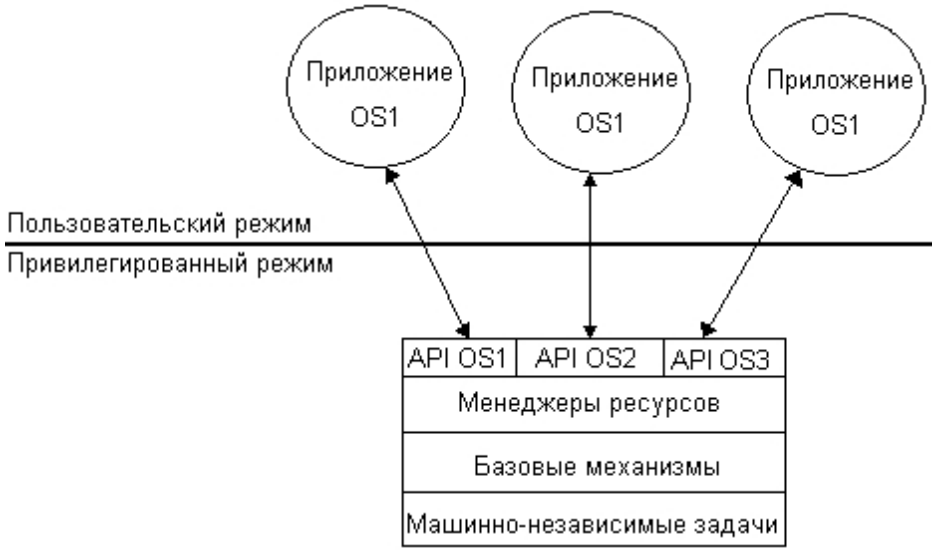


Рис. 1.13. Реализация совместимости на основе нескольких равноправных API

3. *Построение множественных прикладных сред по микроядерному подходу.* 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. 1.14).

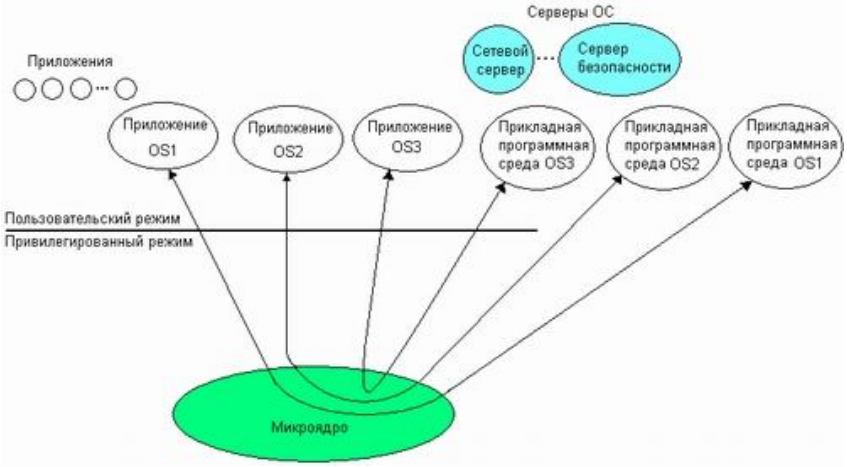


Рис. 1.14. Микроядерный подход к реализации множественных прикладных сред

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.

2. УПРАВЛЕНИЕ ПРОЦЕССАМИ

2.1 Понятие процесса и потока

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. Хотя и в однопрограммной ОС необходимо решать задачи управления ресурсами (например, распределение памяти между приложением и ОС), главные сложности возникают в мультипрограммных ОС, в которых за ресурсы конкурируют сразу несколько приложений. Именно поэтому большая часть всех проблем, рассматриваемых в этом материале, относится к мультипрограммным системам.

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

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 пользование или в совместное пользование с другими процессами. Некоторые из ресурсов выделяются процессу при его создании, а некоторые — динамически по запросам во время выполнения. Ресурсы могут быть приписаны процессу на все время его жизни или только на определенный период.

В настоящее время в большинстве операционных систем определены два типа единиц работы. Более крупная единица работы, обычно носящая название процесса, или задачи, требует для своего выполнения нескольких более мелких работ, для обозначения которых используют термины «поток», или «нить».

Процесс (задача) *– заявка к операционной системе на потребление ресурсов вычислительной системы необходимых для функционирования приложения.*

Многопоточная обработка (multithreading) *– механизм распараллеливания вычислений, учитывающий тесные связи между отдельными ветвями вычислений одного и того же приложения.*

Понятию ***«поток»*** *соответствует последовательный переход процессора от одной команды программы к другой. ОС распределяет процессорное время между потоками.* Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками. 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.

Мультипрограммирование более эффективно на уровне потоков, а не процессов. 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.

2.2 Управление процессами и потоками

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. 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.

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.

Создание процесса *— создание описателя 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.*

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

2.2.1 Планирование

Планирование *- работа по определению того, в какой момент необходимо прервать выполнение текущего активного потока и какому потоку предоставить возможность выполняться. Планирование, по существу, включает в себя решение двух задач:*

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

Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. При планировании могут приниматься во внимание приоритет потоков, время их ожидания в очереди, накопленное время выполнения, интенсивность обращений к вводу-выводу и другие факторы.

Существует множество различных алгоритмов планирования потоков, по-своему решающих каждую из приведенных выше задач. Алгоритмы планирования могут преследовать различные цели и обеспечивать разное качество мультипрограммирования.

В настоящее время *основными наиболее общими используемыми алгоритмами планирования процессов являются динамическое и статистическое планирование.*

Динамическое планирование (on-line)*, когда 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 используют никаких предположений о мультипрограммной смеси.

Динамическое планирование делится на:

- *вытесняющие алгоритмы планирования;*
- *невытесняющие алгоритмы планирования.*

Статическое планирование *- используется в специализированных системах, в которых весь набор одновременно выполняемых задач определен заранее*, например в системах реального времени. *В этом случае планировщик 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.

2.2.2 Диспетчеризация

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 содержит:

- *состояние аппаратуры компьютера в момент прерывания потока* (значение счетчика команд, содержимое регистров общего назначения, режим работы процессора, флаги, маски прерываний и другие параметры).
- *параметры операционной среды* (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.

2.2.3 Состояния потока

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

1. ***выполнение*** *— активное состояние потока, во время которого поток обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;*
2. ***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);
3. ***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).

В течение своей жизни каждый поток переходит из одного состояния в другое в соответствии с алгоритмом планирования потоков, принятым в данной операционной системе. Типичный граф, соответствующий поведению процесса в системе приведен на рис. 2.1.

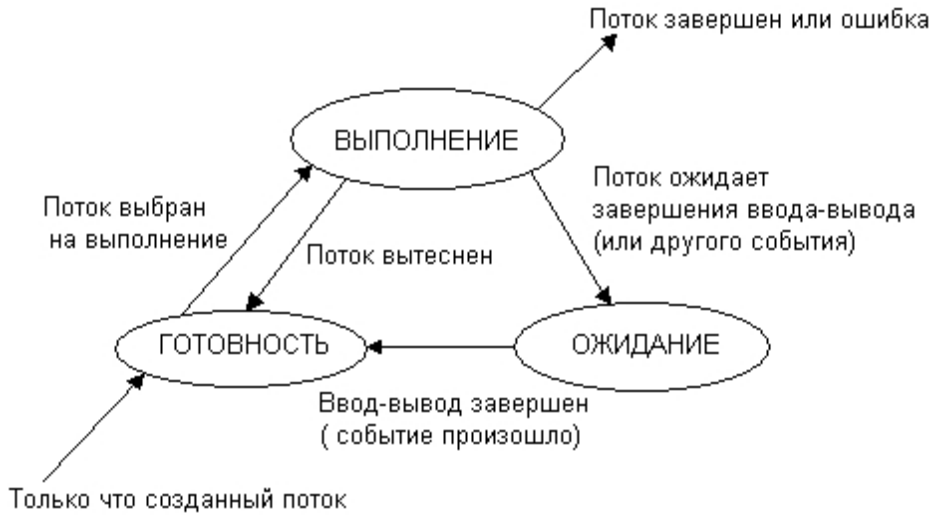


Рис. 2.1. Граф состояний потока в многозадачной среде

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 переходит в состояние выполнения и находится в нем до тех пор, пока либо он 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.

2.3 Алгоритмы планирования процессов

2.3.1 Вытесняющие и невытесняющие алгоритмы планирования

С самых общих позиций множество алгоритмов планирования можно разделить на два класса:

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

Основным различием между вытесняющими и невытесняющими алгоритмами является степень централизации механизма планирования потоков.

Почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений (UNIX,

Windows NT/2000/XP, Linux), реализованы вытесняющие алгоритмы планирования потоков (процессов).

2.3.2 Концепция квантования

В основе многих вытесняющих алгоритмов планирования лежит ***концепция квантования.*** *В соответствии с ней каждому потоку поочередно для выполнения предоставляется ограниченный непрерывный период процессорного времени — квант. 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, а на выполнение в соответствии с определенным правилом выбирается новый поток из очереди готовых. Граф состояний потока, изображенный на рис. 2.1, соответствует алгоритму планирования, основанному на квантовании.

Кванты, выделяемые потокам, могут быть одинаковыми для всех потоков или различными.

2.3.3 Приоритетные алгоритмы планирования

Другой 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 которой 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, и др.

Во многих ОС предусматривается возможность изменения приоритетов в течение жизни потока. Изменение приоритета могут 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, приоритетов.

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.

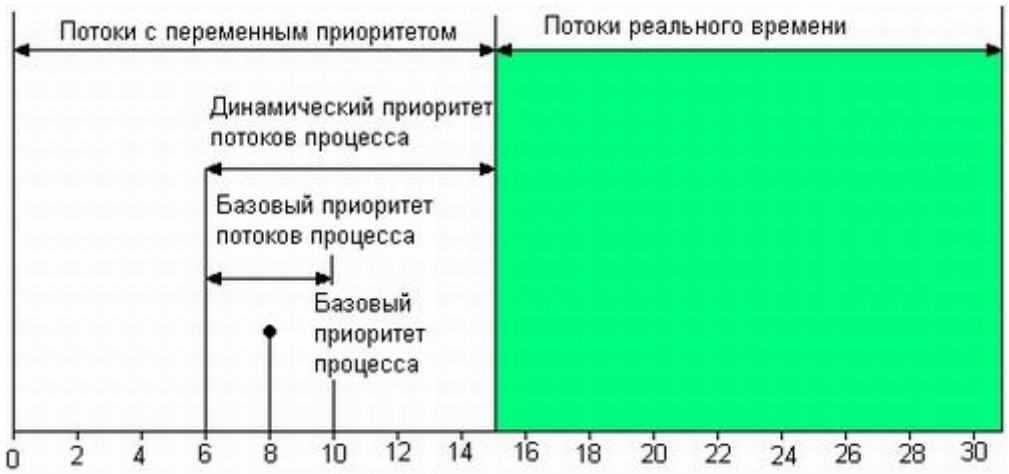


Рис. 2.2. Схема назначения приоритетов в ОС семейства Windows NT

Существуют две разновидности приоритетного планирования:

- ***Обслуживание с относительными приоритетами.*** В системах с относительными приоритетами активный поток выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ожидания (или же произойдет ошибка, или поток завершится).
- ***Обслуживание с абсолютными приоритетами.*** В системах с абсолютными приоритетами выполнение активного потока прерывается кроме указанных выше причин, еще при одном

условии: если в очереди готовых потоков появился поток, приоритет которого выше приоритета активного потока.

2.3.4 Смешанные алгоритмы планирования

Во многих операционных системах алгоритмы планирования построены с использованием как концепции 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 системе Windows NT, в которой 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.*

Процессы реального времени также используют 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 истечении планировщик снимает процесс с выполнения.

2.4 Синхронизация процессов и потоков

Существует достаточно обширный класс средств операционной системы, с помощью которых обеспечивается взаимная синхронизация процессов и потоков.

Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурс об вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Рассмотрим, например (рис. 2.3), задачу ведения базы данных клиентов некоторого предприятия. 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 **А**, 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.

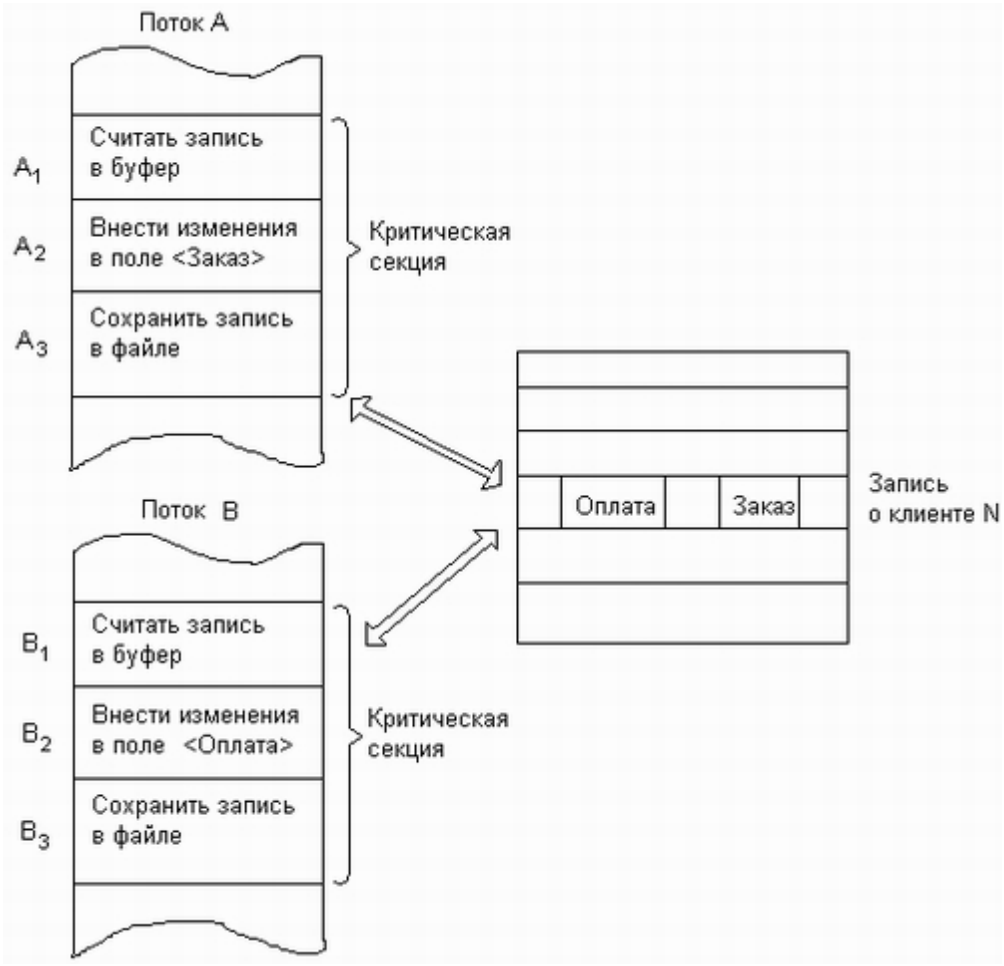


Рис. 2.3. Возникновение гонок при доступе к разделяемым данным

Обозначим соответствующие шаги для потока **A** как **Al**, **A2** и **A3**, а для потока **B** как **Bl**, **B2** и **B3**. Предположим, что в некоторый момент поток **A** обновляет поле **Заказ** записи о клиенте **N**. Для этого он считывает эту запись в свой буфер (шаг **A1**), модифицирует значение поля **Заказ** (шаг **A2**), но внести запись в базу данных (шаг **A3**) 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 **B1**) и выполнить

обновление поля **Оплата** (шаг **B2**), 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, запишет запись о клиенте **N** с модифицированным полем **Заказ** в базу данных (шаг **A3**). 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. 2.4, 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. 2.4, б) или, напротив, все исправления были успешно внесены (рис. 2.4, в).

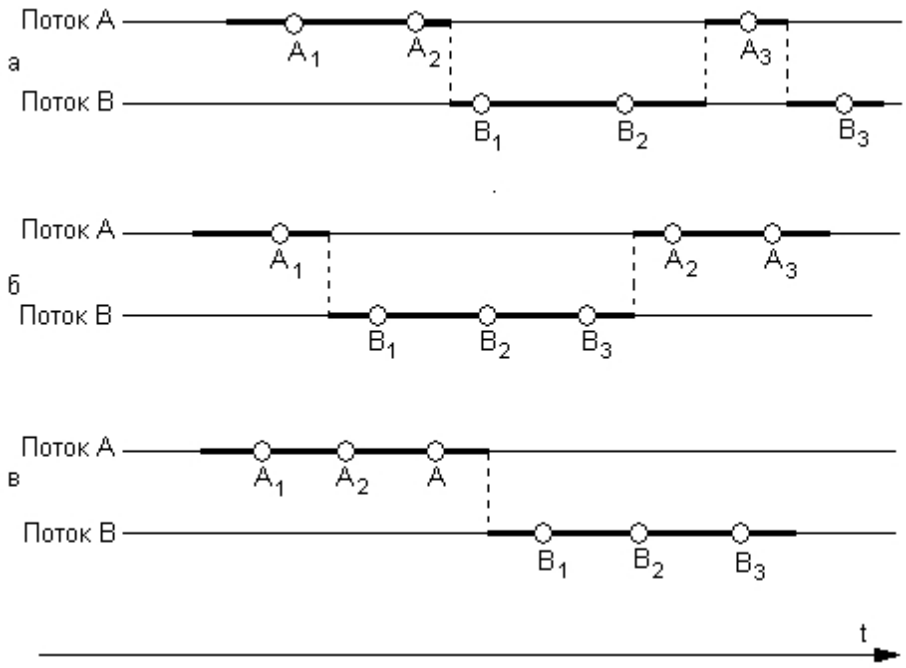


Рис. 2.4. Влияние относительных скоростей потоков на результат решения задачи

Все определяется взаимными скоростями потоков и моментами их прерывания. Поэтому отладка взаимодействующих потоков является сложной задачей.

Гонка *- когда два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков* (ситуации, рассмотренные выше).

Тупики *– состояние системы, когда два или более процессов взаимно блокируют друг друга.*

Рассмотрим пример тупика. Пусть двум процессам, выполняющимся в режиме мультипрограммирования, для выполнения их работы нужно два ресурса, например, принтер и диск. На рисунке 2.5,а показаны фрагменты соответствующих программ. И пусть после того, как процесс **А** занял принтер (установил блокирующую переменную), он был прерван. Управление получил процесс **В**, который сначала занял диск, но при выполнении следующей команды был заблокирован, так как принтер оказался уже занятым процессом **А**. Управление снова получил процесс **А**, который в соответствии со своей программой сделал попытку занять диск и был заблокирован: диск уже распределен процессу **В**. В таком положении процессы **А** и **В** могут находиться сколь угодно долго.

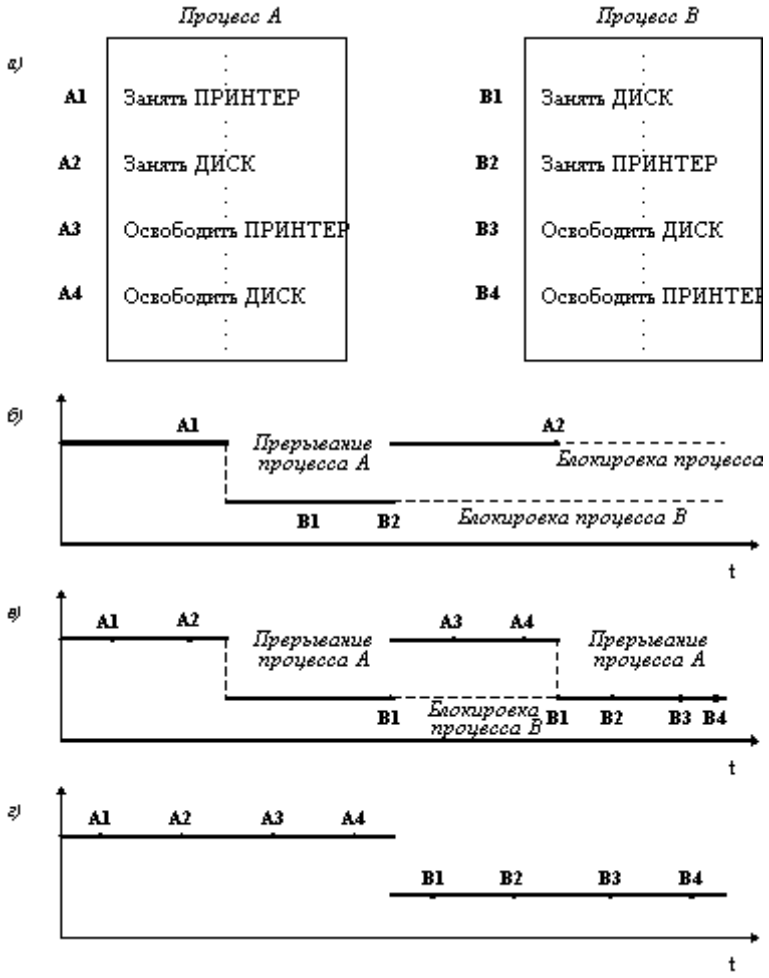


Рис. 2.5. а) фрагменты программ А и В, разделяющих принтер и диск;
б) взаимная блокировка (клинч); в) очередь к разделяемому диску;
г) независимое использование ресурсов.

2.4.1 Критическая секция

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.

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. При этом неважно, находится этот поток в активном или в приостановленном состоянии. Этот прием называют взаимным исключением. Операционная система использует разные способы реализации взаимного исключения. Некоторые способы пригодны для взаимного исключения при вхождении в критическую секцию только потоков одного процесса, в то время как другие могут обеспечить взаимное исключение и для потоков разных процессов.

2.4.2 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. 2.6 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 (F(D) = 1), то значение переменной F(D) устанавливается в 0 и поток входит в критическую секцию.* После того как поток выполнит все действия с данными O, значение переменной F(D) снова устанавливается равным 1.

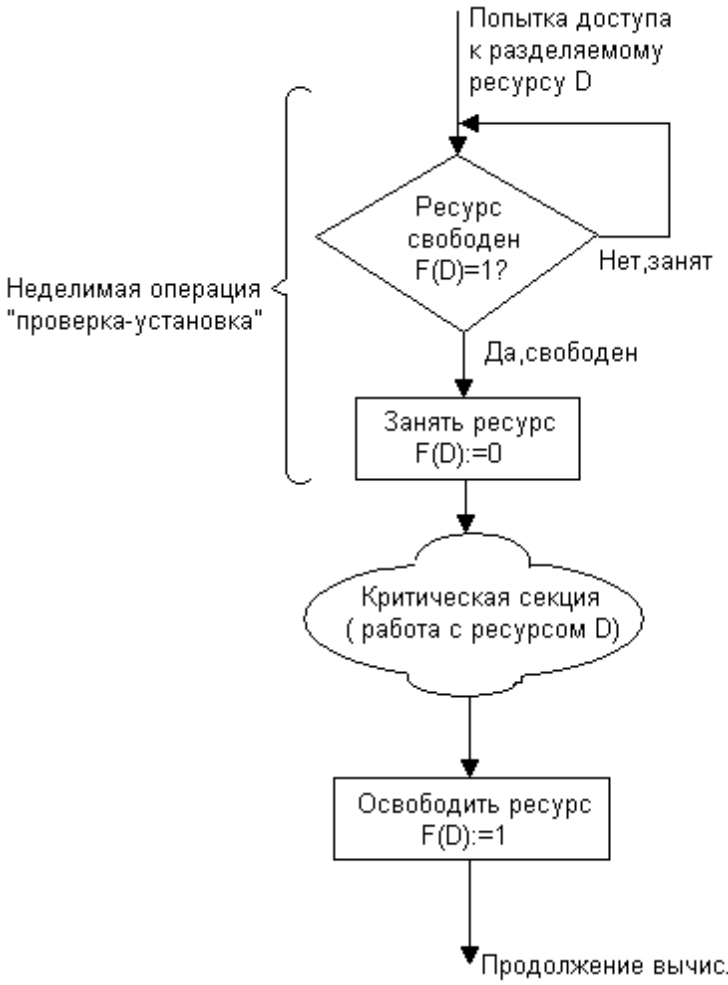


Рис. 2.6. Реализация критических секций с использованием блокирующих переменных

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.

2.4.3 Семафоры

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

Для работы с семафорами вводятся два примитива POST и WAIT, 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.
– PCS): 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 являются неделимой операцией.

Никакие прерывания во время выполнения примитивов V и P недопустимы.

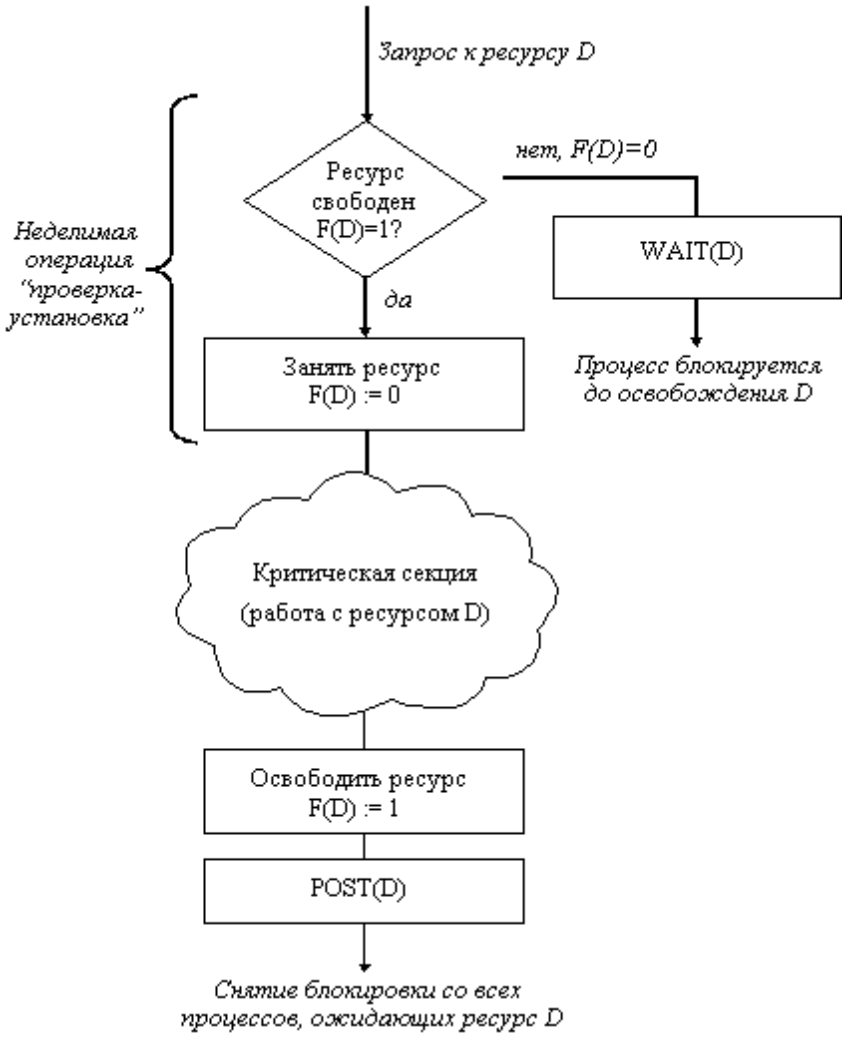


Рис. 2.7. Реализация критической секции с использованием системных функций WAIT(D) и POST(D)

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 процессов, выполняющихся в режиме мультипрограммирования, 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. Процесс "писатель" должен приостанавливаться, когда все буфера оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, процесс "читатель" приостанавливается, когда все буферы пусты, и активизируется при появлении хотя бы одной записи.

3. УПРАВЛЕНИЕ ПАМЯТЬЮ

3.1 Иерархия памяти

Память вычислительной машины представляет собой иерархию запоминающих устройств (ЗУ), отличающихся средним временем доступа к данным, объемом и стоимостью хранения одного бита (рис. 3.1)

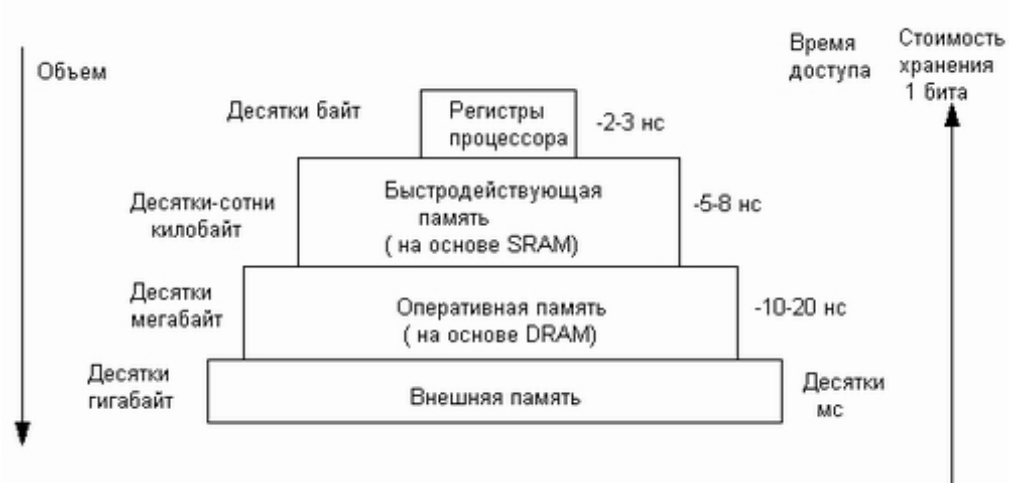


Рис. 3.1. Иерархия запоминающих устройств

Фундаментом этой пирамиды запоминающих устройств служит внешняя память, как правило, представляемая жестким диском. Она имеет большой объем (десятки и сотни гигабайт), но скорость доступа к данным является невысокой. Время доступа к диску измеряется миллисекундами.

На следующем уровне располагается более быстродействующая (время доступа[1] равно примерно 10-20 наносекундам) *и менее объемная* (от десятков мегабайт до нескольких гигабайт) *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 (Кэш), объем которых составляет от нескольких десятков до нескольких сотен килобайт, а время доступа к данным обычно не превышает 8 нс.

***Кэш-память**, или просто кэш (cache), — это способ совместного функционирования двух типов запоминающих устройств, который за счет динамического копирования в «быстрое» ЗУ наиболее часто используемой информации из «медленного» ЗУ позволяет, с одной стороны, уменьшить среднее время доступа к данным, а с другой стороны, экономить более дорогую быстродействующую память.*

Верхушку в этой пирамиде составляют внутренние регистры процессора, которые также могут быть использованы для промежуточного хранения данных. Общий объем регистров составляет несколько десятков байт, а время доступа определяется быстродействием процессора и равно в настоящее время примерно 2-3 нс.

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.

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. Однако пользователю хотелось бы иметь и недорогую, и быструю память. Кэш-память представляет некоторое компромиссное решение этой проблемы.

3.2 Управление памятью

Под памятью (memory) *как правило подразумевается оперативная память компьютера*. В отличие от памяти жесткого диска, которую называют внешней памятью (**storage**), оперативной памяти для сохранения информации требуется постоянное электропитание.

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. *Особая роль памяти объясняется тем, что процессор может выполнять инструкции протравы 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 свободной и занятой памяти;*
- *выделение памяти процессам и освобождение памяти по завершении процессов*;
- *вытеснение кодов и данных процессов из оперативной памяти на диск* (полное или частичное), когда размеры основной памяти не достаточны 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*.

Помимо первоначального выделения памяти процессам при их создании ОС должна также заниматься *динамическим распределением памяти*, то есть выполнять запросы приложений на выделение им

дополнительной памяти во время выполнения. 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.

3.3 Типы адресации

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*** *присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.*
- ***Виртуальные адреса****, называемые иногда математическими, или логическими адресами, вырабатывает транслятор, переводящий программу на машинный язык.* Поскольку во время трансляции в общем случае не известно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) 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****.*

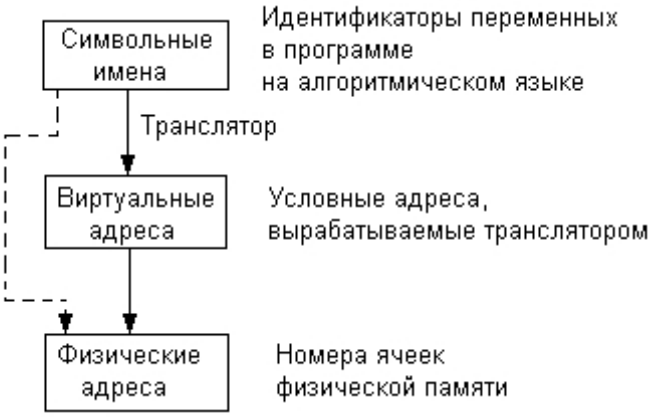


Рис. 3.1. Типы адресов

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 $FFFFFFFF_{16}$. Тем не менее каждый процесс имеет собственное виртуальное адресное пространство — транслятор присваивает виртуальные адреса переменным и кодам каждой программе независимо (рис. 3.2). Совпадение виртуальных адресов переменных и команд различных процессов не приводит к конфликтам, так как в том случае, когда эти переменные одновременно присутствуют в памяти, операционная система отображает их на разные физические адреса.

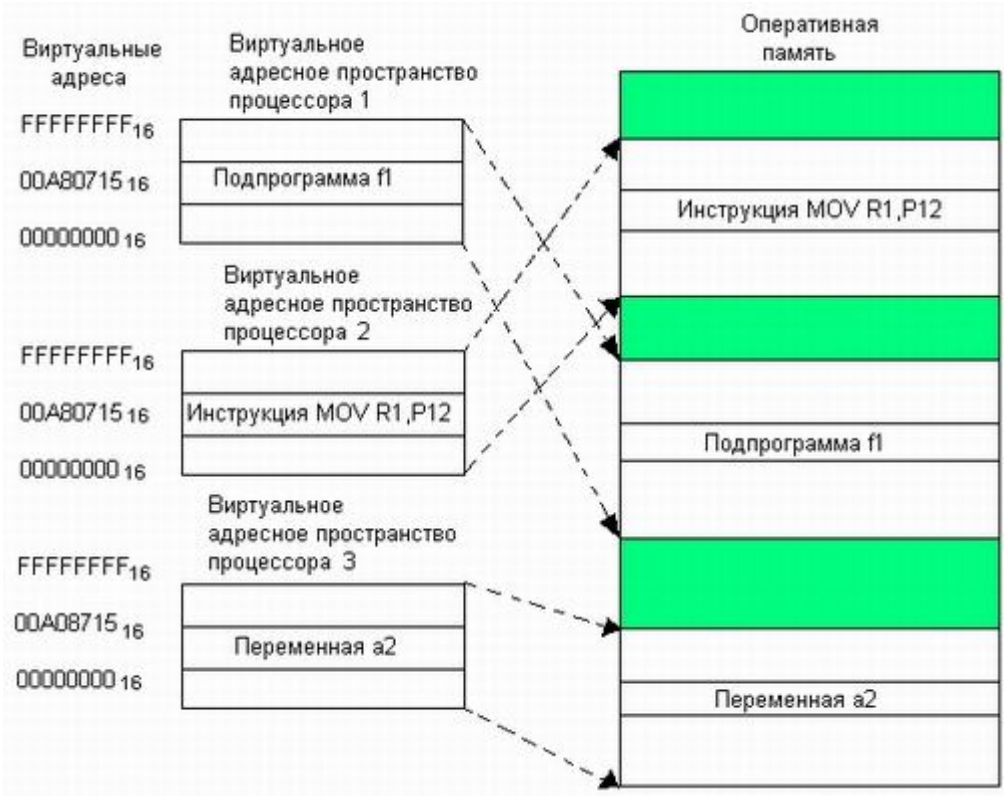


Рис. 3.2. Виртуальные адресные пространства нескольких программ

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 линейной последовательности виртуальных адресов. Такую структуру адресного пространства называют также* ***плоской*** **(*flat*)**. *При этом виртуальным адресом является единственное число, представляющее собой смещение относительно начала (обычно это значение 000...000) виртуального адресного пространства* (рис. 3.3, а). *Адрес такого типа называют линейным виртуальным адресом.*
- В других ОС *виртуальное адресное пространство делится на части, называемые* ***сегментами (или секциями, или областями, или другими терминами)***. *В этом случае помимо линейного адреса*

может быть использован виртуальный адрес, представляющий собой пору чисел (и, т), где n определяет сегмент, а т — смещение внутри сегмента (рис. 3.3, б).

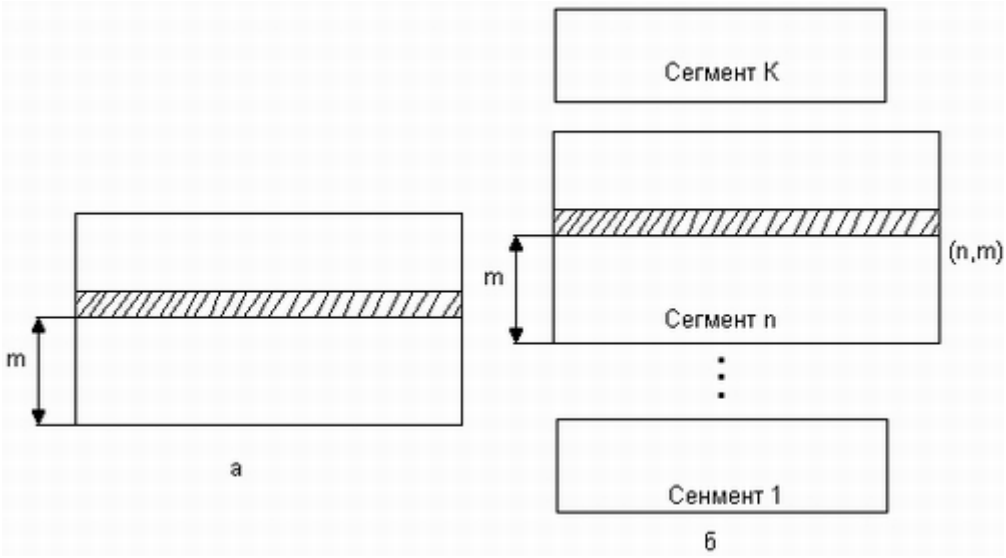


Рис. 3.3. Типы виртуальных адресных пространств:
плоское (а), сегментированное (б)

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 на общую физическую память.

Существуют два принципиально отличающихся подхода к преобразованию виртуальных адресов в физические:

1. *замена виртуальных адресов на физические 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.
2. *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.

Во втором случае 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. 3.4.

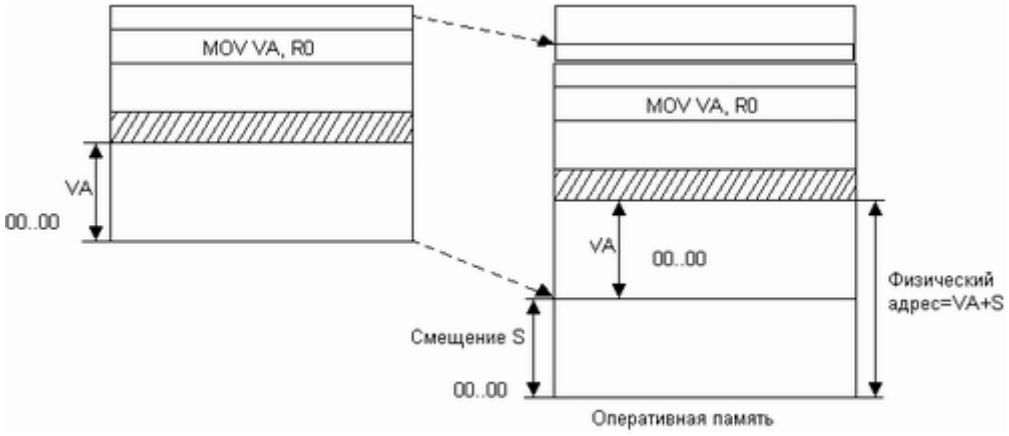


Рис. 3.4. Схема динамического преобразования адресов

Пусть, например, некоторая программа, работающая под управлением этой ОС, загружена в физическую память начиная с физического адреса S. ОС 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 пересылки данных, находящихся по адресу VA, виртуальный адрес VA заменяется физическим адресом VA+S.

Такой 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.

3.4 Виртуальная память и свопинг

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. 3.4, а). Именно на этом принципе основана **виртуальная память** —

наиболее совершенный механизм, используемый в операционных системах для управления памятью.

Однако соотношение объемов виртуальной и физической памяти может быть и обратным. Так, в мини-компьютерах 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. 3.4, б).

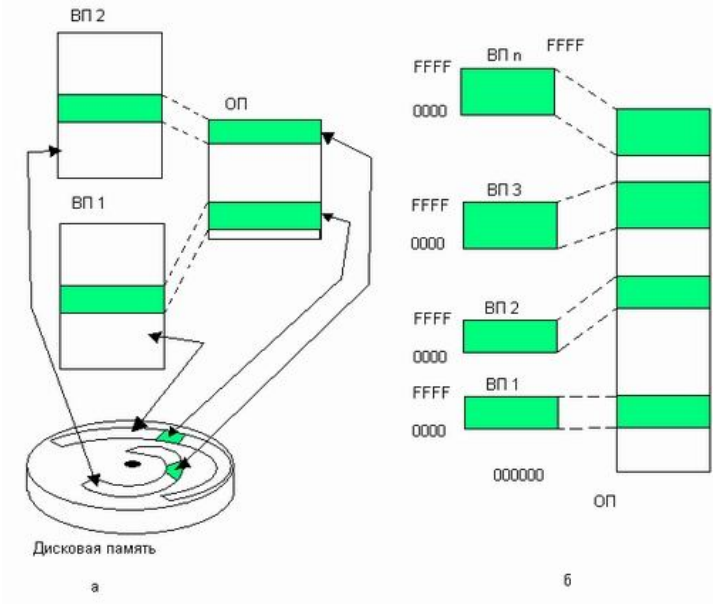


Рис. 3.4. Соотношение объемов виртуального адресного пространства и физической памяти: виртуальное адресное пространство превосходит объем физической памяти (а), виртуальное адресное пространство меньше объема физической памяти (б)

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.

Подмена (виртуализация) оперативной памяти дисковой памятью позволяет повысить объем оперативной памяти компьютера поскольку суммарный объем памяти, занимаемой процессами, может существенно превосходить имеющийся объем оперативной памяти.

Виртуальным *называется ресурс, который пользователю или пользовательской программе представляется обладающим свойствами, которыми он в действительности не обладает.* В данном случае в распоряжение прикладного программиста предоставляется виртуальная

оперативная память, размер которой намного превосходит всю имеющуюся в системе реальную оперативную память.

Виртуализация оперативной памяти осуществляется совокупностью программных модулей ОС и аппаратных схем процессора и включает 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; Q преобразование виртуальных адресов в физические.

Виртуализация памяти может быть осуществлена на основе двух различных подходов:

- ***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, более простой в реализации способ совместного использования оперативной памяти и диска. Однако подкачке 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.

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 форме 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).

3.5 Алгоритмы управления памятью

Рассмотрим 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. 3.7 ***все алгоритмы распределения памяти разделены на два класса:***

- *алгоритмы, в которых используется перемещение сегментов процессов между оперативной памятью и диском,*
- *алгоритмы, в которых внешняя память не привлекается.*

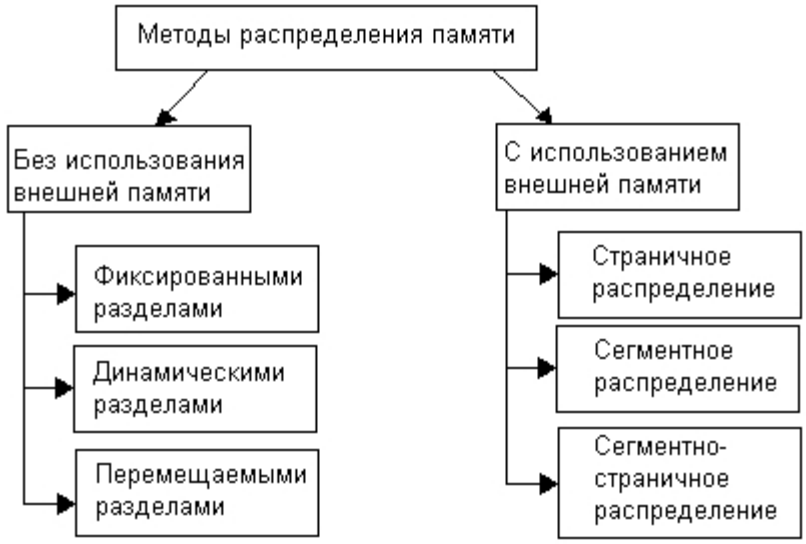


Рис. 3.7. Классификация методов распределения памяти

3.5.1 Алгоритмы управления памятью без использования механизма виртуальной памяти

3.5.1.1 Распределение памяти фиксированными разделами

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. 3.8, а), либо в очередь к некоторому разделу (рис. 3.8, б).

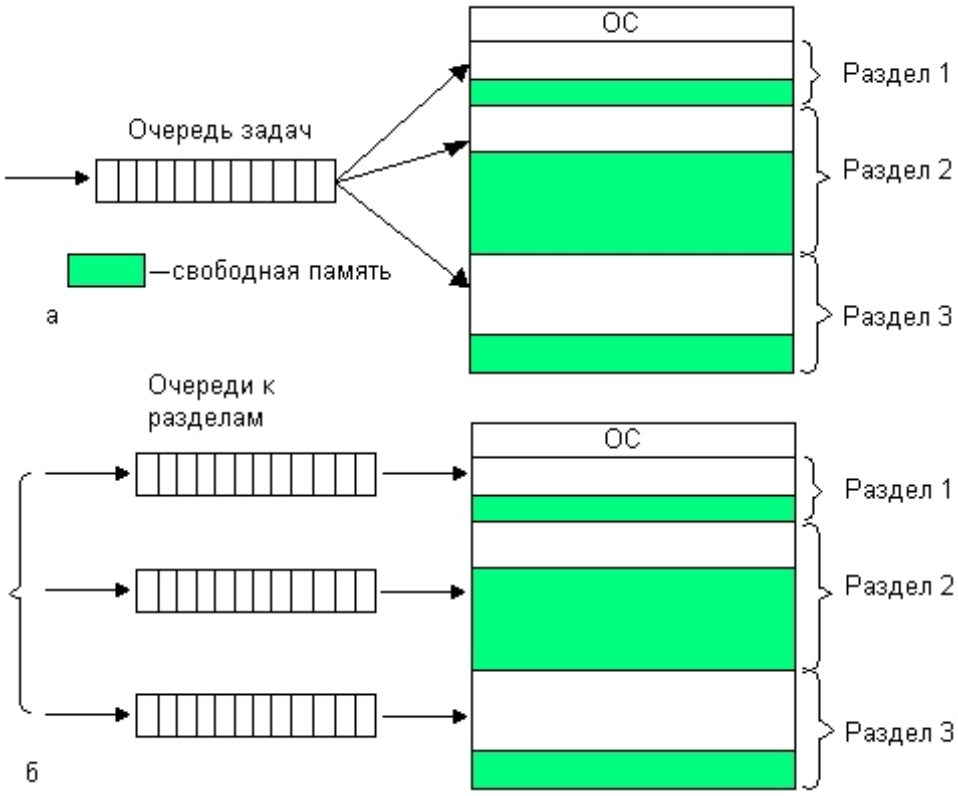


Рис. 3.8. Распределение памяти фиксированными разделами: с общей очередью (а), с отдельными очередями (б)

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. С другой стороны, 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.* Такой способ управления памятью применялся в ранних мультипрограммных ОС.

3.5.1.2 Распределение памяти динамическими разделами

В этом случае 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 него 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. 3.9 показано состояние памяти в различные

моменты времени при использовании динамического распределения. Так, в момент t_0 в памяти находится только ОС, а к моменту t_1 память разделена между 5 процессами, причем процесс П4, завершаясь, покидает память. На освободившееся от процесса П4 место загружается процесс П6, поступивший в момент t_3.

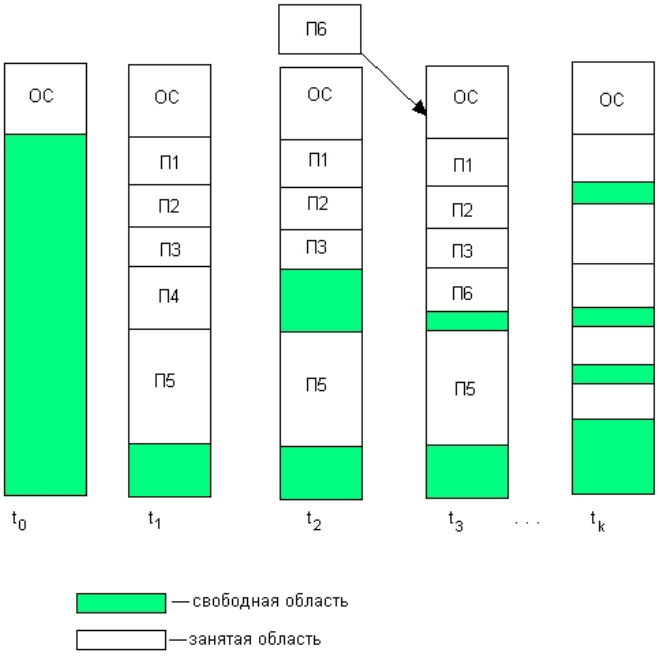


Рис. 3.9. Распределение памяти динамическими разделами

По сравнению с методом распределения памяти фиксированными разделами данный метод обладает гораздо большей гибкостью, но ему присущ очень серьезный недостаток — фрагментация памяти.

Фрагментация *— это наличие большого числа несмежных участков свободной памяти очень маленького размера (фрагментов).* Настолько маленького, 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 операционной системы, как OS/360.

3.5.1.3 Перемещаемые разделы

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших или младших адресов, так, чтобы вея свободная память образовала единую свободную область (рис 3.10). *ОС должна еще время от времени копировать содержимое*

разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется ***сжатием****.*

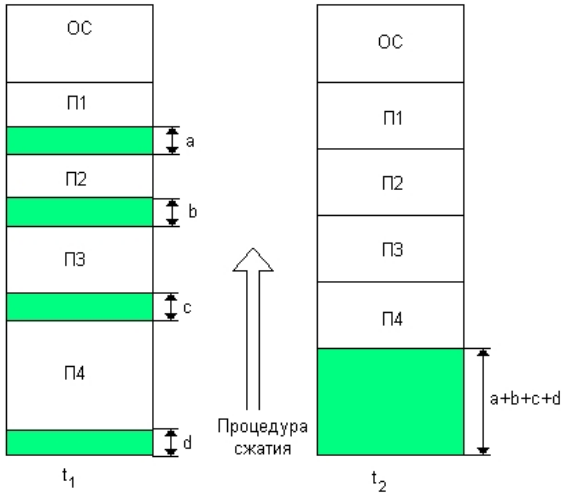


Рис. 3.10. Распределение памяти перемещаемыми разделами

Хотя процедура сжатия и приводит к более эффективному использованию памяти, она может потребовать значительного времени, что часто перевешивает преимущества данного метода.

Так как программы перемещаются по оперативной памяти в ходе своего выполнения, то в данном случае невозможно выполнить настройку адресов с помощью перемещающего загрузчика. Здесь более подходящим оказывается динамическое преобразование адресов.

3.5.2 Алгоритмы управления памятью с использованием виртуальной памяти

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.

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.*
- ***Сегментная виртуальная память предусматривает перемещение данных сегментами*** *— частями виртуального адресного пространства произвольного размера, полученными с учетом смыслового значения данных.*

– ***Сегментно-страничная виртуальная память*** *использует 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.

3.5.2.1 Страничное распределение

На рис. 3.11 показана схема страничного распределения 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.

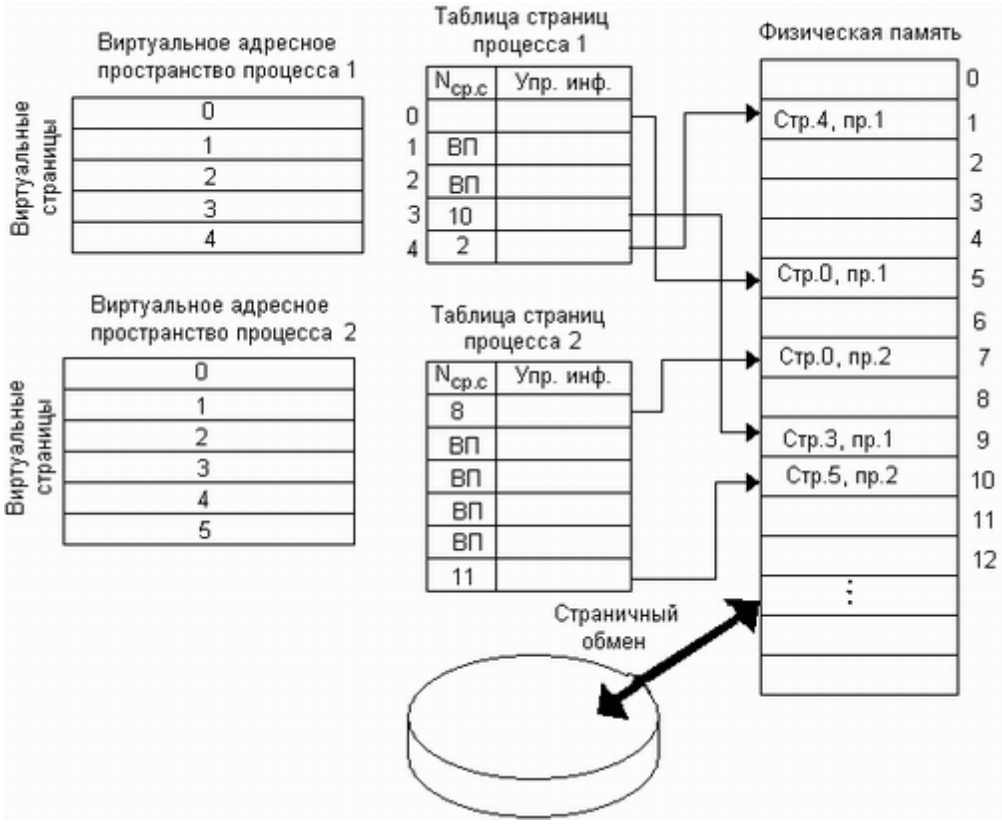


Рис. 3.11. Страничное распределение памяти

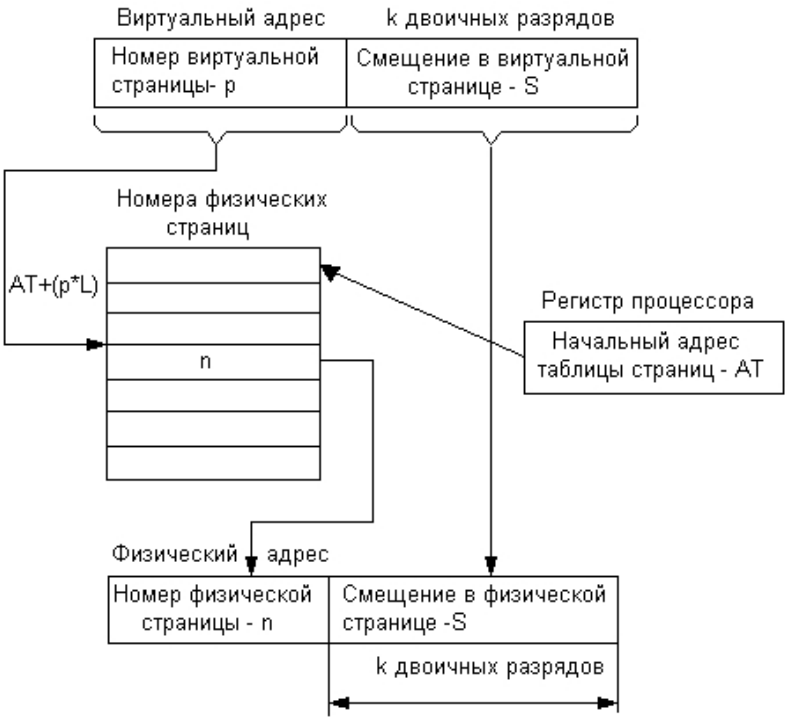


Рис. 3.12. Схема преобразования виртуального адреса в физический

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. *Организация перемещение данных между памятью и диском ведется страницами — частями виртуального адресного пространства, фиксированного и сравнительно небольшого размера*

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.

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.

3.5.2.2 Сегментное распределение

При страничной организации виртуальное адресное пространство процесса делится на равные части механически, без учета смыслового значения данных. Такой подход не позволяет обеспечить дифференцированный доступ к разным частям программы, а это свойство могло бы быть очень полезным во многих случаях. *Кроме того, разбиение виртуального адресного пространства на «осмысленные» части делает возможным совместное использование фрагментов программ разными процессами. При отображении в физическую память сегменты, содержащие коды подпрограммы из обоих виртуальных пространств, проецируются на одну и ту же область физической памяти. Таким образом оба процесса получат доступ к одной и той же копии подпрограммы* (рис. 3.13).

Итак, 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. 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. 3.14).

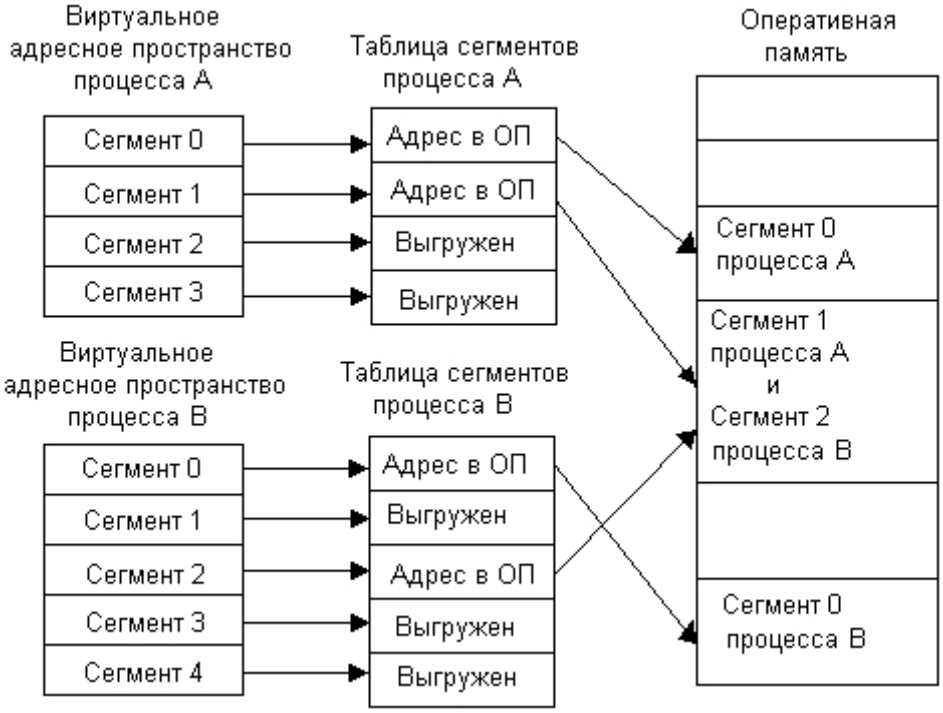


Рис. 3.13. Распределение памяти сегментами

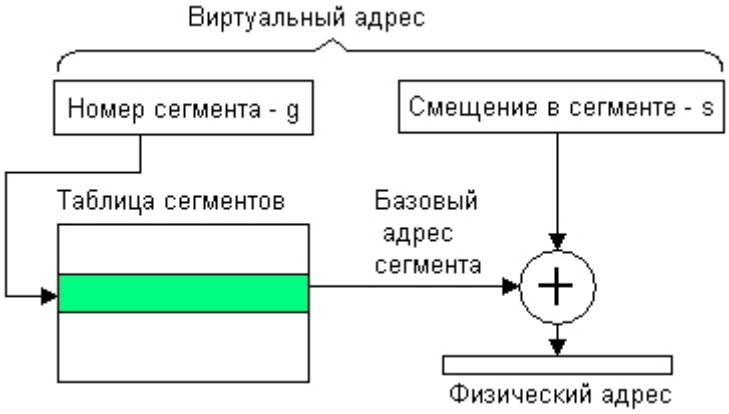


Рис. 3.14. Преобразование виртуального адреса при сегментной организации памяти

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 выше критериям выбора страниц при страничном способе управления памятью.

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. 3.14).

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.

3.5.2.3 Сегментно-страничное распределение

Метод ***сегментно-страничного распределения памяти*** *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 разделено 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 сегментной организации памяти (рис. 3.15, а) все виртуальные сегменты образуют одно непрерывное линейное виртуальное адресное пространство (рис. 3.16, б).

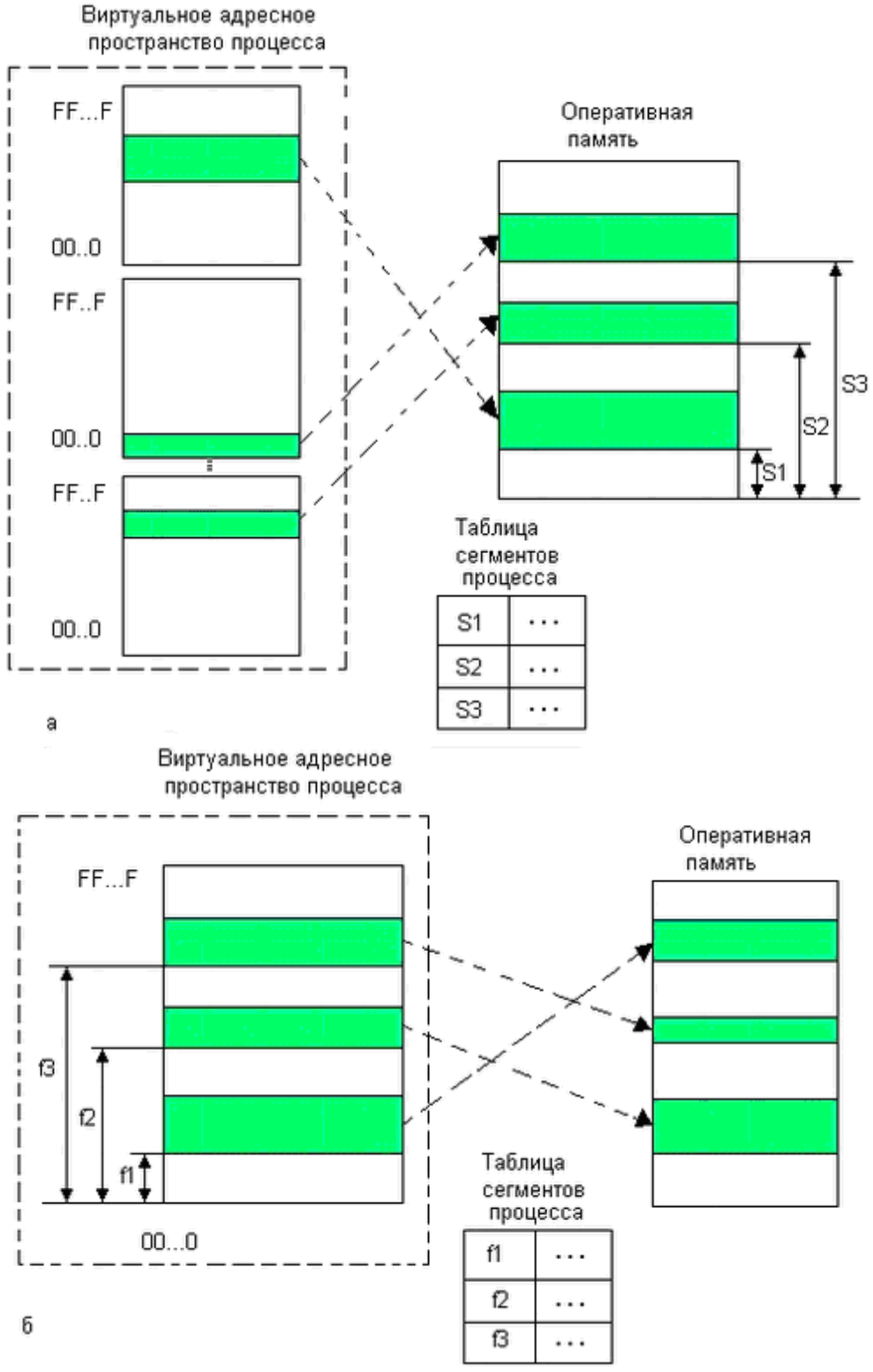


Рис. 3.16. Два способа сегментации при сегментно-страничной организации памяти

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. 3.16 базовые физические адреса обозначены SI, S2, S3, а базовые виртуальные адреса — fl, f2, f3).

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.

Преобразование виртуального адреса в физический происходит в два этапа (рис. 3.17):

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.

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. При этом в старших разрядах содержится номер виртуальной страницы, а в младших — смещение искомого элемента относительно начала страницы.

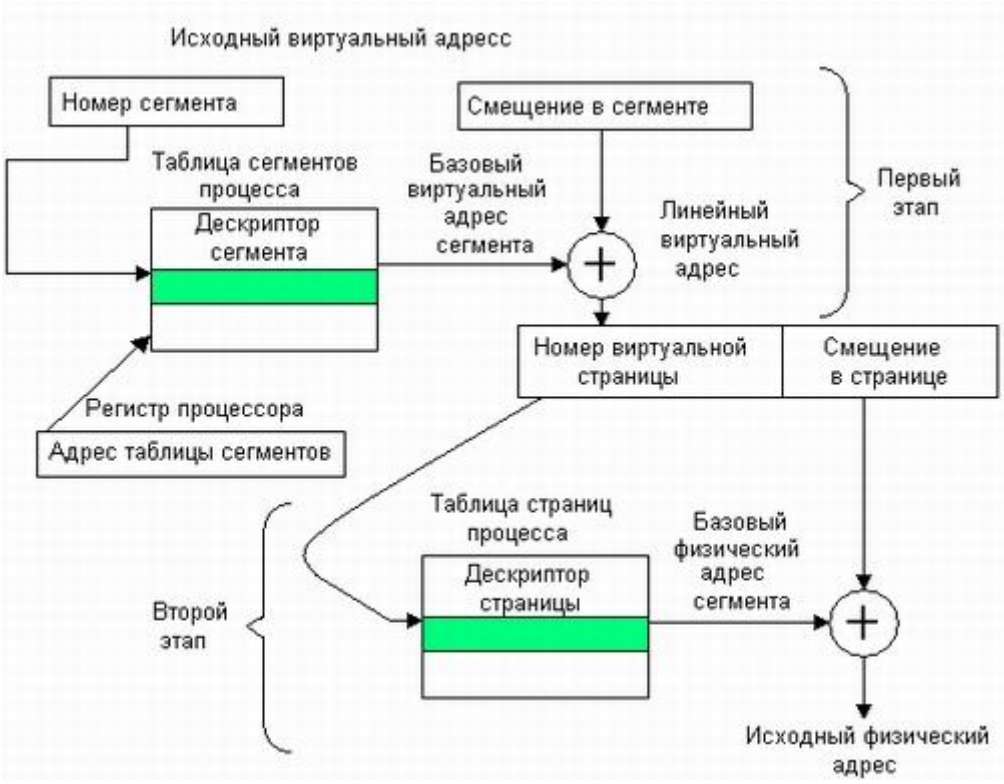


Рис. 3.17. Преобразование виртуального адреса в физический при сегментно -страничной организации памяти

4. ПРЕРЫВАНИЯ

«Прерывания названы так весьма удачно,
поскольку они прерывают нормальную работу системы».

Скотт Максвелл. Ядро Linux в комментариях. — К. ДиаСофт, 2000.

4.1 Понятие прерывания

Система прерываний переводит процессор на выполнение потока команд, отличного от того, который выполнялся до сих пор, с последующим возвратом к исходному коду. 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, которое происходит 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.*

В зависимости от источника вызывающего прерывание, последние делятся на три больших класса:

1. ***Внешние прерывания*** *могут возникать в результате действий пользователя или оператора за терминалом, или же в результате поступления сигналов от аппаратных устройств — сигналов завершения операций ввода-вывода*, вырабатываемых контроллерами внешних устройств компьютера. *Внешние прерывания называют также* ***аппаратными***, отражая тот факт, что прерывание возникает вследствие подачи некоторой аппаратурой электрического сигнала, который передается на специальный вход прерывания процессора. *Данный класс прерываний является* ***асинхронным*** *по отношению к потоку инструкций прерываемой программы.*
2. ***Внутренние прерывания****, называемые также* ***исключениями (exeption)****, происходят синхронно выполнению программы при*

появлении аварийной ситуации в ходе исполнения некоторой инструкции программы. Примерами исключений являются деление на нуль, ошибки защиты памяти, обращения по несуществующему адресу, попытка выполнить привилегированную инструкцию в пользовательском режиме и т. п.

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

Прерываниям приписывается приоритет, с помощью которого они ранжируются по степени важности и срочности.

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:*

- *Аппаратные прерывания обрабатываются драйверами соответствующих внешних устройств,*
- *исключения — специальными модулями ядра,*
- *программные прерывания — процедурами ОС, обслуживающими системные вызовы.*

Кроме этих модулей в операционной системе может находиться так называемый диспетчер прерываний, который координирует работу отдельных обработчиков прерываний.

4.2 Механизм прерываний

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

Существуют два основных способа, выполнения прерывания*, причем в обоих способах процессору предоставляется информация об уровне приоритета прерывания на шине подключения внешних устройств:*

1. ***Векторный (vectored).*** *В случае векторных прерываний в процессор передается также информация о начальном адресе программы обработки возникшего прерывания — обработчика прерываний.* Устройствам, которые используют векторные прерывания,

назначается вектор прерываний, представляющий 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.

2. ***Опрашиваемый (polled).*** *При использовании опрашиваемых прерываний процессор получает от запросившего прерывание устройства только информацию об уровне приоритета прерывания. С каждым уровнем прерываний связано несколько устройств и соответственно несколько программ — обработчиков прерываний. При возникновении прерывания процессор определяет, какое устройство запросило прерывание путем опроса 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.

Механизм прерываний аппаратной платформы может сочетать векторный и опрашиваемый типы прерываний. Контроллеры периферийных устройств выставляют на шину не вектор, а сигнал запроса прерывания определенного уровня IRQ. 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 работать 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.*

Маскирование *— при обслуживании некоторого запроса все запросы с равным или более низким приоритетом маскируются, то есть не обслуживаются. Схема маскирования предполагает возможность временного маскирования (приостановки) прерываний любого класса независимо от уровня приоритета.*

Обобщенно последовательность действий аппаратных и программных средств по обработке прерывания можно описать следующими этапами.

1. *При возникновении сигнала (для аппаратных прерываний) или условия (для внутренних прерываний) прерывания происходит первичное аппаратное распознавание типа прерывания.* В зависимости от поступившей в процессор информации (уровень прерывания, вектор прерывания или тип условия внутреннего прерывания) *происходит автоматический вызов процедуры обработки 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, а также нескольких регистров общего назначения, которые требуются программе обработки прерывания. Может быть сохранен и полный контекст процесса, если ОС обслуживает данное прерывание со сменой процесса.

3. *Одновременно с загрузкой адреса процедуры обработки прерываний в счетчик команд может автоматически выполняться загрузка нового значения слова состояния машины, которое определяет режимы работы процессора при обработке прерывания, в том числе работу в привилегированном режиме.* 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.* Многие процессоры автоматически устанавливают признак запрета прерываний в начале цикла обработки прерывания, в противном случае это делает программа обработки прерываний.

5. *После того как прерывание обработано ядром операционной системы, прерванный контекст восстанавливается и работа потока возобновляется с прерванного места.*

4.3 Функции централизованного диспетчера прерываний

Прерывания выполняют очень полезную для вычислительной системы функцию — они позволяют реагировать на асинхронные но отношению к вычислительному процессу события. Трудности обработки прерываний связаны с непредвиденными переходами управления от одной процедуры к другой, возникающими в результате прерываний от контроллеров внешних устройств. А также возникновение в непредвиденные моменты времени исключений, связанных с ошибками во время выполнения инструкций.

Операционная система должна упорядочивать выполнения системных процедур, вызываемых по прерываниям во времени так же, как планировщик упорядочивает многочисленные пользовательские потоки. Кроме того, сам планировщик потоков является системной процедурой, вызываемой по прерываниям. Поэтому правильное планирование процедур, вызываемых по прерываниям, является необходимым условием правильного планирования пользовательских потоков.

Для упорядочения работы обработчиков прерываний в операционных системах применяется тот же механизм, что и для упорядочения работы пользовательских процессов — механизм приоритетов. В операционной системе выделяется программный модуль, который занимается диспетчеризацией обработчиков прерываний, называемый ***диспетчером прерываний****.*

При возникновении прерывания диспетчер прерываний вызывается первым. Он запрещает ненадолго 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 приоритет. 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.

4.4 Процедуры обработки прерываний вызванные из текущего процесса

Важной особенностью процедур, выполняемых по запросам прерываний, является то, что они выполняют работу, чаще всего никак не связанную с текущим процессом. Процедуры обработки прерываний работают с ресурсами, которые были выделены им при инициализации соответствующего драйвера или инициализации самой операционной системы. Эти ресурсы принадлежат операционной системе, а не

конкретному процессу. В частности, память выделяется драйверам из системной области. ***Поэтому обычно говорят, что процедуры обработки 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.

Диспетчеризация прерываний является важной функцией ОС, и эта функция реализована практически во всех мультипрограммных операционных системах.

В общем случае в операционной системе реализуется двухуровневый механизм планирования работ:

1. *Верхний уровень планирования выполняется диспетчером прерываний*, который распределяет процессорное время между потоком поступающих запросов на прерывания различных типов — внешних, внутренних и программных.
2. *Оставшееся процессорное время распределяется другим диспетчером — диспетчером потоков, на основании дисциплин квантования.*

4.5 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 обмен 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 полезно использовать векторные свойства системы программных прерываний, имеющиеся во многих процессорах, то есть закрепить за каждым системным вызовом определенное значение вектора прерывания. Приложение при таком способе вызова непосредственно указывает в аргументе запроса значение вектора, после чего управление немедленно передается требуемой процедуре операционной системы (рис. 4.1, а) *– то есть используется* ***децентрализованный способ передачи управления***.

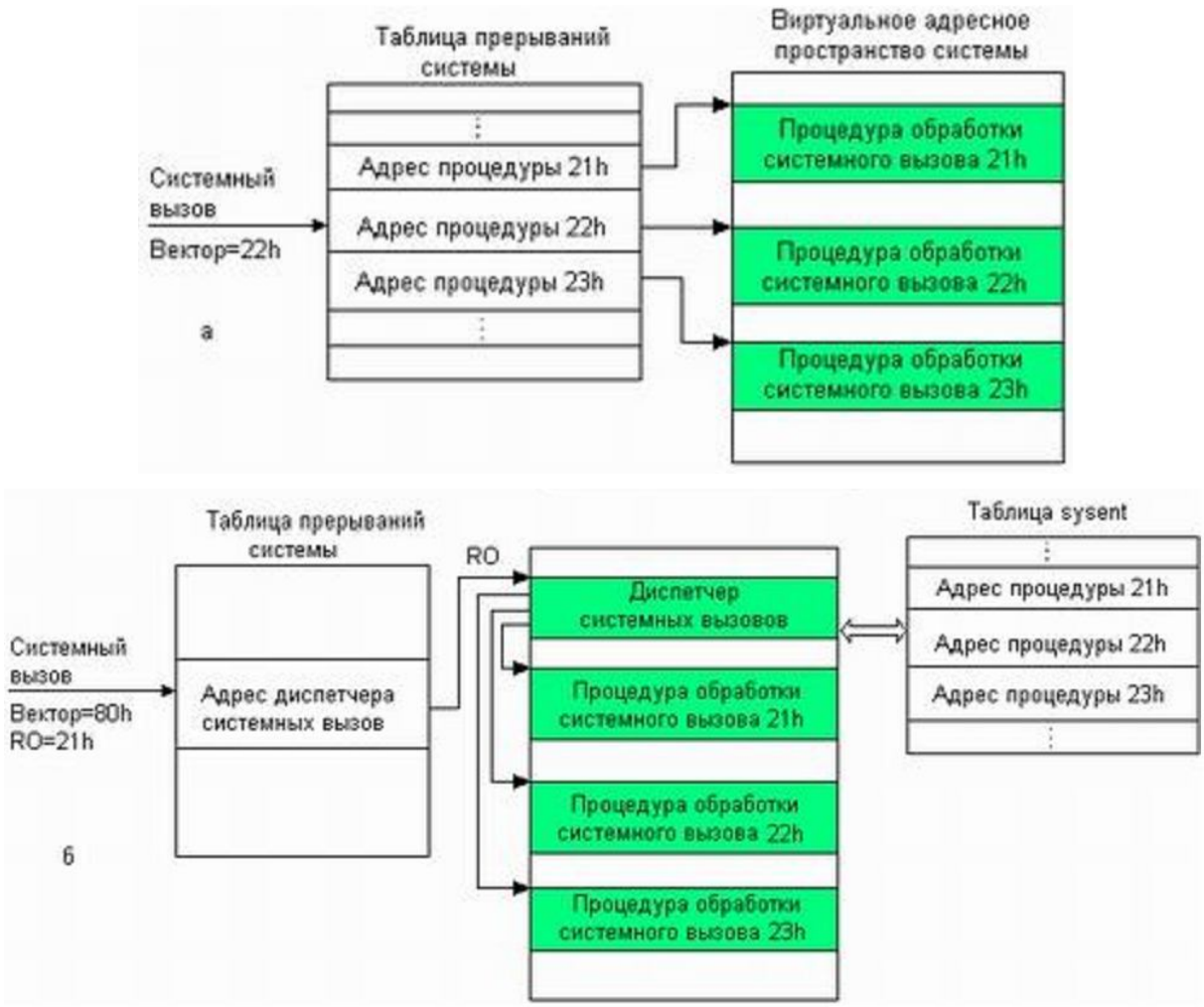


Рис. 4.1. Децентрализованная (а) и централизованная (б) схемы обработки системных вызовов

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.* Способ передачи зависит от реализации, например номер

можно поместить в определенный регистр общего назначения процессора 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 как помещаться в регистры общего назначения, так и передаваться через стек или массив, находящийся в оперативной памяти.

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 системного вызова к нескольким внутренним процедурам ядра ОС, принадлежащим к 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.

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

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. 4.2, 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 вывода-вывода, управление возвращается прикладному процессу* (рис. 4.2, б).

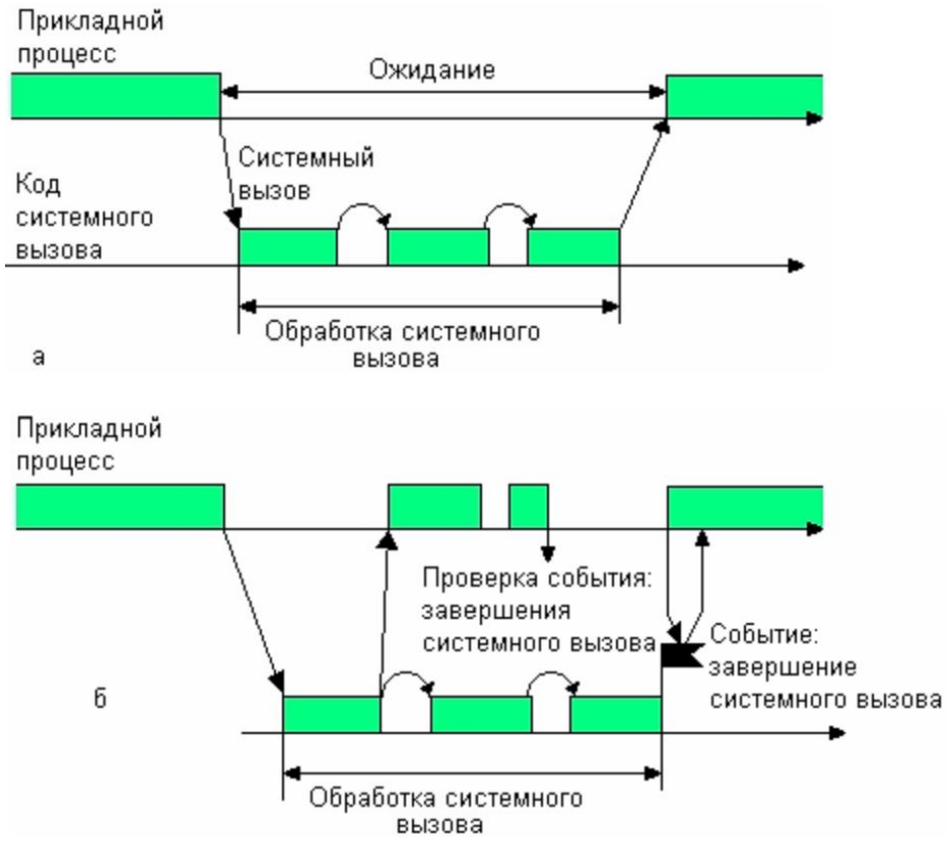


Рис. 4.2. Синхронные (а) и асинхронные (б) системные вызовы

Большинство системных вызовов в операционных системах являются синхронными, так как этот режим избавляет приложение от работы по выяснению момента появления результата вызова. Асинхронные системные вызовы применяются в операционных системах на основе микроядерного подхода, так как при этом в пользовательском режиме работает часть ОС, которым необходимо иметь полную свободу в организации своей работы, а такую свободу дает только асинхронный режим обслуживания вызовов микроядром.

5. УПРАВЛЕНИЕ ВВОДОМ-ВЫВОДОМ

5.1 Организация взаимодействия ОС с устройствами ввода-вывода

Каждое устройство ввода-вывода вычислительной системы (диск, принтер, терминал и т. п.) *снабжено специализированным блоком управления, называемым контроллером. Контроллер взаимодействует с драйвером — системным программным модулем, предназначенным для управления данным устройством.* 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 терминала или записать в определенный сектор диска).

Основными компонентами подсистемы ввода-вывода являются драйверы, управляющие внешними устройствами, и файловая система.

Достоинством подсистемы ввода-вывода любой универсальной ОС является наличие разнообразного набора драйверов для наиболее популярных периферийных устройств.

***Драйвер** взаимодействует, с одной стороны, с модулями ядра ОС (модулями подсистемы ввода-вывода, модулями системных вызовов, модулями подсистем управления процессами и памятью и т. д.), а с другой стороны — с контроллерами внешних устройств. Поэтому* существуют два типа интерфейсов:

- интерфейс «драйвер-ядро» (Driver Kernel Interface, DKI),
- интерфейс «драйвер-устройство» {Driver Device Interface, DDF).

Подсистема ввода-вывода (Input-Output Subsystem) мультипрограммной ОС при обмене данными с внешними устройствами компьютера должна решать следующие задачи:

- организация параллельной работы устройств ввода-вывода и процессора;
- согласование скоростей обмена и кэширование данных;
- разделение устройств и данных между процессами;
- обеспечение удобного логического интерфейса между устройствами и остальной частью системы;
- 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.

Операция ввода-вывода может выполняться по отношению к программному модулю, запросившему операцию:

- ***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 мультипрограммном режиме одновременно с операцией ввода-вывода* (рис. 7.1, б)*.*

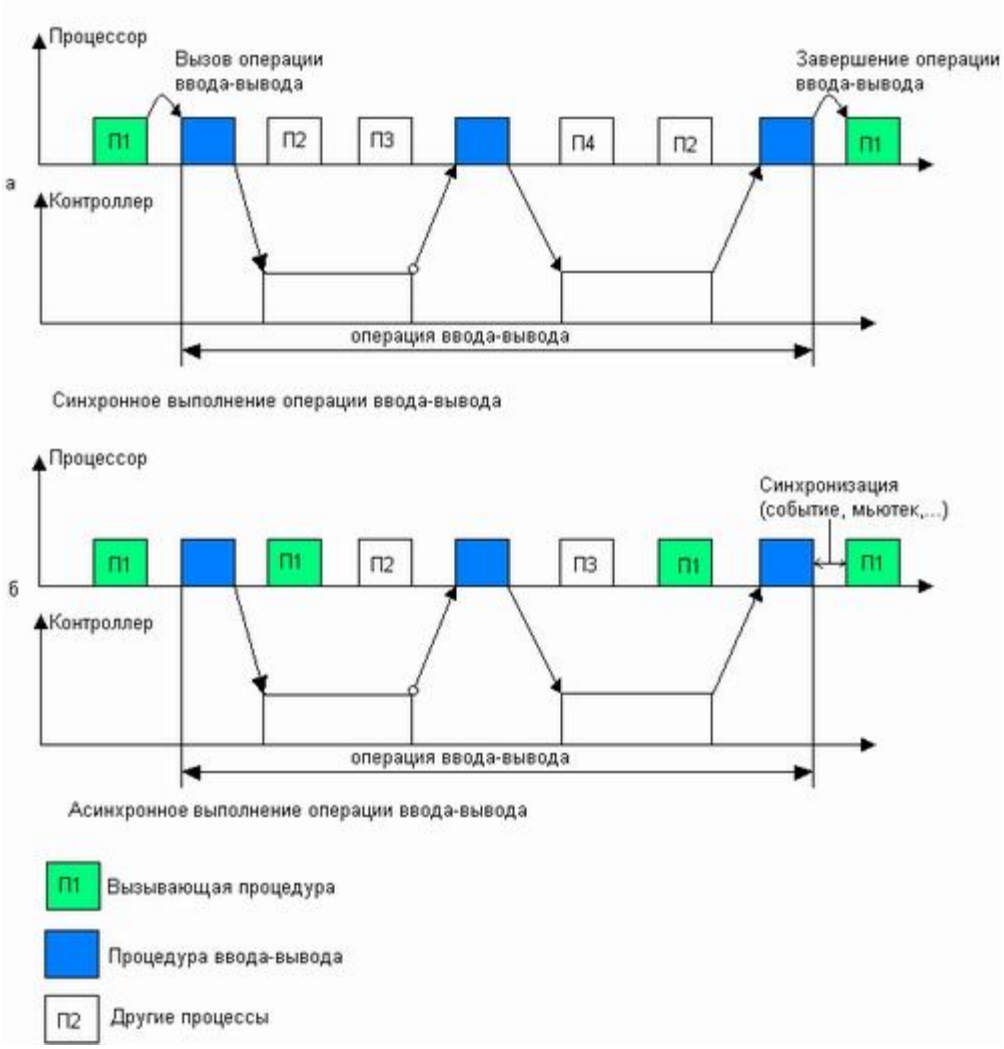


Рис. 5.1. Два режима выполнения операций ввода-вывода

5.2 Многослойная модель подсистемы ввода-вывода

Многослойное построение программного обеспечения, характерно при построении подсистемы ввода-вывода. При этом нижние слои подсистемы ввода-вывода должны включать индивидуальные драйверы, написанные для конкретных физических устройств, а верхние слои должны обобщать процедуры управления этими устройствами, предоставляя общий

интерфейс для групп устройств, обладающих некоторыми общими характеристиками.

В самом общем виде программное обеспечение ввода-вывода можно разделить на четыре слоя (рисунок 5.2)*:*

1. *Обработка прерываний,*
2. *Драйверы устройств,*
3. *Независимый от устройств слой операционной системы,*
4. *Пользовательский слой программного обеспечения.*

Рис. 5.2. Многоуровневая организация подсистемы ввода-вывода

В более частном виде структура подсистемы ввода-вывода, характерная для современных ОС представлена на рис. 5.3.

Большая часть программного обеспечения ввода-вывода является независимой от устройств. Точная граница между драйверами и независимыми от устройств программами определяется системой, так как некоторые функции, которые могли бы быть реализованы независимым способом, в действительности выполнены в виде драйверов для повышения эффективности функционирования.

Рис. 5.3. Структура подсистемы ввода-вывода современной ОС

Типичными функциями для независимого от устройств слоя являются:

- обеспечение общего интерфейса к драйверам устройств,
- именование устройств,
- защита устройств,
- обеспечение независимого размера блока,
- буферизация,
- распределение памяти на блок-ориентированных устройствах,
- распределение и освобождение выделенных устройств,
- уведомление об ошибках.

5.3 Менеджеры ввода-вывода

В подсистеме ввода-вывода наряду с модулями, отражающими специфику внешних устройств и образующими вертикальные подсистемы, существуют модули универсального назначения.

Менеджер ввода-вывода*. – модуль ОС организующий согласованную работу всех остальных компонентов подсистемы ввода-вывода, взаимодействие с пользовательскими процессами и другими подсистемами ОС. Причем функции управления устройствами, распределены по всем уровням, образуя оболочку.*

Верхний слой менеджера *составляют системные вызовы ввода-вывода, которые принимают от пользовательских процессов запросы на ввод-вывод 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.

5.4 Драйверы устройств

Под ***драйвером*** *понимается программный модуль, который обладает следующими свойствами и функциями:*

- *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.

В операционной системе только драйвер устройства знает о конкретных особенностях какого-либо устройства.

Порядок функционирования драйвера устройства:

- *Драйвер устройства принимает запрос от программного слоя и решает, как его выполнить.* Если драйвер был свободен во время поступления запроса, то он начинает выполнять запрос немедленно.

Если же он был занят обслуживанием другого запроса, то вновь поступивший запрос присоединяется к очереди уже имеющихся запросов, и он будет выполнен, когда наступит его очередь.

- *Преобразование запроса ввода-вывода из абстрактной формы в конкретную.* Для дискового драйвера это означает преобразование номеров блоков в номера цилиндров, головок, секторов, проверку, работает ли мотор, находится ли головка над нужным цилиндром.
- *Передача команд контроллеру и принятие решения должен ли драйвер блокировать ли себя до окончания заданной операции или нет.* Если операция занимает значительное время, как при печати некоторого блока данных, то драйвер блокируется до тех пор, пока операция не завершится, и обработчик прерывания не разблокирует его. Если команда ввода-вывода выполняется быстро (например, прокрутка экрана), то драйвер ожидает ее завершения без блокирования.
- *Возвращение управления программе (вызвавшей драйвер) с результатом операции ввода-вывода.*

6. ФАЙЛОВАЯ СИСТЕМА

6.1 Организация файловой системы

Одной из основных задач операционной системы является предоставление удобств пользователю при работе с данными, хранящимися на дисках. Для этого ОС подменяет физическую структуру хранящихся данных некоторой удобной для пользователя логической моделью. *Логическая модель файловой системы материализуется в виде дерева каталогов, выводимого на экран такими утилитами, как Windows Explorer, в символьных составных именах файлов, в командах работы с файлами.* Базовым элементом этой модели является файл, который так же, как и файловая система в целом, может характеризоваться как логической, так и физической структурой.

Файл *— это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные.* Файлы хранятся в памяти, на зависящей от энергопитания, обычно — на магнитных дисках.

Основные цели использования файла перечислены ниже.

- *Долговременное и надежное хранение информации.* Долговременность достигается за счет использования запоминающих устройств, не зависящих от питания, 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 легкий способ разделения информации между приложениями и пользователями за счет наличия понятного человеку символьного имени и постоянства хранимой информации и расположения файла.

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 над файлами, такие как создание, уничтожение, чтение, запись, именование и поиск файлов.*

Файловая система распределяет дисковую память, поддерживает именование файлов, отображает имена файлов в соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление файлов. Таким образом, ФС 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.

Основные функции в такой 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.*
- *защита файлов одного пользователя от несанкционированного доступа другого пользователя.*

6.2 Типы файлов

Файлы бывают нескольких типов:

1 обычные файлы:
  1.1 текстовые,
  1.2 двоичные;
2 специальные файлы:
  2.1 блок-ориентированные,
  2.2 байт-ориентированные;
3 файлы-каталоги.

Обычные файлы *в свою очередь подразделяются на текстовые файлы и двоичные файлы.*

Текстовые файлы *состоят из строк символов*, представленных в ASCII-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 ASCII-коды, они часто имеют сложную внутреннюю структуру, например, объектный код программы или архивный файл.* Все операционные системы должны уметь распознавать хотя бы один тип файлов - их собственные исполняемые файлы.

Специальные файлы *- это файлы, ассоциированные с устройствами ввода-вывода, которые позволяют пользователю выполнять операции ввода-вывода, используя обычные команды записи в файл или чтения из файла.* Эти

команды обрабатываются вначале программами файловой системы, а затем на некотором этапе выполнения запроса преобразуются ОС в команды управления соответствующим устройством. *Специальные файлы, так же как и устройства ввода-вывода, делятся на:*

- ***блок-ориентированные**,*
- ***байт-ориентированные**.*

***Каталог** - это, с одной стороны, группа файлов, объединенных пользователем исходя из некоторых соображений (например, файлы, содержащие программы игр, или файлы, составляющие один программный пакет), а с другой стороны - это файл, содержащий системную информацию о группе файлов, его составляющих.* В каталоге содержится список файлов, входящих в него, и устанавливается соответствие между файлами и их характеристиками (атрибутами).

В разных файловых системах могут использоваться в качестве атрибутов разные характеристики, например:

- *информация о разрешенном доступе,*
- пароль для доступа к файлу,
- владелец файла,
- создатель 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,
- указатель на ключевое поле в записи,
- длина ключа,
- *времена создания, последнего доступа и последнего изменения,*
- текущий размер файла,
- максимальный размер файла.

***Параметры прав доступа** в самом общем случае 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* (рисунок 6.1).

В некоторых системах пользователи могут быть разделены на отдельные категории. Для всех пользователей одной категории определяются

единые права доступа. Например, в системе UNIX все пользователи подразделяются на три категории:

- владельца файла,
- членов его группы
- всех остальных.

Имена файлов

Имена пользователей	modern.txt	win.exe	class.dbf	unix.ppt
kira	читать	выполнять	–	выполнять
genya	читать	выполнять	–	выполнять читать
nataly	читать	–	–	выполнять читать
victor	читать писать	–	создать	–

Рис. 6.1. Матрица прав доступа

Различают два основных подхода к определению прав доступа:

1. ***избирательный доступ****, когда для каждого файла и каждого пользователя сам владелец может определить допустимые операции;*
2. ***мандатный подход****, когда система наделяет пользователя определенными правами по отношению к каждому разделяемому ресурсу (в данном случае файлу) в зависимости от того, к какой группе пользователь отнесен.*

6.3 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* (рис. 6.2).

Граф, описывающий иерархию каталогов, может быть ***деревом*** *или* ***сетью****. Каталоги образуют* ***дерево****, если файлу разрешено входить только в один каталог* (рис. 6.2, б)*, и* ***сеть*** *— если файл может входить сразу в несколько каталогов* (рис. 6.2, в).

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).

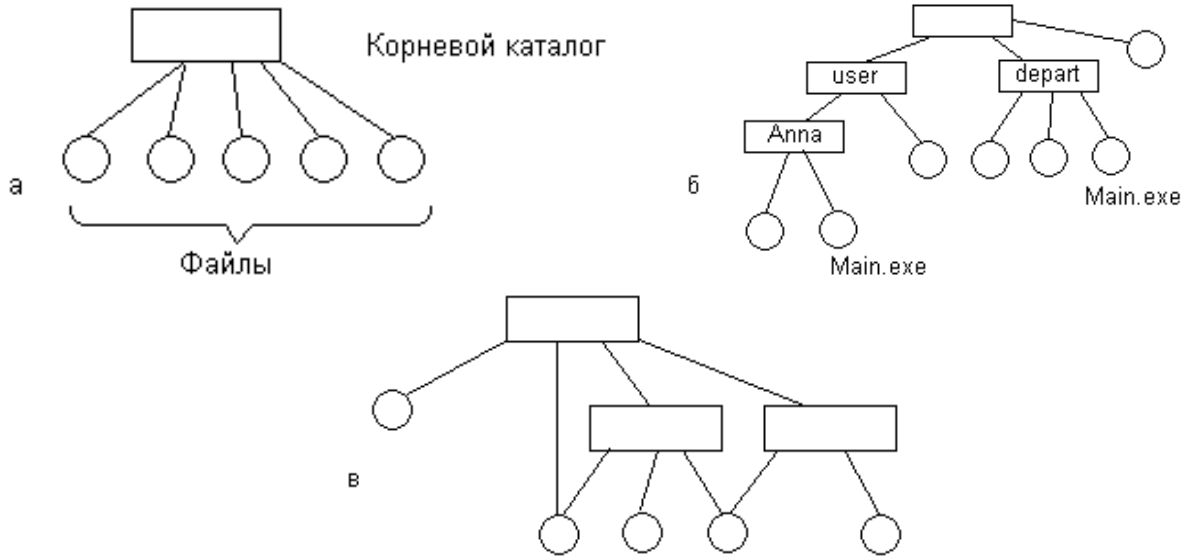


Рис. 6.2. Иерархия файловых систем

Все файлы имеют символьные имена. ***В иерархически организованных файловых системах обычно используются три типа имен файлов:***

- *простые,*
- *составные*
- *относительные (NFS).*

Простое, или короткое, символьное имя *идентифицирует файл в пределах одного каталога.* Простые имена присваивают файлам пользователи при этом они должны учитывать ограничения ОС как на номенклатуру символов, так и на длину имени. Так, в популярной файловой системе FAT длина имен ограничивались схемой 8+3 (8 символов — собственно имя, 3 символа — расширение имени).

Современные файловые системы, а также усовершенствованные варианты уже существовавших файловых систем, как правило, поддерживают длинные простые символьные имена файлов. Например, в файловых сиетемах NTFS и FAT32, входящих в состав операционной системы Windows NT, имя файла может содержать до 255 символов.

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****, в котором простые имена отделены друг от друга принятым в ОС разделителем.* Часто в качестве разделителя используется прямой или обратный слеш, при этом принято не указывать имя корневого каталога. На рис. 6.2, 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.

6.4 Понятие о монтировании

Монтирование *– операция в результате которой пользователю предоставляется возможность объединять файловые системы, находящиеся на разных устройствах, в единую файловую систему, описываемую единым деревом каталогов.*

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 (рис. 6.3), причем один, из дисков является системным.

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 выполнения монтирования выбранный каталог man становится корневым каталогом второй файловой системы. Через этот каталог монтируемая файловая система подсоединяется как поддерево к общему дереву (рис. 6.4).

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 она с самого начала была единой.

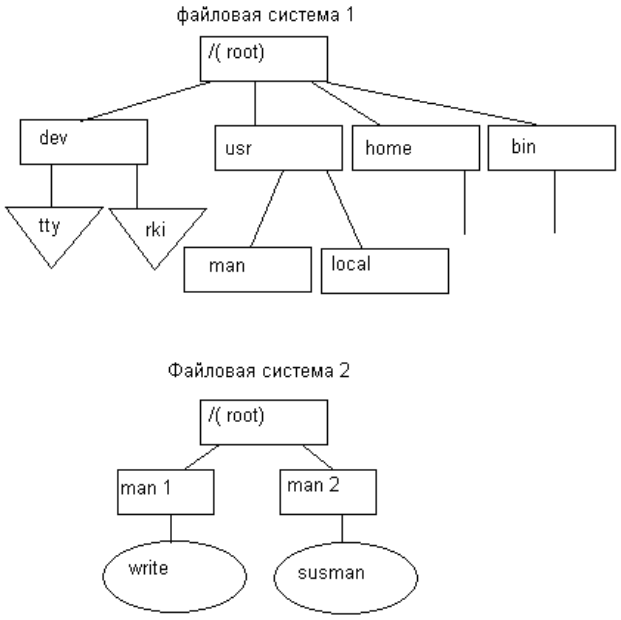


Рис. 6.3. Две файловые системы до монтирования

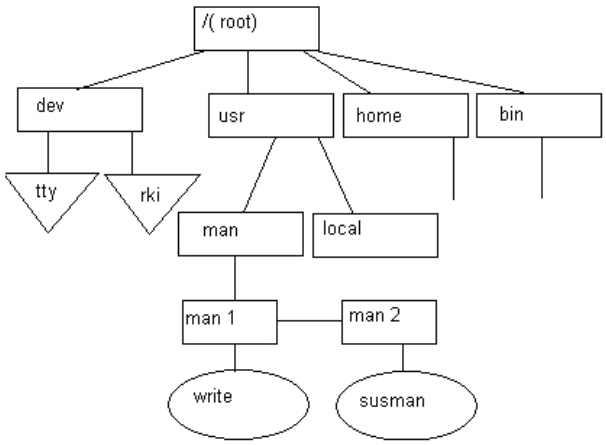


Рис. 6.4. Общая файловая система после монтирования

6.5 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 байт, 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**.

Основным типом устройства, которое используется в современных вычислительных системах для хранения файлов, являются ***дисковые накопители****.* 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. 6.5).

На каждой стороне каждой пластины размечены тонкие концентрические кольца — ***дорожки (traks)****, на которых хранятся данные.* Количество дорожек зависит от типа диска. *Когда диск вращается, элемент, называемый* ***головкой****, считывает двоичные данные с магнитной дорожки или записывает их на магнитную дорожку.*

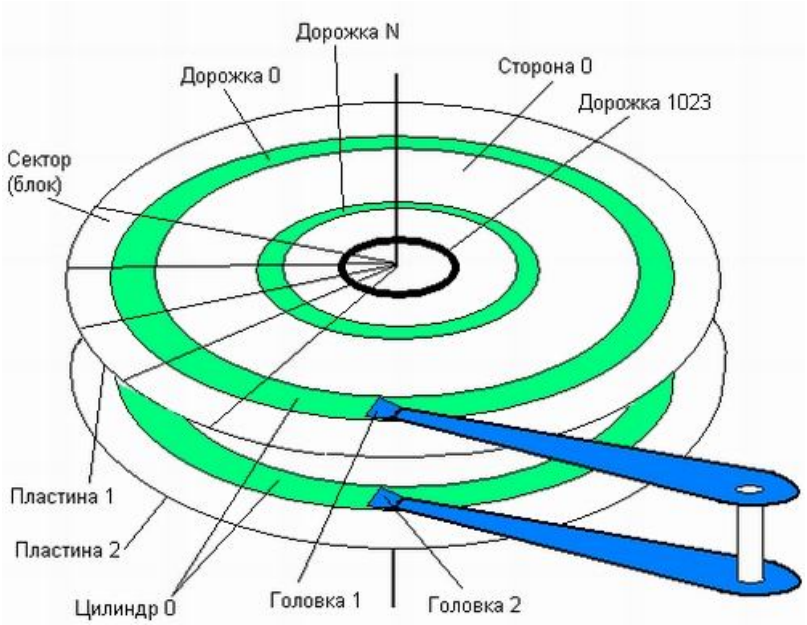


Рис. 6.5. Схема устройства жесткого диска

Головка позиционируются над заданной дорожкой. Головки перемещаются над поверхностью диска дискретными шагами, каждый шаг соответствует сдвигу на одну дорожку. В дисках имеется по головке на каждую дорожку. 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.*

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 можно максимально записать одно и то же число байт*.

Сектор *— наименьшая адресуемая единица обмена данными дискового устройства с оперативной памятью. Для того чтобы контроллер мог найти на диске нужный сектор, необходимо задать ему все составляющие адреса сектора:*

- *номер цилиндра,*
- *номер поверхности*
- *номер сектора.*

Дорожки и секторы создаются в результате выполнения процедуры физического, или низкоуровневого, форматирования диска, предшествующей использованию диска.

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,*
- *информация о поврежденных областях.*
- *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 (используются также названия логический диск и логический раздел)*[1]*. Логическое устройство функционирует так, как если бы это был отдельный физический диск.* Именно с логическими устройствами работает пользователь, обращаясь к ним по символьным именам, используя, например, обозначения A:, B:, C:. *На каждом логическом устройстве может создаваться только одна файловая система.*

На разных логических устройствах одного и того же физического диска могут располагаться файловые системы разного типа. На рис. 6.6 показан пример диска, разбитого на три раздела, в которых установлены две файловых системы NTFS (разделы C: и E:) и одна файловая система FAT (раздел D:).

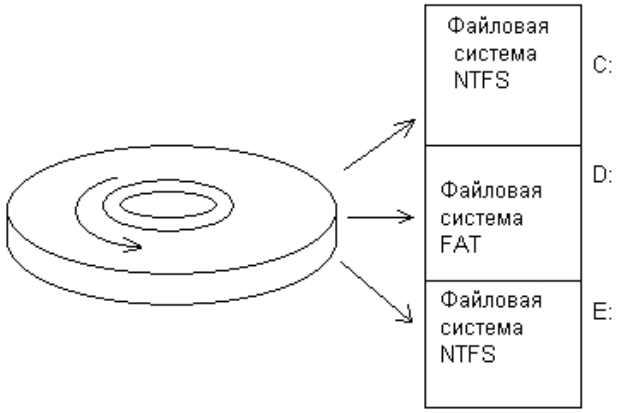


Рис. 6.6. Разбиение диска на разделы

6.6 Общая модель файловой системы

Функционирование любой файловой системы можно представить многоуровневой моделью (рисунок 6.7)*, в которой каждый уровень*

предоставляет некоторый интерфейс (набор функций) вышележащему уровню, а сам, в свою очередь, для выполнения своей работы использует интерфейс (обращается с набором запросов) нижележащего уровня.

Рис. 6.7. Общая модель файловой системы

*Задачей **символьного уровня** является определение по символьному имени файла его уникального имени.* В файловых системах, в которых один и тот же файл может иметь несколько символьных имен, на данном уровне просматривается цепочка каталогов для определения уникального имени файла.

*На следующем, **базовом уровне** по уникальному имени файла определяются его характеристики: права доступа, адрес, размер и другие.* Как уже было сказано, характеристики файла могут входить в состав каталога или храниться в отдельных таблицах. При открытии файла его характеристики перемещаются с диска в оперативную память, чтобы уменьшить среднее время доступа к файлу.

*Следующим этапом реализации запроса к файлу является **проверка прав доступа** к нему.* Для этого сравниваются полномочия пользователя или процесса, выдавших запрос, со списком разрешенных видов доступа к данному файлу. Если запрашиваемый вид доступа разрешен, то выполнение запроса продолжается, если нет, то выдается сообщение о нарушении прав доступа.

*На **логическом уровне** определяются координаты запрашиваемой логической записи в файле, то есть требуется определить, на каком расстоянии (в байтах) от начала файла находится требуемая логическая*

запись. При этом абстрагируются от физического расположения файла, он представляется в виде непрерывной последовательности байт.

*На **физическом уровне** файловая система определяет номер физического блока, который содержит требуемую логическую запись, и смещение логической записи в физическом блоке*. Для решения этой задачи используются результаты работы логического уровня - смещение логической записи в файле, адрес файла на внешнем устройстве, а также сведения о физической организации файла, включая размер блока.

После определения номера физического блока, файловая система обращается к системе ввода-вывода для выполнения операции обмена с внешним устройством. В ответ на этот запрос в буфер файловой системы будет передан нужный блок, в котором на основании полученного при работе физического уровня смещения выбирается требуемая логическая запись.

6.7 Понятие о журналируемых файловых системах

***Журналируемые файловые системы** — это класс файловых систем, характерная черта которых — ведение журнала, хранящего список изменений, в той или иной степени помогающего сохранить целостность файловой системы.*

Причиной отсутствия целостности в файловой системе может быть некорректное размонтирование, сбой в момент обновления данных, а также ошибки (отсутствие целостности) в файлах данных. Сюда же можно включить ошибки в метаданных файловой системы, которые могут привести к потерям файлов и другим серьезным проблемам.

*Для минимизации проблем, связанных с целостностью, **журналируемая файловая система** хранит список изменений, которые она будет проводить с файловой системой перед фактической записью изменений. Эти записи хранятся в отдельной части файловой системы, называемой **«журналом»**. Как только изменения файловой системы безопасно внесены в журнал, журналируемая файловая система применяет эти изменения к файлам или метаданным, а затем удаляет эти записи из журнала.* Записи журнала организованы в наборы связанных изменений файловой системы, что очень похоже на то, как изменения добавляемые в базу данных организованы в транзакции.

Наличие журнала повышает вероятность сохранения целостности файловой системы, потому что записи в лог-файл ведутся до проведения фактических изменений, и эти записи хранятся до тех пор, пока они не будут целиком и безопасно применены. *В случае сбоя в компьютере программа монтирования может гарантировать целостность журналируемой файловой системы простой проверкой «журнала» на наличие ожидаемых, но не произведенных изменений и последующей записью их в файловую*

систему. Т.о. при наличии журнала в большинстве случаев системе не нужно проводить проверку целостности файловой системы, а это означает, что компьютер будет доступен для работы практически сразу после перезагрузки. Соответственно, шансы потери данных в связи с проблемами в файловой системе значительно снижаются.

6.8 Физическая организация и адресация в файле

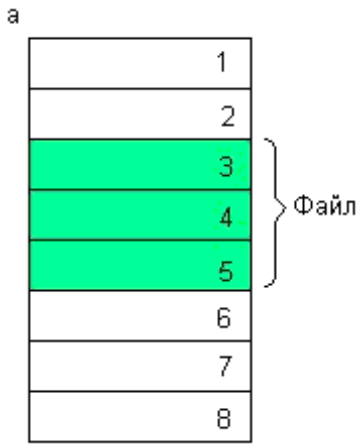
Важным компонентом физической организации файловой системы является ***физическая организация файла****, то есть способ размещения файла на диске.*

Основными критериями эффективности физической организации файлов являются:

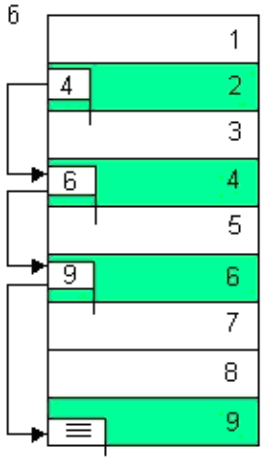
- *скорость доступа к данным;*
- *объем адресной информации файла;*
- *степень фрагментированности дискового пространства;*
- *максимально возможный размер файла.*

Непрерывное размещение *— простейший вариант физической организации* (рис. 6.8, 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. Серьезной проблемой является фрагментация. Поэтому на практике используются методы, в которых файл размещается в нескольких, в общем случае несмежных областях диска.

Размещение файла в виде связанного списка кластеров *дисковой памяти* (рис. 6.9, б) *в этом 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, 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.

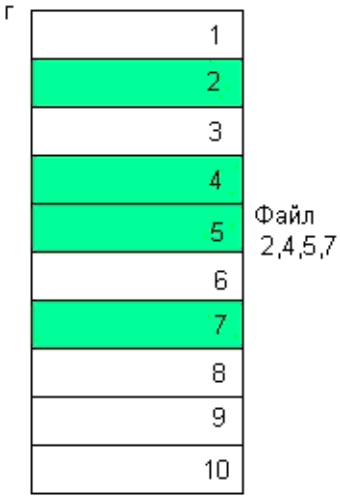


а) непрерывное размещение



б) связанный список кластеров

в) связанный список индексов



г) перечень номеров кластеров

Рис. 6.9. Физическая организация файла

Использование связанного списка индексов (рис. 6.9, в) - *применяется, в файловой системе FAT. 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 отдельной 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.

Перечисление номеров кластеров, занимаемых этим файлом (рис. 6.9, г)*, при этом данный перечень 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.

7. ОСОБЕННОСТИ ПОСТРОЕНИЯ СОВРЕМЕННЫХ ФАЙЛОВЫХ СИСТЕМ

7.1 Файловая система FAT

Логический раздел, содержащий файловую систему FAT, состоит из следующих областей.

- *Загрузочный сектор содержит программу начальной загрузки операционной системы. Вид этой программы зависит от типа операционной системы, которая будет загружаться из этого раздела.*
- *Основная копия FAT содержит информацию о размещении файлов и каталогов на диске.*
- *Резервная копия 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.*

Файловая система FAT поддерживает всего два типа файлов:

- *обычный файл,*
- *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 в поле записи каталога «номер первого кластера» фиксируется номер этого указателя. 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 «последний кластер файла». *Если 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* (рис. 7.1). *Процесс повторяется до тех пор, пока не будут размещены все данные файла.* Таким образом, создается связный список всех кластеров файла.

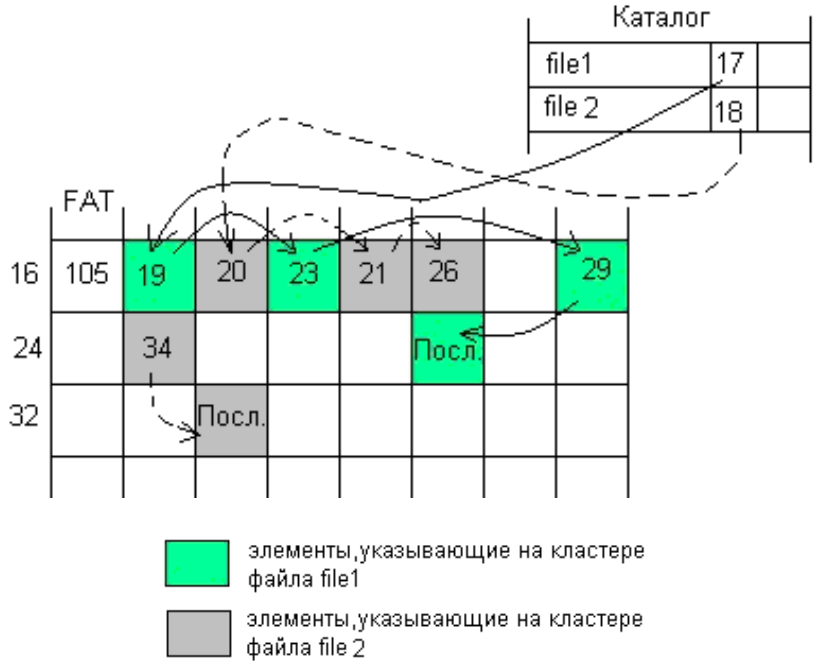


Рис. 7.1. Списки указателей файлов в FAT

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:*

- FAT16 - используются 16-разрядные указатели, что позволяет поддерживать до 65 536 кластеров в области данных диска.
- FAT32, использует 32-разрядные указатели, поддерживая для более чем 4 миллиардов кластеров.

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.

Резервная копия FAT всегда синхронизируется с основной копией при любых операциях с файлами, поэтому резервную копию нельзя использовать 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 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 получили большое распространение благодаря их применению в операционных системах MS-DOS и Windows 3.x — самых массовых операционных системах первого десятилетия эры персональных компьютеров. По этой причине эти файловые системы поддерживаются сегодня и другими ОС, такими как UNIX, OS/2, Windows NT/2000 и Windows. Однако из-за постоянно растущих объемов жестких дисков, а также возрастающих требований к надежности, эти файловые системы быстро вытесняются как системой FAT32, впервые появившейся в Windows 95 OSR2, так и файловыми системами других типов.

Ограничения FAT в Windows

Штатными средствами Windows 2000, Windows XP, Windows Vista и Windows 7 невозможно создать разделы FAT32 более 32 ГБ, однако, с такими разделами возможно работать, если они были предварительно созданы в других ОС. Причина этого заключается в том, что, по мнению Microsoft, при увеличении размера тома FAT32 выше 32 ГБ резко падает производительность, и что более подходящее решение — использование NTFS, то есть родной формат файловой системы для Windows 2000 и Windows XP. Но поскольку NTFS нецелесообразно использовать на флеш-накопителях, то была разработана специальная файловая система exFAT, снимающая ряд ограничений.

Windows XP работает с томами объемом до 2 ТБ (из справки Windows XP).

Максимально возможный размер файла для тома FAT32 — ~ 4 ГБ — 4 294 967 295 байт (232-1 — 4 294 967 295 байт) — это весьма важный фактор для смены файловой системы. FAT32 не поддерживает установку разрешений на доступ к файлам и папкам и некоторые другие функции современных файловых систем. Все эти причины привели к тому, что сейчас наблюдается тенденция отказа от FAT32 в пользу более продвинутых файловых систем, таких как NTFS, Ext2/Ext3.

Файловая система exFAT

exFAT (от англ. *Extended FAT* — «расширенная FAT») — проприетарная файловая система, предназначенная главным образом для флэш-накопителей.

Основными преимуществами exFAT перед предыдущими версиями FAT являются:

- Уменьшение количества перезаписей одного и того же сектора, что очень важно для флеш-накопителей, у которых ячейки памяти необратимо изнашиваются после определённого количества операций записи. Это была основная причина разработки ExFAT.
- Теоретический лимит на размер файла 264 байт (16 экзабайт).
- Максимальный размер кластера увеличен до 225 байт (32 Мбайта).
- Улучшение распределения свободного места за счёт введения бит-карты свободного места, что может уменьшать фрагментацию диска.
- Отсутствие лимита на количество файлов в одной директории.
- Введена поддержка списка прав доступа.
- Поддержка транзакций (опциональная возможность, должна поддерживаться устройством).

7.2 Файловая система NTFS

Файловая система NTFS 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.

Основными отличительными свойствами NTFS являются:

- *поддержка больших файлов и больших дисков объемом до 2 байт;*
- *восстанавливаемость после сбоев и отказов программ и аппаратуры управления дисками;*
- *высокая скорость операций, в том числе и для больших дисков;*
- *низкий уровень фрагментации, в том числе и 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.*

7.2.1 Структура тома NTFS

В отличие от разделов FAT *все пространство* ***тома*** (том - раздел диска в терминологии Windows) *NTFS представляет собой либо файл, либо часть файла.*

Основой структуры тома NTFS является главная таблица файлов ***MFT (Master File Table),*** *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, используемых сегодня, размер записи MFT равен 2 Кбайт, который мы далее будет считать размером записи по умолчанию.

Все файлы на томе NTFS идентифицируются номером файла, который определяется позицией файла в MFT. Весь том NTFS состоит из последовательности кластеров.

Порядковый номер кластера в томе NTFS называется ***логическим номером кластера LCN (Logical Cluster Number)****.*

Файл NTFS также состоит из последовательности кластеров, при этом порядковый номер кластера внутри файла называется ***виртуальным номером кластера VCN (Virtual Cluster Number)****.*

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, 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. 7.2. 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. 7.2. Структура тома NTFS

Далее располагается первый отрезок MFT, содержащий 16 стандартных, создаваемых при форматировании записей о системных файлах NTFS. Назначение этих файлов описано в приведенной ниже таблице MFT.

Таблица 7.1 - Записи о системных файлах NTFS в MFT

Номер записи	**Системный файл**	**Имя файла**	**Назначение файла**
0	Главная таблица файлов	**$Mft**	Содержит полный список файлов тома NTFS
1	Копия главной таблицы файлов	**SMftMirr**	Зеркальная копия первых трех записей MFT
2	Файл журнала	**SLogFile**	Список транзакций, который используется для восстановления файловой системы после сбоев
3	Том	**SVolume**	Имя тома, версия NTFS и другая информация о томе
4	Таблица определения атрибутов	**SAttrDef**	Таблица имен, номеров и описаний атрибутов
5	Индекс корневого каталога	**$.**	Корневой каталог
6	Битовая карта кластеров	**SBitmap**	Разметка использованных кластеров тома
7	Загрузочный сектор раздела	**SBoot**	Адрес загрузочного сектора раздела
8	Файл плохих кластеров	**SBadClus**	Файл, содержащий список всех обнаруженных на томе плохих кластеров
9	Таблица квот	**SQuota**	Квоты используемого пространства на диске для каждого пользователя
10	Таблица преобразования регистра символов	**SUpcase**	Используется для преобразования регистра символов для кодировки Unicode
11-15	Зарезервированы для будущего использования		

В NTFS файл целиком размещается в записи таблицы MFT, если это позволяет сделать его размер. В том же случае, когда размер файла больше размера записи MFT, 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 частью, а остальные части — нерезидентными. Адресная информация об отрезках, содержащих нерезидентные части файла, размещается в атрибутах резидентной части.

7.2.2 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 кроме атрибутов у файла нет никаких других компонентов.

Каждый атрибут файла NTFS состоит из полей: тип атрибута, длина атрибута, значение атрибута и, возможно, имя атрибута. Имеется системный набор атрибутов, определяемых структурой тома NTFS. Системные атрибуты имеют фиксированные имена и коды их типа, а также определенный формат.

Файлы NTFS в зависимости от способа размещения делятся на

- *небольшие,*
- *большие,*
- *очень большие*
- *сверхбольшие.*

Небольшие файлы (small) - *файлы небольшого размера, целиком располагаться внутри одной записи MFT, имеющей, например, размер 2 Кбайт.*

Небольшие файлы NTFS состоят по крайней мере из следующих атрибутов (рис. 7.3)*:*

- *стандартная информация (SI — standard information);*
- *имя файла (FN — file name);*
- *данные (Data);*
- *дескриптор безопасности (SD — security descriptor).*

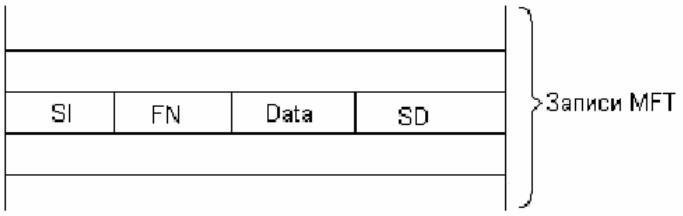


Рис. 7.2. Небольшой файл NTFS

Большие файлы (large). *Если данные файла не помещаются в одну запись MET, 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. 7.3).

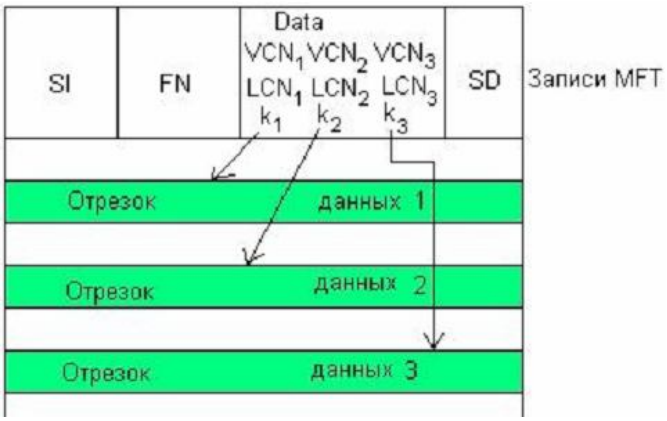


Рис. 7.3. Большой файл

Сверхбольшие файлы (extremely huge). *Для сверхбольших файлов в атрибуте Attribute List можно указать несколько атрибутов, расположенных в дополнительных записях MFT* (рис. 7.4)*. Кроме того, можно использовать двойную косвенную адресацию, когда нерезидентный атрибут будет ссылаться на другие нерезидентные атрибуты, поэтому в NTFS не может быть атрибутов слишком большой для системы длины.*

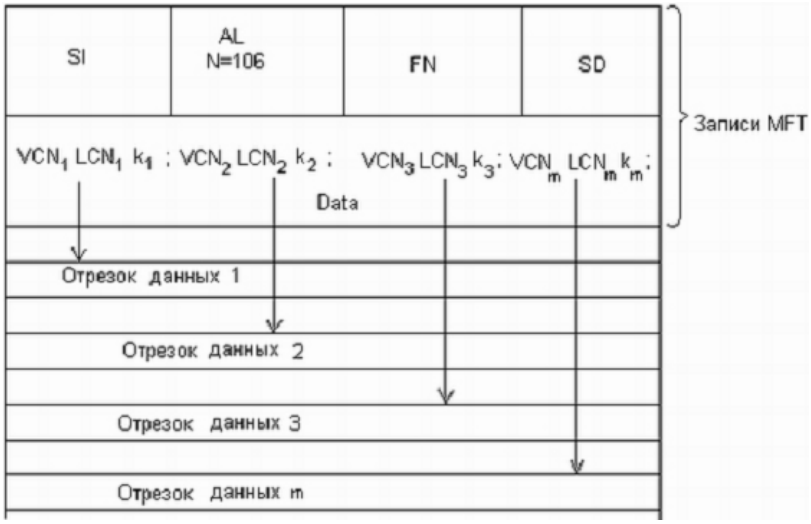


Рис. 7.4. Очень большой файл

Очень большие файлы (huge). *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. 7.5). Эта ссылка содержится в атрибуте Attribute List. Сам атрибут данных по-прежнему содержит адреса нерезидентных отрезков данных.

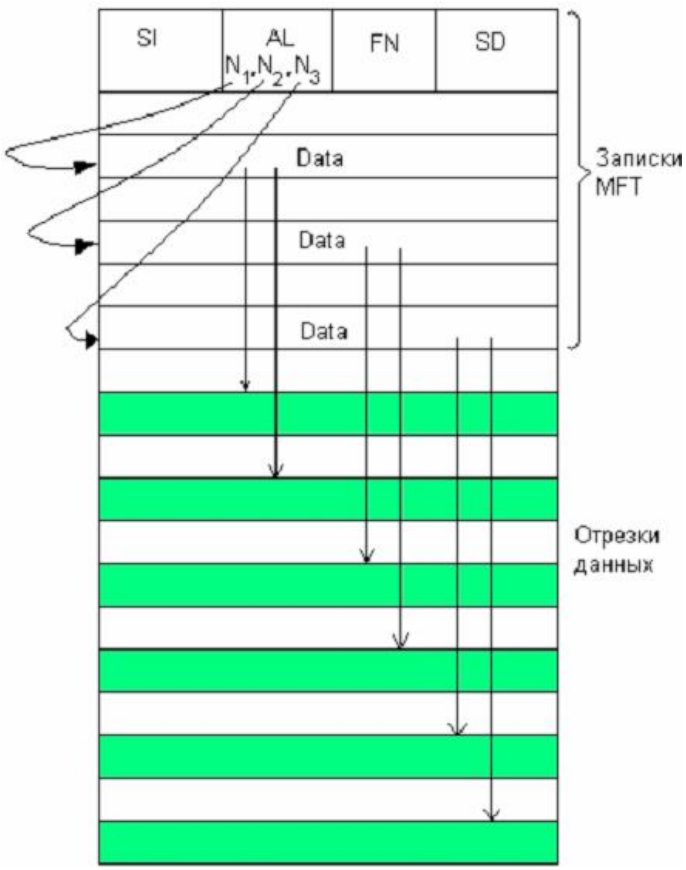


Рис. 7.5. Сверхбольшой файл

7.2.3 Каталоги NTFS

Каталог NTFS *представляет собой один вход в таблицу MFT, который содержит атрибут Index Root. Индекс содержит список файлов, входящих в каталог.* 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.

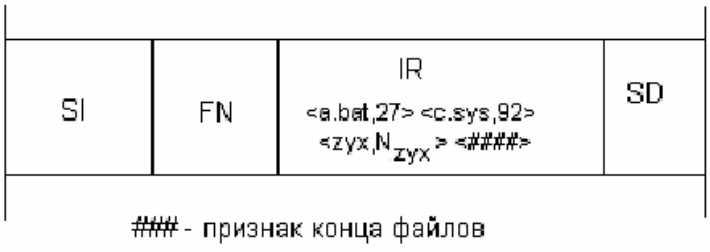


Рис. 7.6 Небольшой каталог

Имеются две формы хранения списка файлов:

– ***Небольшие каталоги (small indexes).*** *Если количество файлов в каталоге невелико, то список файлов может быть резидентным в*

записи в MFT, являющейся каталогом (рис. 7.6). 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 того как каталог растет, список файлов может потребовать нерезидентной формы хранения. Однако начальная часть списка всегда остается резидентной в корневой записи каталога в таблице MFT* (рис. 7.7). 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.

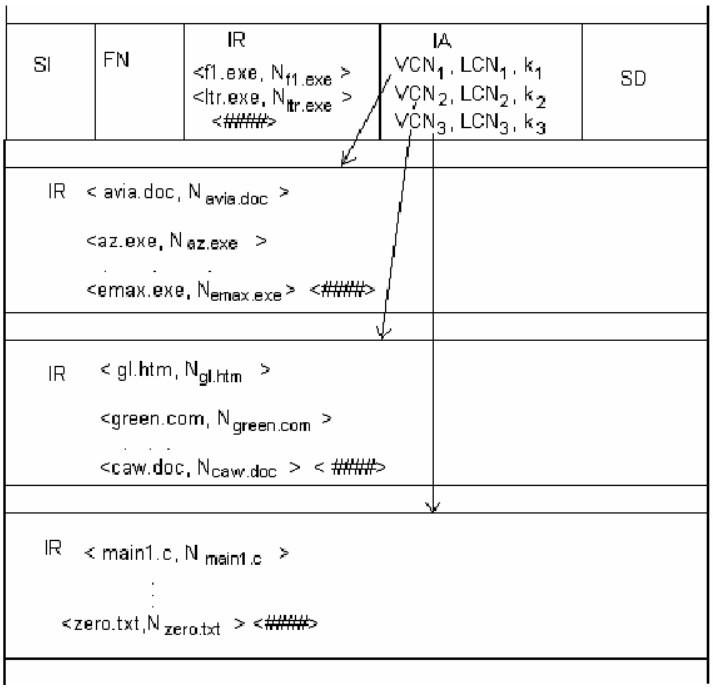


Рис. 7.7. Большой каталог

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, то это означает, что его нужно искать в первой нерезидентной группе, для чего из атрибута Index Allocation извлекается адрес отрезка (VCN_j, LCN_j K_j), хранящего имена файлов первой группы. Среди имен этой группы поиск осуществляется прямым перебором имен 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.

7.3 Файловая система Ext 2/3

На заре развития Linux использовала файловую систему Minix. Эта файловая система была довольно стабильна, но была 16 разрядной и как следствие имела жесткое ограничение в 64 Мегабайта на раздел. Также присутствовало ограничение имени файла: оно составляло 14 символов. Эти и не только ограничения повлекли появление в апреле 1992 года «расширенной файловой системы» (extended file system), решавшей две главные проблемы Minix. Новая файловая система расширила ограничения на размер файла до 2 гигабайт и установила предельную длину имени файла в 255 символов. Но она все равно имела проблемы: не было поддержки раздельного доступа, временных меток модификации данных. Решением всех проблем стала новая файловая система, разработанная в январе 1993 года. В ext2 были сразу реализованы соответствующие стандарту POSIX списки контроля доступа ACL и расширенные атрибуты файлов.

7.3.1 Логическая организация файловой системы ext2

Граф, описывающий иерархию каталогов, файловой системы ext2 представляет собой сеть, это достигается тем, что один файл может входить сразу в несколько каталогов.

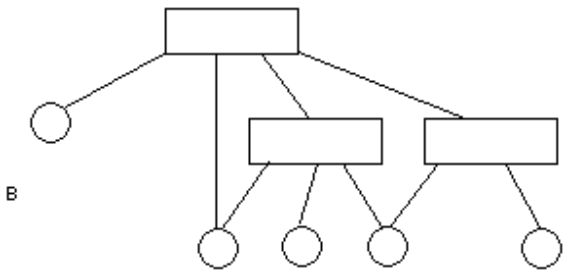


Рис. 7.8. Сетевая иерархия каталогов файловой системы ext2

Все типы файлов имеют символьные имена. Ограничения на простое имя состоят в том что, его длина не должна превышать 255 символов, а также в имени не должны присутствовать символ NUL и ‘/’. Ограничения на символ NUL связаны с представлением строк на языке Си, а на символ ‘/’ с тем, что он используются как разделительный символ между каталогами.

Полное имя представляет собой цепочку простых символьных имен всех каталогов, через которые проходит путь от корня до данного файла. В файловой системе ext2 файл может входить в несколько каталогов, а значит, иметь несколько полных имен; здесь справедливо соответствие

«один файл — много полных имен». В любом случае полное имя однозначно определяет файл.

***Атрибуты файлов** хранятся не в каталогах, как это сделано в ряде простых файловых систем, а в специальных таблицах. В результате **каталог** имеет очень простую структуру, состоящую всего из двух частей:*

- *номера индексного дескриптора*
- *имени файла.*

7.3.2 Структурная организация файловой системы ext2

Как и в любой файловой системе UNIX, ***в составе ext2 можно выделить следующие составляющие:***

- *блоки и группы блоков;*
- *индексный дескриптор;*
- *суперблок.*

Всё пространство раздела диска разбивается на блоки фиксированного размера, кратные размеру сектора — 1024, 2048 и 4096 байт. Размер блока указывается при создании файловой системы на разделе диска. Меньший размер блока позволяет экономить место на жестком диске, но также ограничивает максимальный размер файловой системы. Все блоки имеют порядковые номера. *С целью уменьшения фрагментации и количества перемещений головок жесткого диска при чтении больших массивов данных блоки объединяются в группы блоков.*

*Базовым понятием файловой системы является **индексный дескриптор (информационный узел), information node,** или **inod**e*. Это специальная структура, которая содержит информацию об атрибутах и физическом расположении файла.

Рис. 7.9. Обобщенная структурная схема ФС ext2

Суперблок *— основной элемент файловой системы ext2. Он содержит общую информацию о файловой системе:*

- *общее число блоков и индексных дескрипторов в файловой системе;*
- *число свободных блоков и индексных дескрипторов в файловой системе;*
- *размер блока файловой системы;*
- *количество блоков и индексных дескрипторов в группе;*
- *размер индексного дескриптора;*
- *идентификатор файловой системы.*

От целостности суперблока напрямую зависит работоспособность файловой системы. Операционная система создает несколько резервных копий суперблока для возможности его восстановления в случае повреждения.

Описание группы блоков, представляет собой массив, содержащий общую информацию обо всех блоках раздела.

Битовая карта блоков *— это структура, каждый бит которой показывает, отведен ли соответствующий ему блок какому-либо файлу. Если бит равен 1, то блок занят. Аналогичную функцию выполняет* ***битовая карта индексных дескрипторов****, показывая какие именно индексные дескрипторы заняты, а какие нет.*

Все оставшееся место, обозначенное как ***данные*** *и отводится для хранения файлов.*

7.3.3 Система адресации данных в файловой системе ext2

Система адресации данных *— это одна из самых существенных составных частей файловой системы. Именно система адресации позволяет находить нужный файл среди множества как пустых, так и занятых блоков на диске.*

Файловая система ext2 использует следующую схему адресации блоков файла. Для хранения адреса файла выделено 15 полей, каждое из которых состоит из 4 байт. Если размер файла меньше или равен 12 блоков, то номера этих кластеров непосредственно перечисляются в первых двенадцати полях адреса. Если размер файла превышает 12 блоков, то следующее 13-е поле содержит адрес кластера, в котором могут быть расположены номера следующих блоков файла.

Таким образом, 13-й элемент адреса используется для косвенной адресации. При максимальном размере блока равном 4096 байт, 13-й элемент, может содержать до 1024 номеров следующих кластеров данных файла. Если размер файла превышает 12+1024 блоков, то используется 14-е поле, в котором находится номер блока, содержащего 1024 номеров блоков,

каждый из которых хранят 1024 номеров блоков данных файла. Здесь применяется уже двойная косвенная адресация. И наконец, если файл включает более 12+1024+1048576 = 1049612 блоков, то используется последнее 15-е поле для тройной косвенной адресации.

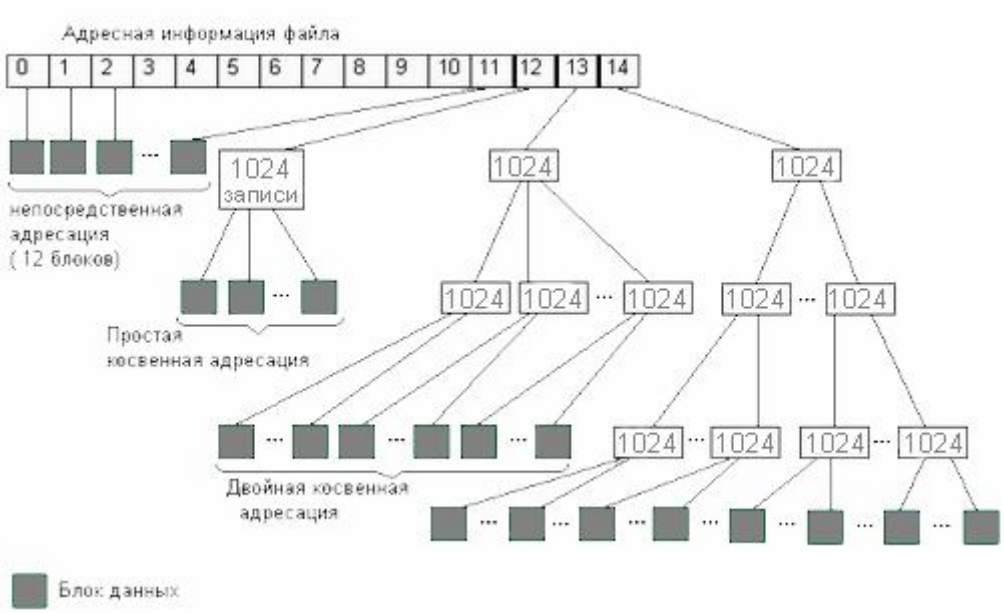


Рис. 7.10. Система адресации ФС ext2

Таким образом, описанная выше система адресации, позволяет при максимальном размере блока 4 Кб иметь файлы размера до 2 терабайт.

7.3.4 Особенности файловой системы ext3

Ext3 *(или 3-я расширенная файловая система) — файловая система, основанная на ext2, включающая в себя элементы журналирования. То есть в ней предусмотрена запись специальных данных в «журнал», что позволяет восстановить файловую систему при сбоях в работе компьютера.*

Стандартом предусмотрено три режима журналирования:

- **writeback**: в журнал записываются только метаданные файловой системы, то есть информация о её изменении. Не может гарантировать целостности данных, однако позволяет обнаружить ошибки ФС, что заметно сокращает время проверки по сравнению с ext2;
- **ordered**: то же, что и writeback, но запись данных в файл производится гарантированно до записи информации о изменении этого файла. Немного снижает производительность, также не может гарантировать целостности данных (хотя и увеличивает вероятность их сохранности при дописывании в конец существующего файла);
- **journal**: полное журналирование как метаданных ФС, так и пользовательских данных. Самый медленный, но и самый безопасный режим; может гарантировать целостность данных при

хранении журнала на отдельном разделе (а лучше — на отдельном жёстком диске).

Файловая система ext3 может поддерживать файлы размером до 1 ТБ. С Linux-ядром 2.4 объем файловой системы ограничен максимальным размер блочного устройства, что составляет 2 терабайта. В Linux 2.6 (для 32-разрядных процессоров) максимальный размер блочных устройств составляет 16 ТБ, однако ext3 поддерживает только до 4 ТБ. (линк)

7.4 Сравнительный анализ файловых систем

В настоящее время все популярные ОС поддерживают максимальное количество файловых систем. Сравнительный анализ распространенных файловых систем приведен в таблице 7.2.

Таблица 7.2 - Сравнительный анализ файловых систем

Параметры	**Тип файловой системы**				
	FAT 32	**NTFS**	**ext 2**	**ext 3**	**Reiser FS**
Создатель	Microsoft	Microsoft	Rémy Card	Stephen Tweedie	Namesys
Дата представления	1996	1993	1993	1999	2001
Родная ОС	Windows 95	Windows NT	GNU/ Linux	GNU/ Linux	GNU/ Linux
Допустимые символы в названиях	Любые символы, кроме NUL * *	Любые символы, кроме NUL, " / \ * ? < > | :	Любые символы, кроме NUL, / *	Любые символы, кроме NUL, / *	Любые символы, кроме NUL, / *
Максимальная длина пути файла	Нет ограни-чений	32 767 симв., каждый эле-мент пути - до 255 симв.	Нет ограни-чений	Нет ограни-чений	Нет ограни-чений
Максималь-ный размер файла	4 ГВ	16 ЕВ	16 ГВ — 2 ТВ	16 ГВ — 2 ТВ	8 ТВ
Максималь-ный размер тома	512 МВ — 8 ТВ	16 ЕВ	2 ТВ — 32 ТВ	2 ТВ — 32 ТВ	16 ТВ
Запись владельца файла	Нет	Да	Да	Да	Да
Создание временных меток	Да	Да	Нет	Нет	Нет
Временные метки доступа/чтения	Да	Да	Да	Да	Да
Контрольные суммы/ECC	Нет	Нет	Нет	Нет	Нет
Жёсткие ссылки	Нет	Да	Да	Да	Да
Мягкие ссылки	Нет	Да	Да	Да	Да
Журналирование блоков	Нет	Нет	Нет	Да	Да
Журналирование метаданных	Нет	Да	Нет	Да	Да
Чувствительно к регистру	Нет	Частично	Да	Да	Да
Лог. изменений файлов	Нет	Да	Нет	Нет	Нет
Прозрачная компрессия	Нет	Да	Нет	Нет	Да

8. СЕТЕВЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

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 — более высокая производительность, отказоустойчивость, масштабируемость и приближение к пользователю.

8.1 Модели сетевых служб и распределенных приложений

Типичным является сетевое приложение, состоящее из двух частей. 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:

1. способ разделения приложения на части, выполняющиеся на разных компьютерах сети;
2. выделение специализированных серверов в сети, на которых выполняются некоторые общие для всех приложений функции;
3. способ взаимодействия между частями приложений, работающих на разных компьютерах.

8.1.1 Способы разделения приложений на части

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 из системы меню, выбор элемента из списка 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 над файлами и файловой системой, которые обычно являются функциями операционной системы.

На основе этой модели можно построить несколько схем распределения частей приложения между компьютерами сети.

8.1.2 Двухзвенные схемы разделения приложений

Наиболее распространенной является двухзвенная схема, распределяющая приложение между двумя компьютерами. Перечисленные выше типовые функциональные части приложения можно разделить между двумя компьютерами различными способами.

Рассмотрим типовые реализации двухзвенной схемы.

В централизованной схеме (рис. 8.1, а) *компьютер пользователя работает как терминал, выполняющий лишь функции представления данных, тогда как все остальные функции передаются центральному компьютеру.* Ресурсы компьютера пользователя используются 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 мыши.

В схеме ***«файловый сервер»*** (рис. 8.1, б) *на клиентской машине выполняются все части приложения, кроме файловых операций. При этом в сети имеется компьютер, который 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 к удаленным файлам вместо локальных. Такая схема обладает хорошей масштабируемостью, так как дополнительные пользователи и приложения добавляют лишь незначительную 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.

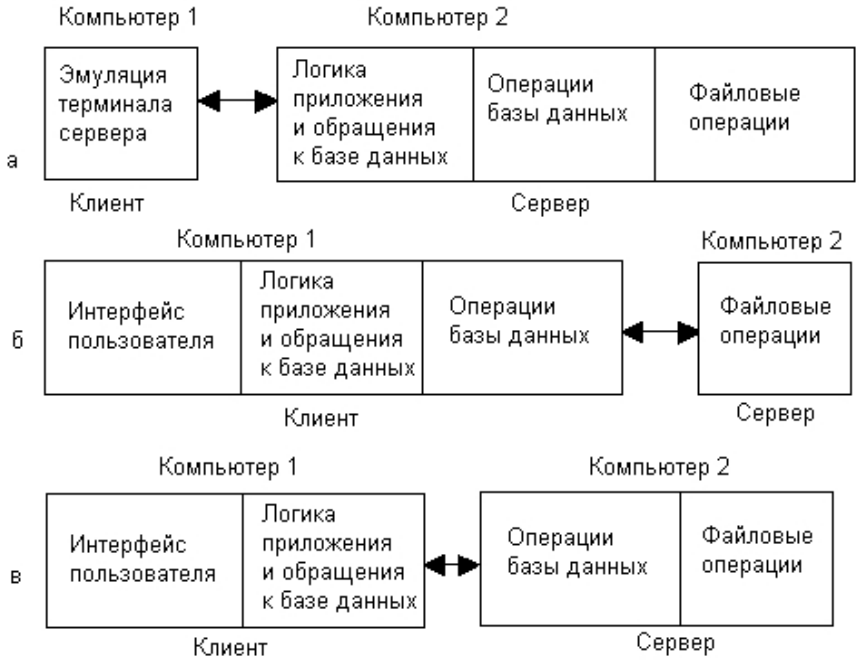


Рис. 8.1. Варианты распределений частей приложения по двухзвенной схеме централизированная (а); «файловый сервер» (б); равномерное распределение функций (в)

Другая часто используется ***схема, в которой на серверный компьютер возлагаются функции проведения внутренних операций базы данных и файловых операций*** (рис. 9.1, 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.

8.1.3 Трехзвенные схемы разделения приложений

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, при которой на клиентской машине выполняются средства представления и логика представления, а также поддерживается программный интерфейс для вызова частей приложения второго звена — промежуточного сервера (рис. 8.2).

Рис. 8.2. Трехзвенная схема распределения частей приложения

Промежуточный сервер *называют в этом варианте* ***сервером приложений****, так как на нем выполняются прикладная логика и логика обработки данных, представляющих собой наиболее специфические и важные части большинства приложений.* ***Слой логики обработки данных*** *вызывает внутренние операции* ***базы данных****, которые реализуются третьим звеном схемы —* ***сервером баз данных****.*

Сервер баз данных, как и в двухзвенной модели, выполняет функции двух последних слоев — операции внутри базы данных и файловые операции.

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 сети можно применить 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.

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 классу middleware — то есть промежуточному слою, располагающемуся между индивидуальной для каждого приложения логикой и сервером баз данных.

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.

8.2 Механизмы передачи сообщений в распределенных системах

Единственным по-настоящему важным отличием распределенных систем от централизованных является способ взаимодействия между процессами.

Принципиально межпроцессное взаимодействие может осуществляться одним из двух способов:

1. *с помощью совместного использования одних и тех же данных (разделяемая память);*
2. *путем передачи друг другу данных в виде сообщений.*

В централизованных системах связь между процессами, как правило, предполагает наличие разделяемой памяти. В этом случае один процесс пишет в разделяемый буфер, а другой читает из него. 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* (как находящихся в пользовательской фазе, так и в системной, то есть выполняющих код операционной системы) *может осуществляться только путем передачи сообщений через сеть. 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 его 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 любой сетевой ОС имеется подсистема передачи сообщений, называемая также ***транспортной подсистемой****, которая обеспечивает набор средств для организации взаимодействия процессов по сети.* ***Назначение этой системы*** *— экранировать детали сложных сетевых протоколов от программиста. Подсистема позволяет процессам взаимодействовать посредством достаточно простых примитивов.*

Транспортная подсистема сетевой ОС имеет обычно сложную структуру, отражающую структуру семиуровневой модели взаимодействия открытых систем (Open System Interconnection, OSI). Представление сложной задачи сетевого взаимодействия компьютеров в виде иерархии нескольких частных задач позволяет организовать это взаимодействие максимально гибким образом. В то же время каждый уровень модели OSI экранирует особенности лежащих под ним уровней от вышележащих уровней, что делает средства взаимодействия компьютеров все более универсальными по мере продвижения вверх по уровням. Таким образом, в процесс выполнения примитивов send и receive вовлекаются средства всех нижележащих коммуникационных протоколов (рис. 8.3).

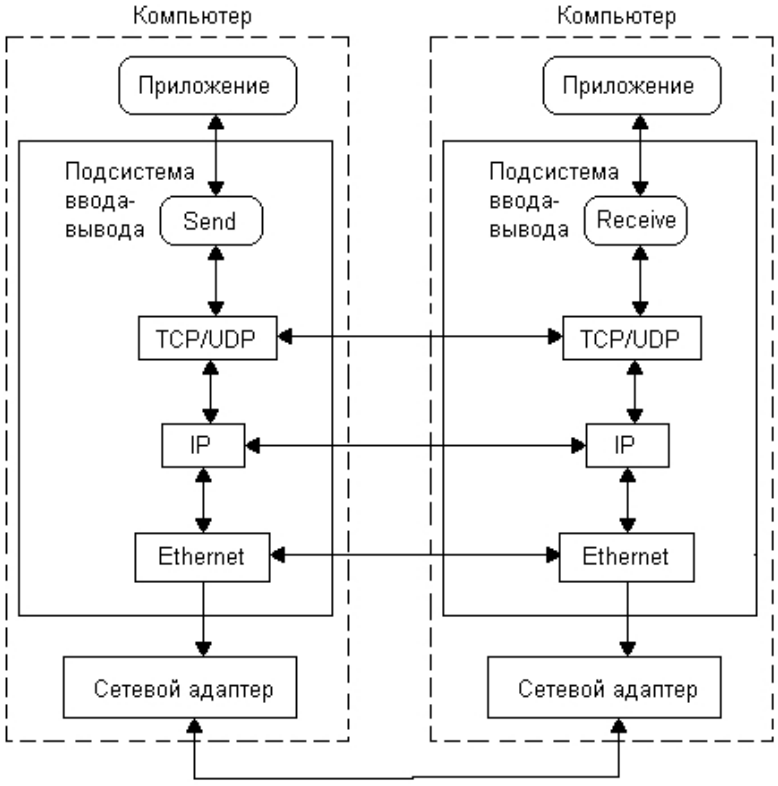


Рис. 8.3. Примитивы обмена сообщениями и транспортные средства подсистемы ввода-вывода

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 и на другие вопросы. В сети всегда имеется один получатель или их может быть несколько? Требуется ли гарантированная доставка сообщений? Должен ли отправитель дождаться ответа на свое сообщение, прежде чем продолжать свою работу? Как отправитель, получатель и подсистема передачи сообщений должны реагировать на отказы узла или коммуникационного канала во время взаимодействия? Что нужно делать, если приемник не готов принять сообщение, нужно ли отбрасывать сообщение или сохранять его в буфере? А если сохранять, 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.

8.3 Синхронизация в распределенных системах

Центральным вопросом взаимодействия процессов в сети является способ их синхронизации, который полностью определяется используемыми в операционной системе коммуникационными примитивами.

В этом отношении коммуникационные примитивы делятся на

- *блокирующие (синхронные),*
- *неблокирующие (асинхронные),*

причем смысл данных терминов в целом соответствует смыслу аналогичных терминов, применяемых при описании системных вызовов и операций ввода- В отличие от локальных системных вызовов при выполнении коммуникационных примитивов завершение запрошенной операции в общем случае зависит не только от некоторой работы локальной ОС, но и от работы удаленной ОС.

8.4 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.

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.

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 вызовов локальных процедур. 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 RPC от локального вызова является то, что он обязательно использует нижележащую систему обмена сообщениями, однако это не должно быть явно видно ни в определении процедур, ни в самих процедурах. Выполнение вызывающей программы и вызываемой локальной процедуры в одной машине реализуется в рамках единого процесса. Но в реализации RPC участвуют как минимум два процесса — по одному в каждой машине.

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 выполняет роль сервера процедур, помещается оригинальный код вызываемой процедуры, а также еще один стаб, называемый* ***серверным стабом*** (рис. 8.4).

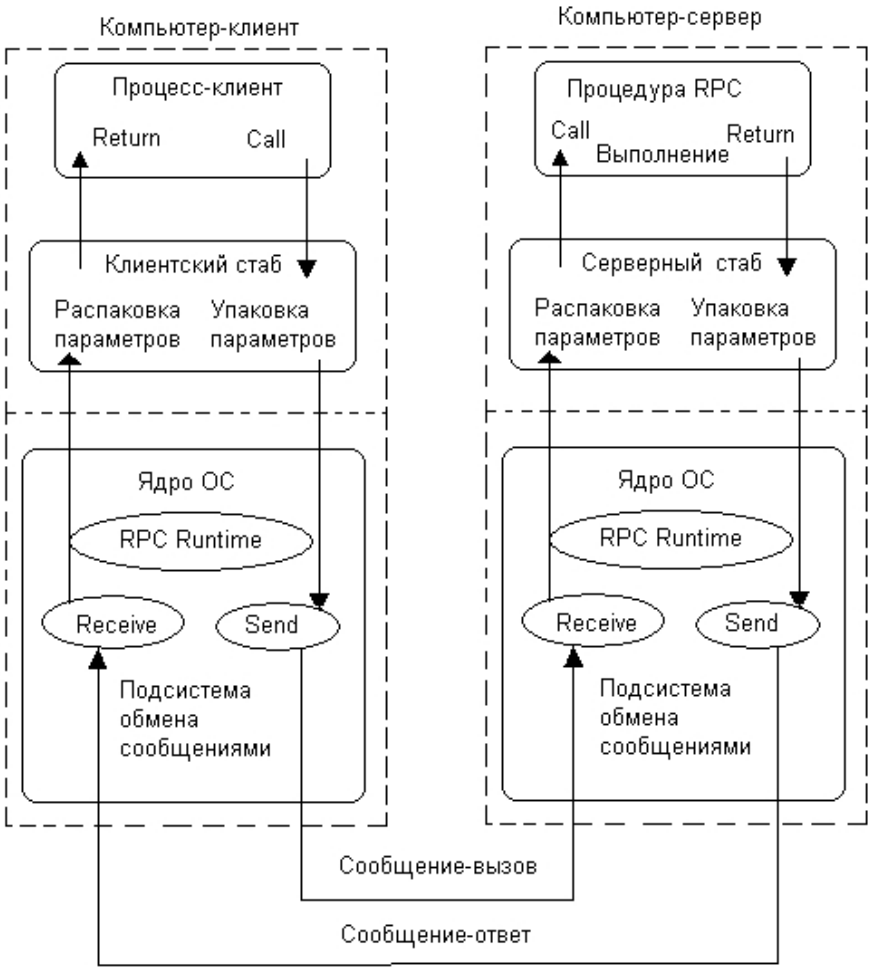


Рис. 8.4. Выполнение удаленного вызова процедуры

Назначение клиентского и серверного стабов - организовать передачу параметров вызываемой процедуры и возврат значения процедуры через сеть, при этом код оригинальной процедуры, помещенной на сервер, должен быть полностью сохранен. Стабы используют для передачи данных через сеть средства подсистемы обмена сообщениями, то есть существующие в ОС примитивы send и receive. *Подобно оригинальной процедуре, клиентский стаб вызывается путем обычной передачи параметров, однако затем вместо выполнения системного вызова, работающего с локальным ресурсом, происходит формирование сообщения, содержащего имя вызываемой процедуры и ее параметры*

9. СЕТЕВЫЕ ФАЙЛОВЫЕ СИСТЕМЫ

9.1 Модель сетевой файловой системы

Ключевым компонентом любой распределенной системы является файловая система, которая также является в этом случае распределенной. В распределенной системе функцией файловой системы является хранение программ и данных и предоставление доступа к ним по мере необходимости. Распределенная файловая система поддерживается одним или более компьютерами, хранящими файлы. Эти компьютеры, которые позволяют пользователям сети получать доступ к своим файлам, обычно называют файловыми серверами. Файловые серверы отрабатывают запросы на чтение или запись файлов, поступающие от других компьютеров сети, которые в этом случае являются клиентами файловой службы. Каждый посланный запрос проверяется и выполняется, а ответ отсылается обратно. Файловые серверы обычно содержат иерархические файловые системы, каждая из которых имеет корневой каталог и каталоги более низких уровней. Во многих сетевых файловых системах клиентский компьютер может подсоединять и монтировать эти файловые системы к своим локальным файловым системам, обеспечивая пользователю удобный доступ к удаленным каталогам и файлам. При этом данные монтируемых файловых систем физически никуда не перемещаются, оставаясь на серверах.

Сетевая файловая система (ФС) в общем случае включает следующие элементы (рис. 10.1):

- *локальная файловая система;*
- *интерфейс локальной файловой системы;*
- *сервер сетевой файловой системы;*
- *клиент сетевой файловой системы;*
- *интерфейс сетевой файловой системы;*
- *протокол клиент-сервер сетевой файловой системы.*

Клиенты сетевой ФС *— это программы, которые работают на 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 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, передать запрос локальной файловой системе для отработки. После получения ответа от локальной файловой системы*

сервер передает его по сети клиенту, а тот, в свою очередь, — приложению, обратившемуся с запросом.

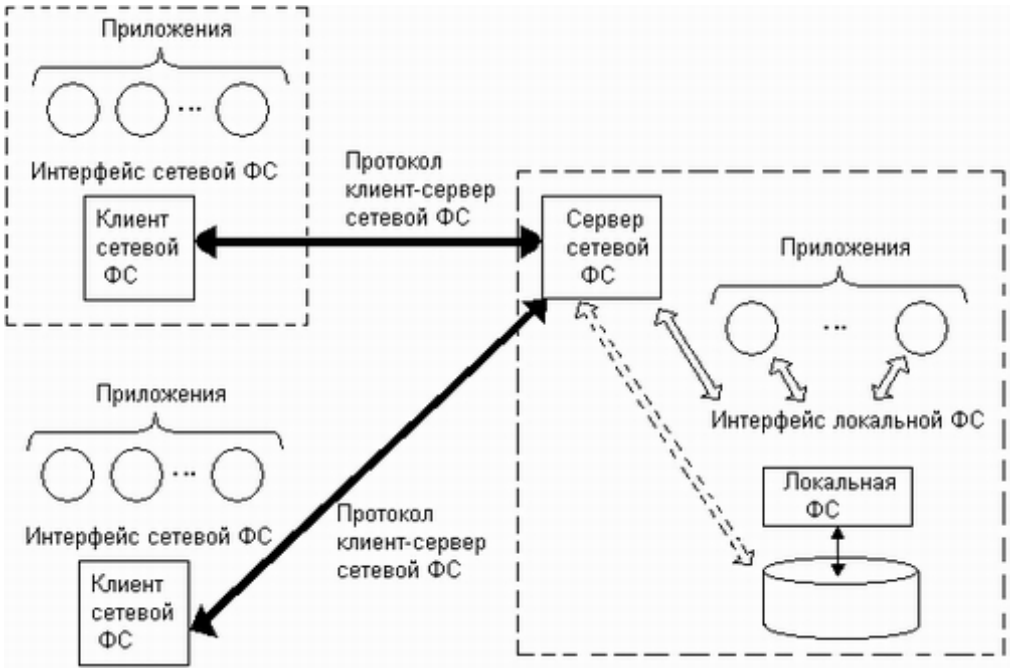


Рис. 9.1. Модель сетевой файловой системы

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.

Клиент и сервер сетевой файловой системы взаимодействуют друг с другом по сети по определенному протоколу. В его функции будет входить ретрансляция серверу запросов, принятых клиентом от приложений, с которыми тот затем будет обращаться к локальной файловой системе. Одним из механизмов, используемых для этой ретрансляции, может быть механизм RPC.

Примером такого протокола является ***SMB (Server Message Block****), разработанный компаниями Microsoft, Intel и IBM работает на прикладном уровне модели OSI. Для передачи по сети своих сообщений протокол SMB использует различные транспортные протоколы. Сообщения SMB могут передаваться и с помощью других протоколов, например TCP/UDP и IPX.*

Для одной и той же локальной файловой системы могут существовать различные протоколы сетевой файловой системы.

Так, к файловой системе NTFS сегодня можно получить доступ с помощью различных протоколов (рис. 9.2), в том числе таких распространенных, как SMB, NCP (NetWare Control Protocol — основной протокол доступа к файлам и принтерам сетевой ОС NetWare компании

Novell) и NFS (Network File System — протокол сетевой файловой системы от компании Sun Microsystems, чрезвычайно популярный в различных вариантах ОС семейства UNIX).

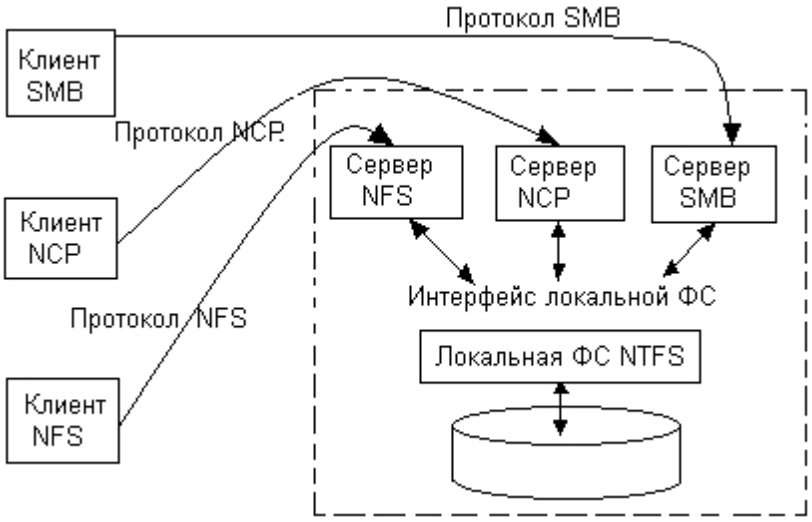


Рис. 9.2. Доступ к одной локальной файловой системе с помощью нескольких протоколов клиент-сервер

С другой стороны, с помощью одного и того же протокола может реализовываться удаленный доступ к локальным файловым системам разного типа.

Например, протокол SMB используется для доступа не только к файловой системе FAT, но и к файловым системам NTFS и HPFS (рис. 9.3). Эти файловые системы могут располагаться как на разных, так и на одном компьютере.

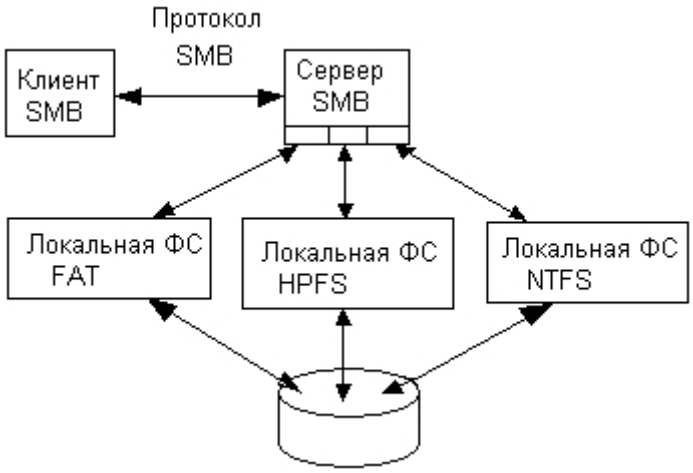


Рис. 9.3. Доступ к локальным файловым системам различного типа с помощью одного протокола клиент-сервер

За достаточно долгий срок развития сетей в них утвердилось несколько сетевых файловых систем. В среде операционной системы UNIX наибольшее распространение получили две сетевые файловые системы — FTP (File Transfer Protocol) и NFS (Network File System). Они первоначально разрабатывались для доступа к локальной файловой системе ОС семейства UNIX. Со временем в крупных сетях стали одновременно применяться несколько сетевых файловых систем разных типов, например NetWare и SMB или NetWare и NFS. Это часто происходило при объединении нескольких сетей в одну.

На рис. 9.3 показан *вариант организации неоднородной сетевой файловой системы, в которой на компьютере с локальной файловой системой NTFS работает несколько файловых серверов, поддерживающих различные протоколы клиент-сервер.*

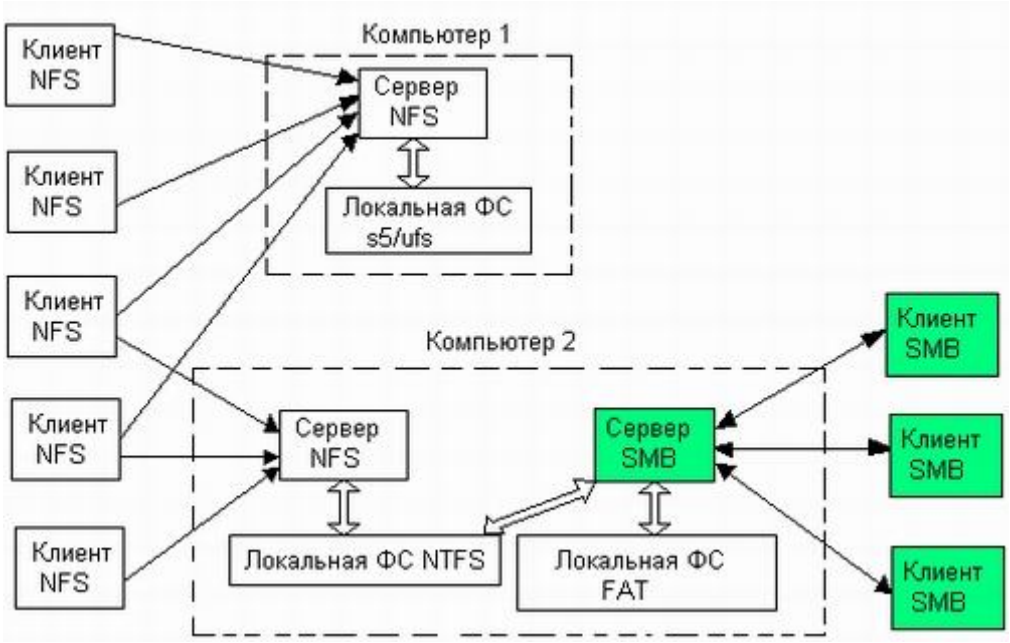


Рис. 9.4. Неоднородная сетевая файловая система

9.2 Интерфейс сетевой файловой службы

Структура файла. Во многих системах, таких как UNIX и Windows, ***файл*** *— это не интерпретируемая последовательность байтов.* Значение и структура информации в файле является заботой прикладных программ, операционную систему это не интересует.

Модифицируемость файлов - *возможность модификации файла после его создания. В большинстве сетевых файловых систем файлы могут модифицироваться, но в некоторых распределенных системах единственными операциями с файлами являются create (создать) и read (прочитать).* Такие файлы называются неизменяемыми. Для неизменяемых файлов намного легче осуществить кэширование файла и его репликацию

(тиражирование), так как исключаются все проблемы, связанные с обновлением всех копий файла при его изменении.

***Семантика разделения файлов**. порядок чтения и записи в случае когда два или более пользователей разделяют один файл, чтобы избежать проблем с интерпретацией результирующих данных файла.*

Различают следующие семантики разделения файлов:

- ***Семантика UNIX.** В операционных системах UNIX, обычно определяется, что когда операция чтения следует за операцией записи, то читается только что обновленный файл. Аналогично, 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.*
- ***Сеансовая семантика.** В соответствии с этой моделью изменения в открытом файле сначала видны только процессу, который модифицирует файл, и только после закрытия файла эти изменения могут видеть другие процессы. При использовании сеансовой семантики возникает проблема одновременного использования одного и того же файла двумя или более клиентами.*
- ***Семантика неизменяемых 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).*
- ***Транзакционная семантика**, т. е. использование механизма неделимых транзакций.*

***Контроль доступа** С каждым разделяемым файлом обычно связан список управления доступом (Access Control List, ACL), обеспечивающий защиту данных от несанкционированного доступа.* 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.

***Единица доступа**. Файловый интерфейс может быть отнесен к одному из двух типов в зависимости от того, поддерживает ли он модель загрузки-выгрузки или модель удаленного доступа.*

*В **модели загрузки-выгрузки** пользователю предлагаются средства чтения или записи файла целиком. Эта модель предполагает следующую схему обработки файла:*

- *чтение файла с сервера на машину клиента,*

– *обработка файла на машине клиента,*
– *запись обновленного файла на сервер.*

***Модель удаленного доступа**, которая предполагает поддержку большого количества операций над файлами:*

– открытие и закрытие файлов,
– чтение и запись частей файла,
– позиционирование в файле,
– проверка и изменение атрибутов файла и т. д.

В то время как в модели загрузки-выгрузки файловый сервер обеспечивал только хранение и перемещение файлов, *в случае модели удаленного доступа все файловые операции выполняются на серверах, а клиенты только генерируют запросы на их отработку.*

9.3 Размещение клиентов и серверов по компьютерам и в операционной системе

При организации в сети взаимодействия между ОС по концепции клиент-сервер, имеются различные способы распределения серверной и клиентской частей между компьютерами.

1. *В некоторых файловых системах* (например, NFS или файловых системах Windows 95/98/NT/2000) *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.
2. *В некоторых случаях выпускается так называемая серверная версия ОС* (например, Windows NT Server), *которая использует то же программное обеспечение файловой службы, но только позволяющее* (за счет выделения файловому серверу большего количества ресурсов (в основном оперативной памяти) *обслуживать одновременно большее число пользователей, чем версии файлового сервера для клиентских компьютеров.*
3. *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 являются модулями ядра ОС, работающими в привилегированном режиме. *В современных ОС эти компоненты оформляются как*

высокоуровневые драйверы, работающие в составе подсистемы ввода-вывода.

9.4 Кэширование данных

Кэширование данных в оперативной памяти, может существенно повысить скорость доступа к файлам, хранящимся на дисках. *Кэширование широко используется в сетевых файловых системах, где оно позволяет не только повысить скорость доступа к удаленным данным (это по-прежнему является основной целью кэширования), но и улучшить масштабируемость и 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: модель загрузки-выгрузки, переносящая файл 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, имеются три различных места для хранения кэшируемых файлов и их частей:*

- *память сервера,*
- *диск клиента (если имеется),*
- *память клиента.*

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.

Для решения этой проблемы необходимо, чтобы модификации данных, выполненные над одной из копий, были своевременно распространены на все остальные копии. Существует несколько вариантов распространения модификаций:

- ***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.* Данный вариант распространения модификаций обеспечивает семантику разделения файлов в стиле UNIX.

– ***принятие сеансовой семантики****, в соответствии с которой запись файла на сервер производится только после закрытия файла.* Этот алгоритм называется «запись по закрытию» и приводит к тому, что если две копии одного файла кэшируются на разных машинах и последовательно записываются на сервер, то второй записывается поверх первого.

Проверка достоверности кэша*.* Распространение модификаций решает только проблему согласования главной копии файла, хранящейся на сервере, с клиентскими копиями. 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 проверять, являются ли данные в кэше клиента достоверными. В противном случае данные кэша должны быть повторно считаны с сервера. Существуют два подхода к решению этой проблемы*.

1. ***Инициирование проверки клиентом****, в этом случае клиент 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.*** Этот способ дискредитирует саму идею кэширования, так как каждое обращение к файлу вызывает обмен по сети с сервером. Но зато это обеспечивает семантику разделения файлов UNIX.
   – ***Периодические проверки.*** Улучшают производительность, но делают семантику разделения неясной, зависящей от временных соотношений.
   – ***Проверка при открытии файла.*** Этот способ подходит для сеансовой семантики. Необходимо отметить, что сеансовая семантика требует одновременного применения модели доступа загрузки-выгрузки, метода распространения модификаций «запись по закрытию» и проверки достоверности кэша при открытии файла.
2. ***Инициирование проверки сервером.*** *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 данную информацию и использует ее при доступе к файлу других клиентов.*

   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.

9.5 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) *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.*

Имеется две главные причины для применения репликации:

1. *Увеличение надежности за счет наличия независимых копий каждого файла, хранящихся на разных файловых серверах.* При отказе одного из них файл остается доступным.
2. *Распределение 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.

Репликация похожа на кэширование файлов на стороне клиентов тем, что в системе создается несколько копий одного файла. Однако существуют и принципиальные отличия - 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.

9.6 Примеры сетевых файловых служб: FTP и NFS

9.6.1 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. Серверы и клиенты FTP имеются практически в каждой ОС семейства UNIX, а также во многих других сетевых ОС. Клиенты FTP 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 *(File Transfer Protocol) 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 особенно 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.

Клиент FTP состоит из нескольких функциональных модулей:

- User Interface — пользовательский интерфейс, принимающий от пользователя символьные команды и отображающий состояние сеанса FTP на символьном экране.
- User-Pi — интерпретатор команд пользователя. Этот модуль взаимодействует с соответствующим модулем сервера FTP.
- 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-PL
- Server-DTP — модуль, управляющий передачей данных файла по командам от модуля Server-PL Взаимодействует с локальной файловой системой сервера.

Клиент и сервер FTP 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 по этапам приведена ниже.

1. Сервер FTP всегда открывает управляющий порт TCP 21 для прослушивания, ожидая приход запроса на установление управляющего сеанса FTP 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 клиент FTP (по умолчанию клиент может использовать для передачи данных порт управляющего сеанса), а сервер должен использовать порт, на единицу меньший порта клиента.

9.6.2 Файловая система NFS

Файловая система NFS (Network File System) создана компанией Sun Microsystems. В настоящее время это стандартная сетевая файловая система для ОС семейства UNIX, кроме того, клиенты и серверы NFS реализованы для многих других ОС. Принципы ее организации на сегодня стандартизованы сообществом Интернета, последняя версия NFS v. 4 описывается спецификацией, выпущенной в декабре 2000 года.

Файловая система NFS *(Network File System) представляет собой систему, поддерживающую схему удаленного доступа к файлам. Работа пользователя с удаленными файлами производится после выполнения операции монтирования становится полностью прозрачной — поддерево*

файловой системы сервера NFS становится поддеревом локальной файловой системы.

Одной из целей разработчиков NFS была поддержка неоднородных систем с клиентами и серверами, работающими под управлением различных ОС на различной аппаратной платформе.

Для обеспечения устойчивости клиентов к отказам серверов в NFS принят подход stateless, то есть серверы при работе с файлами не хранят данных об открытых клиентами файлах.

Основная идея NFS *— позволить произвольной группе пользователей разделять общую файловую систему.* Чаще всего все пользователи принадлежат одной локальной сети, но не обязательно. Можно выполнять NFS и в глобальной сети.

Каждый NFS-сервер предоставляет один или более своих каталогов для доступа удаленным клиентам. Каталог объявляется доступным со всеми своими подкаталогами. Клиенты получают доступ к экспортируемым каталогам путем монтирования. *Многие бездисковые рабочие станции, могут монтировать удаленную файловую систему к корневому каталогу, при этом вся файловая система целиком располагается на сервере.* 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.

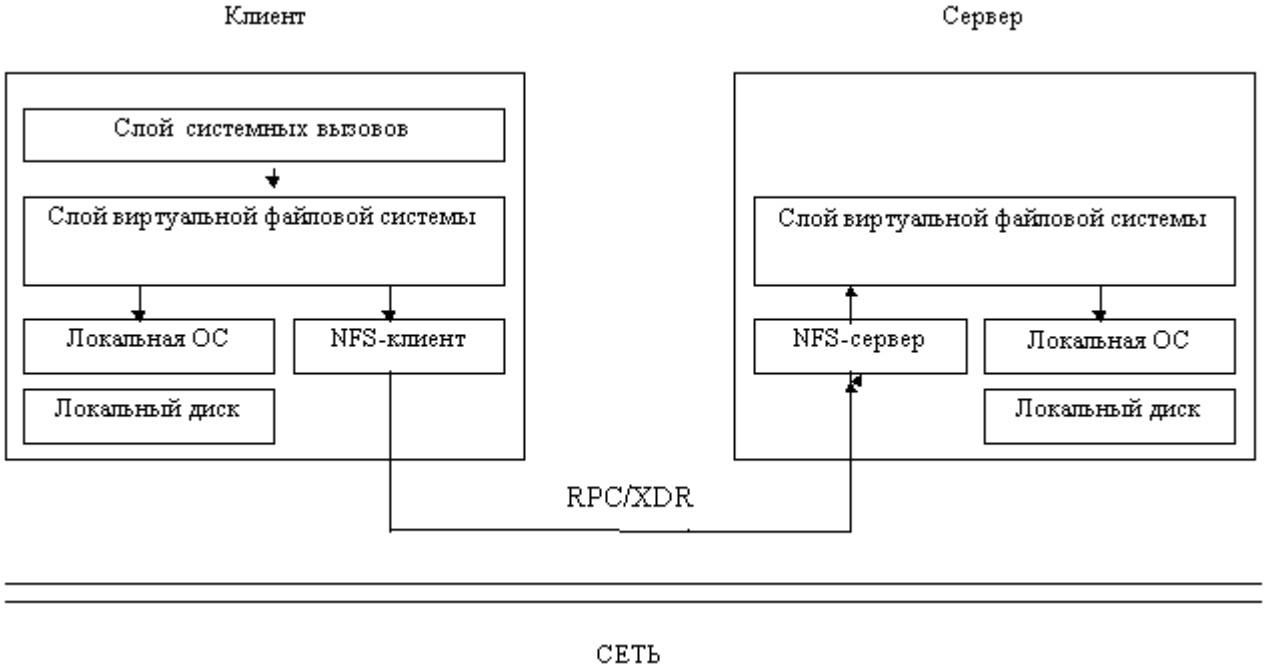


Рис. 9.5. Реализация NFS

В своей работе файловая система NFS использует два протокола.

Первый NFS-протокол управляет монтированием. Клиент посылает серверу полное имя каталога и запрашивает разрешение на монтирование этого каталога в какую-либо точку собственного дерева каталогов. Получив имя, сервер проверяет законность этого запроса и возвращает клиенту дескриптор файла, являющегося удаленной точкой монтирования.

Второй NFS-протокол используется для доступа к удаленным файлам и каталогам. Клиенты могут послать запрос серверу для выполнения 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 используется кэширование на стороне клиента, данные в кэш переносятся поблочно и применяется упреждающее чтение, при котором чтение блока в кэш по требованию приложения всегда сопровождается чтением следующего блока по инициативе системы. 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, файл удаляется из кэша и от сервера получается новая копия 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 свежими.

Репликация в NFS не поддерживается.

9.7 Служба каталогов

Подобно большой организации, большая компьютерная сеть нуждается в централизованном хранении как можно более полной справочной информации о самой себе. Решение многих задач в сети опирается на информацию о пользователях сети — их именах, используемых для логического входа в систему, паролях, правах доступа к ресурсам сети, а также о ресурсах и компонентах сети: серверах, клиентских компьютерах, маршрутизаторах, шлюзах, томах файловых систем, принтерах и т. п.

Примеры наиболее важных задач централизованной базы справочной информации сети:

- ***Аутентификация пользователей**, на основе которой затем выполняется авторизация доступа.* В сети должны каким-то образом централизованно храниться учетные записи пользователей, содержащие имена и пароли.
- ***Поддержка прозрачности доступа к сетевым ресурсам**. В такой базе должны храниться имена этих ресурсов и отображения имен*

на числовые идентификаторы (например, IP-адреса), позволяющие найти этот ресурс в сети. Прозрачность может обеспечиваться при доступе к серверам, томам файловой системы, интерфейсам процедур RPC, программным объектам распределенных приложений и многим другим сетевым ресурсам.

- ***Электронная почта с единой для сети справочной службой***, хранящей данные о почтовых именах пользователей.
- ***Средства управления качеством обслуживания трафика*** (Quality of Service, QoS), *которые также требуют наличия сведений обо всех пользователях и приложениях системы, их требованиях к параметрам качества обслуживания трафика, а также обо всех сетевых устройствах, с помощью которых можно управлять трафиком* (маршрутизаторах, коммутаторах, шлюзах и т. п.).
- ***Организация распределенных приложений*** может существенно упроститься, если в сети имеется *база, 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 управления сетью*** *должна располагать базой для хранения информации о топологии сети и характеристиках всех сетевых элементов, таких как маршрутизаторы, коммутаторы, серверы и клиентские компьютеры*. Наличие полной информации о составе сети и ее связях позволяет системе автоматизированного управления сетью правильно идентифицировать сообщения об аварийных событиях и находить их первопричину.

Результатом развития систем хранения справочной информации стало появление в сетевых операционных системах специальной службы — так называемой службы каталогов (Directory Services), называемой также справочной службой (directory — справочник, каталог).

Служба каталогов (DS - Directory Services) *хранит информацию обо всех пользователях и ресурсах сети в виде унифицированных объектов с определенными атрибутами, а также позволяет отражать взаимосвязи между хранимыми объектами*, такие как принадлежность пользователей к определенной группе, права доступа пользователей к компьютерам, вхождение нескольких узлов в одну подсеть, коммуникационные связи между подсетями, производственную принадлежность серверов и т. д.

Служба каталогов позволяет *выполнять над хранимыми объектами набор некоторых базовых операций*, таких как добавление и удаление объекта, включение объекта в другой объект, изменение значений атрибута объекта, чтение атрибутов и некоторые другие.

Обычно над службой каталогов строятся различные специфические сетевые приложения, которые используют информацию службы для решения конкретных задач: управления сетью, аутентификации пользователей, обеспечения прозрачности служб и других, перечисленных выше.

Служба каталогов обычно строится на основе модели клиент-сервер: серверы хранят базу справочной информации, которой пользуются клиенты, передавая серверам по сети соответствующие запросы. Для клиента службы каталогов она представляется единой централизованной системой, хотя большинство хороших служб каталогов имеют распределенную структуру, включающую большое количество серверов, но эта структура для клиентов прозрачна.

Проблемы сохранения производительности и надежности при увеличении масштаба сети обычно решаются за счет распределенных баз данных справочной информации. Разделение данных между несколькими серверами снижает нагрузку на каждый сервер, а надежность при этом достигается за счет наличия нескольких реплик каждой части базы данных.

10. СОВРЕМЕННЫЕ КОНЦЕПЦИИ И ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ОПЕРАЦИОННЫХ СИСТЕМ

10.1 Требования предъявляемые к современной операционной системе

Операционная система является сердцевиной сетевого программного обеспечения, она создает среду для выполнения приложений и во многом определяет, какими полезными для пользователя свойствами эти приложения будут обладать. В связи с этим рассмотрим требования, которым должна удовлетворять современная ОС.

Очевидно, что ***главным требованием, предъявляемым к операционной системе, является*** *способность выполнения основных функций: эффективного управления ресурсами и обеспечения удобного интерфейса для пользователя и прикладных программ.*

Современная ОС, как правило, должна реализовывать:

- *мультипрограммную обработку,*
- *виртуальную память, свопинг,*
- *поддерживать многооконный интерфейс,*
- *и выполнять многие другие, совершенно необходимые функции.*

Кроме этих функциональных требований к операционным системам предъявляются не менее важные требованиям:

- ***Расширяемость.*** *Код должен быть написан таким образом, чтобы можно было легко внести дополнения и изменения, если это потребуется, и не нарушить целостность системы.*
- ***Переносимость.*** *Код должен легко переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которая включает наряду с типом процессора и способ организации всей аппаратуры компьютера) одного типа на аппаратную платформу другого типа.*
- ***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 в состоянии наносить вред ОС.*
- ***Совместимость.*** *ОС должна иметь средства для выполнения прикладных программ, написанных для других операционных систем. Кроме того, пользовательский интерфейс должен быть совместим с существующими системами и стандартами.*
- ***Безопасность. ОС*** *должна обладать средствами защиты ресурсов одних пользователей от других.*

– ***Производительность***. *Система должна обладать настолько хорошим быстродействием и временем реакции, насколько это позволяет аппаратная платформа.*

Рассмотрим более подробно некоторые из этих требований.

10.1.1 Требования по расширяемости

Операционные системы всегда эволюционно изменяются со временем, и эти изменения более значимы, чем изменения аппаратных средств. Изменения ОС обычно представляют собой приобретение ею новых свойств. Например, поддержка новых устройств, таких как CD-ROM, возможность связи с сетями нового типа, поддержка многообещающих технологий, таких как графический интерфейс пользователя или объектно-ориентированное программное окружение, использование более чем одного процессора. Сохранение целостности кода, какие бы изменения не вносились в операционную систему, является главной целью разработки.

Расширяемость достигаться путем использования следующих подходов:

1. ***За счет модульной структуры ОС****, при которой программы строятся из набора отдельных модулей, взаимодействующих только через функциональный интерфейс. Новые компоненты могут быть добавлены в операционную систему модульным путем, они выполняют свою работу, используя интерфейсы, поддерживаемые существующими компонентами.*
2. ***Использование объектов*** *для представления системных ресурсов также улучшает расширяемость системы.*

   Объекты *- это абстрактные типы данных, над которыми можно производить только те действия, которые предусмотрены специальным набором объектных функций.* Объекты позволяют единообразно управлять системными ресурсами. Добавление новых объектов не разрушает существующие объекты и не требует изменений существующего кода.
3. *Прекрасные возможности для расширения предоставляет* ***подход к структурированию ОС по типу клиент-сервер*** *с использованием микроядерной технологии. В соответствии с этим подходом ОС строится как совокупность привилегированной управляющей программы и набора непривилегированных услуг-серверов.* Основная часть ОС может оставаться неизменной в то время, как могут быть добавлены новые серверы или улучшены старые.
4. ***Средства вызова удаленных процедур (RPC)*** *позволяют добавить новые программные процедуры в любую машину сети и немедленно*

поступить в распоряжение прикладных программ на других машинах сети.

5. ***Загружаемые драйверы**, которые могут быть добавлены в систему во время ее работы.* Новые файловые системы, устройства и сети могут поддерживаться путем написания драйвера устройства, драйвера файловой системы или транспортного драйвера и загрузки его в систему.

10.1.2 Требования по переносимости

Требование переносимости кода тесно связано с расширяемостью. *Расширяемость позволяет улучшать операционную систему, в то время как переносимость дает возможность перемещать всю систему на машину, базирующуюся на другом процессоре или аппаратной платформе, делая при этом по возможности небольшие изменения в коде.*

Написание переносимой ОС аналогично написанию любого переносимого кода - нужно следовать некоторым правилам.

- *Во-первых, большая часть кода должна быть написана на языке, который имеется на всех машинах, куда вы хотите переносить систему.* Обычно это означает, что код должен быть написан на языке высокого уровня, предпочтительно стандартизованном, например, на языке С. Программа, написанная на ассемблере, не является переносимой, если только вы не собираетесь переносить ее на машину, обладающую командной совместимостью с вашей.
- *Во-вторых, следует учесть, в какое физическое окружение программа должна быть перенесена. Различная аппаратура требует различных решений при создании ОС.* Например, ОС, построенная на 32-битовых адресах, не может быть перенесена на машину с 16-битовыми адресами (разве что с огромными трудностями).
- *В-третьих, важно минимизировать или, если возможно, исключить те части кода, которые непосредственно взаимодействуют с аппаратными средствами.* Некоторые очевидные формы зависимости включают прямое манипулирование регистрами и другими аппаратными средствами.
- *В-четвертых, если аппаратно зависимый код не может быть полностью исключен, то он должен быть изолирован в нескольких хорошо локализуемых модулях. Аппаратно-зависимый код не должен быть распределен по всей системе.* Например, можно спрятать аппаратно-зависимую структуру в программно-задаваемые данные абстрактного типа. Другие модули системы будут работать с этими данными, а не с аппаратурой, используя набор некоторых

функций. Когда ОС переносится, то изменяются только эти данные и функции, которые ими манипулируют.

Для легкого переноса ОС при ее разработке должны быть соблюдены нижеуказанные требования.

- ***Переносимый язык высокого уровня.*** *Большинство переносимых ОС написано на языке С* (стандарт ANSI X3.159-1989). *Ассемблер используется только для тех частей системы, которые должны непосредственно взаимодействовать с аппаратурой* (например, обработчик прерываний). Однако непереносимый код должен быть тщательно изолирован внутри тех компонентов, где он используется.
- ***Изоляция процессора.*** *Некоторые низкоуровневые части ОС должны иметь доступ к процессорно-зависимым структурам данных и регистрам.* Однако код, который делает это, должен содержаться в небольших модулях, которые могут быть заменены аналогичными модулями для других процессоров.
- ***Изоляция платформы.*** *Зависимость от платформы заключается в различиях между рабочими станциями разных производителей, построенными на одном и том же процессоре*. Должен быть введен программный уровень, абстрагирующий аппаратуру (кэши, контроллеры прерываний ввода-вывода и т. п.) вместе со слоем низкоуровневых программ таким образом, чтобы высокоуровневый код не нуждался в изменении при переносе с одной платформы на другую.

10.1.3 Требования по совместимости

Совместимость *- способность ОС выполнять программы, написанные для других ОС или для более ранних версий данной операционной системы, а также для другой аппаратной платформы.*

Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений.

Двоичная совместимость *достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение на другой ОС. Для этого необходимы: совместимость на уровне команд процессора, совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.*

Совместимость на уровне исходных текстов *требует наличия соответствующего компилятора в составе программного обеспечения, а также совместимости на уровне библиотек и системных вызовов.* При этом

необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль.

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

Гораздо сложнее достичь двоичной совместимости между процессорами, основанными на разных архитектурах. Для того, чтобы один компьютер выполнял программы другого (например, DOS-программу на Mac), *этот компьютер должен работать с машинными командами, которые ему изначально непонятны.* Выходом в таких случаях является использование так называемых прикладных сред. Учитывая, что основную часть программы, как правило, составляют вызовы библиотечных функций, прикладная среда имитирует библиотечные функции целиком, используя заранее написанную библиотеку функций аналогичного назначения, а остальные команды эмулирует каждую по отдельности.

Соответствие стандартам POSIX также является средством обеспечения совместимости программных и пользовательских интерфейсов. Использование стандарта POSIX (IEEE стандарт 1003.1 - 1988) позволяет создавать программы стиле UNIX, которые могут легко переноситься из одной системы в другую.

10.1.4 Требования по безопасности

Правила безопасности определяют такие свойства, как защита ресурсов одного пользователя от других и установление квот по ресурсам для предотвращения захвата одним пользователем всех системных ресурсов (таких как память).

Обеспечение защиты информации от несанкционированного доступа является обязательной функцией сетевых операционных систем.

Основы стандартов в области безопасности были заложены "*Критериями оценки надежных компьютерных систем*". Этот документ, изданный в США в 1983 году национальным центром компьютерной безопасности (NCSC - National Computer Security Center), часто называют **Оранжевой Книгой**.

В соответствии с требованиями Оранжевой книги ***безопасной считается такая система****, которая "посредством специальных механизмов защиты контролирует доступ к информации таким образом, что только имеющие соответствующие полномочия лица или процессы,*

выполняющиеся от их имени, могут получить доступ на чтение, запись, создание или удаление информации".

Иерархия уровней безопасности, приведенная в Оранжевой Книге, помечает низший уровень безопасности как D, а высший - как A.

1. *В* ***класс D*** *попадают системы, оценка которых выявила их несоответствие требованиям всех других классов.*
2. *Основными свойствами, характерными для* ***C-систем****, являются: наличие подсистемы учета событий, связанных с безопасностью, и избирательный контроль доступа.*

Уровень **C** делится на 2 подуровня:

- ***уровень C1****, обеспечивающий защиту данных от ошибок пользователей, но не от действий злоумышленников.*
- *На* ***уровне C2*** *должны присутствовать*
  - *средства секретного входа,* обеспечивающие идентификацию пользователей путем ввода уникального имени и пароля перед тем, как им будет разрешен доступ к системе.
  - *Избирательный контроль доступа*, требуемый на этом уровне позволяет владельцу ресурса определить, кто имеет доступ к ресурсу и что он может с ним делать. Владелец делает это путем предоставляемых прав доступа пользователю или группе пользователей.
  - *Средства учета и наблюдения* (*auditing*) - обеспечивают возможность обнаружить и зафиксировать важные события, связанные с безопасностью, или любые попытки создать, получить доступ или удалить системные ресурсы.
  - *Защита памяти* - заключается в том, что память инициализируется перед тем, как повторно используется. На этом уровне система не защищена от ошибок пользователя, но поведение его может быть проконтролировано по записям в журнале, оставленным средствами наблюдения и учета.

3. *Системы* ***уровня B*** *основаны на помеченных данных и распределении пользователей по категориям, то есть реализуют мандатный контроль доступа.* Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к данным только в соответствии с этим рейтингом. *Этот уровень в отличие от уровня C защищает систему от ошибочного поведения пользователя.*
4. ***Уровень A*** *является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня B выполнения*

формального, математически обоснованного доказательства соответствия системы требованиям безопасности.

Различные коммерческие структуры (например, банки) особо выделяют необходимость учетной службы, аналогичной той, что предлагают государственные рекомендации С2. Любая деятельность, связанная с безопасностью, может быть отслежена и тем самым учтена. Это как раз то, что требует С2 и то, что обычно нужно банкам.

10.2 Тенденции в структурном построении ОС

Как уже отмечалось выше, для удовлетворения требований, предъявляемых к современной ОС, большое значение имеет ее структурное построение. Операционные системы прошли длительный путь развития от монолитных систем к хорошо структурированным модульным системам, способным к развитию, расширению и легкому переносу на новые платформы.

10.2.1 Монолитные системы

В общем случае ***"структура" монолитной системы*** *представляет собой отсутствие структуры* (рисунок 10.1)*. ОС написана как набор процедур, каждая из которых может вызывать другие, когда ей это нужно.* При использовании этой техники каждая процедура системы имеет хорошо определенный интерфейс в терминах параметров и результатов, и каждая вольна вызвать любую другую для выполнения некоторой нужной для нее полезной работы.

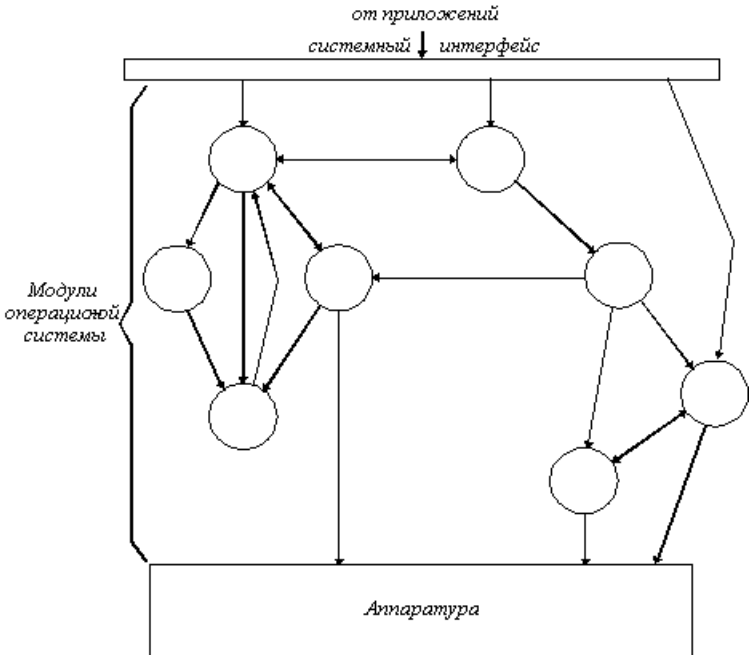


Рис. 10.1. Монолитная структура ОС

Для построения монолитной системы необходимо скомпилировать все отдельные процедуры, а затем связать их вместе в единый объектный файл с помощью компоновщика Каждая процедура видит любую другую процедуру (в отличие от структуры, содержащей модули, в которой большая часть информации является локальной для модуля, и процедуры модуля можно вызвать только через специально определенные точки входа).

Однако даже такие монолитные системы могут быть немного структурированными. Такая организация ОС предполагает следующую структуру:

- *Главная программа, которая вызывает требуемые сервисные процедуры.*
- *Набор сервисных процедур, реализующих системные вызовы.*
- *Набор утилит, обслуживающих сервисные процедуры.*

В этой модели для каждого системного вызова имеется одна сервисная процедура. Утилиты выполняют функции, которые нужны нескольким сервисным процедурам. Это деление процедур на три слоя показано на рисунке 4.2.

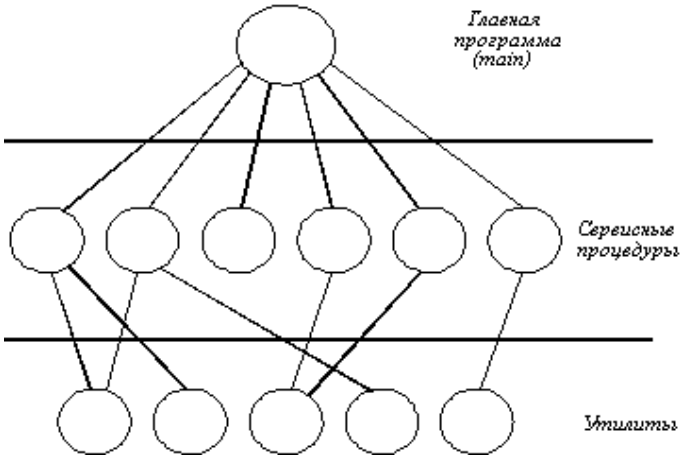


Рис. 4.2. Простая структуризация монолитной ОС

10.2.2 Многоуровневые системы

Обобщением предыдущего подхода является ***организация ОС как иерархии уровней****. Уровни образуются группами функций операционной системы - файловая система, управление процессами и устройствами и т.п. Каждый уровень может взаимодействовать только со своим непосредственным соседом - выше- или нижележащим уровнем. Прикладные программы или модули самой операционной системы передают запросы вверх и вниз по этим уровням.*

Первой системой, построенной таким образом была простая пакетная система THE, которую построил Дейкстра и его студенты в 1968 году. Система имела 6 уровней.

- Уровень 0 занимался распределением времени процессора, переключая процессы по прерыванию или по истечении времени.
- Уровень 1 управлял памятью - распределял оперативную память и пространство на магнитном барабане для тех частей процессов (страниц), для которых не было места в ОП, то есть слой 1 выполнял функции виртуальной памяти.
- Слой 2 управлял связью между консолью оператора и процессами. С помощью этого уровня каждый процесс имел свою собственную консоль оператора.
- Уровень 3 управлял устройствами ввода-вывода и буферизовал потоки информации к ним и от них. С помощью уровня 3 каждый процесс вместо того, чтобы работать с конкретными устройствами, с их разнообразными особенностями, обращался к абстрактным устройствам ввода-вывода, обладающим удобными для пользователя характеристиками.
- На уровне 4 работали пользовательские программы, которым не надо было заботиться ни о процессах, ни о памяти, ни о консоли, ни об управлении устройствами ввода-вывода.
- Процесс системного оператора размещался на уровне 5.

Многоуровневый подход был также использован при реализации различных вариантов ОС UNIX.

Хотя такой структурный подход на практике обычно работал неплохо, сегодня он все больше воспринимается монолитным. В системах, имеющих многоуровневую структуру было нелегко удалить один слой и заменить его другим в силу множественности и размытости интерфейсов между слоями. Добавление новых функций и изменение существующих требовало хорошего знания операционной системы и массы времени.

Когда стало ясно, что операционные системы должны иметь возможности развития и расширения, *монолитный подход стал давать трещину, и на смену ему пришла модель клиент-сервер и тесно связанная с ней концепция микроядра*.

10.2.3 Модель клиент-сервер и микроядра

Модель клиент-сервер - это еще один подход к структурированию ОС.

Модель клиент-сервер *предполагает наличие программного компонента - потребителя какого-либо сервиса -* ***клиента****, и программного компонента - поставщика этого сервиса -* ***сервера.*** *Взаимодействие между клиентом и сервером стандартизуется, так что сервер может обслуживать клиентов, реализованных различными способами и, может быть, разными производителями.*

Инициатором обмена обычно является клиент, который посылает запрос на обслуживание серверу, находящемуся в состоянии ожидания запроса. Один и тот же программный компонент может быть клиентом по отношению к одному виду услуг, и сервером для другого вида услуг.

Модель клиент-сервер успешно применяется не только при построении ОС, но и на всех уровнях программного обеспечения, и имеет в некоторых случаях более узкий, специфический смысл, сохраняя, естественно, при этом все свои общие черты.

Применительно к структурированию ОС идея «клиент-сервер» состоит в разбиении ее на несколько процессов - серверов, каждый из которых выполняет отдельный набор сервисных функций - например, управление памятью, создание или планирование процессов.

Каждый сервер выполняется в пользовательском режиме. Клиент, которым может быть либо другой компонент ОС, либо прикладная программа, запрашивает сервис, посылая сообщение на сервер. Ядро ОС (называемое здесь микроядром), работая в привилегированном режиме, доставляет сообщение нужному серверу, сервер выполняет операцию, после чего ядро возвращает результаты клиенту с помощью другого сообщения (рисунок 10.3).

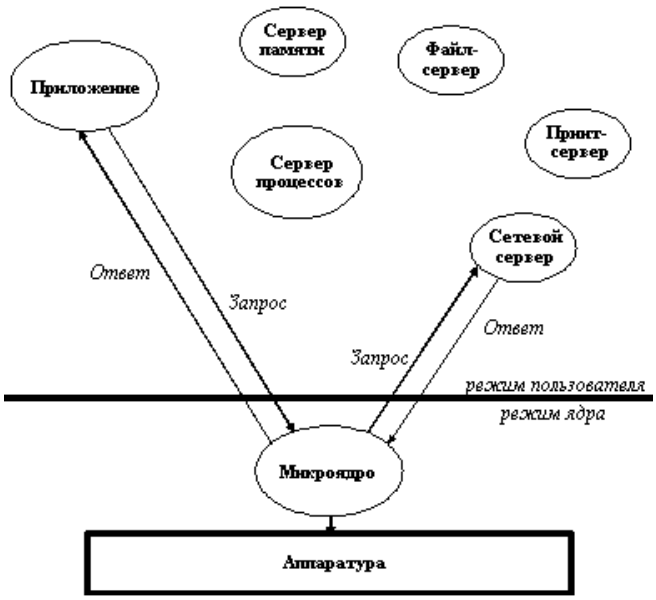


Рис. 10.3. Структура ОС клиент-сервер

Подход с использованием микроядра *заменил вертикальное распределение функций операционной системы на горизонтальное. Компоненты, лежащие выше микроядра, хотя и используют сообщения, пересылаемые через микроядро, взаимодействуют друг с другом непосредственно. Микроядро играет роль регулировщика.* Оно проверяет

сообщения, пересылает их между серверами и клиентами, и предоставляет доступ к аппаратуре.

Данная теоретическая модель является идеализированным описанием системы клиент-сервер, в которой ядро состоит только из средств передачи сообщений. В действительности различные варианты реализации модели клиент-сервер в структуре ОС могут существенно различаться по объему работ, выполняемых в режиме ядра.

На одном краю этого спектра находится разрабатываемая фирмой IBM на основе микроядра Mach операционная система Workplace OS, придерживающаяся чистой микроядерной доктрины, состоящей в том, что все несущественные функции ОС должны выполняться не в режиме ядра, а в непривилегированном (пользовательском) режиме. На другом - Windows NT, в составе которой имеется исполняющая система (NT executive), работающая в режиме ядра и выполняющая функции обеспечения безопасности, ввода-вывода и другие.

Микроядро реализует жизненно важные функции, лежащие в основе операционной системы. Это базис для менее существенных системных служб и приложений. В общем случае, подсистемы, бывшие традиционно неотъемлемыми частями операционной системы - файловые системы, управление окнами и обеспечение безопасности - становятся периферийными модулями, взаимодействующими с ядром и друг с другом.

Главный принцип разделения работы между микроядром и окружающими его модулями - включать в микроядро только те функции, которым абсолютно необходимо исполняться в режиме супервизора и в привилегированном пространстве. Под этим обычно подразумеваются

- *машиннозависимые программы (включая поддержку нескольких процессоров),*
- *некоторые функции управления процессами,*
- *обработка прерываний,*
- *поддержка пересылки сообщений,*
- *некоторые функции управления устройствами ввода-вывода, связанные с загрузкой команд в регистры устройств.*

Эти функции операционной системы трудно, если не невозможно, выполнить программам, работающим в пространстве пользователя.

В настоящее время именно операционные системы, построенные с использованием модели клиент-сервер и концепции микроядра, в наибольшей степени удовлетворяют требованиям, предъявляемым к современным ОС нижеуказанные свойства.

- **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.

- Технология микроядер является основой построения множественных прикладных сред, которые **обеспечивают совместимость программ**, написанных для разных ОС. Абстрагируя интерфейсы прикладных программ от расположенных ниже операционных систем, микроядра позволяют гарантировать, что вложения в прикладные программы не пропадут в течение нескольких лет, даже если будут сменяться операционные системы и процессоры.
- **Расширяемость** также является одним из важных требований к современным операционным системам. Увеличивающаяся сложность монолитных операционных систем делала трудным, если вообще возможным, внесение изменений в ОС с гарантией надежности ее последующей работы. Ограниченный набор четко определенных интерфейсов микроядра открывает путь к упорядоченному росту и эволюции ОС.
- Использование модели клиент-сервер повышает **надежность**. Каждый сервер выполняется в виде отдельного процесса в своей собственной области памяти, и таким образом защищен от других процессов. Более того, поскольку серверы выполняются в пространстве пользователя, они не имеют непосредственного доступа к аппаратуре и не могут модифицировать память, в которой хранится управляющая программа. И если отдельный сервер может потерпеть крах, то он может быть перезапущен без останова или повреждения остальной части ОС.
- Эта модель хорошо **подходит для распределенных вычислений**, так как отдельные серверы могут работать на разных процессорах мультипроцессорного компьютера или даже на разных компьютерах.

10.2.4 Объектно-ориентированный подход

В настоящее время для цели обеспечения расширяемости в наибольшей степени соответствует объектно-ориентированный подход, при котором каждый программный компонент является функционально изолированным от других.

Основным понятием этого подхода является "объект".

Объект *- это единица программ и данных, взаимодействующая с другими объектам посредством приема и передачи сообщений. Объект может быть представлением как некоторых конкретных вещей -*

прикладной программы или документа, так и некоторых абстракций - процесса, события.

Программы (функции) объекта *определяют перечень действий, которые могут быть выполнены над данными этого объекта.* ***Объект-клиент*** *может обратиться к другому объекту, послав сообщение с запросом на выполнение какой-либо функции* ***объекта-сервера****.*

Для ***обеспечения преемственности*** *при переходе к более детальному описанию разработчикам предлагается механизм наследования свойств уже существующих объектов, то есть механизм, позволяющий порождать более конкретные объекты из более общих.*

Например, при наличии объекта "текстовый документ" разработчик может легко создать объект "текстовый документ в формате Word 6.0", добавив соответствующее свойство к базовому объекту. Механизм наследования позволяет создать иерархию объектов, в которой каждый объект более низкого уровня приобретает все свойства своего предка.

Внутренняя структура данных объекта скрыта от наблюдения. Нельзя произвольно изменять данные объекта. Для того, чтобы получить данные из объекта или поместить данные в объект, необходимо вызывать соответствующие объектные функции. Это изолирует объект от того кода, который использует его. Разработчик может обращаться к функциям других объектов, или строить новые объекты путем наследования свойств других объектов, ничего не зная о том, как они сконструированы. Это свойство называется инкапсуляцией.

Таким образом, объект предстает для внешнего мира в виде "черного ящика" с хорошо определенным интерфейсом. С точки зрения разработчика, использующего объект, пока внешняя реакция объекта остается без изменений, не имеют значения никакие изменения во внутренней реализации. Это дает возможность легко заменять одну реализацию объекта другой, например, в случае смены аппаратных средств; при этом сложное программное окружение, в котором находятся заменяемые объекты, не потребует никаких изменений.

Использование объектно-ориентированного подхода особенно эффективно при создании активно развивающегося программного обеспечения, например, при разработке приложений, предназначенных для выполнения на разных аппаратных платформах.

Полностью объектно-ориентированные операционные системы очень привлекательны так как, используя объекты системного уровня, программисты смогут залезать вглубь операционных систем для приспособления их к своим нуждам, не нарушая целостность системы.

Но особенно большие перспективы имеет этот подход в реализации распределенных вычислительных сред. В то время, как сейчас разные

пакеты, работающие в данный момент в сети, представляют собой статически связанные наборы программ, в будущем, с использованием объектно-ориентированного подхода, они могут превратиться в единую совокупность динамически связываемых объектов, где каждый объект оперативно устанавливает и разрывает связи с другими объектами для выполнения актуальных в данный момент задач. Приложения, созданные для такой сетевой среды, основанной на объектах, могут выполняться, динамически обращаясь к множеству объектов, независимо от их местонахождения в сети и независимо от их операционной среды.

10.2.5 Множественные прикладные среды

В то время как некоторые идеи (например, объектно-ориентированный подход) непосредственно касаются только разработчиков и лишь косвенно влияют на конечного пользователя, ***концепция множественных прикладных сред*** *приносит пользователю долгожданную возможность выполнять на своей ОС программы, написанные для других операционных систем и других процессоров.*

Множественные прикладные среды обеспечивают совместимость данной ОС с приложениями, написанными для других ОС и процессоров, на двоичном уровне, а не на уровне исходных текстов.

При реализации множественных прикладных сред разработчики сталкиваются с **противоречивыми требованиями**.

1. С одной стороны, задачей каждой прикладной среды является выполнение программы по возможности так, как если бы она выполнялась на "родной" ОС.
2. С другой стороны потребности этих программ могут входить в конфликт с конструкцией современной операционной системы:
   - Специализированные драйверы устройств могут противоречить требованиям безопасности.
   - Могут конфликтовать схемы управления памятью и оконные системы.
   - Чисто экономические вопросы (например, стоимость лицензирования программ и угроза судебного преследования) также могут повлиять на дизайн чужих прикладных сред.
   - Большой потенциальной проблемой является производительность - прикладная среда должна выполнять программы с приемлемой скоростью.

Для сокращения времени на выполнение чужих программ прикладные среды используют имитацию программ на уровне библиотек. Эффективность этого подхода связана с тем, что большинство сегодняшних

программ работают под управлением GUI (графических интерфейсов пользователя). Они непрерывно выполняют вызовы библиотек GUI для манипулирования окнами и для других связанных с GUI действий. И это то, что позволяет прикладным средам возместить время, потраченное на эмулирование команды за командой. Тщательно сделанная прикладная среда имеет в своем составе библиотеки, имитирующие внутренние библиотеки GUI, но написанные на родном коде, то есть она совместима с программным интерфейсом другой ОС. Иногда такой подход называют трансляцией для того, чтобы отличать его от более медленного процесса эмулирования кода по одной команде за раз.

С позиции использования прикладных сред более предпочтительным является способ написания программ, при котором программист для выполнения некоторой функции обращается с вызовом к операционной системе, а не пытается более эффективно реализовать эквивалентную функцию самостоятельно, работая напрямую с аппаратурой.

Модульность операционных систем нового поколения позволяет намного легче реализовать поддержку множественных прикладных сред. В отличие от старых операционных систем, состоящих из одного большого блока для всех практических применений, разбитого произвольным образом на части, новые системы являются модульными, с четко определенными интерфейсами между составляющими. Это делает создание дополнительных модулей, объединяющих эмуляцию процессора и трансляцию библиотек, значительно более простым.

Существует много разных стратегий по воплощению идеи множественных прикладных сред, и некоторые из этих стратегий диаметрально противоположны.

- В случае UNIX, транслятор прикладных сред обычно делается, как и другие прикладные программы, плавающим на поверхности операционной системы.
- В более современных операционных системах типа Windows NT модули прикладной среды выполняются более тесно связанными с операционной системой, хотя и обладают по-прежнему высокой независимостью.
- В OS/2 с ее более простой, слабо структурированной архитектурой средства организации прикладных сред встроены глубоко в операционную систему.

Использование множественных прикладных сред обеспечит пользователям большую свободу выбора операционных систем и более легкий доступ к более качественному программному обеспечению.

11. ОСОБЕННОСТИ ПОСТРОЕНИЯ ОПЕРАЦИОННЫХ СИСТЕМ СЕМЕЙСТВА WINDOWS

11.1 Краткая история создания ОС Windows

Операционные системы корпорации Microsoft можно условно разделить на три группы:

1. MS-DOS и MS-DOS+Windows 3.1,
2. потребительские (consumer) версии Windows (Windows 95/98/Me),
3. линия ОС, ведущих свое начало от Windows NT (Windows NT/2000/XP/Vista).

Однозадачная **16-разрядная ОС MS-DOS** была выпущена в начале 80-х годов и затем широко применялась на компьютерах с процессором x86. Вначале MS-DOS была довольно примитивна, ее оболочка занималась, главным образом, обработкой командной строки, но в последующие версии было внесено много улучшений, заимствованных, главным образом, из ОС Unix. Затем под влиянием успехов дружественного графического интерфейса корпорации Macintosh была разработана **система Windows**. Особенно широкое распространение получили версии **Windows 3.0, 3.1 и 3.11**. Первоначально это была не самостоятельная ОС, а скорее многозадачная (с невытесняющей многозадачностью) графическая оболочка MS-DOS, которая контролировала компьютер и файловую систему.

В 1995 г. была выпущена 32-разрядная **ОС Windows 95**, где была реализована вытесняющая многозадачность. ОС Windows 95 включала большой объем 16-разрядного кода, главным образом для обеспечения преемственности с приложениями MS-DOS. Другой проблемой данной версии Windows, была нереентерабельность существенной части кода ядра. Так, если один из потоков был занят модификацией данных в ядре, другой поток, чтобы не получить эти данные в противоречивом состоянии, вынужден был ждать, то есть не мог воспользоваться системными сервисами. Это, зачастую, сводило на нет преимущества многозадачности.

Операционная система **Windows NT** (New Technology) - новая 32-разрядная ОС, совместимая с предшествующими версиями Windows по интерфейсу. Работу над созданием системы возглавил Дэвид Катлер, один из ключевых разработчиков ОС VAX VMS. Ряд идей системы VMS присутствует в NT. Заметна преемственность в системе управления большим адресным пространством и резидентным множеством процесса, в системе приоритетов обычных процессов и процессов реального времени, в средствах синхронизации и т.д. Вместе с тем Windows NT - это совершенно новый амбициозный проект разработки системы с учетом новейших достижений в области архитектуры микроядра.

В начале 1999 г. была выпущена Windows NT 5.0, переименованная в **Windows 2000**. Следующая версия этой ОС данной серии – **Windows XP** появилась в 2001 г., а Windows Server 2003 - в 2003 г. В настоящее время выпущена **Windows Vista**, ранее известная под кодовым именем Longhorn, - новая версия Windows, продолжающая линейку Windows NT.

Объем исходных текстов ядра ОС Windows неизвестен. По некоторым оценкам, объем ядра Windows NT 3.5 составляет приблизительно 10 Мб, а с каждой новой версией ОС Windows этот объем неуклонно увеличивается в полтора-два раза.

11.2 Возможности системы Windows

Перечень возможностей системы достаточно широк, вот лишь некоторые из них. Операционная система Windows:

- является истинно 32-разрядной, поддерживает вытесняющую многозадачность;
- работает на разных аппаратных архитектурах и обладает способностью к сравнительно легкому переносу на новые аппаратные архитектуры;
- поддерживает работу с виртуальной памятью;
- является полностью реентерабельной;
- хорошо масштабируется в системах с симметричной мультипроцессорной обработкой;
- является распределенной вычислительной платформой, способной выступать в роли как клиента сети, так и сервера;
- защищена как от внутренних сбоев, так и от внешних деструктивных действий. У приложений нет возможности нарушить работу операционной системы или других приложений;
- совместима, то есть, ее пользовательский интерфейс и API совместимы с предыдущими версиями Windows и MS-DOS. Она также умеет взаимодействовать с другими системами вроде UNIX, OS/2 и NetWare;
- обладает высокой производительностью независимо от аппаратной платформы;
- обеспечивает простоту адаптации к глобальному рынку за счет поддержки Unicode;
- поддерживает многопоточность и объектную модель.

11.3 Общая структура ОС Windows

Архитектура ОС Windows претерпела ряд изменений в процессе эволюции. Первые версии системы имели микроядерный дизайн. Архитектура более поздних версий системы микроядерной уже не является.

Причина заключается в постепенном преодолении основного недостатка микроядерных архитектур - дополнительных накладных расходов, связанных с передачей сообщений. По Microsoft, чисто микроядерный дизайн коммерчески невыгоден, поскольку неэффективен. Поэтому большой объем системного кода, в первую очередь управление системными вызовами и экранная графика, был перемещен из адресного пространства пользователя в пространство ядра и работает в привилегированном режиме. В результате в ядре ОС Windows переплетены элементы микроядерной архитектуры и элементы монолитного ядра (комбинированная система).

Сегодня микроядро ОС Windows слишком велико (более 1 Мб), чтобы носить приставку "микро". *Основные компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как и положено в микроядерных операционных системах. В тоже время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром.*

Высокая модульность и гибкость первых версий Windows NT позволила успешно перенести систему на такие отличные от Intel платформы, как Alpha (корпорация DEC), Power PC (IBM) и MIPS (Silicon Graphic). Более поздние версии ограничиваются поддержкой архитектуры Intel x86.

ОС Windows состоит из компонентов, работающих в режиме ядра, и компонентов, работающих в режиме пользователя (рис. 11.1).

В схеме, представленной на рис. 11.1, отчетливо просматриваются несколько функциональных уровней, каждый из которых пользуется сервисами более низкого уровня:

- *Задача* ***слоя абстрагирования от оборудования*** (hardware abstraction layer, HAL) - *скрыть аппаратные различия аппаратных архитектур для потенциального переноса системы с одной платформы на другую.* HAL предоставляет выше лежащим уровням аппаратные устройства в абстрактном виде, что позволяет изолировать ядро, драйверы и исполнительную систему ОС Windows от специфики оборудования.
- ***Ядром*** *обычно называют все компоненты ОС, работающие в привилегированном режиме работы процессора или в режиме ядра.* Корпорация Microsoft называет *ядром (kernel)* компонент, находящийся в невыгружаемой памяти и содержащий низкоуровневые функции операционной системы: диспетчеризация прерываний и исключений, планирование потоков и др.

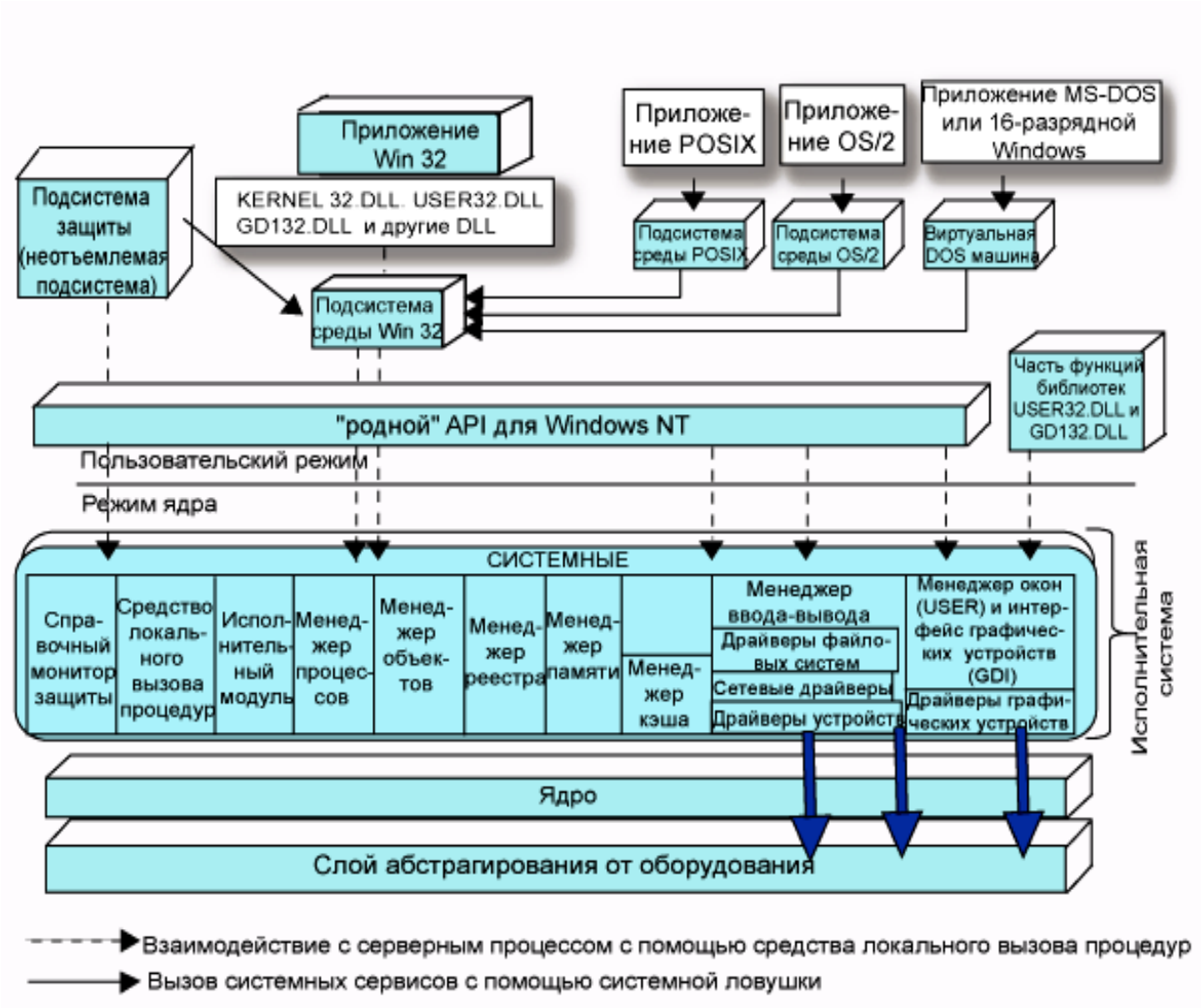


Рис. 11.1. Упрощенная архитектурная схема ОС Windows

– ***Исполнительная система (executive)*** обеспечивает управление памятью, процессами и потоками, защиту, ввод-вывод и взаимодействие между процессами.

– ***Менеджер ввода-вывода и драйверы устройств*** *содержат аппаратно-зависимый код и обеспечивают трансляцию пользовательских вызовов в запросы, специфичные для конкретных устройств.*

– ***Менеджер окон и графики*** *реализует функции графического пользовательского интерфейса (GUI)*, более известные как Win-32-функции модулей USER и GDI

– *В* ***режиме пользователя*** *работают разнообразные сервисы (аналоги демонов в Unix), управляемые диспетчером сервисов и решающие системные задачи*. Пользовательские приложения (user applications) бывают пяти типов: Win32, Windows 3.1, MS-DOS, POSIX и OS/2 1.2. Среду для выполнения пользовательских процессов предоставляют три подсистемы окружения: Win32,

POSIX и OS/2. Таким образом, пользовательские приложения не могут вызывать системные вызовы ОС Windows напрямую, а вынуждены обращаться к DLL подсистем.

DLL (динамически подключаемая библиотека) - *Набор вызываемых подпрограмм, включенных в один двоичный файл, который приложения, использующие эти подпрограммы, могут динамически загружать в процессе своего выполнения.*

Ядро и HAL являются аппаратно-зависимыми и написаны на языках Си и ассемблера. Верхние уровни написаны на языке Си и являются машинно-независимыми.

Основные компоненты ОС Windows реализованы в следующих системных файлах, находящихся в каталоге system32:

- **ntoskrnl.exe** - исполнительная система и ядро;
- **ntdll.dll** - внутренние функции поддержки и интерфейсы диспетчера системных сервисов с функциями исполнительной системы;
- **hal.dll** - уровень абстрагирования от оборудования;
- **win32k.sys** - часть подсистемы Win32, работающая в режиме ядра;
- **kernel32.dll, advapi32.dll, user32.dll, gdi32.dll** - основные dll подсистемы Win32.

11.4 Подсистема Win32

Взаимодействие между приложением и операционной системой осуществляется при помощи системных вызовов (системных сервисов в терминологии Microsoft). Однако приложение не может вызвать системный вызов напрямую (более того, системные вызовы не документированы). Вместо этого приложение должно воспользоваться программным интерфейсом ОС - Win32 API.

Win32 API (Application Programming Interface) - прикладной интерфейс программирования в семействе операционных систем Microsoft Windows.

В состав Win32 подсистемы (см. рис. 11.2) входят:

- серверный процесс подсистемы окружения csrss.exe,
- драйвер режима ядра Win32k.sys,
- dll - модули подсистем (kernel32.dll, advapi32.dll, user32.dll и gdi32.dll), экспортирующие Win32-функции и драйверы графических устройств.

В процессе эволюции структура подсистемы претерпела изменения. Например, функции окон и рисования с целью повышения производительности были перенесены из серверного процесса, работающего в режиме пользователя, в драйвер режима ядра Win32k.sys.

Приложение, ориентированное на использование Win32 API, может работать практически на всех версиях Windows, несмотря на то, что сами системные вызовы в различных системах различны (см. рис. 11.2). Таким путем корпорация Microsoft обеспечивает преемственность своих операционных систем.

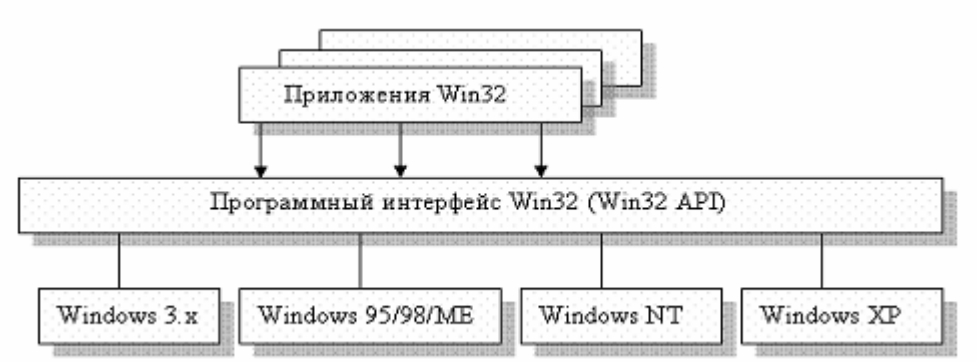


Рис. 11.2. Поддержка единого программного интерфейса для различных версий Windows

При вызове приложением одной из Win32-функций dll-подсистем может возникнуть одна из трех ситуаций (см. рис. 11.3).

1. Функция полностью выполняется внутри данной dll (шаг 1).
2. Для выполнения функции привлекается сервер csrss, для чего ему посылается сообщение (шаг 2a, за которым обычно следуют шаги 2b и 2c).
3. Данный вызов транслируется в системный сервис (системный вызов), который обычно обрабатывается в модуле ntdll.dll (шаги 3a и 3b).

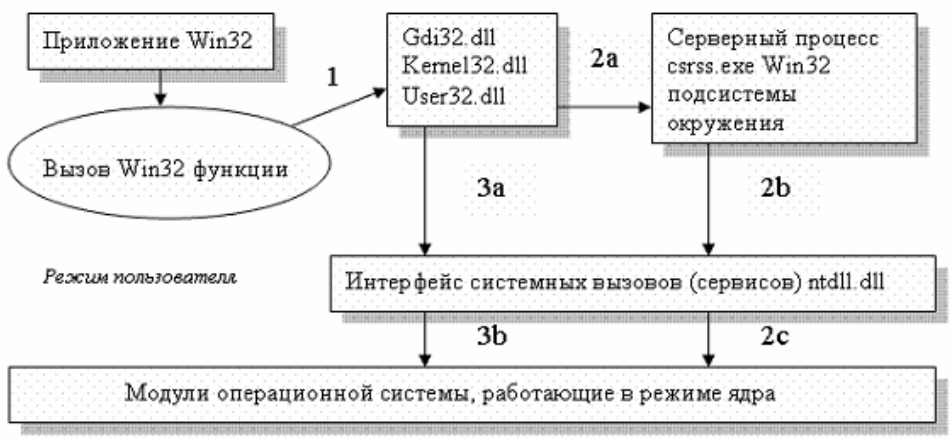


Рис. 11.3. Различные маршруты выполнения вызовов Win32 API

Помимо перечисленных, наиболее важных dll-библиотек, в системном каталоге system32 имеется большое количество других dll-файлов. В настоящее время количество вызовов API составляет несколько десятков тысяч.

11.5 Основные механизмы в ОС Windows

Рассмотрим реализацию основных механизмов операционной системы в ОС Windows. Следует отметить, что терминология корпорации Microsoft несколько отличается от общепринятой. Например, ***системные вызовы*** *называются* **системными сервисами***, а под* ***программным прерыванием*** *понимается выполнение специфичных функций ядра, требующих прерывания работы текущего процесса.*

11.5.1 Ловушки

Общим *для реализации рассматриваемых основных механизмов является необходимость сохранения состояния текущего потока с его последующим восстановлением. Для этого в ОС Windows используется механизм ловушек (trap).*

В случае возникновения требующего обработки события (прерывания, исключения или вызова системного сервиса) процессор переходит в привилегированный режим и передает управление обработчику ловушек, входящему в состав ядра. Обработчик ловушек создает в стеке ядра прерываемого потока фрейм ловушки, содержащий часть контекста потока для последующего восстановления его состояния, и в свою очередь передает управление определенной части ОС, отвечающей за первичную обработку произошедшего события.

В типичном случае сохраняются и впоследствии восстанавливаются:

- программный счетчик;
- регистр состояния процессора;
- содержимое остальных регистров процессора;
- указатели на стек ядра и пользовательский стек;
- указатели на адресное пространство, в котором выполняется поток (каталог таблиц страниц процесса).

Адрес части ядра ОС, ответственной за обработку данного конкретного события определяется из вектора прерываний*, который номеру события ставит в соответствие адрес процедуры его первичной обработки.* Это оказывается возможным, поскольку все события типизированы и их число ограничено. Для асинхронных событий их номер определяется контроллером прерываний, а для синхронных - ядром.

То же самое происходит в случае возникновения исключений и прерываний. Простые исключения могут быть обработаны диспетчером ловушек, а более сложные обрабатываются диспетчером исключений, который может в случае возникновения исключения вернуть управление вызвавшему это исключение приложению.

11.5.2 Приоритеты

В большинстве операционных систем аппаратные прерывания имеют приоритеты, которые определяются контроллерами прерываний. Однако ОС Windows имеет свою аппаратно-независимую шкалу приоритетов, которые называются *уровни запросов прерываний* (interrupt request levels, IRQL), и охватывает не только прерывания, а все события, требующие системной обработки.

Можно сказать, что в ОС Windows действует двухуровневая схема планирования. Приоритеты высшего уровня (в данном случае IRQLs) определяются аппаратными или программными прерываниями, а приоритеты низшего уровня (в своем диапазоне от 0 до 31) устанавливаются для пользовательских потоков, выполняемых на нулевом уровне IRQL, и контролируются планировщиком.

11.5.3 Планирование потоков

Процедура планирования обычно связана с весьма затратной процедурой диспетчеризации - переключением процессора на новый поток, поэтому планировщик должен заботиться об эффективном использовании процессора. Принадлежность потоков к процессу при планировании не учитывается, то есть единицей планирования в ОС Windows является именно поток.

В ОС Windows реализовано вытесняющее приоритетное планирование, когда каждому потоку присваивается определенное числовое значение - приоритет, в соответствии с которым ему выделяется процессор. Потоки с одинаковыми приоритетами планируются согласно алгоритму Round Robin (карусель).

В системе предусмотрено 32 уровня приоритетов. Шестнадцать значений приоритетов (16-31) соответствуют группе приоритетов реального времени, пятнадцать значений (1-15) предназначены для обычных потоков, и значение 0 зарезервировано для системного потока обнуления страниц (см. рис. 11.4).

Рис. 11.4. Приоритеты потоков

Чтобы избавить пользователя от необходимости запоминать числовые значения приоритетов и иметь возможность модифицировать планировщик,

разработчики ввели в систему *слой абстрагирования приоритетов*. Совокупность из шести классов приоритетов процессов и семи классов приоритетов потоков образует 42 возможные комбинации и позволяет сформировать так называемый базовый приоритет потока (см. табл. 11.1).

Таблица 11.1. Формирование базового приоритета потока из класса приоритета процесса и относительного приоритета потока

	Приоритеты потоков						
Классы приоритетов процессов	Критичный ко времени	Самый высокий	Выше нормы	Нормаль-ный	Ниже нормы	Самый низкий	Неработа-ющий
Неработающий	15	6	5	4	3	2	1
Ниже нормы	15	8	7	6	5	4	1
Нормальный	15	10	9	8	7	6	1
Выше нормы	15	12	11	10	9	8	1
Высокий	15	15	14	13	12	11	1
Реального времени	31	26	25	24	23	22	16

Базовый приоритет процесса и первичного потока по умолчанию равен значению из середины диапазонов приоритетов процессов (24, 13, 10, 8, 6 или 4). Смена приоритета процесса влечет за собой смену приоритетов всех его потоков, при этом их относительные приоритеты остаются без изменений.

11.6 Реализация файловой системы

Типовая совокупность действий пользователя в отношении файловой системы на диске состоит из форматирования диска, создания на нем структуры каталогов, заполнения их файлами, а также выполнения разнообразных действий с этими файлами. Кроме того, файловые службы должны решать проблемы совместного доступа к данным, проблемы проверки и сохранения целостности файловой системы, проблемы повышения производительности и ряд других. В ОС Windows файловая система интегрирована в систему ввода-вывода (см. рис. 11.5), построенную в виде набора разнообразных драйверов, и также реализована в виде драйвера, например, драйвера NTFS или драйвера FAT. Общение драйверов организовано путем посылки так называемых IRP (I/O request packet) пакетов.

Подобно многим современным операционным системам ОС Windows поддерживает несколько файловых систем (CDFS, UDF, FAT, NTFS, удаленные FS). Эта возможность заложена в архитектуре системы ввода-вывода. Список зарегистрированных файловых систем можно "увидеть" с

помощью утилиты WinObj. *Базовой файловой системой в ОС Windows является NTFS.*

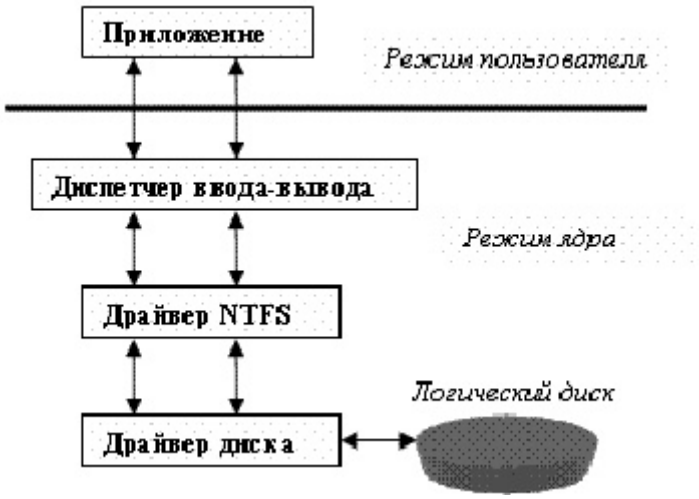


Рис. 11.5. Доступ к файловой системе через систему ввода-вывода

11.6.1 Монтирование файловых систем

Монтирование базовых дисков осуществляется автоматически при первом обращении к диску. Это делает диспетчер монтирования (Mountmgr.sys). Сведения о монтированных дисках имеются в реестре в разделе HKLM\SYSTEM\MountedDevices.

Создание точек монтирования (mount points) - связывание каталога NTFS реализовано с помощью точек повторного анализа. Связывание файлов - техника, заимствованная из Unix, - образование для файла или каталога нескольких родительских каталогов, см. рис. 11.6.

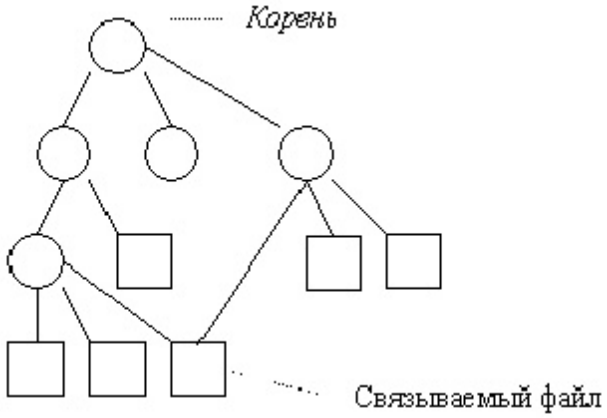


Рис. 11.6. Образование связей в файловой системе

ОС Windows (как и Unix) поддерживает два вида связей - жесткие (hard link) и символические (symbolic link). В случае жесткой связи запись о файле

появляется в новом каталоге, а MFT-запись этого файла включает счетчик количества ссылок на данный файл.

Символическая линковка - создание нового файла, который содержит путь к связываемому файлу. Символическую связь (иногда говорят точку соединения, junction) можно образовать при помощи входящей в состав ресурсов Windows утилиты linkd.exe или при помощи свободно распространяемой утилиты junction.exe с сайта www.sysinternals.com.

11.6.2 Устройство кэша файловой системы

Устройство кэша файловой системы ОС Windows отличается от традиционного. В традиционной реализации кэш - буфер в оперативной памяти, содержащий ряд блоков диска и расположенный между файловой системой и системой ввода-вывода.

В ОС Windows кэш работает на более высоком уровне, нежели файловая система (см. рис. 11.7). В результате запрос на чтение с текущей позиции может быть непосредственно удовлетворен из кэша. Если же нужных байтов файла в кэше нет, то файловая система вычисляет логический номер блока в файле (LCN), затем логический номер блока на диске (VCN). Подобная организация позволяет системе поддерживать единый централизованный кэш для всех используемых файловых систем (NTFS, FAT, CDFS, NFS и др.), а файловые системы не обязаны управлять своими кэшами.

Рис. 11.7. Место менеджера кэша в системе ввода-вывода

11.6.3 Поддержание целостности файловой системы

В современных ОС Windows предусмотрены меры, которые позволяют свести к минимуму ущерб от порчи файловой системы и затем полностью или частично восстановить ее целостность.

Одним из средств поддержки целостности является *журналирование*. Последовательность действий с объектами во время транзакции протоколируется, и, если произошел останов системы, то, имея в наличии протокол, можно осуществить откат системы назад в исходное целостное состояние, в котором она пребывала до начала транзакции.

Если нарушение целостности файловой системы все же произошло, то можно прибегнуть к помощи специализированных *утилит* (chkdsk, scandisk и др.). Они могут запускаться после загрузки или после сбоя и осуществляют многократное сканирование разнообразных структур данных файловой системы в поисках противоречий.

11.7 Средства управления безопасностью

Для управления системой безопасности в ОС Windows имеются разнообразные и удобные инструментальные средства. В частности, потребуется умение управлять учетными записями пользователей при помощи панели "Пользователи и пароли". Кроме того понадобится контролировать привилегии пользователей при помощи панели "Назначение прав пользователям". Рекомендуется также освоить работу с утилитой просмотра данных маркера доступа процесса WhoAmI.exe, утилитами просмотра и редактирования списков контроля доступа (cacls.exe, ShowACLs.exe, SubInACL.exe, SvcACL.exe), утилитой просмотра маркера доступа процесса PuList.exe и рядом других.

11.7.1 Система управления доступом

Подсистема защиты данных является одной из наиболее важных. В центре системы безопасности ОС Windows находится система контроля доступа.

С каждым процессом или потоком, то есть активным компонентом (**субъектом**), связан маркер доступа, а у каждого защищаемого объекта (например, файла) имеется дескриптор защиты. Проверка прав доступа обычно осуществляется в момент открытия объекта и заключается в сопоставлении прав **субъекта** списку прав доступа, который хранится в составе дескриптора защиты **объекта**

Защищаемые объекты Windows включают:

- файлы,
- устройства,

- каналы,
- события,
- мьютексы,
- семафоры,
- разделы общей памяти,
- разделы реестра и др.

Сущность, от которой нужно защищать объекты, называется **"субъектом"**. Субъектами в Windows являются процессы и потоки, запускаемые конкретными пользователями. Субъект безопасности - активная системная составляющая, а объект - пассивная.

Помимо дискреционного доступа Windows поддерживает управление ***привилегированным*** **доступом**. Это означает, что в системе имеется пользователь-администратор с неограниченными правами.

Кроме того, для упрощения администрирования пользователи Windows объединены в **группы**. Пользователь, как член группы, облекается, таким образом, набором полномочий, необходимых для его деятельности, и играет определенную роль. Подобная стратегия называется **управление *ролевым* доступом**.

Ключевая цель системы защиты Windows - следить за тем, кто и к каким объектам осуществляет доступ. Система защиты хранит информацию, относящуюся к безопасности для каждого пользователя, группы пользователей и объекта. Модель защиты ОС Windows требует, чтобы субъект на этапе открытия объекта указывал, какие операции он собирается выполнять в отношении этого объекта.

Единообразие контроля доступа *к различным объектам (процессам, файлам, семафорам и др.) обеспечивается тем, что с каждым процессом (потоком) связан маркер доступа, а с каждым объектом - дескриптор защиты. Маркер доступа в качестве параметра имеет идентификатор пользователя, а дескриптор защиты - списки прав доступа. ОС может контролировать попытки доступа, которые прямо или косвенно производятся процессами и потоками, инициированными пользователем.*

11.7.2 Пользователи и группы пользователей

Каждый пользователь (и каждая группа пользователей) системы должен иметь учетную запись (account) в базе данных системы безопасности. Учетные записи идентифицируются именем пользователя и хранятся в базе данных SAM (Security Account Manager) в разделе HKLM/SAM реестра.

Учетная запись пользователя содержат набор сведений о пользователе, такие, как имя, пароль (или реквизиты), комментарии и адрес.

Наиболее важными элементами учетной записи пользователя являются:

- список привилегий пользователя в отношении данной системы,
- список групп, в которых состоит пользователь,
- идентификатор безопасности **SID** (Security IDentifier).

Идентификаторы безопасности генерируются при создании учетной записи. Они (а не имена пользователей, которые могут не быть уникальными) служат основой для идентификации субъектов внутренними процессами ОС Windows.

Учетные записи групп, созданные для упрощения администрирования, содержат список учетных записей пользователей, а также включают сведения, аналогичные сведениям учетной записи пользователя (SID группы, привилегии члена группы и др.).

SID пользователя (и группы) является уникальным внутренним идентификатором и представляют собой структуру переменной длины с коротким заголовком, за которым следует длинное случайное число. Это числовое значение формируется из ряда параметров, причем утверждается, что вероятность появления двух одинаковых SID практически равна нулю. В частности, если удалить пользователя в системе, а затем создать его под тем же именем, то SID вновь созданного пользователя будет уже другим.

Узнать свой идентификатор безопасности пользователь легко может при помощи утилит *whoami* или getsid из ресурсов Windows.

Система хранит идентификаторы безопасности в бинарной форме, однако существует и текстовая форма представления SID. Текстовая форма используется для вывода текущего значения SID, а также для интерактивного ввода (например, в реестр).

В текстовой форме каждый идентификатор безопасности имеет определенный формат. Вначале находится префикс S, за которым следует группа чисел, разделенных дефисами. Например, SID администратора системы имеет вид: S-1-5-<домен>-500, а SID группы everyone, в которую входят все пользователи, включая анонимных и гостей - S-1-1-0.

11.7.3 Объекты. Дескриптор защиты

В ОС Windows все типы объектов защищены одинаковым образом. С каждым объектом связан *дескриптор защиты (security descriptor)*. Связь объекта с дескриптором происходит в момент создания объекта.

Дескриптор защиты (см. рис. 11.8) содержит:

- SID владельца объекта,
- SID групп для данного объекта?

– два указателя на списки DACL (Discretionary ACL) и SACL (System ACL) контроля доступа.

DACL и SACL содержат разрешающие и запрещающие доступ списки пользователей и групп, а также списки пользователей, чьи попытки доступа к данному объекту подлежат аудиту.

Структура каждого ACL списка проста. Это набор записей ACE (Access Control Entry), каждая запись содержит SID и перечень прав, предоставленных субъекту с этим SID.

На примере, изображенном на рис. 11.8, владелец файла Александр имеет право на все операции с данным файлом, всем остальным обычно дается только право на чтение, а Павлу запрещены все операции. Таким образом, **список DACL** описывает все права доступа к объекту. Если этого списка нет, то все пользователи имеют все права; если этот список существует, но он пустой, права имеет только его владелец.

Рис. 11.8. Структура дескриптора защиты для файла

В списке ACL есть записи ACE двух типов - разрешающие и запрещающие доступ. Разрешающая запись содержит SID пользователя или группы и битовый массив (access mask), определяющий набор операций, которые процессы, запускаемые этим пользователем, могут выполнять с данным объектом. Запрещающая запись действует аналогично, но в этом случае процесс не может выполнять перечисленные операции.

Кроме списка DACL дескриптор защиты включает также список SASL, который имеет такую же структуру, что и DACL, то есть состоит из таких же ACE записей, только вместо операций, регламентирующих доступ к объекту, в нем перечислены операции, подлежащие аудиту. В примере на рис. 11.8

операции с файлом процессов, запускаемых Сергеем, описанные в соответствующем битовом массиве будут регистрироваться в системном журнале.

11.7.4 Субъекты безопасности. Процессы, потоки. Маркер доступа

Так же как и объекты, субъекты должны иметь отличительные признаки - контекст пользователя, для того, чтобы система могла контролировать их действия. Сведения о контексте пользователя хранятся в маркере (употребляются также термины "токен", "жетон") доступа.

При интерактивном входе в систему пользователь обычно вводит свое имя и пароль. Система (процедура Winlogon) по имени находит соответствующую учетную запись, извлекает из нее необходимую информацию о пользователе, формирует список привилегий, ассоциированных с пользователем и его группами, и все это объединяет в структуру данных, которая называется маркером доступа.

Маркер также хранит некоторые параметры сессии, например, время окончания действия маркера. Таким образом, именно маркер является той визитной карточкой, которую субъект должен предъявить, чтобы осуществить доступ к какому-либо объекту.

Основные компоненты маркера доступы показаны на рис. 11.9.

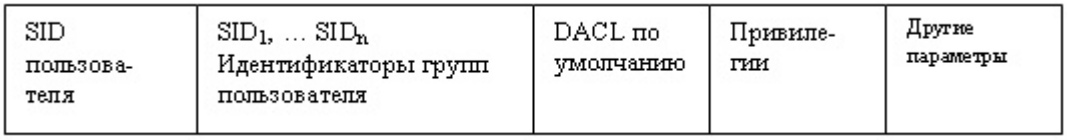


Рис. 11.9. Основные компоненты маркера доступа

Включая в маркер информацию о защите, в частности, DACL, Windows упрощает создание объектов со стандартными атрибутами защиты. Как уже говорилось, если процесс не позаботится о том, чтобы явным образом указать атрибуты безопасности объекта, на основании списка DACL, присутствующего в маркере, будут сформированы права доступа к объекту по умолчанию.

11.7.5 Проверка прав доступа

После формализации атрибутов защиты субъектов и объектов можно перечислить основные этапы проверки прав доступа (рис. 11.10).

Этапов проверки довольно много. Наиболее важные этапы из них:

– *Если SID субъекта совпадает с SID владельца объекта и запрашиваются стандартные права доступа, то доступ предоставляется независимо от содержимого DACL.*

– *Далее система последовательно сравнивает SID каждого ACE из DACL с SID маркера. Если обнаруживается соответствие, выполняется сравнение маски доступа с проверяемыми правами. Для запрещающих ACE даже при частичном совпадении прав доступ немедленно отклоняется. Для успешной проверки разрешающих элементов необходимо совпадение всех прав.*

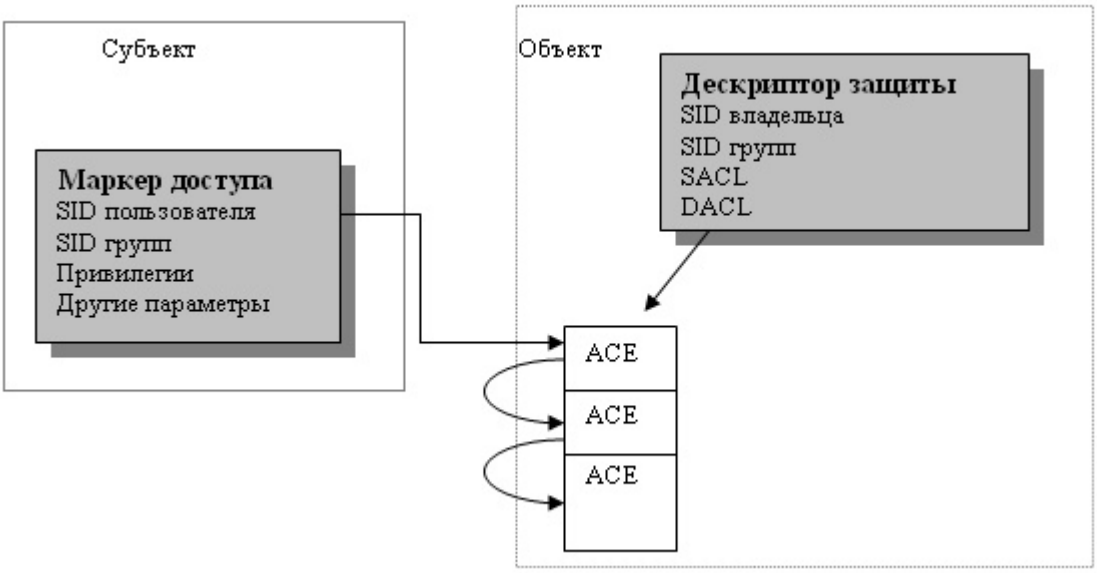


Рис. 11.10. Пример проверки прав доступа к защищенному объекту

Очевидно, что для процедуры проверки важен порядок расположения ACE в DACL. Поэтому Microsoft предлагает так называемый предпочтительный порядок размещения ACE. Например, для ускорения рекомендуется размещать запрещающие элементы перед разрешающими.

11.7.6 Основные компоненты системы безопасности ОС Windows

Система контроля дискреционного доступа - центральная концепция защиты ОС Windows, однако перечень задач, решаемых для обеспечения безопасности, этим не исчерпывается. В данном разделе будут проанализированы структура, политика безопасности и API системы защиты.

Изучение структуры системы защиты помогает понять особенности ее функционирования. Несмотря на слабую документированность ОС Windows по косвенным источникам можно судить об особенностях ее функционирования.

Система защиты ОС Windows состоит из следующих компонентов (см. рис. 11.11).

– **Процедура регистрации (Logon Processes),** которая обрабатывает запросы пользователей на вход в систему. Она включают в себя начальную интерактивную процедуру, отображающую начальный

диалог с пользователем на экране, и удаленные процедуры входа, которые позволяют удаленным пользователям получить доступ с рабочей станции сети к серверным процессам Windows NT. Процесс Winlogon реализован в файле Winlogon.exe и выполняется как процесс пользовательского режима. Стандартная библиотека аутентификации Gina реализована в файле Msgina.dll.

- **Подсистема локальной авторизации (Local Security Authority, LSA)**, которая гарантирует, что пользователь имеет разрешение на доступ в систему. Этот компонент - центральный для системы защиты Windows NT. Он порождает маркеры доступа, управляет локальной политикой безопасности и предоставляет интерактивным пользователям аутентификационные услуги. LSA также контролирует политику аудита и ведет журнал, в котором сохраняются сообщения, порождаемые диспетчером доступа. Основная часть функциональности реализована в Lsasrv.dll.
- **Менеджер учета (Security Account Manager, SAM)**, который управляет базой данных учета пользователей. Эта база данных содержит информацию обо всех пользователях и группах пользователей. Данная служба реализована в Samsrv.dll и выполняется в процессе Lsass.
- **Диспетчер доступа (Security Reference Monitor, SRM)**, проверяющий, имеет ли пользователь право на доступ к объекту и на выполнение тех действий, которые он пытается совершить. Этот компонент обеспечивает легализацию доступа и политику аудита, определяемые LSA. Он предоставляет услуги для программ супервизорного и пользовательского режимов, чтобы гарантировать, что пользователи и процессы, осуществляющие попытки доступа к объекту, имеют необходимые права. Данный компонент также порождает сообщения службы аудита, когда это необходимо. Это компонент исполнительной системы: Ntoskrnl.exe.

Все компоненты активно используют базу данных Lsass, содержащую параметры политики безопасности локальной системы, которая хранится в разделе HKLM\SECURITY реестра.

Реализация модели дискреционного контроля доступа связана с наличием в системе одного из ее важнейших компонентов - **монитора безопасности**. Это особый вид субъекта, который активизируется при каждом доступе и в состоянии отличить легальный доступ от нелегального и не допустить последний. Монитор безопасности входит в состав диспетчера доступа (SRM), который, согласно описанию, обеспечивает также управление ролевым и привилегированным доступом.

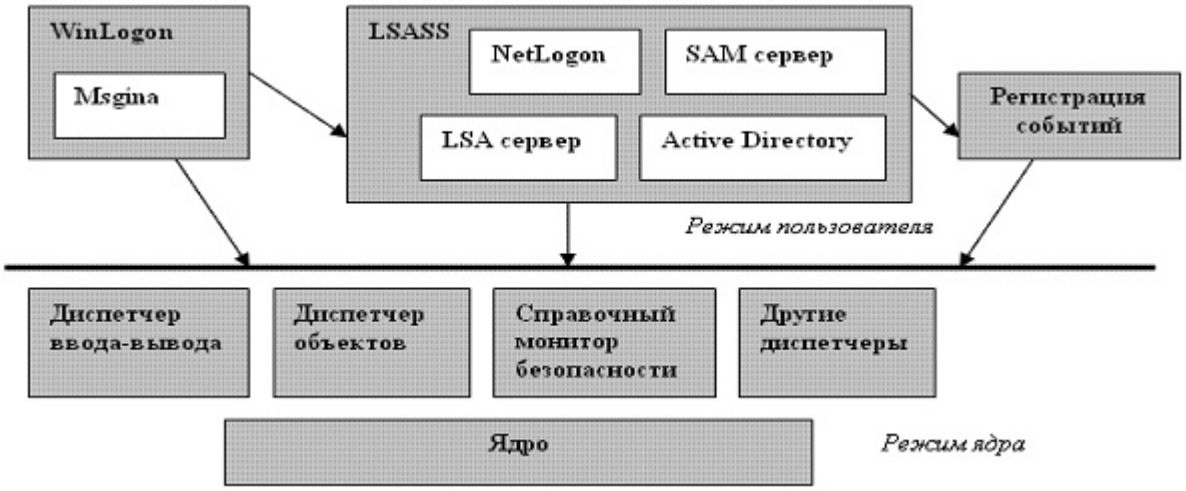


Рис. 11.11. Структура системы безопасности ОС Windows

11.7.7 Политика безопасности

Система безопасности ОС Windows отвечает требованиям класса С2 "оранжевой" книги и требованиям стандарта Common Criteria, которые составляют основу *политики безопасности системы*.

Политика безопасности подразумевает ответы на следующие вопросы:

- какую информацию защищать,
- какого рода атаки на безопасность системы могут быть предприняты,
- какие средства использовать для защиты каждого вида информации.

Требования, предъявляемые к системе защиты, таковы.

- Каждый пользователь должен быть идентифицирован уникальным входным именем и паролем для входа в систему. Доступ к компьютеру предоставляется лишь после аутентификации. Должны быть предприняты меры предосторожности против попытки против применения фальшивой программы регистрации (механизм безопасной регистрации).
- Система должна быть в состоянии использовать уникальные идентификаторы пользователей, чтобы следить за их действиями (управление избирательным или дискреционным доступом). Владелец ресурса (например, файла) должен иметь возможность контролировать доступ к этому ресурсу.
- Управление доверительными отношениями. Необходима поддержка наборов ролей (различных типов учетных записей). Кроме того, в системе должны быть средства для управления привилегированным доступом.
- ОС должна защищать объекты от повторного использования. Перед выделением новому пользователю все объекты, включая память и файлы, должны быть проинициализированы.

- Системный администратор должен иметь возможность учета всех событий, относящихся к безопасности (аудит безопасности).
- Система должна защищать себя от внешнего влияния или навязывания, такого, как модификация загруженной системы или системных файлов, хранимых на диске.

Надо отметить, что, в отличие от большинства операционных систем, ОС Windows была изначально спроектирована с учетом требований безопасности, и это является ее несомненным достоинством. Посмотрим теперь, как в рамках данной архитектуры обеспечивается выполнение требований политики безопасности.

11.7.8 Ролевой доступ. Привилегии

С целью гибкого управления системной безопасностью в ОС Windows реализовано **управление доверительными отношениями (trusted facility management)**, которое требует поддержки набора ролей (различных типов учетных записей) для разных уровней работы в системе. В системе имеется **управление привилегированным доступом**, то есть функции администрирования доступны только одной группе учетных записей - Administrators (Администраторы).

В соответствии со своей ролью каждый пользователь обладает определенными ***привилегиями*** **и правами** на выполнение различных операций в отношении системы в целом, например, право на изменение системного времени или право на создание страничного файла. Аналогичные права в отношении конкретных объектов называются ***разрешениями***. И права, и привилегии назначаются администраторами отдельным пользователям или группам как часть настроек безопасности.

Каждая привилегия имеет два текстовых представления: дружественное имя, отображаемое в пользовательском интерфейсе Windows, и программное имя, используемое приложениями, а также Luid - внутренний номер привилегии в конкретной системе. Помимо привилегий в Windows имеются близкие к ним права учетных записей.

Важно, что даже администратор системы по умолчанию обладает далеко не всеми привилегиями. Это связано с **принципом предоставления минимума привилегий**. В каждой новой версии ОС Windows, в соответствии с этим принципом, производится ревизия перечня предоставляемых каждой группе пользователей привилегий, и общая тенденция состоит в уменьшении их количества. С другой стороны общее количество привилегий в системе растет, что позволяет проектировать все более гибкие сценарии доступа.

Назначение и отзыв привилегий - прерогатива **локального администратора безопасности** LSA (Local Security Authority), поэтому,

чтобы программно назначать и отзывать привилегии, необходимо применять функции LSA.

Локальная политика безопасности системы означает наличие набора глобальных сведений о защите, например, о том, какие пользователи имеют право на доступ в систему, а также о том, какими они обладают правами. Поэтому говорят, что каждая система, в рамках которой действует совокупность пользователей, обладающих определенными привилегиями в отношении данной системы, является объектом политики безопасности. Объект политики используется для контроля базы данных LSA. Каждая система имеет только один объект политики, который создается администратором LSA во время загрузки и защищен от несанкционированного доступа со стороны приложений.

Управление привилегиями пользователей включает в себя задачи перечисления, задания, удаления, выключение привилегий и ряд других.

12. ОСОБЕННОСТИ ПОСТРОЕНИЯ ОПЕРАЦИОННЫХ СИСТЕМ СЕМЕЙСТВА UNIX

12.1 История создания ОС семейства Unix

Первая система UNIX была разработана в конце 1960-х — начале 1970-х годов в подразделении Bell Labs компании AT&T. С тех пор было создано большое количество различных UNIX-систем. Юридически лишь некоторые из них имеют полное право называться «UNIX»; остальные же, хотя и используют сходные концепции и технологии, объединяются термином «UNIX-подобные» (англ. Unix-like). Для краткости под UNIX-системами будем подразумевать как истинные UNIX, так и UNIX-подобные ОС. Некоторые отличительные признаки UNIX-систем включают в себя:

- использование простых текстовых файлов для настройки и управления системой;
- широкое применение командной строки;
- представление устройств и некоторых средств межпроцессного взаимодействия как файлов;
- использование конвейеров из нескольких программ, каждая из которых выполняет одну задачу.

В настоящее время UNIX используются в основном на серверах, а также как встроенные системы для различного оборудования. На рынке ОС для рабочих станций и домашнего применения UNIX уступили другим операционным системам, в первую очередь Microsoft Windows. UNIX-системы имеют большую историческую важность, поскольку благодаря им распространились некоторые популярные сегодня концепции и подходы в области ОС и программного обеспечения.

12.1.1 Первые UNIX

В 1957 году в Bell Labs была начата работа по созданию операционной системы для собственных нужд. Под руководством Виктора Высотского (русского по происхождению) была создана система BESYS.

В 1964 году появились компьютеры третьего поколения, для которых возможности BESYS уже не подходили. Высотский и его коллеги приняли решение не разрабатывать новую собственную операционную систему, а подключиться к совместному проекту General Electric и Массачусетского технологического института Multics. Телекоммуникационный гигант AT&T, в состав которого входили Bell Labs, оказал проекту существенную поддержку, но в 1969 году вышел из проекта, поскольку он не приносил никаких финансовых выгод.

Первоначально UNIX была разработана в конце 1960-х годов сотрудниками Bell Labs, в 1969 году Кен Томпсон, стремясь реализовать идеи, что были положены в основу ОС MULTICS, но на более скромном аппаратном обеспечении (DEC PDP-7), написал первую версию новой операционной системы, а Брайан Керниган придумал для неё название — UNICS (UNIplexed Information and Computing System). Позже это название сократилось до UNIX. В 1971 году вышла версия для PDP-11, наиболее успешного семейства миникомпьютеров 1970-х (в СССР оно известно как СМ ЭВМ). Эта версия получила название «первая редакция» (Edition 1) и была первой официальной версией. Системное время все реализации UNIX отсчитывают с 1 января 1970.

Первые версии UNIX были написаны на ассемблере и не имели встроенного компилятора с языка высокого уровня. Примерно в 1969 году Кен Томпсон при содействии Дениса Ритчи разработал и реализовал язык Би (B). В 1969—1973 годах на основе Би был разработан компилируемый язык, получивший название Си (C).

В 1973 году вышла третья редакция UNIX, со встроенным компилятором с Си. 15 октября того же года появилась четвёртая редакция, с переписанным на Си системным ядром, а в 1975 — пятая редакция, полностью переписанным на Си. С 1974 года UNIX стал бесплатно распространяться среди университетов и академических учреждений. С 1975 года началось появление новых версий, разработанных за пределами Bell Labs, и рост популярности системы.

К 1978 г. система была установлена более чем на 600 машинах, прежде всего, в университетах. Седьмая редакция была последней единой версией UNIX. Именно в ней появился близкий к современному интерпретатор командной строки Bourne shell.

12.1.2 Создание BSD UNIX

С 1978 года начинает свою историю BSD UNIX, созданный в университете Беркли. Его первая версия, была основана на шестой редакции. В 1979 выпущена новая версия, названная 3BSD, основанная на седьмой редакции. BSD поддерживал такие полезные свойства, как виртуальную память и замещение страниц по требованию. Автором BSD был Билл Джой.

В начале 1980-х компания AT&T, которой принадлежали Bell Labs, осознала ценность UNIX и начала создание коммерческой версии UNIX. Эта версия, поступившая в продажу в 1982 году, носила название UNIX System III и была основана на седьмой версии системы.

Важной причиной раскола UNIX стала реализация в 1980 году стека протоколов TCP/IP. До этого межмашинное взаимодействие в UNIX пребывало в зачаточном состоянии — наиболее существенным способом

связи был UUCP (средство копирования файлов из одной UNIX-системы в другую, изначально работавшее по телефонным сетям с помощью модемов).

Было предложено два интерфейса программирования сетевых приложений:

- Berkley sockets
- интерфейс транспортного уровня TLI (англ. Transport Layer Interface).

Интерфейс Berkley sockets был разработан в университете Беркли и использовал стек протоколов TCP/IP, разработанный там же. TLI был создан AT&T в соответствии с определением транспортного уровня модели OSI и впервые появился в системе System V версии 3. Хотя эта версия содержала TLI и потоки, первоначально в ней не было реализации TCP/IP или других сетевых протоколов, но подобные реализации предоставлялись сторонними фирмами. Реализация TCP/IP официально и окончательно была включена в базовую поставку System V версии 4. Это, как и другие соображения (по большей части, рыночные), вызвало окончательное размежевание между двумя ветвями UNIX — BSD (университета Беркли) и System V (коммерческая версия от AT&T). Впоследствии, многие компании, лицензировав System V у AT&T, разработали собственные коммерческие разновидности UNIX, такие, как AIX, HP-UX, IRIX, Solaris.

В середине 1983 года была выпущена BSD версии 4.2, поддерживающая работу в сетях Ethernet и Arpanet. Система стала весьма популярной. Между 1983 и 1990 годами в BSD были добавлено много новых возможностей, таких как отладчик ядра, сетевая файловая система NFS, виртуальная файловая система VFS, и существенно улучшены возможности работы с файловыми сетями.

Тем временем AT&T выпускала новые версии своей системы, названной System V. В 1983году была выпущена версия 1 (SVR1 — System V Release 1), включавшая полноэкранный текстовый редактор vi, библиотеку curses, буферизацию ввода-вывода, кеширование inode. Версия 2 (SVR2), выпущенная в 1984 году, реализовывала монопольный доступ к файлам (file locking), доступ к страницам по требованию (demand paging), копирование при записи (copy-on-write). Версия 3 вышла в 1987 году и включала, среди прочего, TLI, а также систему поддержки удалённых файловых систем RFS. Версия 4 (SVR 4), разработанная в сотрудничестве с фирмой Sun и вышедшая 18 октября 1988, поддерживала многие возможности BSD, в частности TCP/IP, сокеты, новый командный интерпретатор csh. Кроме того, там было много других добавлений, таких как символические ссылки, командный интерпретатор ksh, сетевая файловая система NFS (заимствованная у SunOS) и т. д.

Современные реализации UNIX, как правило, не являются системами V или BSD в чистом виде. Они реализуют возможности как System V, так и BSD.

12.1.3 Свободные UNIX-подобные операционные системы

В 1983 году Ричард Столлмэн объявил о создании проекта GNU — попытки создания свободной UNIX-подобной операционной системы с нуля, без использования оригинального исходного кода. Большая часть программного обеспечения, разработанного в рамках данного проекта, — такого, как GNU toolchain, Glibc (стандартная библиотека языка Си) и Coreutils — играет ключевую роль в других свободных операционных системах. Однако работы по созданию замены для ядра UNIX, необходимые для полного выполнения задач GNU, продвигались крайне медленно. В настоящее время GNU Hurd — попытка создать современное ядро на основе микроядерной архитектуры Mach — всё ещё далека от завершения.

В 1991 году, когда Линус Торвальдс опубликовал ядро Linux и привлёк помощников, использование инструментов, разработанных в рамках проекта GNU, было очевидным выбором. Операционная система GNU и ядро Linux вместе составляют ОС, известную, как GNU/Linux. Дистрибутивы этой системы (такие как Red Hat и Debian), включающие ядро, утилиты GNU и дополнительное программное обеспечение стали популярными как среди любителей, так и среди представителей бизнеса.

В результате урегулирования юридического дела, возбуждённого UNIX Systems Laboratories против университета Беркли и Berkeley Software Design Inc., было установлено, что университет может распространять BSD UNIX, в том числе и бесплатно. После этого были возобновлены эксперименты, связанные с BSD-версией UNIX. Вскоре разработка дистрибутива BSD была продолжена в нескольких направлениях одновременно, что привело к появлению проектов, известных как FreeBSD, NetBSD, OpenBSD, TrustedBSD и DragonFlyBSD.

В настоящий момент GNU/Linux и представители семейства BSD быстро отвоёвывают рынок у коммерческих UNIX-систем и одновременно проникают как на настольные компьютеры конечных пользователей, так и на мобильные и встраиваемые системы. Одним из свидетельств данного успеха служит тот факт, что, когда фирма Apple искала основу для своей новой операционной системы, она выбрала NEXTSTEP — операционную систему со свободно распространяемым ядром, разработанную фирмой NeXT и переименованную в Darwin после приобретения фирмой Apple. Данная система относится к семейству BSD и основана на ядре Mach. Применение Darwin BSD UNIX в Mac OS X делает его одной из наиболее широко используемых версий UNIX.

12.1.4 Современное состояние ОС семейства Unix

После разделения компании AT&T, товарный знак UNIX и права на оригинальный исходный код неоднократно меняли владельцев, в частности, длительное время принадлежали компании Novell.

В 1993 году Novell передала права на товарный знак и на сертификацию программного обеспечения на соответствие этому знаку консорциуму X/Open, который затем объединился с Open Software Foundation, образовав консорциум The Open Group. Он объединяет ведущие компьютерные корпорации и государственные организации, в том числе IBM, Hewlett-Packard, Sun, NASA и многие другие. Консорциум занимается разработкой открытых стандартов в области операционных систем, самым важным из которых является Single UNIX Specification, ранее известный как POSIX. С точки зрения The Open Group, название UNIX могут носить только системы, прошедшие сертификацию на соответствие Single UNIX Specification.

Эволюция UNIX систем приведена на рисунке 12.1.

Идеи, заложенные в основу UNIX, оказали огромное влияние на развитие компьютерных операционных систем. В настоящее время UNIX-системы признаны одними из самых исторически важных ОС.

UNIX была написана на языке высокого уровня, а не на ассемблере (доминировавшем в то время). Она содержала значительно упрощённую, по сравнению с современными ей операционными системами, файловую модель. Файловая система включала как службы, так и устройства (такие как принтеры, терминалы и жёсткие диски) и предоставляла внешне единообразный интерфейс к ним, но дополнительные механизмы работы с устройствами (такие как IOCTL и биты доступа) не вписывались в простую модель «поток байтов».

UNIX популяризовала идею иеархической файловой системы с произвольной глубиной вложенности. Другие операционные системы того времени позволяли разбивать дисковое пространство на каталоги или разделы, но число уровней вложенности было фиксировано и, зачастую, уровень вложенности был только один. Позднее все основные фирменные операционные системы обрели возможность создания рекурсивных подкаталогов.

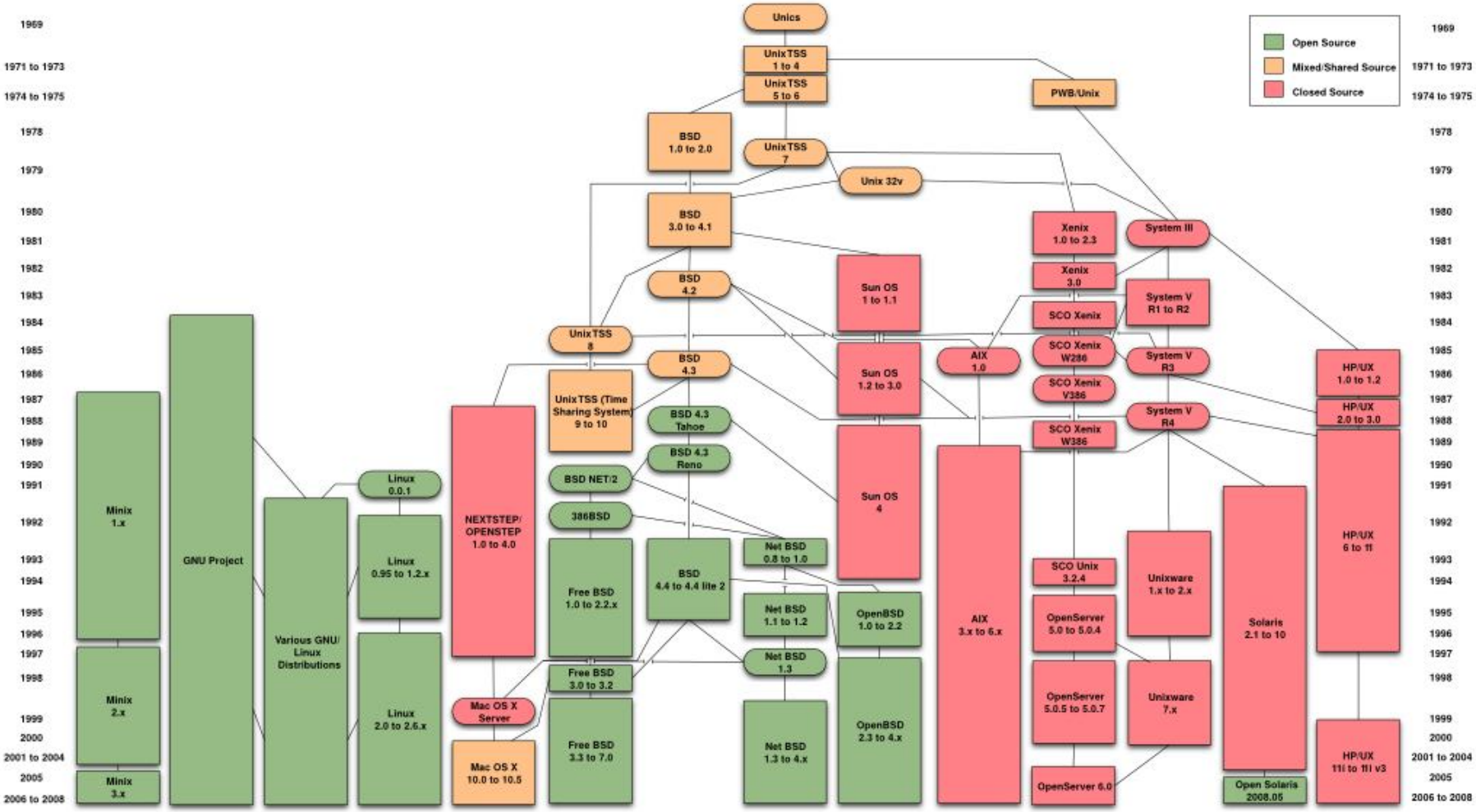


Рис. 12.1. Эволюция систем семейства UNIX

То, что интерпретатор команд стал просто одной из пользовательских программ, а в качестве дополнительных команд выступают отдельные программы, является ещё одной инновацией, популяризированной UNIX. Язык командной оболочки UNIX используется пользователем как для интерактивной работы, так и для написания скриптов, то есть не существует отдельного языка описания заданий. Так как оболочка и команды операционной системы являются обычными программами, пользователь может выбирать их в соответствии со своими предпочтениями, или даже написать собственную оболочку. Наконец, новые команды можно добавлять к системе без перекомпиляции ядра. Новый, предложенный в командной строке UNIX, способ создания цепочек программ, последовательно обрабатывающих данные, способствовал использованию параллельной обработки данных.

Существенными особенностями UNIX были полная ориентация на текстовый ввод-вывод и предположение, что размер машинного слова кратен восьми битам. Ориентация на текстовый восьмибитный байт сделала UNIX более масштабируемой и переносимой, чем другие операционные системы. Со временем текстовые приложения одержали победу и в других областях, например, на уровне сетевых протоколов, таких как Telnet, FTP, SMTP, HTTP и других.

UNIX способствовала широкому распространению регулярных выражений, которые были впервые реализованы в текстовом редакторе ed для UNIX. Возможности, предоставляемые UNIX-программам, стали основой стандартных интерфейсов операционных систем (POSIX).

Широко используемый в системном программировании язык Си, созданный изначально для разработки UNIX, превзошёл UNIX по популярности. Си был первым высокоуровневым языком, предоставляющим доступ ко всем возможностям процессора, таким как ссылки, таблицы, битовые сдвиги, приращения и т. п. Первые разработчики UNIX способствовали внедрению принципов модульного программирования и повторного использования в инженерную практику.

UNIX предоставлял возможность использования протоколов TCP/IP на сравнительно недорогих компьютерах, что привело к быстрому росту Интернета. Это, в свою очередь, способствовало быстрому обнаружению нескольких крупных уязвимостей в системе безопасности, архитектуре и системных утилитах UNIX.

Со временем ведущие разработчики UNIX разработали культурные нормы разработки программного обеспечения, которые стали столь же важны, как и сам UNIX

12.2 Цели и возможности ОС семейства UNIX

Операционная система UNIX проектировалась как инструментальная система для разработки программного обеспечения. Своей уникальностью система обязана во многом тому обстоятельству, что она была, по сути, создана всего двумя разработчиками, причем создававшие ее люди делали систему для себя, и первое время ее использовали на мини-ЭВМ с очень скромными вычислительными ресурсами. По этой причине UNIX, прежде всего, обладает простым, но очень мощным командным языком и независимой от устройств файловой системой. Поскольку при создании этой ОС использовался язык высокого уровня, на котором пишутся не только системные, но и прикладные программы (речь идет о языке С), то система и приложения, выполняющиеся в ней, получились легко переносимыми.

Первой целью при разработке этой системы было стремление сохранить простоту и обойтись минимальным количеством функций. Все реальные сложности оставлялись пользовательским программам.

Второй целью была общность. Одни и те же методы и механизмы должны были использоваться во многих случаях. Поэтому общность в UNIX-системах проявляется во многих аспектах, и в частности:

- обращения к файлам, устройствам ввода/вывода и буферам межпроцессных сообщений выполняются с помощью одних и тех же примитивов;
- одни и те же механизмы именования, присвоения альтернативных имен и защиты от несанкционированного доступа применяются к файлам с данными и директориями и устройствам;
- одни и те же механизмы работают в отношении программно и аппаратно инициируемых прерываний.

Наконец, третья цель заключалась в создании операционной среды, в которой большие задачи можно было бы решать, комбинируя существующие небольшие программы, а не разрабатывая программы заново.

К основным функциям операционной системы UNIX можно отнести следующее.

- Обработка прерываний.
- Создание и уничтожение процессов.
- Переключение процессов из одного состояния в другое.
- Диспетчеризация.
- Приостановка и активизация процессов.
- Синхронизация процессов.
- Организация взаимодействия между процессами.
- Манипулирование блоками управления процессами.

– Поддержка операции ввода-вывода.
– Поддержка операции распределения и перераспределения памяти.
– Поддержка работы файловых систем.
– Поддержка механизма вызова-возврата по обращению к процедурам.

12.3 Типовая структура ОС семейства UNIX

В структуре ОС можно выделить три основные части (рис. 12.2):

1. Самая низкоуровневая часть ОС - ядро. Оно непосредственно взаимодействует с аппаратными средствами и обеспечивает переносимость всего остального ПО на компьютеры с разным аппаратным обеспечением. Ядро предоставляет программам определенный набор системных API, с помощью которых производятся создание процессов, управление ими, их взаимодействие и синхронизация, а также файловый ввод/вывод.
2. Более высокий уровень - уровень конкретных служебных программ и языков программирования. На этом уровне система получает ресурсы через обращение к ядру ОС (т.е. по прерываниям).
3. Уровень вспомогательных процедур, интерпретаторов, компиляторов. На данной основе строятся пользовательские приложения (текстовые редакторы, графические интерфейсы и собственно приложения).

Рис. 12.2. Структура ОС UNIX

Таким образом, система UNIX состоит из ядра, демонов (сервисов) и утилит. В системе принят ряд соглашений, которые соблюдаются всеми разработчиками программ под UNIX, в частности, поддержка POSIX.

Ядро - это набор системных таблиц и подпрограмм работы с ними. В ядро также входят драйверы устройств. Ядро состоит из статической части, которая загружается при старте системы, и модулей. Модули могут динамически загружаться при старте системы или во время работы, при необходимости поддержки той или иной функции. В частности, подсистема поддержки NFS и драйверы внешних устройств оформлены в виде модулей.

Демоны - это серверные резидентные приложения, отвечающие за обработку запросов от других (клиентских) программ.

Утилиты - это программы, которые нужны для выполнения разных базовых работ в системе: копирования файлов, управления процессами, восстановления файловой системы и т.п.

Рассмотрим ядро системы. Оно позволяет всем остальным программам общаться с периферийными устройствами, регулирует доступ к файлам, управляет память и процессами. Основным достоинством ядра является строгая стандартизация системных API. За счет этого во многом достигается переносимость кода между разными версиями UNIX и абсолютно различным аппаратным обеспечением.

Рис. 12.3. Структура ядра UNIX

Подсистема управления файлами - почти единственная из всех работает с драйверами, которые являются модулями ядра. "Почти", потому что есть еще и сетевая подсистема, которая работает, например, с драйвером сетевой карты и с драйверами различных современных сетевых устройств.

Обмен данными с драйверами может проходить двумя способами:

- с помощью буфера,
- с помощью потока.

Суть первого метода заключается в том, что для информации выделяется кэш (или сверхоперативная память, как его называли раньше), в который заносится необходимый блок данных. Далее информация из кэша передается к драйверу. Драйвер - единственный элемент ядра, способный управлять периферийными устройствами. Но подсистема управления файлами может взаимодействовать с драйвером и через поток. Поток представляет собой посимвольную передачу данных драйверу. Следует отметить, что способ взаимодействия с драйвером определяется не пользователем и не приложением. Он является характеристикой того устройства, которым управляет драйвер. Очевидно, что потоковое общение позволяет взаимодействовать более оперативно, чем общение через буфер. Ведь на заполнение буфера тратится время и, следовательно, возрастает время отклика.

Подсистема управления процессами - отвечает за синхронизацию и взаимодействие процессов, распределение памяти и планирование выполнения процессов. Для всех этих целей в подсистему управления процессами включены три модуля, которые наглядно продемонстрированы на схеме.

Рассмотрим вызовы, служащие для работы с процессами:

- fork (создает новый процесса),
- exec (выполняет процесс),
- exit (завершает исполнение процесса),
- wait (один из способов синхронизации),
- brk (управляет памятью, выделенной процессу),
- signal (обработчики исключений) и др.

Модуль распределения памяти, позволяет избежать нехватки оперативной памяти. Используя механизмы свопинга и виртуальной памяти модуль выполняет очень важную функцию - он определяет какому процессу сколько выделить памяти

Диспетчер процессов (SWOPPER) – организует исполнение нескольких процессов и переключение между ними: планирование исполнения, переключение контекста, поддержку механизма приоритетов, обмен информацией между процессами и синхронизацию. могут также обмениваться между собой информацией.

На **уровне аппаратного управления** происходит обработка прерываний и связь ядра с железом.

12.4 Обработка процессов в ОС семейства UNIX

UNIX - это многопользовательская многопроцессная система, т.е. в ней может одновременно быть запущено несколько процессов от имени разных пользователей. Число процессов, которые можно одновременно запустить, ограничивается размерами таблицы процессов в ядре и другими настройками ядра.

Схемы взаимодействия между процессами соответствуют механизму сопрограмм. Использование сопрограмм упрощает логику ядра системы и требует одного выделенного процесса, который создается нестандартным образом. С него начинается работа системы после запуска. В ОС UNIX этот процесс называется "диспетчерским" ("swapper"), он не имеет пользовательской фазы.

Все процессы в ОС UNIX, кроме диспетчерского, создаются операцией "Порождение". В этой операции участвуют два процесса: порождающий и порожденный. Порождающий выполняет системный вызов (fork), в результате появляется порожденный процесс. При запуске процесс получает уникальный идентификатор процесса (Process IDentifier, PID), по которому он становится доступен другим процессам и планировщику.

На рисунке 12.4 представлена блок-схема жизненного цикла процесса в ОС UNIX.

Для создания системного процесса используется системный вызов fork (разветвление), в результате которого получаются два идентичных процесса, называемые родительский процесс и порожденный (дочерний) процесс. Они не имеют общей первичной памяти, но совместно используют все открытые файлы. Для уничтожения процесса имеется вызов EXIT, который завершает работу данного процесса и передает код возврата (завершения) родительскому процессу. Сегменты данного процесса уничтожаются. Остается структура пользования для родительского процесса. Когда родительский процесс подучил информацию об уничтожении порожденного им процесса, тогда уничтожается структура пользования и освобождается место в таблице процессов.

В большинстве систем UNIX реализована возможность выполнять несколько параллельных подпроцессов внутри процесса. Эти подпроцессы называют потоками (threads).

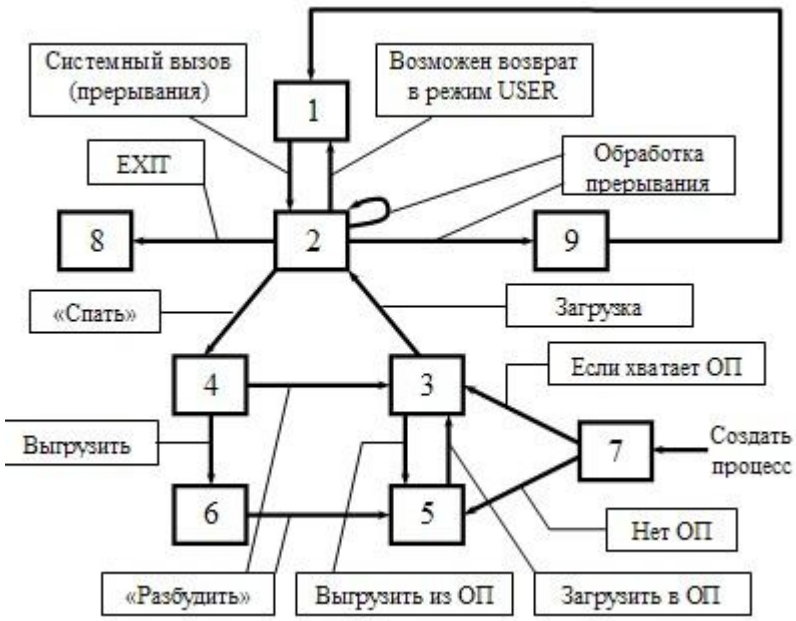


Рис.12.4. Блок схема жизненного цикла процесса в ОС UNIX

Пояснения к схеме : 1 - процесс выполняет инструкции программы (USER RUNNING); 2 - процесс выполняет системный вызов (KERNEL RUNNING); 3 - процесс готов к выполнению (READY_TO_RUN); 4 - процесс "спит" и ждет события (ASLEEP_IN_MEMORY); 5 - процесс готов к выполнению, но он находится на внешнем носителе (т.е. выгружен) (READY_TO_ RUN, SWAPPED); 6 - процесс "спит" и находится на внешнем носителе (SLEEP, SWOPPED); 7 - процесс только что создан (процесс родился) (CREATED); 8 - процесс уничтожен, завершен (ZOMBIE); 9 - процесс прерван по приоритету (PREEMPTED).

Каждый процесс в UNIX работает в своем собственном адресном пространстве, поэтому сбой в работе одного процесса никак не влияет на работу других. Подсистема виртуальной памяти, являющаяся частью ядра, запрещает процессам обращаться к чужим адресным пространствам.

Адресное пространство процесса состоит из 3 сегментов:

- текстового сегмента (инструкции);
- сегмент данных;
- сегмент стека.

Одной из функций ядра является планирование процессов, т.е. передача управления от одного процесса к другому. Для этого в ядре есть отдельная подпрограмма, называемая планировщиком задач. Процессы получают управление от планировщика задач в соответствии со своим приоритетом. Планировщик задач через определенное количество микросекунд решает, следует ли передать управление следующему в очереди процессу. Распределением ресурсов между процессами занимается так же ядро ОС.

Процесс может находится: в режиме пользователя или в режиме системы. Ядро представляет собой отдельный процесс, выполняющийся с наивысшим приоритетом.

Управление процессами осуществляется в ОС UNIX с помощью двух структур.

1. PROC-STRUCTRE (блок управления процессом). Составляющие блока:

- состояние процесса;
- размер и адрес процесса;
- кому принадлежит процесс;
- идентификация процесса;
- канал ожидания;
- поле сигналов;
- таймер и счетчик используемого времени.

2. USER-STRUCTRE (структура использования) - содержит информацию о процессе, которая должна быть доступна только на уровне исполнения. Содержание структуры:

- параметры ввода-вывода (I/O), т.е. адреса буферов и т.д.;
- окружение в файловой системе (текущий каталог, коренной каталог);
- таблица открытых файлов;
- код возврата, номера ошибок;
- поле сигналов (информация, как надо реагировать на сигнал).

На рисунке 12.5 приведена таблица связи между структурами.

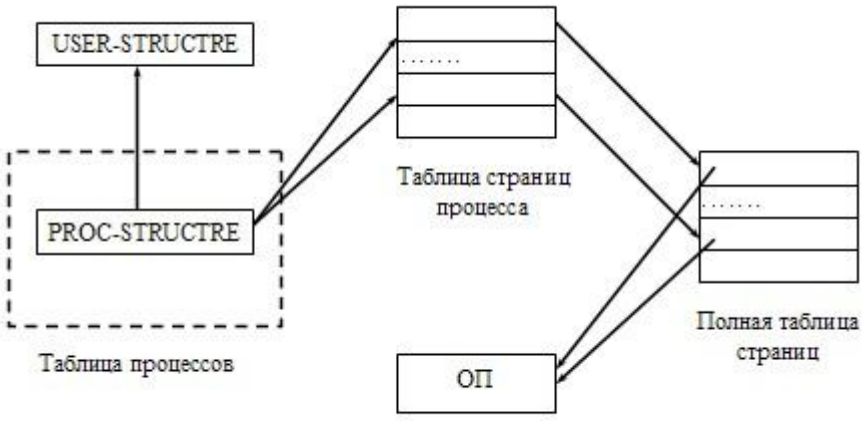


Рис.12.5. Таблица связи между структурами.

В ОС UNIX нет разницы между задачами и заданиями. Работы в системе представлены множеством конкурирующих процессов. Процесс

строго последователен, нельзя выполнять асинхронные действия внутри процесса. Даже операция I/O не может быть выполнена асинхронно.

Надо отметить, что в процесс диспетчеризации процессов ядро UNIX/Linux обрабатывает два вида исключений, которые обычно называют "oops" и "panic". Почти в каждой операционной системе "panic" происходит в тех случаях, когда ядро обнаруживает серьезную неисправность. Если система каким-либо образом повредила сама себя, ей требуется остановиться немедленно, пока она не произведет необратимых критических изменений (типа уничтожения файловой системы). Везде, где только возможно, UNIX/Linux пытается детектировать проблему и справиться с ней без остановки всей системы. Например, многие ситуации типа "oops" приводят к завершению процесса, который нормально запустился, но потом зациклил систему. Бывают, однако, ситуации, когда все настолько плохо, что полная паника является наилучшим выходом. Считается, что пользователи стабильных версий ядра не должны встречать ни "паник", ни "oops". Но в реальном мире они иногда происходят.

12.5 Организация пользователей в ОС семейства UNIX

Каждый пользователь в UNIX имеет свою собственную учетную запись пользователя (account), которая содержит имя пользователя, пароль, идентификатор пользователя (UID), идентификатор главной группы пользователя (GID), описание пользователя, его домашний каталог и путь к командному процессору, который следует запустить при интерактивном входе пользователя в систему.

Пользователь, работающий в UNIX, имеет уникальное имя пользователя и уникальный идентификатор. Идентификатор пользователя (User ID, UID) - это целое число от 0 до 2147483647. Обычные пользователи в Solaris имеют идентификатор в диапазоне от 100 до 60000.

Пользователю не надо знать свой идентификатор, потому что он используется только системой, а для входа в систему пользователь указывает свое имя (username). Пользователи объединены в группы. Каждая группа имеет свое имя и уникальный идентификатор (Group ID, GID). В группе может быть сколько угодно пользователей, и каждый из них может быть участником любого количества групп. Однако у каждого пользователя есть главная группа - она указывается в свойствах любого файла, который создает пользователь. Идентификатор группы имеет значение от 100 до 60000, если только это не специальная группа. Для специальных (предопределенных) групп зарезервирован диапазон от 0 до 99.

Пользователей объединяют в группы для того, чтобы было удобнее администрировать систему.

Концепция прав доступа в UNIX требует объединять пользователей в группы всегда, когда нужно предоставить одинаковые права доступа к файлам или каталогам группе людей.

Домашний каталог и путь к командному процессору играют роль только при интерактивном входе пользователя в систему. В UNIX каждый пользователь может работать с системой как непосредственно (набирая команды ОС на клавиатуре), так и обращаясь через сеть к тем или иным службам, запущенным на компьютере под UNIX.

При установке системы автоматически создаются предопределенные пользователи и группы. Предопределенные группы и пользователи требуются для того, чтобы от их имени работали системные службы, и в то же время доступ к файлам этих служб был ограничен для всех остальных. Кроме того, это делается для того, чтобы было проще управлять правами доступа к системным файлам.

Один из предопределенных пользователей - это пользователь root с UID, равным нулю. Пользователь с таким UID называется суперпользователем (superuser) или привилегированным пользователем и всегда имеет имя root. Он имеет неограниченные права на доступ к любому объекту в системе. Суперпользователь, является как правило системным администратором системы и отвечает за безопасность ОС, ее стабильную работу, добавление и удаление пользователей, регулярное резервное копирование и т.д.

12.4 Работа с файловыми системами

В разных вариантах UNIX используются разные файловые системы, причем часто поддерживается несколько разных файловых систем. Например, Linux умеет работать с ext2, ext3 (это ее родные файловые системы), UFS, HPFS, NTFS, FAT и другими. Solaris, и некоторые другие ОС UNIX, использует файловую систему типа UFS.

Файловая система UNIX характеризуется:

- иерархической структурой,
- согласованной обработкой массивов данных,
- возможностью создания и удаления файлов,
- динамическим расширением файлов,
- защитой информации в файлах,
- трактовкой периферийных устройств (например, таких как терминалы и принтеры) как файлов.

12.4.1 Организация разделов

Одной из целей разделения на разделы является повышение сохранности данных на случай непредвиденных происшествий. Путем разделения жесткого диска на разделы, данные могут быть сгруппированы и разобщены. Когда происходит авария, повреждаются данные только одного раздела, а данные других разделов скорее всего уцелеют. Даже журналируемая файловая система обеспечивает только защиту данных в случае сбоя питания и неожиданного отключения устройств хранения. Она не защищает ваши данные от испорченных блоков и логических ошибок в файловой системе.

Есть два вида основных разделов в системе UNIX:

1. *раздел с данными*: обычные данные системы Linux, включая *корневой раздел*, содержащий все данные для старта и запуска системы;
2. *раздел подкачки*: расширение физической памяти компьютера, представляет собой дополнительную память на жестком диске.

Пространство для подкачки (обозначается как swap) доступно только для самой системы, и скрыто при обычной работе. Раздел подкачки - это механизм, который обеспечивает, как и на обычных системах UNIX, продолжение вашей работы, что бы ни случилось.

Наряду с указанными двумя, UNIX поддерживает множество других типов файловых систем, такие как Reiser, JFS, NFS, FATxx и многие другие файловые системы, изначально доступные на других (проприетарных) операционных системах.

Большинство систем UNIX содержат корневой раздел, один или несколько разделов с данными, и один или несколько разделов подкачки. Системы в смешанных средах, могут содержать разделы данных других систем, такие как разделы файловых системам FAT или VFAT с данными ОС Windows.

Корневой раздел обозначается одиночной косой чертой, «/» и содержит системные конфигурационные файлы, большинство основных команд и серверные программы, системные библиотеки, некоторое временное пространство и домашний каталог пользователя с правами администратора.

Во многих дистрибутивах UNIX ядро находится на отдельном разделе, поскольку это самый важный файл вашей системы. В этом случае такой раздел содержащий ваше ядро (ядра) ОС и сопутствующие файлы данных монтируется в точку /boot.

Остаток жесткого диска(ов) обычно делится на разделы данных обычно по следующему принципу:

- раздел для пользовательских программ (*/usr*)
- раздел, содержащий персональные данные пользователей (*/home*)
- раздел для хранения временных данных, таких как очереди печати и почты (*/var*)
- раздел для дополнительного программного обеспечения (*/opt*)

Все разделы подключаются к системе через точки монтирования. Точка монтирования определяет место расположения конкретных данных в файловой системе. Во время запуска системы, автоматически монтируются все разделы, которые описаны в файле /etc/fstab.

На сервере системные данные стремятся отделить от пользовательских данных. Программы различных служб хранятся отдельно от данных, которые они обрабатывают. На таких системах создаются различные разделы:

- раздел со всеми данными, необходимыми для загрузки машины;
- раздел с конфигурационными данными и серверными программами;
- один или несколько разделов, содержащих серверные данные, такие как таблицы базы данных, почта пользователей, FTP-архив и т.д.;
- раздел с пользовательскими программами и приложениями;
- один или несколько разделов для конкретных пользовательских файлов (домашние каталоги);
- один или несколько разделов подкачки (виртуальная память).

12.4.2 Организация древа файловой системы

Файловая система организована в виде древовидной структуры с одной исходной вершиной, которая называется *корневой директорией* или "корнем" файловой системы (записывается: "/"). Этот каталог, содержит все основные каталоги и файлы.

Имя пути поиска состоит из компонент, разделенных между собой наклонной чертой (/); каждая компонента представляет собой набор символов, составляющих имя вершины (файла), которое является уникальным для каталога (предыдущей компоненты), в котором оно содержится. Полное имя пути поиска начинается с указания наклонной черты и идентифицирует файл (вершину), поиск которого ведется от корневой вершины дерева файловой системы с обходом тех ветвей дерева файлов, которые соответствуют именам отдельных компонент.

Вариант (на примере Linux Red Hat) организации файловой системы UNIX приведен на рисунке 12.6. В зависимости от системного администратора, операционной системы и назначения UNIX-машины,

структура может меняться, и каталоги по желанию могут быть опущены или добавлены. Даже не обязательно соответствие имен, они лишь соглашение.

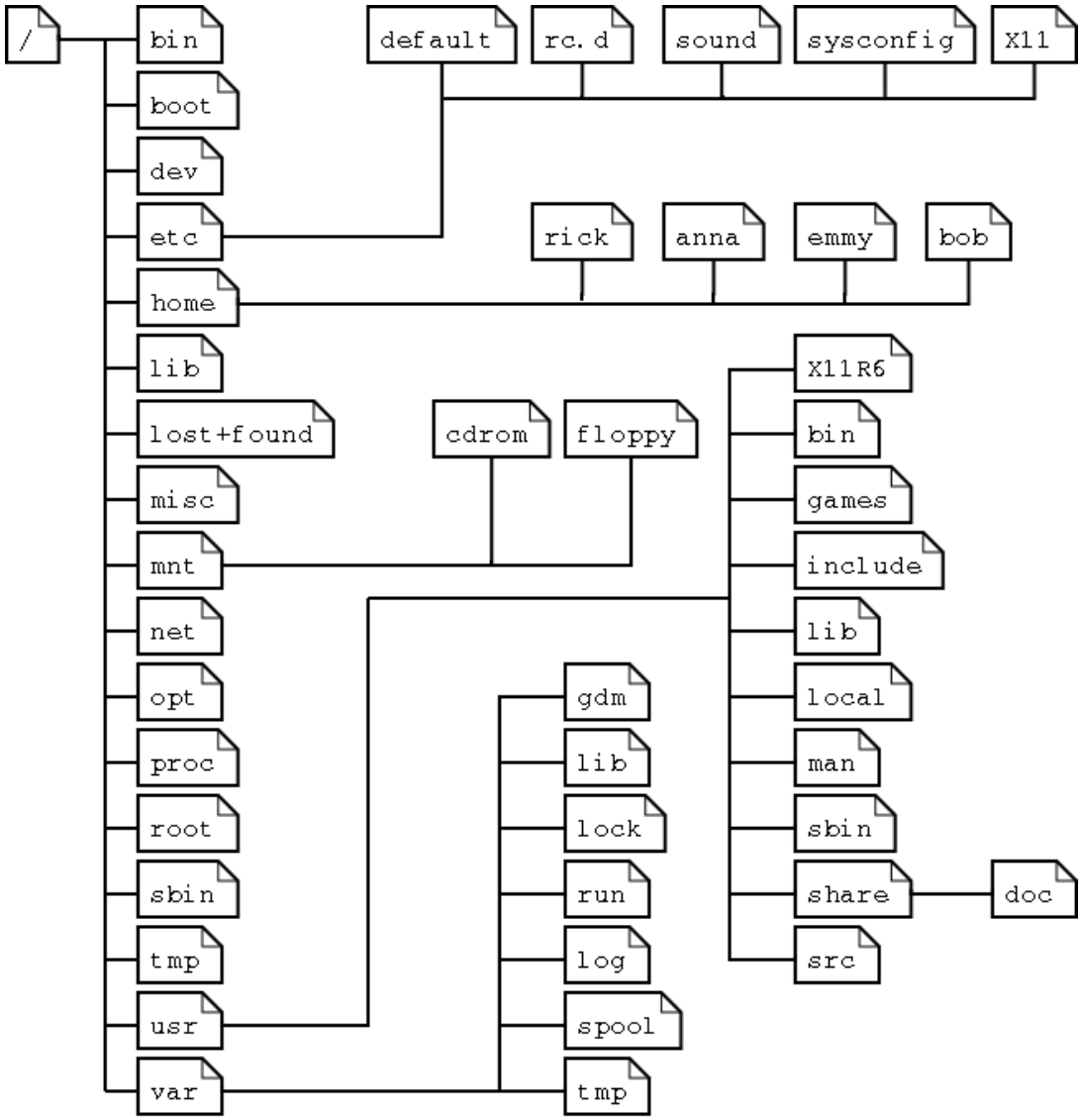


Рис. 12.6. Пример древовидной структуры файловой системы UNIX

Большинство каталогов в файловой системе UNIX хранит специфические данные или содержит отдельные компоненты системы. Назначение основных системных каталогов в файловой системе UNIX приведено в таблице 12.1.

Таблица 12.1 - Назначение основных системных каталогов в файловой системе UNIX

Директория	Содержимое
/bin	Общие программы для совместного использования системой, системным администратором и пользователями.
/boot	Загрузочные файлы и ядро, vmlinuz. В некоторых последних дистрибутивах также данные grub. Grub – это большой единый загрузчик, который представляет собой попытку избавиться от многих различных загрузчиков известных нам на сегодняшний день.
/dev	Содержит ссылки на все периферийные устройства, которые представлены файлами с особыми свойствами.
/etc	Большинство важных системных файлов конфигурации находятся в /etc, этот каталог содержит данные, аналогичные Панели Управления Windows
/home	Домашние каталоги обычных пользователей.
/initrd	Информация для загрузки. (в некоторых дистрибутивах)
/lib	Файлы библиотек, включает файлы для всех разновидностей программ, необходимых системе и пользователям.
/lost+found	Каждый раздел имеет lost+found в его верхней директории. Здесь находятся файлы, которые были спасены во время сбоев.
/misc	Для разных целей.
/mnt	Стандартные точки монтирования для внешних файловых систем, например, CD-ROM'a или цифровой камеры.
/net	Стандартные точки монтирования для удаленных файловых систем
/opt	Как правило, содержит дополнительное ПО и ПО третьих сторон.
/proc	Виртуальная файловая система, содержащая информацию о системных ресурсах. Более подробная информация о назначении файлов в proc можно получить, введя команду man proc в окне терминала. Файл proc.txt рассматривает виртуальную файловую систему в деталях.
/root	Домашняя директория суперпользователя root.
/sbin	Программы для использования системой и системным администратором.
/tmp	Временное место для использования системой, которое очищается после перезагрузки.
/usr	Программы, библиотеки, документация и т.д. для всех пользовательских программ.
/var	Место хранения всех изменяемых и временных файлов, созданных пользователями, такие как log-файлы, почтовые очереди, the print spooler area, место для временного хранения файлов, загружаемых из Интернета, или сохранения образа CD перед записью.

12.4.3 Каталоги и файлы

Каталоги похожи на обычные файлы в одном отношении; система представляет информацию в каталоге набором байтов, но эта информация включает в себя имена файлов в каталоге в объявленном формате для того, чтобы операционная система и программы, такие как ls (выводит список имен и атрибутов файлов), могли их обнаружить.

Для пользователя система UNIX трактует устройства так, как если бы они были файлами. Устройства, для которых назначены специальные файлы устройств, становятся вершинами в структуре файловой системы. Обращение программ к устройствам имеет тот же самый синтаксис, что и обращение к обычным файлам; семантика операций чтения и записи по отношению к устройствам в большой степени совпадает с семантикой операций чтения и записи обычных файлов. Способы защиты устройств совпадает со способом защиты обычных файлов: путем соответствующей установки битов разрешения доступа к ним (файлам).

Поскольку имена устройств выглядят так же, как и имена обычных файлов, и поскольку над устройствами и над обычными файлами выполняются одни и те же операции, большинству программ нет необходимости различать внутри себя типы обрабатываемых файлов.

В общем случае в файловой системе UNIX могут быть следующие объекты

- *Каталоги*: файлы, которые представляют собой списки других файлов.
- *Специальные файлы*: механизм использования ввода-вывода. Большинство специальных файлов находятся в /dev.
- *Ссылки*: механизм обеспечения видимости файла или каталога во множестве частей файлового дерева системы. Мы в деталях поговорим о ссылках.
- *(Домены) сокеты*: особый тип файла, подобный сокетам TCP/IP, обеспечивающий взаимодействие в сети процессов, защищенных контролем файловой системы на доступ.
- *Именованные каналы*: действуют более или менее похоже на сокеты и обеспечивают способ коммуникации между процессами без использования правил поведения сетевых сокетов.

В файловой системе, файл представлен с помощью *inode (индексного дескриптора)*, своего рода серийного номера, содержащего информацию о данных этого файла: кому принадлежит этот файл, и где он находится на жестком диске.

Каждый раздел имеет свой собственный набор индексных дескрипторов; на всей системе с несколькими разделами могут существовать файлы с одним и тем же номером индексного дескриптора.

Каждый inode описывает структуру данных на жестком диске, хранит информацию о свойствах файла, в том числе физическое местоположение его данных. Когда жесткий диск назначается для хранения данных (обычно во время начала процесса установки системы или при добавлении дополнительных дисков к существующей) в разделе создается определенное

количество индексных дескрипторов. Этот число будет максимальным количеством файлов всех типов (в том числе каталогов, специальных файлов, ссылок и т.д.), которые могут существовать в одно и то же время на данном разделе. Как правило, на 1 inode приходится от 2 до 8 килобайт памяти.

Во время создания нового файла, он получает свободный inode. В этом индексном дескрипторе содержится следующая информация:

- владелец и группа-владелеца файла;
- тип файла (обычный, каталог, ...);
- разрешения на файл;
- дата и время создания, последнего открытия и изменения;
- дата и время, когда эта информация была изменена в индексном дескрипторе;
- количество ссылок на этот файл;
- размер файла;
- адрес, определяющий фактическое расположение данных файла.

12.4.3 Права доступа

Права доступа к файлу регулируются установкой специальных битов разрешения доступа, связанных с файлом. Устанавливая биты разрешения доступа, можно независимо управлять выдачей разрешений на чтение, запись и выполнение для трех категорий пользователей:

- владельца файла,
- группы пользователя,
- прочих пользователей.

Пользователи могут создавать файлы, если разрешен доступ к каталогу. Вновь созданные файлы становятся листьями в древовидной структуре файловой системы.

Программы, выполняемые под управлением системы UNIX, не содержат никакой информации относительно внутреннего формата, в котором ядро хранит файлы данных, так как данные в программах представляются как бесформатный поток байтов.

Программы могут интерпретировать поток байтов по своему желанию, при этом любая интерпретация никак не будет связана с фактическим способом хранения данных в операционной системе. Так, синтаксические правила, определяющие задание метода доступа к данным в файле, устанавливаются системой и являются едиными для всех программ, однако семантика данных определяется конкретной программой.

13. ПЕРСПЕКТИВНАЯ ОПЕРАЦИОННАЯ СИСТЕМА QNX NEUTRINO

13.1 Назначение ОС QNX Neutrino

Основным назначением операционной системы QNX Neutrino является реализация программного интерфейса POS1X в масштабируемой, отказоустойчивой форме, подходящей для широкого круга открытых систем, начиная от небольших встроенных систем с ограниченными ресурсами и заканчивая крупными распределенными вычислительными средами. Данная ОС поддерживает несколько семейств процессоров, в том числе x86, ARM, XScale, PowerPC, MIPS и SH-4.

Рис. 13.1. Основные сферы применения ОС QNX Neutrino

Операционная система QNX Neutrino идеально подходит для приложений реального времени. Она может быть масштабирована до компактных конфигураций и способна работать в многозадачном режиме, управлять потоками, осуществлять планирование процессов по приоритетам и выполнять быстрое переключение контекстов. Более того, операционная система предоставляет все эти возможности посредством программного интерфейса, основанного на стандартах POSIX.

13.2 Архитектура ОС QNX Neutrino

Возможности эффективности, модульности и простоты достигаются в ОС QNX Neutrino благодаря двум фундаментальным принципам:

1. микроядерная архитектура;
2. межзадачное взаимодействие на основе обмена сообщениями.

13.2.1 Микроядро

Микроядерноя операционная система построена на основе миниатюрного ядра, обеспечивающего минимальные службы для произвольной группы взаимодействующих процессов, которые, в свою очередь, обеспечивают функциональность более высокого уровня.

Операционная система QNX Neutrino строится на основе компактного микроядра, способного управлять группой взаимодействующих процессов. Как видно на рис. 13.2, структура операционной системы больше напоминает «слаженную команду», чем иерархию, так как несколько равноправных «игроков» взаимодействуют в ней между собой посредством координирующего ядра.

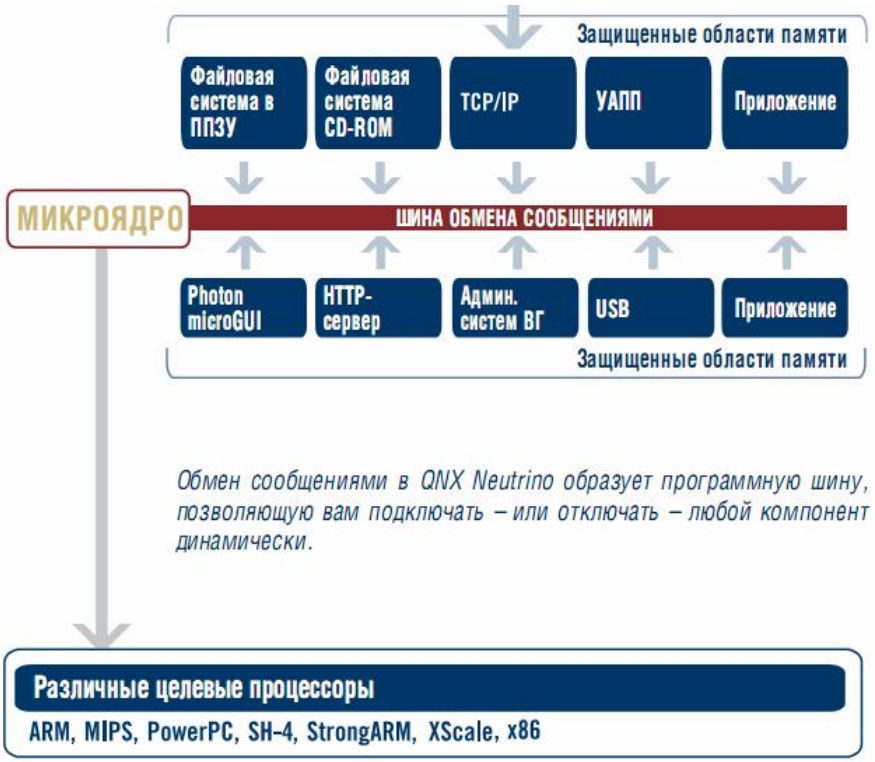


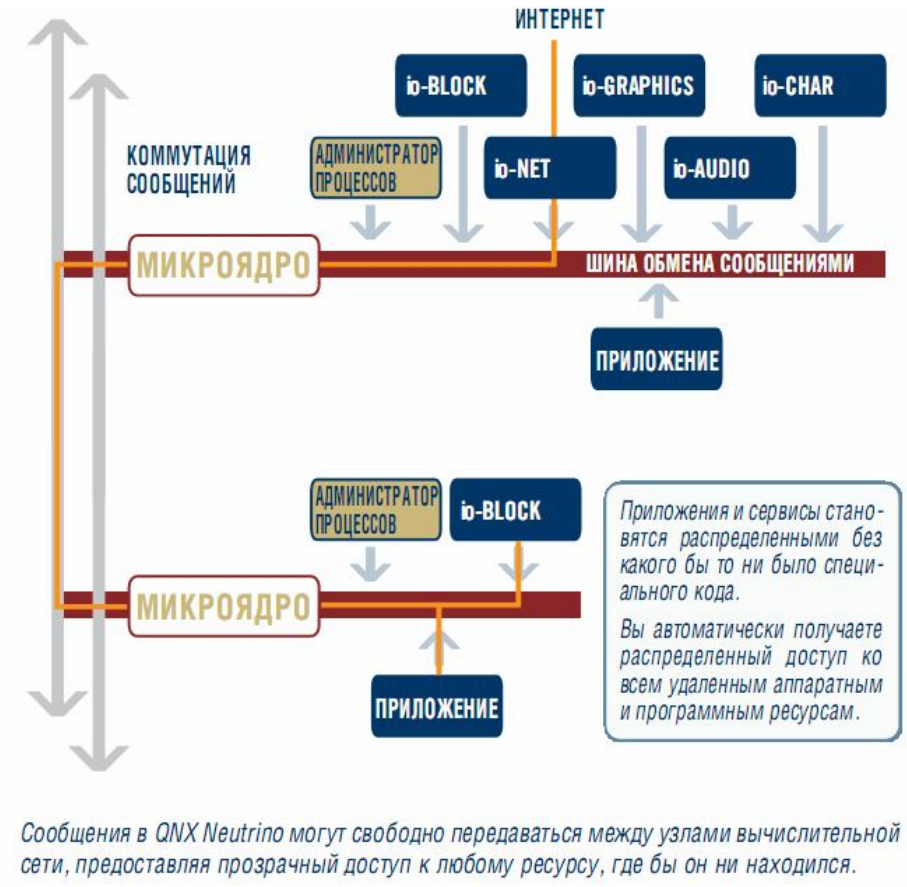
Рис. 13.2 Обмен сообщениями между структурными элементами операционной системы

13.2.2 Межзадачное взаимодействие

Для того чтобы осуществить выполнение нескольких потоков одновременно в многозадачной операционной системе реального времени, эта ОС должна иметь механизмы обеспечения взаимодействия потоков между собой.

Операционная система QNX Neutrino изначально разрабатывалась как сетевая операционная система. В некотором смысле, компьютерная сеть, построенная на основе QNX Neutrino, больше напоминает единую универсальную ЭВМ, чем набор индивидуальных микрокомпьютеров, В

распоряжении пользователей имеется огромный набор ресурсов, которые могут быть применены в любом приложении, Однако в отличие от универсальной ЭВМ, ОС QNX Neutrino является очень гибкой средой, так как на любом ее узле может быть предоставлен необходимый объем вычислительных мощностей в соответствии с потребностями каждого пользователя.



Сообщения в QNX Neutrino могут свободно передаваться между узлами вычислительной сети, предоставляя прозрачный доступ к любому ресурсу, где бы он ни находился.

Рис. 13.3 Обмен сообщениями между отдельными узлами вычислительной сети

В отличие от монолитных систем, QNX Neutrino построена на основе принципа эффективного взаимодействия, который является ключом к эффективному функционированию. Таким образом, механизм обмена сообщениями — это краеугольный камень архитектуры микроядра. Он увеличивает эффективность *всех* транзакций, происходящих между всеми процессами во всей системе независимо от используемой среды передачи, будь это простое прямое соединение между компьютерами или километр витой пары.

Обмен сообщениями в QNX эффективен, поскольку в QNX Neutrino обмен сообщениями - это больше, чем просто форма межзадачного взаимодействия. Это один из фундаментальных механизмов, делающих

системы на основе QNX Neutrino столь гибкими и простыми в разработке. Этот механизм, в частности:

- автоматически синхронизирует выполнение взаимодействующих компонентов;
- избавляет вас от необходимости следить за очередностью доставки данных;
- позволяет вам разбить сложное приложение на четко разграниченные функциональные блоки, которые можно разрабатывать и тестировать по отдельности;
- придает системам изящность, делающую их легкими в эксплуатации и обслуживании;
- действует по всей сети, предоставляя вашим приложениям прозрачный доступ к сервисам и ресурсам удаленных узлов.

Обмен сообщениями в QNX также эффективен, поскольку каждая операция происходит непосредственно между отправителем и получателем. Соответственно, не происходит никакого промежуточного копирования данных, и не требуется дополнительных действий по синхронизации.

13.2.3 Реализация поддержки симметричных многопроцессорных систем

QNX Neutrino в полной мере поддерживающая **симметричные мультипроцессорные системы (**SMP): любой поток любого процесса в ней можно запланировать на выполнение на любом из доступных процессоров.

Рис. 13.4. Реализация поддержки симметричных многопроцессорных систем

Встроенная «прозрачная» поддержка SMP позволяет:

- **Настраивать производительность,** используя **родственность процессоров** - для оптимизации использования процессорного кэша. QNX Neutrino будет всегда пытаться запланировать поток на

процессоре, где он выполнялся в последний раз, если это допустимо. Чтобы иметь возможность дополнительно оптимизировать работу кэша, микроядро QNX Neutrino предоставляет "маску родственности" процессоров, позволяя закрепить поток за одним или несколькими выбранными процессорами.

- **Использовать максимум возможностей каждого процессора** - поскольку QNX Neutrino может запланировать любой поток на любом процессоре, все процессоры можно использовать по максимуму, получая наибольший выигрыш в производительности. При этом, в отличие от традиционных ядер ОС, чтобы поддерживать SMP, микроядру QNX Neutrino не требуется множества модификаций кода, снижающих производительность. Микроядро с поддержкой SMP всего на несколько килобайт больше стандартного.
- **Строить отказоустойчивые кластеры, обладающие высокой вычислительной мощностью** - комбинируя SMP-возможности QNX Neutrino с ее встроенным механизмом распределенных вычислений, вы можете легко конструировать массивные отказоустойчивые кластеры, включающие в себя сотни одно- и многопроцессорных систем. Используя QNX Neutrino, вы получаете уникальную возможность выполнять один и тот же набор бинарных модулей вашего приложения как на однопроцессорных, так и на SMP и кластерных целевых системах.

13.2.4 Реализация поддержки распределенных вычислений

Поддержка распределенных вычислений в QNX Neutrino была заложена изначально. Таким образом, чтобы обращаться к сервисам на удаленных узлах, приложению не нужно выполнять каких-либо специальных действий. Оно будет посылать такие же сообщения, которые использовались бы для доступа к локальному сервису, и эти сообщения будут автоматически перенаправлены по сети на нужный узел. Таким образом, любые удаленные ресурсы - диски, сетевые адаптеры, стеки протоколов, и т.п. - становятся доступны так, как будто они находятся на локальной машине.

Используя распределенные возможности QNX Neutrino, возможно:

- **Уменьшить затраты на оборудование** - при использовании распределенных вычислений узлы сети могут совместно использовать ресурсы вместо их дублирования. Например, если на одном узле расположена большая файловая система в ППЗУ другим узлам иметь такую же не обязательно - они смогут использовать файловую систему того узла. на котором она уже есть. Аналогично, если на одном узле запущен стек TCP/IP, все остальные узлы смогут

использовать этот узел как TCP/IP-шлюз. исключая необходимость в настройке нескольких IP-адресов.

- **Добавлять вычислительную мощность без дополнительных разработок** - чтобы добавить в распоряжение приложения больше вычислительных мощностей или больше физических интерфейсов, достаточно вставить дополнительную процессорную карту или добавить в сеть еще один компьютер. Приложения на уже имеющихся узлах смогут пользоваться ресурсами нового узла без внесения в них каких-либо изменений.
- **Упростить проектирование отказоустойчивых кластеров** - поскольку обмен сообщениями в QNX Neutrino предоставляет прозрачный доступ к сервисам вне зависимости от их местоположения. Приложения могут полностью абстрагироваться от принятия решений о том, как будут обрабатываться запросы от клиента, где этот сервис расположен, и есть ли другие сервисы, способные обработать этот запрос (например, в случае дублирования сервиса на нескольких узлах для обеспечения отказоустойчивости или балансировки нагрузки).
- **Увеличить пропускную способность сети резервированными соединениями** - в QNX Neutrino сообщения могут передаваться по нескольким соединениям одновременно, увеличивая пропускную способность и повышая надежность связи. Например, при отказе одного из соединений QNX Neutrino может перенаправить поток данных по одному или нескольким альтернативным маршрутам. Возможно также настроить QNX Neutrino на балансировку сетевого трафика между всеми доступными соединениями, повысив тем самым суммарную пропускную способность.

13.2.5 Администратор систем высокой готовности

Принятый в QNX Neutrino подход к обеспечению высокой готовности очень прост: *время, расходуемое на перезапуск одного компонента, гораздо меньше времени перезагрузки всей системы.* Например, драйвер или стек протоколов, в котором возникла проблема, может быть немедленно выгружен и перезапущен, зачастую за единицы миллисекунд. Перезагрузка системы не требуется. Именно такой тонкий подход к изоляции сбоев позволяет QNX Neutrino обеспечивать гораздо меньшее, чем у других ОС среднее время восстановления.

В дополнение к этому, возможно расширить встроенные возможности QNX Neutrino, применив администратор систем высокой готовности, , позволяющий системе в случае сбоя самовосстанавливаться автоматически.

Администратор систем высокой готовности обеспечивает:

- **Мгновенные уведомления** об отказах - в администраторе систем высокой готовности реализован механизм квитанций работоспособности, следящий за состоянием каждого компонента системы и позволяющий обнаруживать отказы на самой ранней стадии. Если администратор обнаруживает определенное стечение обстоятельств или отказ, он может автоматически мгновенно оповестить об этом другие компоненты.
- **Настраиваемые сценарии восстановлений** - используя библиотеку администратора систем высокой готовности, приложение может явно указать администратору, какие действия по восстановлению и в каком порядке предпринять в случае сбоя.
- **Автоматическое восстановление соединений** - администратор систем высокой готовности также предоставляет клиентскую библиотеку, которая позволяет системе в случав отказа восстановить прерванные соединения. Эта библиотека содержит замену для стандартных функций ввода/вывода из библиотеки языка С.
- **"Посмертный" анализ** - если процесс завершается некорректно, администратор систем высокой готовности может сохранить его образ для последующей обработки. Анализируя этот образ, возможно определить, какая строка кода вызвала сбой, а также узнать содержимое переменных, чтобы точно определить, что именно произошло.

Администратор систем высокой готовности обладает способностью к самовосстановлению и поэтому устойчив к внутренним сбоям. Если он по какой-либо причине завершается некорректно, он полностью восстанавливает свое предыдущее состояние.

13.3 Файловые системы

В традиционных ОС файловые системы встроены в ядро. *В QNX Neutrino файловые системы расположены вне пределов ядра и выполняются в отдельных защищенных областях памяти как пользовательские процессы. В результате, вы можете запустить, остановить или обновить поддержку той или иной файловой системы «на лету», без необходимости в перезагрузке.*

В дополнение, несколько файловых систем: дисковая, встраиваемая в ППЗУ, CD-ROM, CIFS и т.д. - могут выполняться одновременно на одной и той же целевой системе. Они даже могут работать совместно, расширяя возможности друг друга. Например, файловая система со сжатием может работать совместно со встраиваемой файловой системой, существенно снижая потребности устройства в объеме ППЗУ.

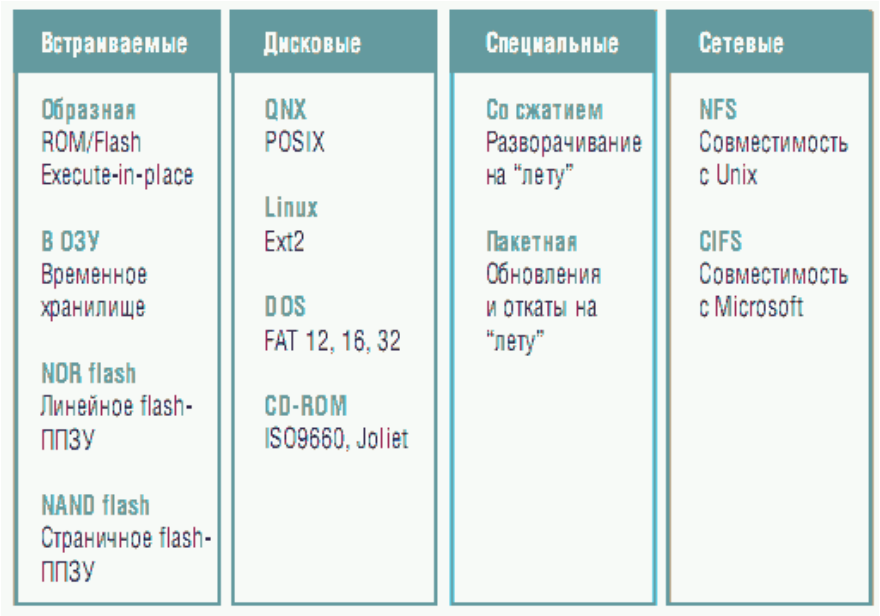


Рис. 13.4. Реализация поддержки файловых систем

13.4 Поддержка сетевых протоколов

В QNX Neutrino все сетевые сервисы выполняются вне пределов ядра как отдельные процессы с защищенными адресными пространствами. Поэтому можно запускать, останавливать или обновлять любой драйвер или протокол "на лету". Сверх того, приложение может пользоваться любым сетевым сервисом через один и тот же стандартный POSIX API, применяемый для работы со всеми остальными сервисами. Поддержка сетевых протоколов позволяет:

- сочетать любое количество сетевых протоколов, включая TCP/IP и распределенную сеть QNX;
- создавать многочисленные виртуальные сети, запуская несколько копий стека TCP/fP на одном и том же физическом интерфейсе;
- использовать богатую базу сетевого кода "третьих" производителей, основанного на POSIX и BSD API.

13.5 Драйвера устройств

QNX Neutrino содержит множество готовых драйверов для различных плат и периферийных устройств. Однако, если необходимо написать драйвер для нестандартного устройства, это реализуется за счет использования *библиотеки администратора ресурсов.*

Библиотека администратора ресурсов предоставляет простой интерфейс для регистрации драйвера в пространстве имен путей и обработки запросов от клиентов. Однако, применение этой библиотеки не ограничивается только драйверами - упрощая процессы установления и

разрыва соединений она помогает ускорить разработку любого сервиса с которым работают множество клиентских приложений.

Рис. 13.5. Использование пакетов драйверов (DDK)

Такой подход к «конструированию» системы драйверов, позволяет:

- **Использовать пакеты разработки драйверов (DDK) для ускорений разработки** - для упрощения разработки драйверов в состав комплекта разработчика QNX Momentics входят пакеты разработки драйверов (DDK) для устройств различного типа, включая аудио-, графические и сетевые адаптеры, терминальные устройства, устройства ввода, принтеры и USB-устройства. В состав пакетов входит детальная документация, исходные тексты, а также готовый программный каркас, в котором весь высокоуровневый аппаратно-независимый код уже реализован.
- **Отлаживать драйверы на уровне исходного текста, используя обычный инструментарий** - поскольку драйверы и прочие администраторы ресурсов выполняются как обычные пользовательские процессы, отлаживать их можно при помощи тех же интегрированных инструментов разработки приложений, которые входят в состав QNX Momentics.
- **Тестировать новые драйверы без перезагрузки** - можно тестировать изменения в коде драйверов без перезагрузки системы, и даже не начиная новую отладочную сессию - просто перекомпилируя и перезапуская драйвер. Мало того, поскольку все драйверы представляют собой обычные процессы, возможно

тестировать и отлаживать их прямо на инструментальном компьютере, еще до того как будет готова целевая аппаратура.

– **Программировать многопогочные сервисы без написания больших объемов кода** - чтобы сделать многопоточные драйверы проще и меньше по объему, библиотека администратора ресурсов включает в себя функции управления пулами потоков, которые автоматически регулируют число потоков в системе в зависимости от загрузки. Драйвер, использующий пулы потоков, изначально поддерживает SMP и может легко масштабироваться в многопроцессорных системах.

13.6 Интегрированный комплекс разработчика

Комплекс разработчика QNX Momentics Professional Edition поддерживает различные языки программирования, различные инструментальные ОС и различные целевые процессоры. Полнофункциональный и в то же время простой в использовании, он позволяет ускорить разработку нового проекта, вне зависимости от его сложности и масштаба.

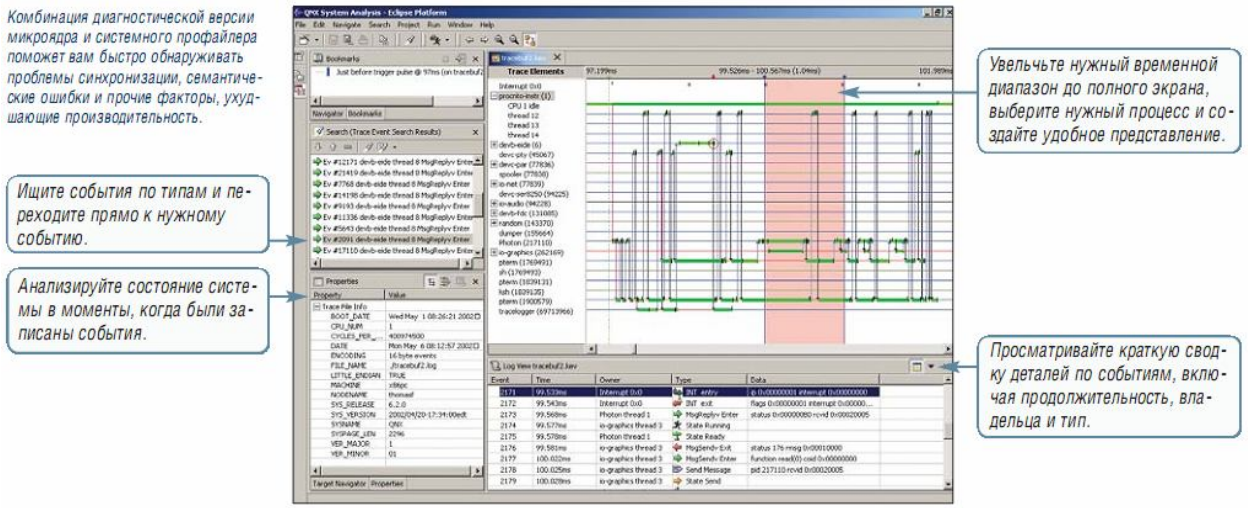


Рис. 13.6. Интерфейс комплекса разработчика QNX Momentics

В состав комплекта входят:

– тесно интегрированный инструментарий для анализа памяти, профилирования кода, анализа событий трассировки, управления версиями. удаленной диагностики, генерации графических интерфейсов. и т.д.;
– мастера, позволяющие экономить время при создании новых проектов, оптимизации целевых образов и организации сессий удаленной отладки;
– богатый выбор пакетов поддержки процессорных плат, библиотек и исходных текстов драйверов различного типа.

Используя QNX Momentics, можно писать на C, C++, встраиваемом C++ или Java, работать в среде Windows, Solaris или QNX Neutrino, и компилировать код для процессоров ARM, MIPS. PowerPC. SH-4, Strong ARM. XScalE или x86 - и все это из одной и той же IDE. Существует даже возможность работать с различными языками и процессорами одновременно.

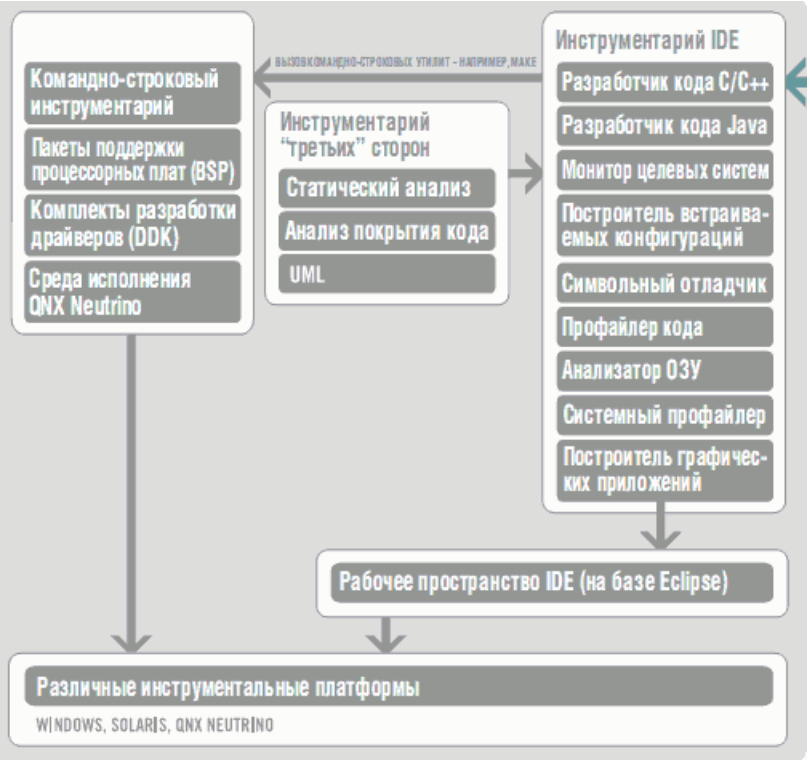


Рис. 13.7. Структура комплекса разработчика QNX Momentics

13.7 Графический интерфейс пользователя Photon

В QNX Neutrino существует Photon micro GUI, модульная графическая оболочка, способная обеспечить даже самому маленькому встраиваемому устройству профессиональный, современный графический интерфейс. По аналогии с самой QNX Neutrino, Photon основан на компактном микроядре и предоставляет большинство своих сервисов посредством опциональных процессов, работающих в защищенных областях памяти.

В отличие от функционально ограниченных графических библиотек, традиционных для других ОС, Photon представляет собой оконную систему, имеющую следующие возможности:

– Строить сложные многослойные дисплеи - Photon предоставляет высокотехнологичные функции типа внеэкранной растеризации, перекрытии изображений, канала прозрачности, подстановки текстуры и прямого режима. В результате, можно создавать плавно работающие многослойные дисплеи, способные отображать комбинацию из графики и живого видео - идеальное решение для

систем динамической навигации, телеприставок с функцией "картинка в картинке", и т.п.

– Создавать уникальный облик - используя механизм, называемый стилями *виджетов (widgetstyles),* можно настраивать внешний вид кнопок, меню, окон и других элементов интерфейса.

– Отображать и вводить текст на нескольких языках одновременно - Photon поддерживает стандарт Unicode.

– Отображать высококачественные шрифты на дисплеях любого размера - администратор шрифтов Photon поддерживает множество шрифтовых форматов, включая растровые и TrueType.

– Подключать различные мультимедийные форматы - расширяемая архитектура медиапроигрывателя Photon предоставляет готовую поддержку множества форматов данных, включая CD-аудио, MP3, потоки MPEG-1, WAV; AIFF, IFF; AU и прочие.

– Обновлять графический интерфейс "на лету" - в Photon большинство графических сервисов (видеодрайверы, менеджеры окон, драйверы устройств ввода и т.п.) реализованы в виде плагинов. Таким образом, можно динамически изменять практически любой компонент интерфейса без перезагрузки.

– Создавать полноценные интерфейсы, не написав ни единой строчки кода - для упрощения разработки графических интерфейсов, Photon поддерживает визуальное средство разработки.

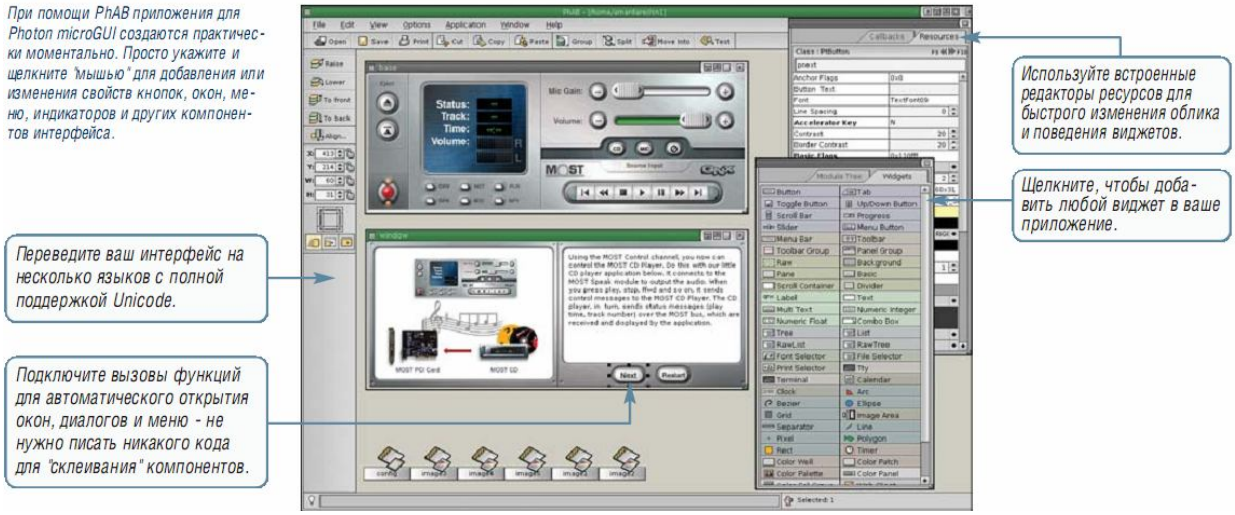


Рис. 13.8. Графический интерфейс пользователя Photon

13.8 Эффективность ОС QNX Neutrino

QNX Neutrino высокую производительность в реальном масштабе времени, поскольку в ней реализованы:

- **Сверхмалые задержки обработки прерывания и переключения контекста** - с временем переключения контекста в 600 наносекунд на процессорах класса Motorola PowerPC 7450, QNX Neutrino позволяет обеспечить максимум производительности вычислительной аппаратуры.
- **Распределенный механизм наследования приоритетов** - в QNX Neutrino драйверы, файловые системы и прочие сервисы могут выполняться с приоритетом клиента, запросившего обслуживание, даже если он расположен на другом узле сети. Такое наследование приоритетов при обмене сообщениями дает гарантию, что задача, выполняемая по заказу низкоприоритетного клиента, всегда будет вытеснена задачей от высокоприоритетного клиента. Инверсия приоритетов исключается.
- **Свобода выбора дисциплины планирования потоков** - QNX Neutrino не просто предоставляет несколько дисциплин планирования, она позволяет назначать каждому потоку свою дисциплину.
- **Гарантированная доступность процессора для задач с жестким графиком** - возможно назначать лимит времени выполнения для потоков в пределах определенного интервала. В результате, потоки будут готовы обработать нерегулярно (асинхронно) возникающие события, не рискуя нарушить график выполнения других процессов и потоков. Эта дисциплина особенно полезна при реализации в системе, обрабатывающей одновременно периодические и апериодические события.
- **Автоматическая синхронизация системных компонентов** - синхронизация, предоставляемая механизмом обмена сообщениями в QNX Neutrino, значительно упрощает реализацию поведения системы в реальном времени. Во многих других ОС такое поведение приходится реализовывать при помощи двухуровневого планирования и с большими накладными расходами.
- **Вложенные прерывания** - предоставляя поддержку вложенных прерываний, в сочетании с фиксированной верхней границей времени реакции.

На базе QNX Neutrino возможно:

- **Создавать системы, способные к самовосстановлению** - в QNX Neutrino любой компонент в случае отказа может быть перезапущен

динамически, не нарушая работу микроядра и других компонентов. Например, если драйвер попытается обратиться к памяти за пределами своего адресного пространства (что для большинства ОС является фатальной ошибкой), QNX Neutrino корректно завершит этот драйвер и освободит все занятые им ресурсы. Возможно, даже автоматически перезапустить этот драйвер, используя администратор систем высокой готовности QNX Neutrino.

- **Использовать одну и ту же ОС во всей своей линейке продуктов** - благодаря исключительной модульности QNX Neutrino, любые уже испытанные и проверенные компоненты - драйверы, приложения, дополнительные сервисы ОС, возможно использовать повторно в других продуктах. Фактически, можно создать универсальный набор бинарных модулей, а затем применять его либо в однопроцессорном устройстве, либо в SMP-системе, либо в вычислительном кластере. Вне зависимости от масштаба и сложности системы, возможно использовать одну и ту же ОС, один и тот же интерфейс прикладного программирования (API) и один и тот же инструментарий разработчика.
- **Производить обновление «на лету»** - поскольку любой компонент в QNX Neutrino может быть добавлен или удален динамически, система может продолжать работать даже в процессе замены или добавления в нее новых приложений, драйверов или стеков протоколов.

Заключение

Решение задач внедрения новой информационной инфраструктуры становится все более актуальной для российских компаний. Это связано и с обострением конкурентной борьбы на внутренних рынках, и с выходом компаний на международный уровень. Многие из этих компаний прошли или проходят серьезную модернизацию как в плане наращивания мощностей ЭВМ так и в области внедрения операционных систем и специального программного обеспечения нового уровня. Бурное развитие информационных технологий приводит к росту спроса на профессиональных специалистов в данной области. Это актуализирует получение образования в области операционных и вычислительных систем, а также широкой востребованости полученных профильных знаний на рынке труда.

Хочется надеяться, что данное учебное пособие поможет будущим профессионалам в информационной сфере получить тот общий набор знаний и умений в области вычислительных и операционных систем, чтобы оказаться востребованными и высокооплачиваемыми сотрудниками престижных компаний.

Список использованных источников

1. Олифер В. Г., Олифер Н. А. Сетевые операционные системы: учебник. – СПб.: Питер, 2003. – 544 с.

2. Операционные системы среды и оболочки: программа дисциплины для студентов по специальности «Прикладная информатика» – Ставрополь: СФ МГГУ им. М. А. Шолохова, 2008. – 20 с.

3. Карпов В. Е., Коньков К. А. Основы операционных систем [Эл. ресурс]. – М.: Интернет-университет информационных технологий, 2005. – URL: www.INTUIT.ru (дата доступа 1.09.2008).

4. Дубаков А. А. Операционные системы: учебное пособие. - Томск: изд. ТПУ, 1999. – 141 с.

5. Коньков К. А. Устройство и функционирование ОС Windows [Эл. ресурс]. – М.: БИНОМ. Лаборатория знаний. Интернет-университет информационных технологий, 2008. – URL: www.INTUIT.ru (дата доступа 1.09.2008).

6. Курячий Г. В. Операционная система Unix [Эл. ресурс]. – М.: Интернет-университет информационных технологий, 2004. – URL: www.INTUIT.ru (дата доступа 1.09.2008).

7. Курячий Г. В., Маслинский К. А. Операционная система Linux [Эл. ресурс]. – М.: Интернет-университет информационных технологий, 2005. - URL: www.INTUIT.ru (дата доступа 1.09.2008).

8. Операционная система реального времени QNX Neutrino 6.3. Системная архитектура: пер. с англ. - СПб.: БХВ-Петербург, 2006. – 336 с.

9. Крюков В. А. Операционные системы распределенных вычислительных систем (распределенные ОС): курс лекций [Эл. ресурс]. – М.: факультет ВМиК МГУ, 2008. – URL: parallel.ru/parallel/russia/people/krukov.html (дата доступа 1.09.2008).

10. Макаренко С. И. Вычислительные системы, сети и телекоммуникации: учебное пособие. – Ставрополь: СФ МГГУ им. М. А. Шолохова, 2008. – 352 с.

11. Макаренко С. И. К вопросу о распределении информационно - вычислительных потоков в сетях реального времени // Тезисы докладов международной молодежной научной конференции «XXXI Гагаринские чтения» / Моск. авиационный техн. ин-т. Том 4.– М.: изд. МАТИ-РГТУ. 2006.

12. Макаренко С. И. Аспекты совершенствования информационной подсистемы АСУ реального времени // Сб. докладов международной НТК посвященной 35-летию со дня основания Московского государственного технического университета гражданской авиации: «Гражданская авиация на

современном этапе развития науки, техники и общества» Московский государственный технический университет гражданской авиации. – М.: изд. МГТУ ГА. 2006. – с. 156-157.

13. Макарено С. И., Кихтенко А. В. Показатели качества обслуживания информационно-вычислительной сети АСУ реального времени в условиях нестационарности потоков данных. // Авиакосмические технологии и оборудование. Казань-2006: мат. Международной научно-практической конференции. 15-16 августа 2006 года. - Казань: изд. КГТУ им. А. Н. Туполева, 2006. – с. 173–174.

14. Макарено С. И., Кихтенко А. В. Анализ методов оценки влияния нестабильности входных потоков данных на показатели качества обслуживания информационно вычислительной сети АСУ реального времени // Передача, обработка и отображение информации: Сб. по материалам докладов Всероссийской научно-технической школы-семинара «Проблемы совершенствования боевых авиационных комплексов, повышение эффективности их ремонта и эксплуатации» г. Терскол, 2006. - Ставрополь: изд. СВВАИУ (ВИ). 2006. – с. 97-99.

15. Макаренко С. И. Анализ математического аппарата расчета качества обслуживания информационно-вычислительной сети на сетевом уровне эталонной модели взаимодействия открытых систем // VII Всероссийская конф. молодых ученых по математическому моделированию и информационным технологиям / ИВТ СО РАН, 2006. [Эл. ресурс] - URL: http://www.ict.nsc.ru/ws/YM2006/10566/article.htm (дата доступа 01.09.2008).

16. Макаренко С. И. Показатели качества обслуживания информационно-вычислительной сети АСУ реального времени в условиях нестационарности потоков данных : Ставропольское Высшее Военное Инженерное училище (военный институт). – Ставрополь: 2006. – 23 с. - Деп. в СИФ ЦВНИ Минобороны РФ 14.05.2007, № 15247. - СИФ ЦВНИ Минобороны РФ, инв. № В6555.

17. Макаренко С. И. Адаптивное управление информационными и сетевыми ресурсами // Научное, экспертно-аналитическое и информационное обеспечение стратегического управления, разработки и реализации приоритетных национальных проектов и программ. Сб. науч. тр. ИНИОН РАН. – М.: ИНИОН РАН 2007. – с. 534-538.