Тема 7. Современные сетевые операционные системы

7.1. Сетевые и распределенные операционные системы

7.2. Виды сетевых операционных систем

7.3. Требования, предъявляемые к сетевым операционным системам

7.3. Требования, предъявляемые к корпоративным сетевым операционным системам

7.4. Серверные операционные системы ведущих производителей

7.5. Тенденции на рынке операционных систем

7.6. Операционная система UNIX

7.7. Операционная система Windows 2000

7.8. Операционная система QNX

7.1. Сетевые и распределенные операционные системы

Большое разнообразие типов компьютеров, используемых в вычислительных сетях, влечет за собой разнообразие операционных систем: для рабочих станций, для серверов сетей уровня отдела и серверов уровня предприятия в целом.

К ним могут предъявляться различные требования по производительности, функциональным возможностям и совместимости.


7.1. Сетевые и распределенные операционные системы

Сетевая ОС предоставляет пользователю виртуальную вычислительную систему, работать с которой проще, чем с реальной сетевой аппаратурой. В то же время эта виртуальная система не полностью скрывает распределенную природу своего реального прототипа.

Термин “сетевая операционная система” используется в двух значениях: 1. Совокупность взаимодействующих ОС всех компьютеров сети. 2. Операционная система отдельного компьютера, позволяющая ему работать в сети.

В идеальном случае сетевая ОС должна предоставлять пользователю сетевые ресурсы в виде ресурсов единой централизованной виртуальной машины. В этом случае сетевая ОС является распределенной ОС. Распределенная операционная система существует как единая ОС в масштабах всей вычислительной системы.

Степень автономности каждого компьютера сети, работающего под управлением сетевой ОС, значительно выше по сравнению с с компьютерами, работающими под управлением распределенной ОС.

7.1. Сетевые и распределенные операционные системы

Сетевые ОС могут быть разделены на три группы: масштаба *рабочей группы* (отдела), масштаба *кампуса* и масштаба *предприятия*.

7.2. Виды сетевых операционных систем

1. **Сети отделов** – используются небольшой группой сотрудников, решающих общие задачи. Имеют 1-2 файловых сервера и не более30 пользователей. Задачи сетевой ОС: разделение локальных ресурсов (приложений, данных, принтеров, модемов).

2. **Сети кампусов** – соединяют несколько сетей отделов внутри одной территории предприятия. Задачи сетевой ОС: взаимодействие между сетями отделов, доступ к базам данных предприятия, доступ к факс-серверам и серверам скоростных модемов, высокоскоростных принтеров и др.

3. **Сети предприятия** (корпоративные сети) – объединяют все компьютеры всех территорий отдельного предприятия. Задачи сетевой ОС: предоставлять доступ к информации и приложениям, находящимся в других рабочих группах, других отделах, подразделениях и штаб-квартирах корпорации, обеспечивать широкий набор сервисов –справочную и почтовую службы, средства коллективной работы,поддержку удаленных пользователей, факс-сервис, обработку голосовых сообщений, организацию видеоконференций и др. Особую важность приобретают вопросы безопасности по причинам, связанным с крупномасштабностью сети.

7.3. Требования, предъявляемые к корпоративным сетевым операционным системам

1. Масштабируемость, т.е. способность обеспечивать работу в широком диапазоне различных количественных характеристик сети.
2. Совместимость с другими продуктами, способность работать в сложной гетерогенной среде интерсети в режиме plug-and-play.
3. Поддержка многообразных ОС конечных пользователей (DOS, UNIX, OS/2, Mac, Windows).
4. Поддержка нескольких стеков протоколов (TCP/IP, IPX/SPX, NetBIOS, DECnet, AppleTalk, OSI), обеспечение простого доступа к удаленным ресурсам и удобных процедур управления сервисами.
5. Поддержка многосерверной сети и эффективная интеграция с другими операционными системами.
6. Наличие централизованной масштабируемой справочной службы.
7. Развитая система сервисов: файл-сервис, принт-сервис, безопасность данных и отказоустойчивость, архивирование данных,служба обмена сообщениями, разнообразные базы данных, вызов удаленных процедур RPC и др.
8. Поддержка сетевого оборудования различных стандартов (Ethernet, Token Ring, ARCnet, FDDI), поддержка стандартов управления сетью.

7.4. Серверные операционные системы ведущих производителей

Windows (Microsoft)

Windows NT. Применение Windows NT Server 4.0 в качестве серверной операционной системы во многих случаях было экономически оправданным, что сделало данную операционную систему весьма популярной у малых и средних предприятий — она до сих пор активно используется многими компаниями.

Windows 2000. Windows 2000 является самой популярной операционной системой Microsoft в корпоративном секторе. К серверным операционным системам этого семейства относятся ***Windows 2000 Server*** — универсальная сетевая операционная система для серверов рабочих групп и отделов, ***Windows 2000 Advanced Server*** — операционная система для эксплуатации бизнес-приложений и приложений для электронной коммерции и ***Windows 2000 Datacenter Server*** — ОС для наиболее ответственных приложений обработки данных.

Windows 2000 Advanced Server поддерживает кластеризацию и баланс нагрузки, что делает возможным выполнение масштабируемых приложений с непрерывным доступом к данным. ***Windows 2000 Datacenter Server*** поддерживает симметричную мультипроцессорную обработку с использованием 32 процессоров, 64 Гбайт оперативной памяти, средства восстановления после отказа на основе четырехузловой кластеризации.

Windows Server 2003

- ***Windows Server 2003 Web Edition*** — операционная система для поддержки Web-приложений и Web-сервисов, включая приложения ASP .NET (Active Server Pages);

- ***Windows Server 2003 Standard Edition*** — сетевая операционная система для выполнения серверной части бизнес-решений и рассчитанная на применение в небольших компаниях и подразделениях, поддерживает до 4 Гбайт оперативной памяти и симметричную многопроцессорную обработку с использованием двух процессоров;

- ***Windows Server 2003 Enterprise Edition*** — предназначена для средних и крупных компаний. Она поддерживает серверы на базе 64-разрядных процессоров (до восьми штук) и объем оперативной памяти до 64 Гбайт и выпускается в версиях для 32- и 64-разрядных платформ;

- ***Windows Server 2003 Datacenter Edition*** — предназначена для создания критически важных технических решений с высокими требованиями к масштабируемости и доступности. К таким решениям относятся приложения для обработки транзакций в режиме реального времени, а также решения, основанные на интеграции нескольких серверных продуктов. В ОС реализована поддержка симметричной многопроцессорной обработки (до 32 процессоров), а также имеются службы балансировки нагрузки и создания кластеров, состоящих из восьми узлов. Эта ОС доступна для 32- и 64-разрядных платформ.

UNIX

Solaris (Sun Microsystems). ***Sun Solaris*** **сегодня входит в число самых известных коммерческих версий UNIX. ОС обладает развитыми средствами поддержки сетевого взаимодействия и представляет собой одну из самых популярных платформ для разработки корпоративных решений — для нее существует около 12 тыс. различных приложений, в том числе серверов приложений и СУБД почти от всех ведущих производителей. ОС Solaris 9 поддерживает до 1 млн. работающих процессов, до 128 процессоров в одной системе и до 848 процессоров в кластере, до 576 Гбайт физической оперативной памяти, поддержку файловых систем размером до 252 Тбайт, наличие средств управления конфигурациями и изменениями, встроенную совместимость с Linux.**

HP-UX (Hewlett-Packard). ***HP-UX 11i*** **имеет средства интеграции с Windows и Linux, средства переноса Java-приложений, разработанных для этих платформ, а также средства повышения производительности Java-приложений. HP-UX 11i поддерживает Linux API, что гарантирует перенос приложений между HP-UX и Linux. Операционная система поддерживает до 256 процессоров и кластеры размером до 128 узлов, подключение и отключение дополнительных процессоров, замену аппаратного обеспечения, динамическую настройку и обновление операционной системы без необходимости перезагрузки, резервное копирование в режиме on-line и дефрагментацию дисков без выключения системы.**


AIX (IBM). AIX является клоном UNIX производства IBM, предназначенным для выполнения на серверах IBM @server pSeries и RS/6000. Операционная система обладает совместимостью с Linux. В числе особенностей *AIX 5L* — наличие полностью 64-разрядных ядра, драйверов устройств и среды исполнения приложений (при этом имеется и 32-разрядное ядро и поддержка 32-рязрядных приложений), поддержка 256 Гбайт оперативной памяти, поддержка файлов объемом до 1 Тбайт, удобные средства администрирования, поддержка кластеров (до 32 компьютеров), развитые средства сетевой поддержки.

Linux и FreeBSD. Операционная система *Linux* — это некоммерческий продукт категории Open Source для платформы Intel. Список серверных продуктов для Linux включает такие популярные продукты, как Web-сервер Apache, серверные СУБД и серверы приложений практически от всех производителей.

Еще одной известной некоммерческой версией UNIX является *FreeBSD*, доступная для платформ Intel и DEC Alpha. Основой FreeBSD послужил дистрибутив BSD UNIX, выпущенный группой Калифорнийского университета (Беркли). Операционная система имеет объединенный кэш виртуальной памяти и буферов файловых систем, совместно используемые библиотеки, модули совместимости с приложениями других версий UNIX, динамически загружаемые модули ядра, позволяющие добавлять во время работы поддержку новых типов файловых систем, сетевых протоколов или эмуляторов без перегенерации ядра.

NetWare (Novell)

Основными особенностями последней версии операционной системы, *Novell NetWare 6.5*, являются возможность создания географически распределенных кластеров, наличие средств поддержки мобильных и удаленных пользователей, инструментов управления удаленными сетевыми ресурсами, а также средств синхронизации информации о пользователях и приведения в соответствие между собой каталогов в смешанных средах. Защита данных в *Novell NetWare 6.5* осуществляется с помощью служб каталогов NDS eDirectory. Данная операционная система обычно применяется в качестве сетевого и файлового сервера, сервера печати и групповой работы.

Mac OS X (Apple)

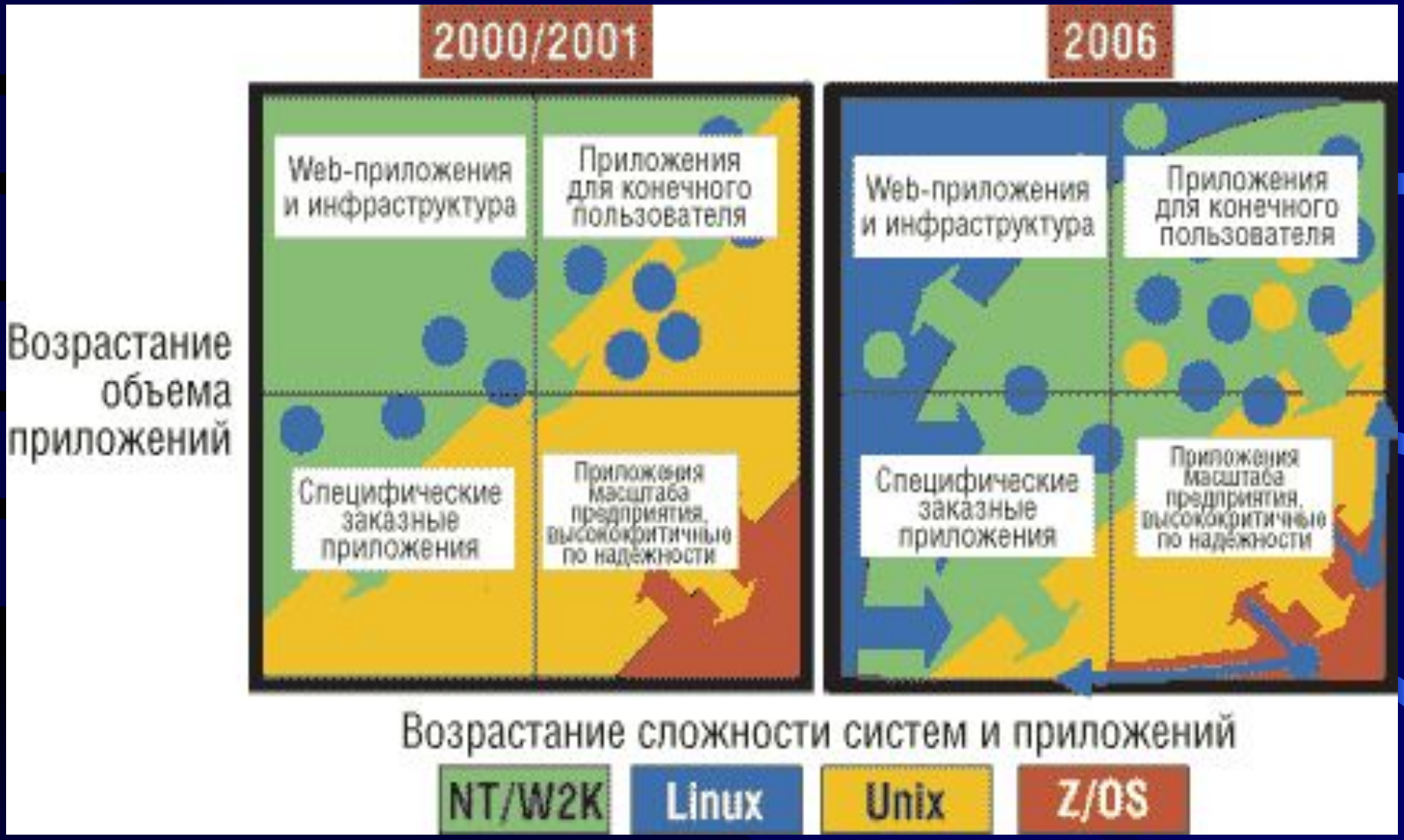
Операционная система *Mac OS X*, созданная компанией Apple совместно с рядом университетских ученых, основана на BSD UNIX. В 1999 году версия *Mac OS X Server* была выпущена в виде продукта Open Source, что позволило разработчикам адаптировать Mac OS X для конкретных заказчиков, а также привлечь их к дальнейшему развитию этой операционной системы. *Mac OS X* характеризуется наличием менеджера виртуальной памяти, возможностью полной изоляции приложений друг от друга, поддержкой многозадачности, сравнимой с аналогичной поддержкой в Windows.

Операционная система Z/OC для высокоуровневых вычислительных устройств eServer z900 (IBM)

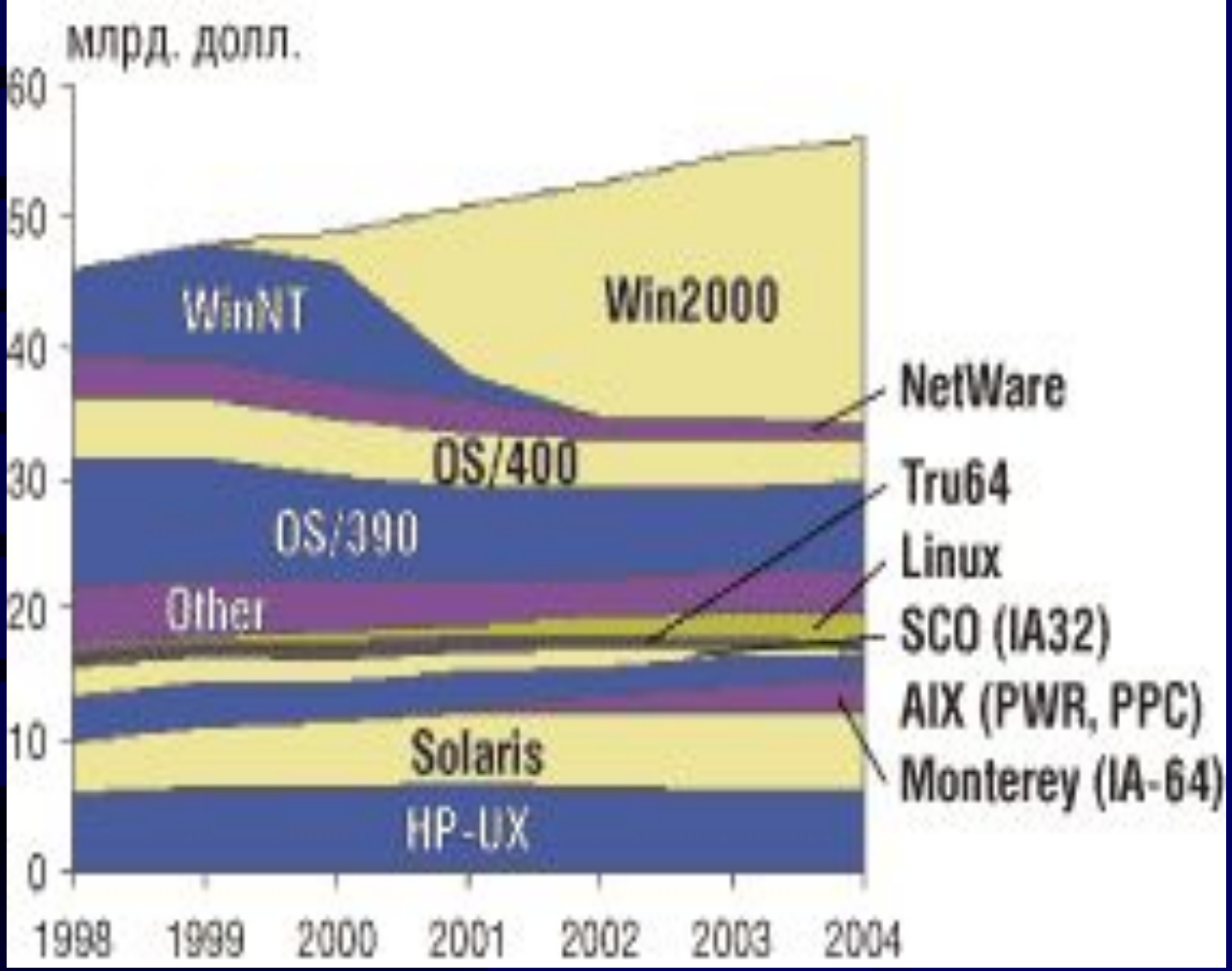
Z/OS является усовершенствованной версией операционной системы OS/390, однако по сравнению с последней новая ОС имеет ряд преимуществ. Среди них zOS – поддержка 64-разрядной адресации, позволяющей ускорить обмен данными между модулями памяти и процессорами и увеличить производительность работы с большими базами данных.

Важным компонентом z/OS является ПО Intelligent Resource Director, предназначенное для автоматического распределения вычислительных мощностей между одновременно выполняющимися приложениями. Другими отличительными особенностями новой ОС является поддержка ПО для платформ Java и Linux, а также простота в настройке и администрировании.

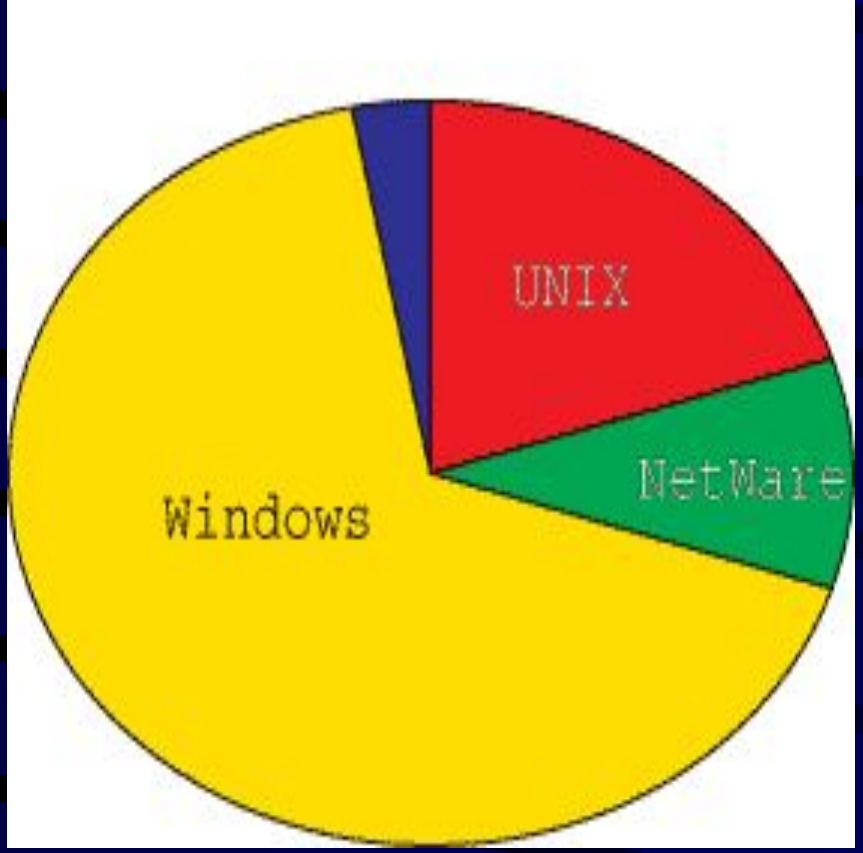
7.5. Тенденции на рынке операционных систем



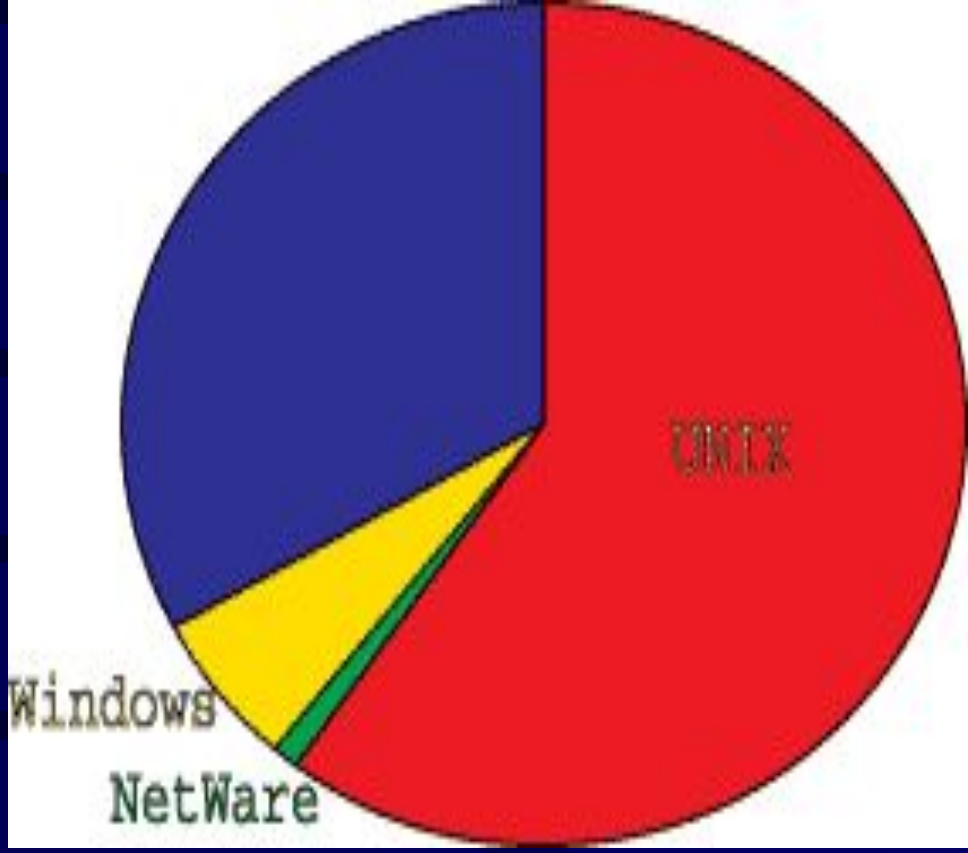
Перераспределение рынка ОС по данным Gartner Group



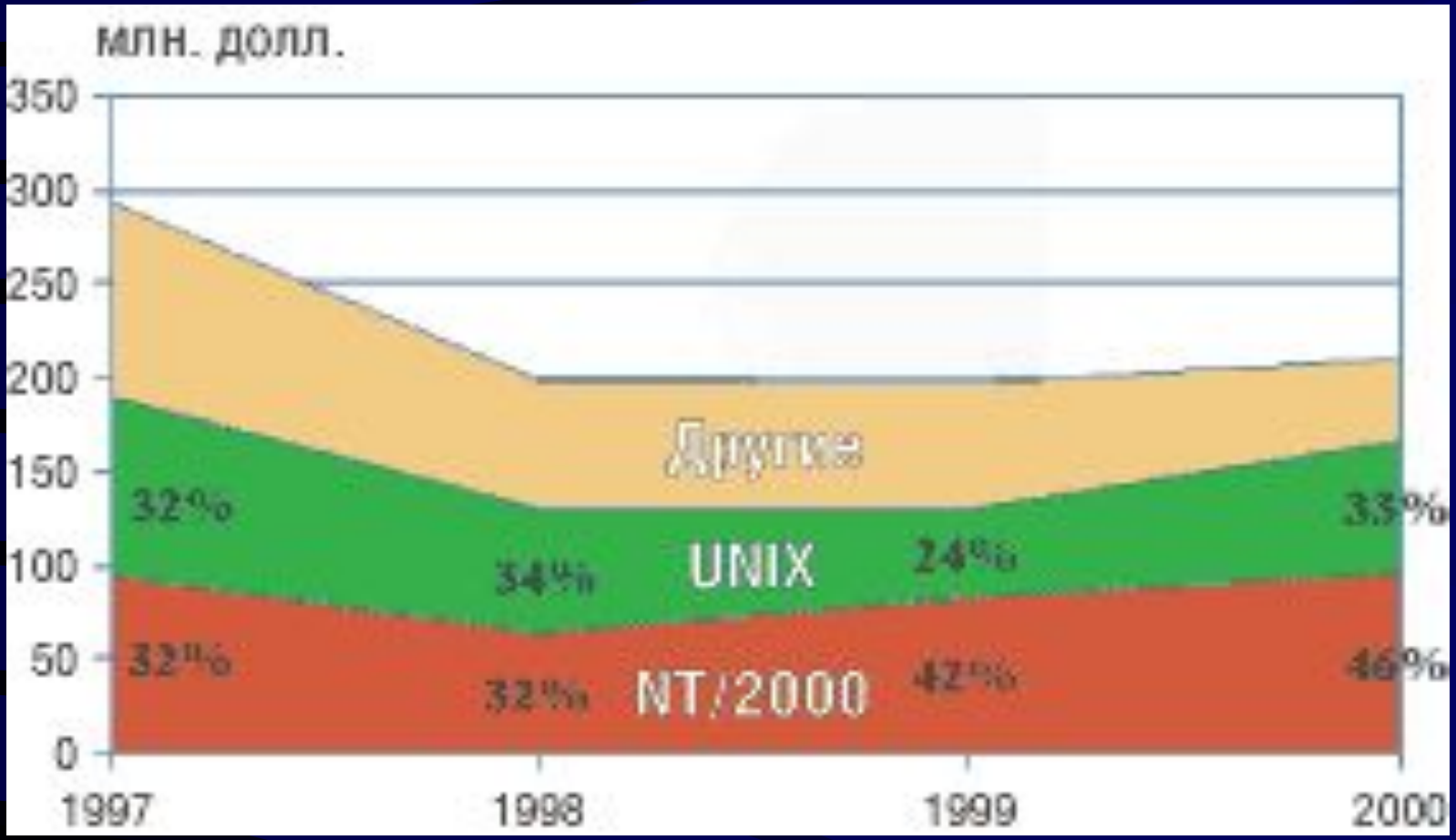
Прогнозы распределения серверного рынка ОС по данным Gartner Group на сентябрь 1999 г.



Распределение от продажи серверов в ценовой категории менее 25 тыс. долл. по операционным системам (по данным Gartner Group, 2001 г.)



Распределение от продажи серверов в ценовой категории более 25 тыс. долл. по операционным системам (по данным Gartner Group, 2001 г.)



Российский рынок серверов (1997-2000) с делением по операционным системам (по данным IDC на 2001 год)

7.6. Операционная система UNIX

7.6.1. История создания

1. Создание CTSS (Compatible Time Sharing System) в МТИ (1961 г.).
2. Создание MULTICS (Multiplexed Information and Computer Service), язык EOL (PL/1), МТИ + Bell Labs + General Electric, 1963 г.
3. Разработка усеченного варианта MULTICS - UNICS (Uniplexed Information and Computer Service) – Кен Томпсон.
4. Создание языка высокого уровня B (упрощение BCPL) и переработка Unix на этом языке – Томпсон.
5. Создание языка C – Ритчи.
6. Переписывание UNIX на C – Томпсон и Ритчи.
7. Статья Томпсона и Ритчи об ОС UNIX, 1974 г.
8. Версия 6 UNIX – 8200 строк C + 900 строк ассемблера –1974 г.
9. Первая переносимая версия UNIX (версия 7) –18000 строк C + 2110 строк ассемблера –1976 г.
10. Выпуск коммерческой версии UNIX фирмой AT&T (System III) – 1984 г., а затем UNIX System V.
11. Развитие UNIX Калифорнийским университетом в Беркли – 1BSD (Fist Berkeley Software Distribution), затем 2BSD, 3BSD, 4BSD.
12. Широкое распространение UNIX – Xenix, Minix, AIX, Sun OS, Solaris, Linux.

7.6.2. Общая характеристика системы UNIX

ОС UNIX – интерактивная система, разработанная программистами и для программистов. Основные требования: простота, элегантность, непротиворечивость, мощь и гибкость.

Общие черты Unix независимо от версии:

1. **Многопользовательский режим со средствами защиты от несанкционированных пользователей.**
2. **Реализация мультипрограммной работы в режиме разделения времени, основанная на использовании алгоритмов вытесняющей многозадачности.**
3. **Использование механизмов виртуальной памяти и свопинга для повышения уровня мультипрограммирования.**
4. **Унификация ввода-вывода на основе расширенного использования понятия файл.**
5. **Иерархическая файловая система, образующая единое дерево каталогов независимо от количества физических устройств, используемых для размещения файлов.**
6. **Переносимость системы за счет написания ее основной части на языке С.**
7. **Разнообразные средства взаимодействия процессов, в том числе через сеть.**
8. **Кэширование дисков для уменьшения среднего времени доступа к файлам.**
9. **Каждая программа выполняет всего одну функцию, но зато делает это хорошо.**

7.6.3. Интерфейс системы UNIX

7.6.3. Интерфейс системы UNIX

У основания пирамиды располагается аппаратное обеспечение, состоящее из центрального процессора, памяти, дисков, терминалов и других устройств.

На голом «железе» работает ОС UNIX. Ее функция заключается в управлении аппаратным обеспечением и предоставлении всем программам интерфейса системных вызовов. Эти системные вызовы позволяют программам создавать процессы, файлы и прочие ресурсы, а также управлять ими.

Программы обращаются к системным вызовам, помещая аргументы в регистры центрального процессора (или иногда в стек) и выполняя команду эмулированного прерывания для переключения из пользовательского режима в режим ядра и передачи управления ОС UNIX.

7.6.3. Интерфейс системы UNIX

Команду *эмулированного прерывания* исполняют библиотечные функции, по одной на системный вызов.

Эти процедуры написаны на ассемблере, но могут вызываться из программ, написанных на языке С.

Каждая такая процедура помещает аргументы в нужное место и выполняет команду эмулированного прерывания **TRAP**.

Стандарт POSIX определяет библиотечные процедуры, соответствующие системным вызовам, их параметры, что они должны делать и какой результат возвращать. В стандарте даже не упоминаются фактические системные вызовы.

7.6.3. Интерфейс системы UNIX

Все версии UNIX содержат большое количество стандартных программ, некоторые из которых описываются стандартом POSIX 1003.2, другие могут различаться в разных версиях системы UNIX.

К таким программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами.

Можно говорить о трех интерфейсах в ОС UNIX: интерфейс системных вызовов, интерфейс библиотечных функций и интерфейс, образованный набором стандартных обслуживающих программ.

7.6.4. Структура ядра системы Unix (4.4 BSD)

7.6.4. Структура ядра системы Unix (4.4 BSD)

Нижний уровень ядра **состоит из драйверов устройств и процедуры диспетчеризации процессов. Все драйверы системы UNIX делятся на два класса: драйверы символьных и блочных устройств.**

Основное различие между этими двумя классами устройств заключается в том, что на блочных устройствах разрешается операция поиска, а на символьных - нет.

Технически сетевые устройства представляют собой символьные устройства, но они обрабатываются по-иному, поэтому они выделены в отдельный класс.

Диспетчеризация процессов производится при возникновении прерывания. При этом низкоуровневая программа останавливает выполнение работающего процессора, сохраняет его состояние в таблице процессов ядра и запускает соответствующий драйвер.

7.6.4. Структура ядра системы Unix (4.4 BSD)

Диспетчеризация процессов производится, когда ядро завершает свою работу и пора снова запустить процесс пользователя. Программа диспетчеризации процессов написана на ассемблере и представляет собой отдельную от процедуры планирования программу.

Символьные устройства могут **использоваться двумя способами.**

Некоторым программам, таким как текстовые редакторы (vi, emacs), требуется каждая нажатая клавиша без какой-либо обработки. Для этого служит **ввод-вывод с необработанного терминала** (телетайпа).

Другое программное обеспечение, например оболочка (sk), принимает на входе уже готовую текстовую строку, позволяя пользователю редактировать ее, пока не будет нажата клавиша Enter. Такое программное обеспечение пользуется **вводом с терминала в обработанном виде и дисциплинами линии связи.**

7.6.4. Структура ядра системы Unix (4.4 BSD)

Сетевое программное обеспечение **часто бывает модульным, с поддержкой множества различных устройств и протоколов.**

Уровень выше сетевых драйверов **выполняет своего рода функции маршрутизации, гарантируя, что правильный пакет направляется правильному устройству или блоку управления протоколами.**

Большинство систем UNIX содержит в своем ядре *полноценный маршрутизатор Интернета*, и хотя его производительность ниже, чем у аппаратного маршрутизатора, эта программа появилась раньше современных аппаратных маршрутизаторов.

Над уровнем маршрутизации **располагается стек протоколов, обязательно включая IP и TCP, но также иногда и некоторые другие протоколы.**

7.6.4. Структура ядра системы Unix (4.4 BSD)

Над сетевыми протоколами **располагается интерфейс сокетов, позволяющий программам создавать сокеты для отдельных сетей и протоколов. Для использования сокетов пользовательские программы получают дескрипторы файлов.**

Над дисковыми драйверами **располагаются буферный кэш и страничный кэш файловой системы.**

В ранних системах UNIX буферный кэш представлял собой фиксированную область памяти, а остальная память использовалась для страниц пользователей.

Во многих современных системах UNIX этой фиксированной границы уже не существует и любая страница памяти может быть схвачена для выполнения любой задачи, в зависимости оттого, что требуется в данный момент.

7.6.4. Структура ядра системы Unix (4.4 BSD)

Над страничным кэшем располагается система виртуальной памяти. **В ней вся логика работы со страницами, например алгоритм замещения страниц. Поверх него находится программа отображения файлов на виртуальную память и высокоуровневая программа управления страничными прерываниями. При возникновении страничного прерывания она сначала проверяет допустимость обращения к памяти и, если все в порядке, определяет место нахождения требуемой страницы и то, как она может быть получена.**

Управление процессами **организовано следующим образом. Над диспетчером располагается планировщик процессов, выбирающий процесс, который будет запущен следующим.**

Если потоками управляет ядро **(в некоторых версиях UNIX), то управление потоками тоже помещается здесь, хотя в некоторых версиях UNIX управление потоками вынесено в пространство пользователя.**

7.6.4. Структура ядра системы Unix (4.4 BSD)

Над планировщиком **расположена программа обработки сигналов и отправки их в требуемом направлении, а также программа, занимающаяся созданием и завершением процессов.**

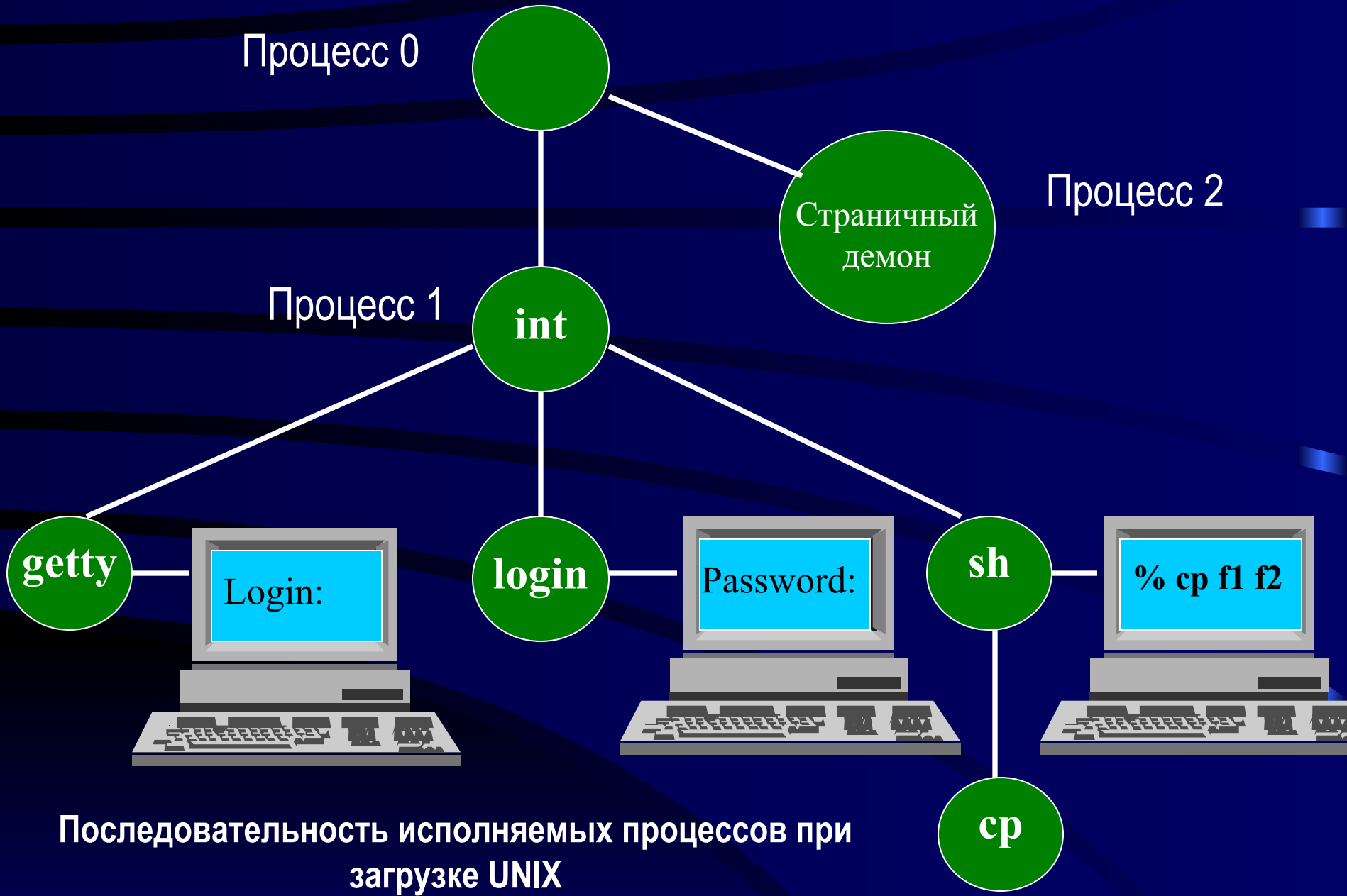
Верхний уровень **представляет собой интерфейс системы. Сюда относится интерфейс системных вызовов и вход для прерывания. Все поступающие системные вызовы направляются одному из модулей низшего уровня в зависимости от природы системного вызова.**

В верхний уровень **поступают также сигналы аппаратных и эмулированных прерываний, включая сигналы, страничные прерывания, разнообразные исключительные ситуации процессора и прерывания ввода-вывода.**

7.6.5. Загрузка системы UNIX (4.4 BSD)

1. Считывание главной загрузочной записи (первый сектор) загружаемого диска
2. Загрузка программы boot в старшие адреса основной памяти
3. Программа boot считывает корневой каталог, затем считывает ядро ОС и передает ему управление
4. Начальная программа ядра (на ассемблере, машинно-зависимая) устанавливает указатель стека, определяет тип ЦП, количество ОП, запрещает прерывания, разрешает работу диспетчера памяти, вызывает процедуру main (на С) для запуска основной части ОС.
5. Программа main выделяет память под буфер сообщений (по мере инициализации в него записываются сообщения о том, что происходит с системой), память для структур данных ядра (буферный кэш, таблицы страниц и т. п.)
6. Определяется конфигурация компьютера и формируются таблицы подключенных устройств, загружаются требуемые драйверы устройств
7. Загружается процесс 0, устанавливается его стек, запущенный процесс 0 программирует таймер реального времени, монтирует корневую ФС, создает процесс 1 (init) и процесс 2 (страничный демон).


8
Проверка флагов процессом init
Нет
9
Процесс многопользовательский ?
Да
13
Установка программы getty
10
Запуск сценария оболочки инициализации системы /etc/rc (монтирование доп. ФС, запуск демонов и др.)
14
Вызов программы регистрации /bin/login
11
Считывание файла /etc/ttys с перечнем терминалов и их свойств, запуск getty для каждого терминала.
15
Запрос пароля и сравнение его с зашифрованным паролем, хранящемся в файле /etc/passwd
12
Вызов программы регистрации /bin/login для каждого терминала
16
Пароль верный ?
Нет
Да
17
Запуск оболочки sh



Последовательность исполняемых процессов при загрузке UNIX

7.6.5. Загрузка системы UNIX (4.4 BSD)

Программа getty устанавливает для каждой линии скорость, после чего выводит на терминале приглашение по входу в систему: login:.

Затем программа getty пытается прочитать имя пользователя, введенное с клавиатуры. После этого программа getty завершает свою работу выполнением программы регистрации /bin/login.

Программа login запрашивает у пользователя его пароль, зашифровывает его и сравнивает с зашифрованным паролем, хранящимся в файле паролей /etc/passwd.

Если пароль введен верно, программа login вместо себя запускает оболочку пользователя, которая ждет первой команды. Если пароль введен неверно, программа login еще раз спрашивает имя пользователя.


7.6.5. Загрузка системы UNIX (4.4 BSD)

На рис выше процесс getty, работающий на терминале 0, все еще ждет ввода.

На терминале 1 пользователь ввел имя регистрации, поэтому программа getty запустила поверх себя процесс login, запрашивающий пароль.

На терминале 2 уже прошла успешная регистрация, в результате чего оболочка напечатала приглашение к вводу (%).

Пользователь ввел команду cp f1 f2, в результате которой оболочка создала дочерний процесс, исполняющий программу cp.

Если бы пользователь на терминале 2 ввел команду cc, то запустилась бы главная программа компилятора C, который, в свою очередь, запустил бы несколько дочерних процессов для выполнения различных проходов компилятора.

7.6.6. Оболочка системы UNIX

Система поддерживает графическое окружение X Windows, но многие программисты предпочитают интерфейс командной строки, создавая множество консольных окон и действуя так, как если бы у них было несколько алфавитно-цифровых терминалов, на каждом из которых работала бы оболочка (shell).

Существует много различных оболочек: sh, ksh, bash и др. После запуска оболочка печатает на экране символ приглашения к вводу (% или $) и ждет, когда пользователь введет командную строку.

После этого оболочка извлекает из нее первое слово и ищет файл с таким именем. Если такой файл удается найти, оболочка запускает его. При этом работа оболочки приостанавливается на время работы запущенной программы. По завершении работы программы оболочка снова печатает приглашение и ждет ввода следующей строки.

7.6.6. Оболочка системы UNIX

У команд оболочки могут быть *аргументы*, которые передаются запускаемой программе в виде текстовых строк. Не все аргументы обязательно должны быть именами файлов.

Аргументы, управляющие работой команды или указывающие дополнительные значения, называются флагами или ключами и, по соглашению, обозначаются знаком тире. Тире требуется, чтобы избежать двусмысленности.

Большинство команд системы UNIX могут принимать несколько флагов и аргументов.

Чтобы было легче указывать группы файлов, оболочка принимает так называемые *волшебные символы* (джокеры).

Например, *символ звездочка* означает все возможные варианты текстовой строки. Команда ls *.c заставляет программу ls вывести список всех файлов, имя которых оканчивается на С.

Другим джокером является *вопросительный знак*, который заменяет один любой символ.

В *квадратных скобках* можно указывать множество символов, из которых программа должна будет выбрать один. Например, команда ls [ape].* указывает на вывод списка файлов, имя которых начинается с символов «а», «р» или «е».

7.6.6. Оболочка системы UNIX

Программа оболочки не должна открывать терминал, чтобы прочитать с него или вывести на него строку. Вместо этого запускаемые программы автоматически получают доступ к файлу, называемому *стандартным устройством ввода* (standard input), и к файлу, называемому *стандартным устройством вывода* (standard output), а также к файлу, называемому *стандартным устройством вывода сообщений об ошибках* (standard error). По умолчанию всем трем устройствам соответствует терминал, т. е. клавиатура и экран.

Многие программы в системе UNIX читают данные со стандартного устройства ввода и пишут на стандартное устройство вывода (например, sort). Стандартные ввод и вывод можно *перенаправлять*. Для этого используются символы < и > соответственно. Разрешается их одновременное использование в одной командной строке.

Программа, считывающая данные со стандартного устройства ввода, выполняющая определенную обработку этих данных и записывающая результат в поток стандартного вывода, называется фильтром.

7.6.6. Оболочка системы UNIX

В системе UNIX часто используются командные строки, в которых первая программа в командной строке формирует вывод, используемый второй программой в качестве ввода. Это можно сделать через временный файл либо через **канал**. Для этого используется **вертикальная черта**, называемая **символом канала**.

Набор команд, соединенных символом канала, называется **конвейером** и может содержать произвольное количество команд.

UNIX - ***универсальная многозадачная система***. Один пользователь может одновременно запустить несколько программ, каждую в виде отдельного процесса. Синтаксис оболочки для **запуска фонового процесса состоит в использовании символа амперсанда в конце строки**. Конвейеры тоже могут работать в фоновом режиме. Можно одновременно запустить несколько фоновых конвейеров.

7.6.6. Оболочка системы UNIX

Примеры командных строк:

1) cp file1 file2 - копировать файл file1, копия – file2
2) head –20 file - печатать первые 20 строк файла file
3) sort < in > out программе sort (сортировка строк) взять в качестве входного файла in и направить вывод в файл out)
4) sort < in > temp; head -30 < temp; rm temp

Сначала запускается программа sort, которая принимает данные из файла in и записывает результат в файл temp. Когда она завершает работу, запускается программа head, распечатывающая 30 строк из файла temp на стандартном устройстве вывода, которым по умолчанию является терминал. После этого временный файл temp удаляется.

5) sort < in | head -30 - канал
6) grep ter*.t | sort | head - 201 tail - 5 > foo

Здесь в стандартное устройство вывода записываются все строки, содержащие строку «ter» во всех файлах, имена которых оканчиваются на t, после чего они сортируются. Первые 20 строк выбираются программой head, которая передает их программе tail, записывающей последние 5 строк в файл foo.

7.6.6. Оболочка системы UNIX

Примеры командных строк:

7) wc-1<a>b&

Запустит программу подсчета количества слов wc, которая сосчитает число строк (флаг -1) во входном файле a и запишет результат в файл b, но будет это делать в фоновом режиме. Как только команда введена пользователем, оболочка напечатает символ приглашения к вводу и перейдет в режим ожидания следующей команды.

8) sort < x | head & - конвейер в фоновом процессе


7.6.6. Оболочка системы UNIX

Список команд оболочки может быть помещен в файл, а затем этот файл с командами может быть выполнен, для чего нужно запустить оболочку с этим файлом в качестве входного аргумента. Вторая программа оболочки выполнит перечисленные в этом файле команды одну за другой, точно так же, как если бы эти команды вводились с клавиатуры.

Файлы, содержащие команды оболочки,называются *сценариями оболочки. В них можно использовать конструкции if, for, while, case*.

7.6.7. Утилиты системы Unix

Кроме оболочки пользовательский интерфейс содержит большое число обслуживающих программ (утилит):

1. **Программы (команды) управления файлами и каталогами.**
2. **Фильтры.**
3. **Средства разработки программ (текстовые редакторы, компиляторы).**
4. **Текстовые процессоры.**
5. **Системное администрирование.**
6. **Разное.**

Стандарты POSIX 1003.2 определяют синтаксис и семантику не менее 100 из этих программ, в основном относящихся к трем первым категориям. Идея стандартизации заключается в том, чтобы можно было писать сценарии оболочки, которые бы работали на всех системах UNIX. Помимо этих стандартных утилит существует еще масса прикладных программ, таких как Web-браузеры, программы просмотра изображений и т. д.

Во всех версиях ОС UNIX есть утилиты, требуемые стандартом POSIX, а также множество других (в зависимости от версии).

7.6.7. Утилиты системы Unix

Программа	Функция (POSIX)
cat	Конкатенация нескольких файлов в стандартный выходной поток
chmod	Изменение режима защиты файла
cp	Копирование файла
cut	Вырезание колонок текста
grep	Поиск определенной последовательности символов в файле
head	Извлечение из файла первых строк
is	Распечатка каталога
make	Компиляция файла для создания двоичного файла
mkdir	Создание каталога
paste	Вставка колонок текста в файл
pr	Форматирование файла для печати
rm	Удаление файла
rmdir	Удаление каталога
sort	Сортировка строк файла по алфавиту
tail	Извлечение из файла последних строк
tr	Преобразование символа из одного набора в другой

7.6.8. Процессы в системе Unix

У каждого пользователя системы может быть одновременно несколько активных процессов, кроме того существуют десятки фоновых процессов (демонов).

Типичный демон – *cron daemon*, предназначенный для планирования и запуска процессов. Он просыпается раз в минуту, проверяя, не нужно ли чего сделать. Если у него есть работа, он ее выполняет и записывает вновь. Этот демон позволяет планировать в системе UNIX активность на минуты, часы, дни и даже месяцы вперед, в том числе запускать периодические задачи.

Другие демоны управляют входящей и исходящей электронной почтой, очередями на принтер, проверяют, осталось ли достаточное количество страниц памяти. Демоны реализуются в UNIX довольно просто, так как каждый из них представляет собой отдельный процесс, не зависимый от всех остальных процессов.

Процессы создаются в ОС UNIX чрезвычайно несложно. Системный вызов *fork* создает точную копию исходного процесса, называемого родительским процессом. Новый процесс называется дочерним. У процессов (родительского и дочернего) собственные образы памяти. Открытые файлы используются совместно. Это значит, что если какой-либо файл был открыт до выполнения вызова fork, он остается открытым в обоих процессах и в дальнейшем.


7.6.8. Процессы в системе Unix

Изменения, произведенные с этим файлом, будут видимы каждому процессу.

Чтобы процессам узнать, кто из них должен исполнять родительскую программу, а кто дочернюю, системный вызов fork (вилка) возвращает дочернему процессу число 0, а родительскому - отличный от нуля PID (Process Identifier - идентификатор процесса) дочернего процесса. Оба процесса могут проверить возвращаемое значение и действовать соответственно, как это показано ниже:

```
pid = fork ( ); /* если fork завершился успешно, pid > 0 в родит. процессе */
if (pid < 0) {
handle_error ( ); /* fork потерпел неудачу (например, память переполнена)*/
} else if (pid > 0) {
                    /* здесь располагается родительская программа */
} else {
                    /* здесь располагается дочерняя программа */
}
```

7.6.8. Процессы в системе Unix

Процессы располагаются по своим PID-идентификаторам. Если дочерний процесс желает узнать свой PID, он может воспользоваться системным вызовом getpid. Идентификаторы процессов используются различным образом. Например, когда дочерний процесс завершается, его PID выдается его родителю.

Процессы взаимодействуют с помощью каналов. Можно создать канал между двумя процессами, в котором один процесс может писать поток байтов, а другой может его читать. Синхронизация процессов достигается путем блокировки процесса при попытке прочитать данные из пустого канала. Например, когда оболочка выполняет строку `sort < f | head` она создаст два процесса, **sort** и **head**, и устанавливает между ними канал. Если канал переполняется, система приостанавливает работу **sort**, пока **head** не удалит хоть сколько-нибудь данных.

Процессы могут взаимодействовать также при помощи программных прерываний посылкой сигналов. Один процесс может послать другому так называемый сигнал. Процессы могут сообщить системе, какие действия следует предпринимать, когда придет сигнал. У процесса есть выбор: проигнорировать сигнал, перехватить его или позволить сигналу убить процесс по умолчанию для большинства сигналов. Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуры обработки сигналов. Когда сигнал прибывает, управление передается обработчику. Когда процедура обработки сигнала завершает свою работу, управление снова возвращается в то место процесса, в котором он находился, когда пришел сигнал.

Обработка сигналов аналогична обработке аппаратных прерываний ввода-вывода. Процесс может посылать сигналы только членам его группы процессов, состоящей из его прямого родителя, всех прародителей, братьев, а также детей (внуков и правнуков). Процесс может послать сигнал сразу всей своей группе за один системный вызов.

7.6.9. Системные вызовы управления процессами в Unix

Для управления процессами используются системные вызовы.

Системный вызов	Описание
pid = fork ( )	Создать дочерний процесс, идентичный родительскому
pid = waitpid (pid, &statloc, opts)	Ждать завершения дочернего процесса
s = execve (name, argv, envp)	Заменить образ памяти процесса
exit (status)	Завершить выполнение процесса и вернуть статус
s = sigaction (sig, &act, &oldact)	Определить действие, выполняемое при приходе сигнала
s = sigreturn (&context)	Вернуть управление после обработки сигнала
s =sigprocmask (how, &set, &old)	Исследовать или изменить маску сигнала
s = sigpending (set)	Получить или установить блокированные сигналы s =
sigsushtnd (sigmask)	Заменить маску сигнала и приостановить процесс
s = kill (pid, sig)	Послать сигнал процессу
s = pause ( )	Приостановить выполнение процесса до след. сигнала

В случае ошибки выполнения системного вызова возвращаемое значение s равно минус 1, pid означает PID процесса, a residual - оставшееся время до предыдущего сигнала будильника.


7.6.9. Системные вызовы управления процессами в Unix

Системный вызов fork представляет собой единственный способ создания новых процессов в системе UNIX. Он создает точную копию оригинального процесса, включая все описатели файлов, регистры и все остальное.

После выполнения системного вызова fork значение всех соответствующих переменных в обоих процессах одинаково, но затем изменения переменных в одном процессе не влияют на переменные другого процесса.

Системный вызов fork возвращает значение, равное нулю для дочернего процесса, и значение идентификатора PID дочернего процесса для родительского. Таким образом два процесса могут определить, кто из них родитель, а кто дочерний процесс.

В большинстве случаев после системного вызова fork дочернему процессу потребуется выполнить программу, отличающуюся от программы, выполняемой родительским процессом.

Работа оболочки происходит следующим образом. Она считывает команды с терминала, с помощью системного вызова fork выполняет введенную команду, затем ждет окончания работы дочернего процесса, после чего считывает следующую команду. Для ожидания завершения дочернего процесса родительский процесс обращается к системному вызову waitpid.

7.6.9. Системные вызовы управления процессами в Unix

У этого системного вызова три параметра. В первом указывается PID процесса, завершение которого ожидается (если указать -1, то это будет означать ожидание завершения любого дочернего процесса).

Второй параметр представляет собой адрес переменной, в которую записывается статус завершения дочернего процесса (нормальное или ненормальное завершение, а также возвращаемое на выходе значение).

Третий параметр показывает, будет ли обращающийся к системному вызову процесс блокирован до завершения дочернего процесса или сразу получит управление после обращения к системному вызову.

В случае оболочки дочерний процесс должен выполнить команду, введенную пользователем. Он выполняет это при помощи системного вызова exec, который заменяет весь образ памяти содержимым файла, указанным в первом параметре системного вызова.

Ниже приводиться очень упрощенный пример использования системных вызовов fork, waitptd и exec (в упрощенной оболочке):

```
while  (TRUE)  {                                  /*Вечный цикл */
      type_prompt  ( );                           /*Вывести приглашение ко вводу*/
      read_command (command, params);  /*Прочитать с клавиатуры строку */
      pid = fork ( );                              /*Ответвить дочерний процесс*/
      if  (pid  < 0)  {
           printf (“Создать процесс невозможно”); /*Ошибка */
           continue;                               /*Следующий цикл */
       }
        if  (pid  != 0)  {
               waitpid ( -1, &status, 0 );          /*Род. пр-с  ждет завершения  доч.  пр-са*/
         }  else  {
         execve (command, params, 0);          /* Дочерний процесс выполняет работу*/
         }
}
```

7.6.9. Системные вызовы управления процессами в Unix

В самом общем случае у системного вызова exec три параметра: имя исполняемого файла, указатель на массив аргументов и указатель на массив строк окружения. Различные варианты этой процедуры, включая excel, execv, execle и execve, позволяют опускать эти параметры или указывать их иными способами.

Все эти процедуры обращаются к одному и тому же системному вызову. Хотя сам системный вызов называется exec, библиотечной процедуры с таким именем нет.

7.6.10. Системные вызовы для управления потоками

Стандартом POSIX (P1003.1c) предусматривается реализация потоков в пространстве пользователя и ядра ОС. Наиболее часто применяемые вызовы управления потоками приведены в таблице. При системной реализации потоков они являются настоящими системными вызовами. При использовании потоков на уровне пользователя они полностью реализуются в динамической библиотеке в пространстве пользователя.

Системный вызов	Описание
pthread_create	Создать поток в адр. простр. вызывающего процесса
pthread_exit	Завершить поток
pthread_ join	Подождать пока не завершится процесс
pthread_mutex_init	Создать новый мьютекс
pthread_ mutex_destroy	Уничтожить мьютекс
pthread_ mutex_lock	Заблокировать мьютекс
pthread_ mutex_unlock	Разблокировать мьютекс
pthread_ cond_init	Создать условную переменную
pthread_ cond_ destroy	Уничтожить условную переменную
pthread_ cond_ wait	Ждать условную переменную
pthread_ cond_ signal	Разблокировать один поток, ждущий условную переменную

err = pthread_create (&tid, attr, function, arg); - этот вызов создает новый поток, в котором работает программа function, а arg передается программе в качестве параметра.

7.6.10. Системные вызовы для управления потоками

Для создания потока **используется системный вызов pthread. Например,**
err = pthread_create (&tid, attr, function, arg);
Этот вызов создает в текущем процессе новый поток, в котором работает программа function, а arg передается этой программе в качестве параметра. Идентификатор нового потока хранится в памяти по адресу, на которой указывает первый параметр. С помощью параметра attr можно задавать для нового потока определенные атрибуты, такие как приоритет планирования.

Поток**, выполнивший свою работу и** ***желающий прекратить свое существование*****, обращается к вызову pthread_exit.**

Поток может подождать**, пока не завершится процесс, обративший к системному вызову pthread_join.**

Если ожидаемый поток завершил свою работу, системный вызов pthread__join выполняется мгновенно. В противном случае обратившийся к нему поток блокируется.

7.6.10. Системные вызовы для управления потоками

Синхронизация потоков может осуществляться при помощи мьютексов. **Как правило, мьютекс охраняет какой-либо ресурс, например буфер, совместно используемый двумя потоками. Чтобы гарантировать, что только один поток в каждый момент времени имеет доступ к общему ресурсу, предполагается, что потоки блокируют (захватывают) его, когда ресурс им больше не нужен.**

Мьютексы **предназначены для** ***кратковременной блокировки***, **например для защиты совместно используемой переменной. Они не предназначаются для долговременной синхронизации, например для ожидания, когда освободится накопитель на МП.**

Для долговременной синхронизации предоставляются переменные состояния. **Они используются следующим образом: один поток ждет, когда переменная примет определенное значение, а другой поток сигнализирует ему изменением этой переменной.**

7.6.11. Реализация процессов в системе Unix

У каждого процесса есть пользовательская часть, в которой работает программа пользователя.

Однако когда один из потоков обращается к системному вызову, происходит эмулированное прерывание с переключением в режим ядра.

После этого поток начинает работать в контексте ядра с полным доступом к ресурсам машины (со своим стеком ядра и счетчиком команд в режиме ядра).

7.6.11. Реализация процессов в системе Unix

Ядро поддерживает две ключевые структуры данных, относящиеся к процессам: **таблицу процессов** (резидентная) и **структуру пользователя** (выгружается на диск, когда процесс отсутствует в памяти).

Таблица процессов содержит:

1. **Параметры планирования**. Приоритеты процессов, процессорное время, потребленное за последний учитываемый период, количество времени, проведенное процессом в режиме ожидания. Используется для выбора активируемого процесса.
2. **Образ памяти**. Указатели на сегменты программы, данных и стека или на таблицы страниц. Когда процесса нет в памяти здесь содержится информация о его месте на диске.
3. **Сигналы.** Маски, характеризующие сигналы (игнорируемые, перехватываемые, доставляемые).
4. **Разное.** Текущее состояние процесса, ожидаемые события, PID процесса, идентификатор пользователя и др.

Структура пользователя включает:

1. **Машинные регистры (сохраняются здесь при переключении процесса).**
2. **Информацию о текущем системном вызове (параметры и результаты).**
3. **Таблицу дескрипторов файлов.**
4. **Учетную информацию. Данные о процессорном времени, использованном в пользовательском и системном режимах.**
5. **Стек ядра для использования процессом в режиме ядра.**

7.6.11. Реализация процессов в системе Unix

Когда выполняется системный вызов fork, вызывающий процесс обращается в ядро и ищет свободную ячейку в таблице процессов, в которую можно записывать данные о дочернем процессе. Если свободная ячейка находится, системный вызов копирует туда информацию из ячейки родительского процесса.

Затем он выделяет память для сегментов данных и для стека дочернего процесса, куда копируются соответствующие сегменты родительского процесса. Структура пользователя копируется вместе со стеком. Программный сегмент может либо копироваться, либо использоваться совместно, если он доступен только для чтения. Начиная с этого момента дочерний процесс может быть запущен.

Так, когда пользователь вводит с терминала команду ls, оболочка создает новый процесс, клонируя свой собственный процесс с помощью системного вызова fork. Новый процесс оболочки затем вызывает системный вызов exec, чтобы считать в свою область памяти содержимое исполняемого файла ls.

7.6.12. Планирование в системеUNIX

В системе UNIX используется двухуровневое планирование.

1. **Низкоуровневый алгоритм** выбирает процесс, готовый к работе из очереди, имеющей высший приоритет (у процессов ядра – отрицательный, у процессов пользователя – положительный). Время кванта – 100 мс.

Раз в секунду приоритет каждого процесса пересчитывается по формуле:

Priority = CPU_usage + nice + base

CPU_usage - среднее значение “тиков” (прерывания) таймера в секунду, при которых работал процесс в течении нескольких последних секунд. Эта величина со временем меняется:

CPU_usage (i) = CPU_usage (i – 1) / 2; i – номер интервала, на котором происходит вычисление нового приоритета.

nice = от – 20 до + 20 (по умолчанию = 0); (Системный администратор может запросить от -20 до -1)

base –назначается ОС (прошиты жестко и отрицательны для свопинга, дискового ввода-вывода и др.)

2. **Высокоуровневый алгоритм** перемещает процессы из памяти на диск и обратно.

7.6.12. Планирование в системеUNIX

Когда процесс эмулирует прерывание для выполнения системного вызова в ядре, он блокируется и удаляется из очереди.
Однако когда происходит событие, которого ждал процесс, он снова помещается в очередь с отрицательным значением.
Выбор очереди определяется событием, которого ждал процесс. Если процесс был заблокирован ожиданием ввода с терминала, то, очевидно, это интерактивный процесс и ему должен быть предоставлен наивысший приоритет, как только он перейдет в состояние готовности, чтобы гарантировать хорошее качество обслуживания интерактивных процессов. Таким образом, процессы, ограниченные производительностью процесса, т. е. находящиеся в положительных очередях, обслуживаются после того, как будут обслужены все процессы, ограниченные вводом-выводом (когда эти процессы окажутся заблокированными в ожидании ввода-вывода).

7.6.13. Управление памятью в системе UNIX

***У каждого процесса в системе UNIX есть адресное пространство, состоящее из трех сегментов*: текста (программы), данных и стека.**

Как правило, программный сегмент разрешен только для чтения, т. е. он не изменяется ни в размерах, ни по содержанию.

Сегмент данных содержит переменные, строки, массивы и другие данные. Он состоит из двух частей: инициализированных и неинициализированных данных.

С точки зрения ОС инициализированные данные не отличаются от текста программы - и тот и другой сегменты содержат сформированные компилятором последовательности битов, загружаемые в память при запуске программы.

Неинициализированные данные по традициям называются BSS (Bulk Storage System - запоминающее устройство большой емкости). Неинициализированные данные необходимы лишь с точки зрения оптимизации использования памяти. Вместо неинициализированных данных компилятор помещает в исполняемый файл слово, содержащее размер области неинициализированных данных в байтах. При запуске программы ОС считывает это слово, выделяет нужное число битов и обнуляет их.

7.6.13. Управление памятью в системе UNIX

В отличие от текстового сегмента, сегмент данных может модифицироваться. Программы изменяют свои переменные, более того, многим программам требуется динамическое выделение памяти во время выполнения. Чтобы реализовать это, ОС разрешает сегменту данных расти и уменьшаться. Программа использует библиотечную процедуру malloc для выделения памяти, которая использует системный вызов brk, увеличивающий сегмент данных.

Третий сегмент - это сегмент стека. На большинстве компьютеров он начинается около старших адресов виртуального адресного пространства и растет вниз к 0. Если указатель стека оказывается ниже нижней границы сегмента стека, происходит прерывание, при котором ОС понижает границу стека на одну страницу памяти. Программы явно не управляют размером стека.

Когда программа запускается, ее стек не пуст. Он содержит все переменные окружения (оболочки), а также командную строку, введенную в оболочке при вызове этой программы. Таким образом, программа может узнать параметры, с которыми она была запущена.

7.6.13. Управление памятью в системе UNIX

Большинство систем UNIX поддерживает *текстовые сегменты совместного использования*. Отображение обеспечивается аппаратным обеспечением виртуальной памяти. Сегменты данных и стека никогда не бывают общими, кроме как после выполнения системного вызова fork, и то только те страницы, которые не модифицируются любым из процессов.

Многие версии UNIX поддерживают *отображение файлов на адресное пространство памяти*. Это позволяет читать из файла и писать файл, как если бы это был массив, хранящийся в памяти. Произвольный доступ к файлу становится существенно более легким, чем при использовании системных вызовов, таких как read и write. Совместный доступ к библиотекам предоставляется именно при помощи этого механизма.

Дополнительно преимущество отображения файла на память заключается в том, что два или более процессов могут одновременно отобразить на свое адресное пространство один и тот же файл. Запись в этот файл одним из процессов мгновенно становится видимой всем остальным.

7.6.13. Управление памятью в системе UNIX

Стандартом POSIX системные вызовы для управления памятью не определены.

На практике в большинстве UNIX-систем есть системные вызовы для управления памятью.

Системный вызов brk указывает размер сегмента данных, задавая адрес первого байта за его пределами. Если новое значение больше старого, сегмент данных увеличивается, в противном случае уменьшается. В случае ошибки возвращаемое значение S= -1.

Системные вызовы mmap и unmap управляют отображением файлов на адресное пространство памяти. Адрес, по которому будет отображаться файл (или его часть), указывается параметром addr. Второй параметр, len, задает количество отображаемых байтов. Он должен быть кратен размеру страницы. Параметр prot задает режим защиты отображаемого файла (чтение, запись, исполнение). Четвертый параметр, flags, определяет, является ли отображаемый файл приватным или доступным для совместного использования. Параметр fd представляет собой дескриптор отображаемого файла. Последний параметр, offset, сообщает, с какого места должен отображаться файл (номер байта).

Системный вызов unmap отменяет отображение файла или его части.

7.6.13. Управление памятью в системе UNIX

До версии 3BSD большинство систем UNIX основывалось на свопинге, работающем следующим образом. Когда загружалось больше процессов, чем могло поместиться в памяти, некоторые из них выгружались на диск. Процесс всегда выгружался целиком. Перемещением между памятью и диском управлял верхний уровень планировщика - свопер (swapper). Выгрузка данных из памяти на диск инициировалась, когда у ядра кончалась свободная память из-за одного из следующих событий:

1. **Системному вызову fork требовалась память для дочернего процесса.**
2. **Системный вызов brk собирался расширить сегмент данных.**
3. **Разросшемуся стеку требовалась дополнительная память.**

Кроме того, когда наступало время запустить процесс, уже достаточно долго находящийся на диске, часто нужно было удалять из памяти другой процесс, чтобы освободить место для запускаемого процесса.

Для удаления свопер рассматривает сначала блокированные процессы. Если такие находились, то выбирался процесс с наивысшим значением суммы приоритета и времени пребывания памяти. Если блокированных процессов нет, то на основании этого же критерия выбирался готовый к выполнению процесс.

7.6.13. Управление памятью в системе UNIX

Каждые несколько секунд свопер исследовал список выгруженных процессов, проверяя процесс, дольше всех находящийся на диске. Затем свопер проверял, будет это легкий или тяжелый свопинг.

Легким считался тот, для которого не требовалось дополнительное высвобождение памяти. Тяжелым свопингом считался случай, когда требовалось удалить один или несколько процессов для размещения нужного процесса в памяти.

Затем весь этот алгоритм повторялся до тех пор, пока не выполнялось одно из следующих двух условий: (1) на диске не осталось процессов, готовых к работе, или (2) в памяти не осталось места для новых процессов. Чтобы не терять большую часть производительности системы на свопинг, ни один процесс не выгружается на диск, если он пробыл в памяти менее 2 с.

Свободное место в памяти и на диске учитывается при помощи связанного стека свободных пространств.

7.6.13. Управление памятью в системе UNIX

***Начиная с версии 3BSD университет в Беркли добавил к системе страничную подкачку*. Все нынешние версии UNIX имеют страничную подкачку по требованию.**

Идея, лежащая в основе страничной подкачки в системе 4BSD, проста. Чтобы работать, процессу не нужно целиком находиться в памяти. Требуется лишь структура пользователя и таблицы страниц. Страницы с сегментами текста, данных и стека загружаются в память по мере обращения к ним.

Страничная подкачка реализуется частично ядром и частично новым процессом, называемым страничным демоном.

Страничный демон - это процесс 2 (процесс 0 -свопер, а процесс 1 - init). Как и все демоны, страничный демон периодически запускается и «смотрит», есть ли доля работы.

Если он обнаруживает, что количество страниц в списке свободных страниц мало, страничный демон инициирует действия по освобождению дополнительных страниц. Память в 4BSD делится на три части. Первые две части - ядро операционной системы и карта памяти - фиксированы в физической памяти.

7.6.13. Управление памятью в системе UNIX

Остальная память делится на страничные блоки, каждый из которых может содержать страницу текста, данных или стека или находиться в списке свободных страниц.

Карта памяти содержит информацию о содержимом страничных блоков. Для каждого такого блока в карте памяти есть запись фиксированной длины.

При килобайтных страничных блоках и 16-байтовых записях карты памяти на нее расходуется менее 2% общего объема памяти. **Первые два поля записи** используются только тогда, когда соответствующий страничный блок находится в списке свободных страниц. В этом случае они сшивают свободные страницы в двухсвязный список.

Следующие три поля используются, когда страничный блок содержит информацию. В них содержится информация о логическом устройстве, содержащем копию страницы, о расположении блока копии страницы на логическом устройстве и хеш-код блока. **Еще три поля** содержат ссылку на запись в таблице процессов, тип хранящегося сегмента и смещение в сегменте. Последнее поле содержит некоторые флаги, нужные для алгоритма страничной подкачки.

7.6.13. Управление памятью в системе UNIX

При запуске процесс может вызвать *страничное прерывание*, если одной или нескольких его страниц не окажется в памяти.

При страничном прерывании ОС берет первый страничный блок из списка свободных блоков, удаляет его из списка и считывает в него требуемую страницу.

Если список свободных страниц пуст, выполнение процесса приостанавливается до тех пор, пока страничный демон не освободит страничный блок.

7.6.13. Алгоритм замещения страниц

***Алгоритм замещения страниц выполняется страничным демоном*. Раз в 250 мс он просыпается, чтобы сравнить количество свободных страничных блоков с системным параметром lotsfree (равным, как правило, 1/4 объема памяти).**

Если число свободных страничных блоков меньше, чем значение этого параметра, страничный демон начинает переносить страницы из памяти на диск, пока число свободных страничных блоков не станет равно lotsfree.

Если количество свободных страничных блоков больше или равно lotsfree, страничный демон снова засыпает. Если у компьютера много памяти и мало активных процессов, страничный демон спит практически все время.

Страничный демон использует модифицированную версию часового алгоритма. Это глобальный алгоритм, т. е. при удалении страницы не учитывается, чья это страница.

7.6.13. Алгоритм замещения страниц

Основной алгоритм часов работает, сканируя в цикле страничные блоки (как если бы они лежали на окружности циферблата часов).

На первом проходе, когда стрелка часов указывает на страничный блок, сбрасывается его бит использования.

На втором проходе у каждого страничного блока, к которому не было доступа с момента первого прохода, бит использования окажется сброшенным, и этот страничный блок будет помещен в список свободных страниц (после записи его на диск, если он «грязный»).

Страничный блок в списке свободных страниц сохраняет свое содержание, что позволяет восстановить страницу, если она потребуется, прежде, чем будет перезаписана.

7.6.13. Алгоритм замещения страниц

Изначально в Berkley UNIX использовался основной алгоритм часов, но затем было обнаружено, что при больших объемах оперативной памяти проходы занимают слишком много времени. Тогда алгоритм был заменен алгоритмом часов с двумя стрелками.

В этом алгоритме страничный демон поддерживает два указателя на карту памяти. При работе он сначала очищает бит использования передней стрелкой, а затем проверяет этот бит задней стрелкой. Если две стрелки находятся близко друг от друга, то только у очень активных используемых страниц появляется шанс, что к ним будет обращение между проходами двух стрелок. Если же стрелки разнесены на 359 градусов (т. е. задняя стрелка находится слегка впереди передней), то получаем исходный алгоритм часов. При каждом запуске демона стрелки проходят не полный оборот, а столько, сколько необходимо, чтобы количество страниц в списке свободных страниц было не менее lotsftee.

7.6.13. Алгоритм замещения страниц

Если операционная система обнаруживает, что частота подкачки страниц слишком высока, а количество свободных страниц все время ниже lotsfree, свопер начинает удалять из памяти один или несколько процессов, чтобы остановить состязание за свободные страничные блоки.

Алгоритм свопинга в системе 4BSD следующий. Сначала свопер проверяет, есть ли процесс, который бездействовал в течение 20 и более секунд. Если такие процессы есть, из них выбирается бездействующий в течение максимального срока и выгружается на диск. Если таких процессов нет, изучаются четыре самых больших процесса, из которых выбирается тот, который находится в памяти дольше всех, и выгружается на диск. При необходимости этот алгоритм повторяется до тех пор, пока не будет высвобождено достаточное количество памяти.

Каждые несколько секунд свопер проверяет, есть ли на диске готовые процессы, которые следует загрузить в память. Каждому процессу на диске присваивается значение, зависящее от времени его пребывания в выгруженном состоянии, размера, значения, использовавшегося при обращении к системному вызову nice, и оттого, как долго этот процесс спал, прежде чем был выгружен на диск.

7.6.13. Алгоритм замещения страниц

Эта функция обычно взвешивается так, чтобы загружать в память процесс, дальше всех находящийся в выгруженном состоянии, если только он не крайне большой.

Свопер загружает в память только структуру пользователя и таблицы страниц. Страницы с текстом, данными и стеком подгружаются при помощи обычной страничной подкачки.

Страничная подкачка в System V во многом организуется так же, как и в ОС 4BSD. Но есть два различия.

Во-первых, в System V вместо алгоритма часов с двумя стрелками используется оригинальный алгоритм часов с одной стрелкой. Более того, вместо того, чтобы помещать страницу в список свободных на втором проходе, она помещается туда только в случае, если она не использовалась в течение нескольких последовательных проходов. Это значительно увеличивает вероятность того, что освобожденная страница не потребуется тут же снова.

Во-вторых, вместо одной переменной lotsfree в System V используются две переменные - min и max. Когда число свободных страниц опускается ниже min, страничный демон начинает освобождать страницы. Этот процесс идет до тех пор, пока число страниц не сравнится с max. Такой подход позволяет избежать неустойчивости, возможной в системе 4BSD.

7.6.14. Ввод-вывод в системе UNIX

Подход, применяемый в UNIX, в построении системы ввода-вывода заключается в интегрировании всех устройств в файловую систему в виде так называемых специальных файлов.

Каждому устройству ввода-вывода назначается имя пути, обычно в каталоге /dev. Например, диск может иметь путь /dev/hdl, принтер - /dev/lp, у сети может быть путь /dev/net.

Доступ к этим специальным файлам осуществляется так же, как к обычным файлам. Для этого не требуется специальных команд или системных вызовов, например команда

cp file/dev/lp

скопирует file на принтер, в результате чего принтер распечатает этот файл.

7.6.14. Ввод-вывод в системе UNIX

Специальные файлы **подразделяются на две категории: блочные и символьные. Блочный специальный файл используется для дисков и позволяет прочитать любой блок по его адресу. Символьные специальные файлы используются для устройств ввода или вывода символьного потока. Эти файлы представляют такие устройства, как клавиатура, принтеры, мыши, сети и т. п.**

С каждым специальным файлом связан драйвер устройства, осуществляющий управление соответствующим устройством.

Примером ввода-вывода системы UNIX является работа с сетью. Ключевым понятием в схеме работы с сетью является *сокет*.

Сокеты подобны почтовым ящикам и телефонным розеткам в том смысле, что они образуют пользовательский интерфейс с сетью, как почтовые ящики формируют интерфейс с почтовой системой, а телефонные розетки - с телефонной сетью.

Сокеты **могут динамически создаваться и разрушаться. При создании сокета вызывающему процессу возвращается дескриптор файла, требующийся для установки соединения, чтения и записи данных, а также разрыва соединения.**

7.6.14. Ввод-вывод в системе UNIX

Каждый сокет поддерживает определенный тип работы в сети, указываемый при создании сокета. Наиболее распространенными типами сокетов являются:

1. Надежный, ориентированный на соединение байтового потока.
2. Надежный, ориентированный на соединение потока пакетов.
3. Ненадежная передача пакетов.

Первый тип сокета позволяет двум процессам на различных машинах установить между собой эквивалент «**трубы**» (канала между процессами на одной машине). Такая система гарантирует, что все посланные байты прибудут на другой конец канала в том порядке, как были отправлены.

Второй тип сокетов отличается тем, что сохраняет границу между пакетами.

Третий тип сокетов не дает гарантий доставки. Сеть может терять пакеты или предоставлять их в неверном порядке. Преимущество этого режима заключается в более высокой производительности, которая в некоторых ситуациях ценится существенно выше, нежели сохранность данных в процессе передачи (например, при доставке мультимедиа).

7.6.14. Ввод-вывод в системе UNIX

При создании сокета одним из параметров указывается **протокол**. Для надежных байтовых потоков используется протокол TCP, для ненадежной передачи - UDP (User Data Protocol). Для надежного потока пакетов протокола нет.

Прежде чем сокет может быть использован для работы в сети, с ним должен быть связан **адрес**.

Как только сокеты созданы на компьютере-источнике и компьютере-приемнике, между ними может быть установлено соединение. **Одна сторона** обращается к системному вызову **listen**, указывая в качестве параметра локальный сокет. При этом системный вызов создает буфер и блокируется до тех пор, пока не прибудут данные.

Другая сторона обращается к системному вызову **connect**, задавая в параметрах дескриптор файла для локального сокета и адрес удаленного сокета. Если удаленный компьютер принимает вызов, система устанавливает соединение между двумя сокетами.

7.7. Операционная система Windows 2000

7.7.1. История создания

1. Появление фирмы MICROSOFT и интерпретатора языка BASIC (1981 г.) для систем на базе микропроцессоров Intel8080 и ZilogZ80.
2. Первый ПК IBM PC, основанный на процессоре Intel8088, и MS DOS 1.0 (1981 г.). ОС состояла из резидентной программы размером 8 Кбайт, довольно близко копирующей CP/M, примитивную ОС для 8-разрядных процессоров Intel8080 и ZilogZ80.
3. В 1983 г. появилась DOS 2.0 – более мощная ОС, состоящая из 24 Кбайт резидентного кода.
4. Появление микропроцессора Intel80286 и PC AT на его основе, поставлявшегося с MS DOS 3.0 (1984 – 1986 гг.), занимавшей 36 Кбайт.
5. С годами в ОС MS DOS появилось много функций (современная наиболее распространенная версия 6.22), но она по-прежнему оставалась системой, ориентированной на командную строку. Сторонние разработчики поставляли различные оболочки, основанные на псевдографике и реализующие табличный интерфейс. Наиболее известной является оболочка Norton Commander.
6. Проект Lisa (графический интерфейс GUI Xerox, ПК Apple, С. Джобс, 1983 г.). Вдохновил Microsoft на создание графической интерфейсной оболочки.

7.7. Операционная система Windows 2000

7.7.1. История создания

7. Интерфейсная графическая оболочка над MS DOS – Windows 1.0 (1985 г.) – была чем-то вроде суррогата.
8. Версия Windows 2.0 для PC AT (1987 г.) – была не намного лучше предыдущей версии.
9. Версия Windows 3.0 для ПК с Intel 386 (1990 г.) – добилась большого коммерческого успеха.
0. Версии Windows 3.1 и 3.11 для ПК с Intel 386, 486 (1992 - 1994 гг.). Имели большой коммерческий успех.
1. Windows 95 с большинством особенностей монолитной ОС на основе MS DOS 7.0. Не была полностью 32-х разрядной. Содержала значительную долю 16-разрядного кода (1995 г.). Использовала файловую систему MS DOS. Поддерживала длинные имена.
2. Windows 98 со значительным наследием MS DOS, содержащая частично 16-разрядный код. Поддерживала большие дисковые разделы. Графический интерфейс интегрировал в себе Интернет и рабочий стол пользователя (1998 г.). Ядро не было реентерабельным, поэтому большинство процессов, зайдя в ядро, получали гигантский мьютекс и часто ждали, пока другой процесс покинет ядро.

У каждого процесса 4-гигабайтное адресное пространство, в котором первые 2 Гбайт полностью принадлежали процессу. Однако следующий 1 Гбайт совместно используется (с возможностью записи) всеми процессами системы. Нижний 1 Гбайт также совместно использовался всеми процессами системы, чтобы они могли получать доступ к векторам прерывания MS DOS. В результате ошибка в одной программе может повредить ключевые структуры данных, используемые посторонними процессами, вследствие чего эти процессы рушатся.

Последний 1 Гбайт совместно используется (с возможностью записи) процессами и ядром и содержит некоторые критические структуры данных. Любая программа, записав поверх этих структур какой-либо мусор (преднамеренно или нет), может вывести из строя всю систему. Однако совместное использование последнего 1 Гбайта необходимо для работы старых программ, написанные для MS DOS.

3. **Windows ME**, в основе повторяющая Windows 98, но с возможностью восстановления настроек ПК при неверной установке параметров (2000 г.). Исправлены некоторые ошибки, добавлены новые функции (улучшенные возможности воспроизведения изображений, музыки, фильмов, домашняя сеть , поддержка кабельных модемов и ADSL и др.

7.7. Операционная система Windows 2000

7.7.1. История создания

4. К концу 80-х годов MS осознала, что построение современной 32-разрядной ОС поверх 16-разрядной MS DOS - не лучшее решение. Компания MS наняла Давида Катлера, одного из ключевых разработчиков ОС VMS (корпорации DEC), и поручила ему возглавить работу над совершенно новой 32-разрядной ОС, совместимой с Windows. Эта новая система, названная позднее Windows NT (NT - New Technology - новая технология), предназначена для деловых приложений, решающих критически важные, ответственные задачи, а также для домашнего использования.
5. В 1993 г. выпущена ОС Windows NT 3.11. Начальный номер версии был выбран так, чтобы он соответствовал номеру версии популярной тогда Windows 3.11. ОС NT требовала значительно больше памяти, чем для Windows 3.1. Не было 32-разрядных программ. Потерпела неудачу на рынке пользователей ПК. Имела некоторый спрос на рынке серверов.
6. В 1994 и 1995 годах было выпущено несколько новых 3.x версий с небольшими изменениями (наиболее распространенной стала версия 3.51). Эти версии стали приобретать сторонников и среди пользователей настольных машин.

7.7. Операционная система Windows 2000

7.7.1. История создания

7. **Windows NT 4.0 (1996 г.) - первое значительное усовершенствование системы NT. Это мощная и надежная современная операционная система. Использует тот же самый пользовательский интерфейс, что и очень популярная в то время Windows 95. Выпускается в двух вариантах: как клиентская (Windows NT Workstation) и как серверная. Имела успех как на рынке пользователей, так и на рынке серверных ОС. Практически полностью написана на языке С с очень небольшими включениями на ассемблере для низкоуровневых функций, как обработка прерываний. Для написания пользовательского интерфейса было использовано некоторое количество строк С++. Обладает высокой переносимостью, различные ее версии работают на компьютере с процессором Pentium, Alpha, MIPS и Power PC. В настоящее время некоторые из этих версий не поддерживаются.**
8. **Windows 2000 (1999 г.) – она же версия NT 5.0. Использует популярный пользовательский интерфейс Windows 98. Является полностью 32-разрядной (планировался переход на 64-разрядную) многозадачной системой с индивидуально защищенными процессами.**

7.7. Операционная система Windows 2000

7.7.1. История создания

У каждого процесса **собственное 32-разрядное виртуальное адресное пространство.**

ОС работает в режиме ядра, тогда как **процессы - в пользовательском режиме**, что обеспечивает полноценную защиту. У процессов может быть один или несколько потоков, видимых для операционной системы и управляемых ею.

ОС **удовлетворяет требованиям безопасности** уровня С2 Министерства обороны США.

ОС обладает **поддержкой симметричных многопроцессорных систем** с числом процессоров от 2 до 32.

ОС Windows 2000 **содержит множество других функций, которые были ранее только в Windows 98**: поддержка устройств plug-and-play, шины USB, стандарта IEE1394 (Fire Wire), IrDA (Infrared Data Association - стандарт на инфракрасную передачу данных и вывод на печать), управление питанием.

Добавлен ряд новых функций, не присутствовавших ранее в других ОС MS: каталоговая служба Active Directory, система безопасности Kerberos, поддержка смарт-карт, инструменты мониторинга системы, лучшая интеграция лэптопов и настольных компьютеров, инфраструктура системного администрирования.

7.7. Операционная система Windows 2000

7.7.1. История создания

Новые свойства получила файловая система NTFS 5.0. Два пользователя могут совместно использовать один связанный файл. Как только один из них начинает запись в этот файл, автоматически создается копия этого файла. Кроме того, новая файловая система NTFS 5.0 допускает **шифрование файлов.**

ОС поддерживает **интернационализацию.** ОС Windows 2000 состоит из единого двоичного кода, работающего во всех странах мира. Для каждой установки системы и даже для каждого пользователя можно выбрать язык, который будет использоваться во время работы системы. Это возможно потому, что все пункты меню, строки диалоговых окон, сообщения об ошибках и другие текстовые строки помещены в специальные каталоги, по одному для каждого языка. Для поддержки языков не использующих латинский алфавит, например русского, греческого, иврита, японского Windows 2000 использует кодировку Unicode.

Есть **интерфейс командной строки.** Это новая 32-разрядная программа, включающая функциональность старой системы MS DOS, а также некоторые новые функции.

ОС Windows 2000 **обладает меньшей переносимостью**, чем NT 4.0. Она работает только на двух платформах Pentium и Intel - IA - 64.

7.7. Операционная система Windows 2000

7.7.1. История создания

Как и предыдущая версия NT, в настоящее время Windows 2000 поставляется в виде нескольких уровней продукта: Professional, Server, Advanced Server, Datacenter Server.

Однако различия между этими версиями незначительны, и в них используется один и тот же исполняемый двоичный код. При установке системы тип продукта записывается во внутренней базе данных (системном реестре). Во время загрузки ОС проверяет содержимое реестра, определяя версию программного продукта. Различия между ними:

Версия	Максимальный размер ОЗУ, Гбайт	CPU	Максимальное число клиентов	Размер кластера	Оптимизация
Professional	4	2	10	0	Время отклика
Server	4	4	Не ограничено	0	Пропускная способность
Advanced Server	8	8	Не ограничено	2	Пропускная способность
Datacenter Server	64	32	Не ограничено	4	Пропускная способность

7.7. Операционная система Windows 2000

7.7.1. История создания

Размер кластера означает способность операционной системы Windows 2000 представить для окружающего мира две или четыре отдельные машины в виде одного сервера.

В Windows 2000 Professional по-другому (по отношению к серверам) настраиваются **параметры по умолчанию**. В этой системе интерактивным процессам предоставляется преимущество перед пакетными заданиями, хотя это можно при необходимости изменить.

С серверными ОС поставляется дополнительное программное обеспечение, а с системой Datacenter Server поставляются дополнительные средства управления большими заданиями.

Формально **различием в версиях управляют** в нескольких местах программы всего две переменные, считываемые из реестра: ProductType и ProductSuite. В зависимости от этих значений выполняется слегка отличный код. Изменение значений этих переменных рассматривается как нарушение лицензии. Кроме того, система перехватывает любые попытки изменить их и регистрирует эти попытки нестираемым способом, так что впоследствии можно доказать факт нарушения лицензии.

7.7. Операционная система Windows 2000

7.7.1. История создания

Кроме основных операционных систем корпорация MS разработала несколько инструментальных программ для продвинутых пользователей.

Windows 2000 представляет собой чрезвычайно сложную систему, на сегодняшний день состоящую более чем из 29 млн. строк на языке С (580 томов по 1000 страниц и 50 строк на странице). Для сравнения - наибольшая по размеру версия UNIX имеет не более 4 млн. строк с учетом графического интерфейса пользователя (X Windows), который не входит собственно в ОС, поскольку считается пользовательским персоналом.

Инструментальные средства MS для продвинутых пользователей (наборы утилит для отладки и мониторинга системы):

1. Support Tools - средства поддержки;
2. Software Development Kit (SDK) – средства разработки программных продуктов;
3. Driver Development Kit (DDK) – средства разработки драйверов;
4. Resource Kit – набор ресурсов.

Инструментарий поддержки распространяется на компакт-диске Windows 2000 (каталог \support\tools).

SDK и DDK можно получить на сайте www.msdn.microsoft.com.

Resource Kit распространяется как розничный продукт MS.

Мощный набор инструментов можно получить на сайте www.sysinternals.com. Некоторые из этих программ предоставляют даже больше информации, чем соответствующие инструменты MS.

7.7.2. Структура системы Windows 2000

ОС Windows можно разделить на 2 части:

1. Основная часть ОС, работающая в режиме ядра (управление процессами, памятью, файловой системой , устройствами и т. д.).

2. Подсистемы окружения (среды), работающие в режиме пользователя (процессы, помогающие пользователям выполнять определенные системные функции).

Основная часть разделена на несколько уровней, каждый из которых пользуется службами лежащего ниже уровня. Основными уровнями являются:

- системные службы (сервисные процессы, являющиеся системными демонами);

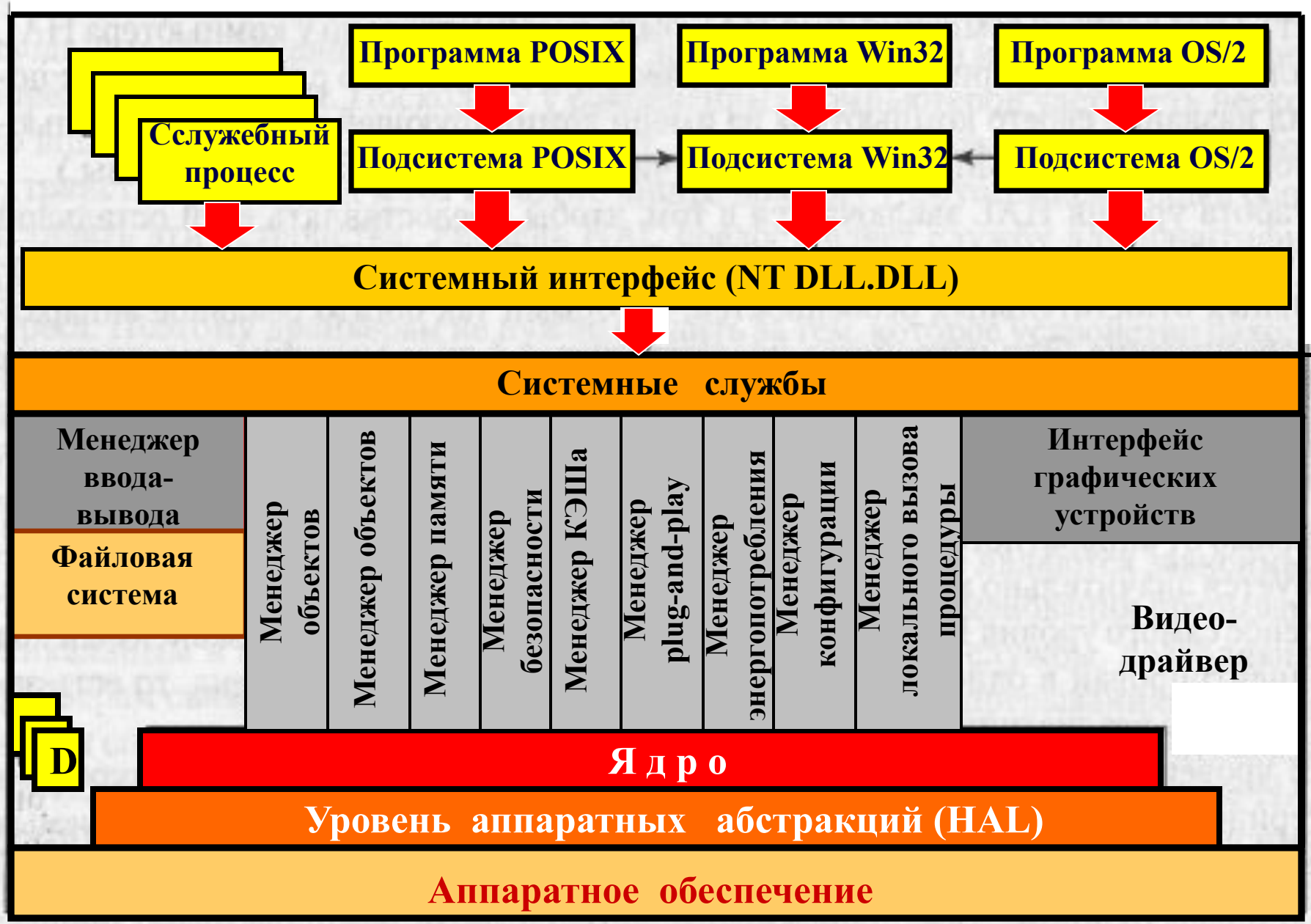
- исполняющая система (супервизор или диспетчер);

- драйверы устройств;

- ядро операционной системы;

- уровень аппаратных абстракций (HAL).

Два нижних уровня написаны на языке С и ассемблере и являются частично машинно-зависимыми. Верхние уровни написаны исключительно на языке С и почти полностью машинно-независимы. Драйверы написаны на С и в некоторых случаях на С++.


Программа POSIX
Программа Win32
Программа OS/2
Сслужебный процесс
Подсистема POSIX
Подсистема Win32
Подсистема OS/2
Системный интерфейс (NT DLL.DLL)
Режим пользователя
Системные службы
Менеджер ввода-вывода
Файловая система
Менеджер объектов
Менеджер объектов
Менеджер памяти
Менеджер безопасности
Менеджер КЭШа
Менеджер plug-and-play
Менеджер энергопотребления
Менеджер конфигурации
Менеджер локального вызова процедуры
Интерфейс графических устройств
Видео-драйвер
D
Я д р о
Уровень аппаратных абстракций (HAL)
Аппаратное обеспечение
Режим ядра

7.7.2. Структура системы Windows 2000

Windows NT 3.x состояла из относительно небольшого ядра и нескольких серверных процессов, работающих в режиме пользователя. Процессы пользователя взаимодействовали с серверными процессами с помощью модели клиент-сервер. В результате система NT получилась легко переносимой на другие компьютеры (Alpha корпорации DEC, Power PC корпорации IBM, MIPS фирмы SGI). Кроме того, такая структура защищала ядро от ошибок в коде серверов.

Однако для увеличения производительности начиная с версии NT 4.0 довольно большая часть операционной системы (например, управление системными вызовами и вся экранная графика) была возвращена в ядро.

Уровень аппаратных абстракций (Hardware Abstraction Layer – HAL)

Работа уровня HAL заключается в том, чтобы предоставить всей остальной системе абстрактные аппаратные устройства, свободные от индивидуальных особенностей аппаратуры. Эти устройства представляются в виде машинно-независимых служб (процедурных вызовов и макросов), которые могут использоваться остальной ОС и драйверами.

В уровень HAL **включены те службы**, которые зависят от от набора микросхем материнской платы и меняются от машины к машине в разумных предсказуемых пределах. Он разработан так, чтобы скрыть различия между материнскими платами разных производителей, но не различия между процессорами.

Программы HAL **находятся в файле hal.dll** каталога %System Root%\system32. Операционная система связывается с драйвером устройства, драйвер - с механизмом HAL, который непосредственно «разговаривает» с аппаратными средствами.


Уровень аппаратных абстракций (Hardware Abstraction Layer – HAL)

К службам уровня HAL относятся доступ к регистрам устройств, адресация к устройствам, не зависящим от шины, обработка прерываний и возврат из прерываний, операции DMA, управление таймером, часами реального времени, блокировками нижнего уровня и синхронизацией многопроцессорных конфигураций, интерфейс с BIOS и доступ к CMOS-памяти.

Уровень HAL не предоставляет абстракций или служб для специфических устройств ввода-вывода - клавиатур, мышей или дисков, а также блоков управления памятью MMU (Memory Management Unit).

Уровень HAL предоставляет три процедуры для чтения регистров устройств и еще три для записи в них:

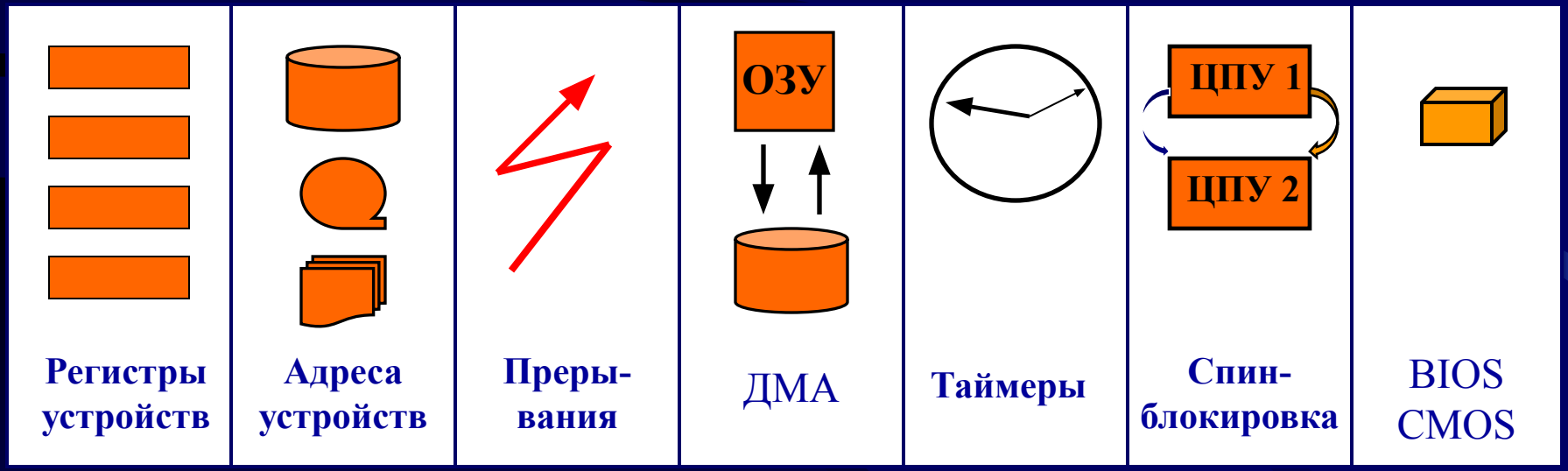
```
uc = READ_PORT_UCHAR(port);      WRITE_PQRT_UCHAR(port, uc);
us = READ_PORT_USHORT(port); WRITE_PORT_ USHORT (port, us);
ul = READ_ PORT_ULONG(port);   WRITE_PORT_ USHORT (port, ul);
```

Эти процедуры читают и пишут соответственно 8-, 16- и 32-разрядные целые числа без знака в указанный порт. Реализацией этих действий в виде обращения к физическим портам или регистрам, отображаемым на память, занимается уровень HAL. Так, как драйвер использует эти процедуры, то он без каких-либо изменений может быть перемещен на другую платформу.

Уровень аппаратных абстракций (Hardware Abstraction Layer – HAL)

После загрузки ОС уровень HAL общается с BIOS и CMOS, чтобы определить, какие шины и устройства ввода-вывода содержатся в системе и как их следует настроить. Затем эта информация помещается в реестр, чтобы другие компоненты системы могли просматривать их, не обращаясь напрямую к BIOS- или CMOS-памяти.

Некоторые функции уровня HAL



Уровень аппаратных абстракций (Hardware Abstraction Layer – HAL)

Функция спин-блокировки необходима, чтобы избежать конфликтов в многопроцессорных системах, и обеспечивает синхронизацию работы процессоров. В частности, такой метод синхронизации применяется в ситуациях, в которых доступ к ресурсу, как правило, получается всего на несколько команд процессора.

Поскольку уровень HAL является в большой степени машинно-зависимым, он должен в совершенстве соответствовать системе, на которой установлен, поэтому набор различных уровней HAL поставляется на компакт-диске Windows 2000. Во время установки системы из них выбирается подходящий уровень и копируется на жесткий диск в каталог \winnt\system32 в виде файла hal.dll. При всех последующих запусках ОС используется эта версия уровня HAL.


7.7.3. Уровень ядра

Назначение ядра – сделать остальную часть ОС независимой от аппаратуры. Для этого ядро на основе низкоуровневых служб HAL формирует абстракции более высоких уровней. Например, у уровня HAL есть вызовы для связывания процедур обработки прерываний с прерываниями и установки их приоритетов. Больше ничего в этом отношении HAL не делает. Ядро же предоставляет полный механизм для переключения контекста и планирования потоков.

Ядро также предоставляет низкоуровневую поддержку двум классам объектов ядра – управляющим объектам и объектам диспетчеризации. Эти объекты используются системой и приложениями для управления ресурсами компьютерной системы: процессами, потоками , файлами и т. д.

Каждый объект ядра – это блок памяти, выделенный ядром, доступный только ему и представляющий собой структуру данных, в которой содержится информация об объекте.

К управляющим объектам ядра относятся объекты, управляющие системой: объекты заданий, процессов, потоков, прерываний, DPC (Deferred Procedure Call – отложенный вызов процедуры), APC (Asynchronous Procedure Call – асинхронный вызов процедуры)

К объектам диспетчеризации ядра относятся объекты, изменение состояния которых могут ждать потоки. Это –семафоры, мьютексы, события, таймеры, очереди, файлы, порты, маркеры доступа и др.

7.7.3. Уровень ядра

Объект DPC используется, чтобы отделить часть процедуры обработки прерываний, для которой время является критичным, от той ее части, для которой время некритично. Как правило, процедура обработки прерываний сохраняет несколько аппаратных регистров, связанных с прерывающим устройством ввода-вывода, чтобы их можно было потом восстановить, и разрешает аппаратуре продолжать работу, но оставляет большую часть обработки на потом.

Например, когда пользователь нажимает на клавишу, процедура обработки прерываний от клавиатуры считывает из регистра код нажатой клавиши и разрешает прерывание от клавиатуры. Но эта процедура не должна немедленно обрабатывать введенный символ, особенно если в данный момент происходит что-то более важное (т. е. с более высоким приоритетом).

Очередь DPC представляет собой напоминание о том, что есть работа, которую следует выполнить позднее.

7.7.3. Уровень ядра

Объект APC похож на отложенный вызов процедуры DPC, но отличается тем, что асинхронный вызов процедуры выполняется в контексте определенного процесса.

Когда обрабатывается нажатая клавиша, не имеет значения, в каком контексте работает DPC, так как всё, что требуется сделать, - это исследовать введенный код и, возможно, поместить его в буфер в ядре.

Однако если по прерыванию потребуется скопировать буфер из пространства ядра в адресное пространство пользовательского процесса (например, по завершении операции чтения модема), тогда процедура копирования должна работать в контексте получателя, который нужен для того, чтобы в таблице страниц одновременно содержались и буфер ядра, и буфер пользователя.

7.7.4. Исполняющая система

Написана на языке С, не зависит от архитектуры машины и относительно просто может быть перенесена на новые машины. Исполняющая система состоит из 10 компонентов, между которыми нет жестких границ. Компоненты одного уровня могут вызывать друг друга. Большинство компонентов представляют собой наборы процедур, которые выполняются потоками системы в режиме ядра.

Менеджер объектов управляет всеми объектами, создаваемыми или известными операционной системе (процессами, потоками, семафорами, файлами и т. д.). Он управляет пространством имен, в которое помещается созданный объект, чтобы к нему можно было обратиться по имени. Остальные компоненты ОС пользуются объектами во время своей работы. При создании объекта менеджер объектов получает в адресном пространстве ядра блок виртуальной памяти и возвращает этот блок в список свободных блоков, когда объект уничтожается.

Менеджер ввода-вывода предоставляет остальной части ОС независимый от устройств ввод-вывод, вызывая для выполнения физического ввода-вывода соответствующий драйвер. К менеджеру ввода вывода относятся и файловые системы, формально являющиеся драйверами устройств. Существует два драйвера для файловых систем FAT и NTFS.

Менеджер процессов управляет процессами и потоками, включая их создание и завершение. Он основывается на объектах потоков и процессов ядра и добавляет к ним дополнительные функции. Это ключевой элемент многозадачности.

7.7.4. Исполняющая система

Менеджер памяти реализует архитектуру виртуальной памяти со страничной подкачкой по требованию ОС. Он управляет преобразованием виртуальных страниц в физические и реализует правила защиты, ограничивающие доступ каждому процессу только теми страницами, которые принадлежат его адресному пространству.

Менеджер безопасности приводит в исполнение сложный механизм безопасности Windows 2000, удовлетворяющий требованиям класса C2 Оранжевой книги Министерства обороны США. В этой книге перечислены правила, которые должна соблюдать система, начиная с аутентификации при регистрации и заканчивая управлением доступом, а также обнулением страниц перед их повторным использованием.

Менеджер кэша хранит в памяти блоки диска, которые использовались последнее время, чтобы ускорить доступ к ним и обслуживает все файловые системы. Взаимодействует с менеджером виртуальной памяти, чтобы обеспечить требуемую непротиворечивость.

Менеджер plug-and-play управляет установкой новых устройств, контролирует их динамическое подключение и осуществляет при необходимости загрузку драйверов.

Менеджер энергопотребления управляет потреблением энергии, следит за состоянием батарей, взаимодействует с ИБП. Он выключает монитор и диски, если к ним не было обращений в течение определенного интервала времени.

7.7.4. Исполняющая система

Менеджер конфигурации отвечает за сохранение реестра. Он добавляет новые записи и ищет запрашиваемые ключи.

Менеджер вызова локальной процедуры обеспечивает высокоэффективное взаимодействие между процессами и их подсистемами. Поскольку этот путь нужен для выполнения некоторых системных вызовов, эффективность оказывается критичной, поэтому здесь не используются стандартные механизмы межпроцессного взаимодействия.

Интерфейс графических устройств GDI (Graphic Device Interface) управляет графическими изображениями для монитора и принтеров, предоставляя системные вызовы для пользовательских программ. (Интерфейс Win32 и модуль GDI превосходят по объему всю остальную исполняющую систему.) Содержит оконный менеджер и драйвер дисплея. Первоначально располагался в пространстве пользователя. Начиная с NT 4.0 перенесен в пространство ядра.

Над исполняющей системой размещаются системные службы. Системные службы предоставляют интерфейс к исполняющей системе. Они принимают системные вызовы Windows 2000 и вызывают необходимые части исполняющей системы для их выполнения.

7.7.4. Исполняющая система

При загрузке операционная система Windows 2000 загружается в память, как набор файлов.

Основная часть ОС, состоящая из ядра и исполняющей системы, хранится в файле **ntoskrnl.exe (1674 Кбайт).** Уровень **HAL**, представляющий собой библиотеку общего доступа, находится в файле **hal.dll (80 Кбайт).** Интерфейс Win32 и графических устройств хранятся в файле **win32.sys (1690 Кбайт)**. Системный интерфейс для связи пользовательского режима с режимом ядра – файл **NTDLL.DLL** содержит **484 Кбайт**. Драйверы хранятся во множестве файлов, как правило, с расширением sys.

Существуют две версии файла ntoskrnl.exe: для однопроцессорных и многопроцессорных систем. Существуют версии для процессора Xeon, способного поддерживать более 4 Гбайт физической памяти, и для процессора Pentium, который поддерживает только 4 Гбайта памяти. Наконец, этот модуль может содержать или не содержать отладочные функции, в зависимости от чего он предназначается либо для отладки системы, либо для продажи.

7.7.5. Драйверы устройств

Существуют четыре вида драйверов: (1) аппаратных средств, (2) файловой системы, (3) фильтров и (4) сетевых устройств переадресации и серверов. Из этих четырех типов только драйверы аппаратных средств имеют непосредственный доступ к физическому оборудованию и согласованно используют механизм HAL.

Драйверы устройств устанавливаются в систему, добавляются в реестр и затем динамически загружаются при каждой загрузке системы. Драйверы не являются частью файла **ntoskrnl.exe.**

Каждый драйвер может управлять одним или несколькими устройствами ввода-вывода, но драйвер устройства может также выполнять действия, не относящиеся к какому-либо специфическому устройству, - шифровать поток данных или даже просто предоставлять доступ к структурам данных ядра.

Существуют драйверы для реально видимых и осязаемых устройств ввода-вывода, таких как диски и принтеры, но есть также драйверы для многих внутренних устройств и микросхем.

Корпорация Microsoft предоставляет специальный программный продукт, который нужно использовать для написания своих драйверов (Driver Development Kit (DDK) - средства разработки драйверов).

7.7.6. Реализация объектов

ОС Windows 2000 является объектно-ориентированной системой. Объект ОС предоставляет однородный и непротиворечивый интерфейс ко всем системным ресурсам и структурам данных, таким как процессы, потоки, семафоры и т. п.

Все объекты характеризуются следующими свойствами:

- **объекты именуются по одной и той же схеме;**
- **доступ к объекту осуществляется с помощью дескриптора объекта, через менеджер объектов;**
- **все проверки, связанные с защитой, производятся в одном месте, так что ни один процесс не может их обойти;**
- **совместное использование объектов организуется по одной и той же схеме;**
- **поскольку все объекты открываются и закрываются через менеджер объектов, легко отследить, какие объекты еще используются, а какие можно безопасно удалить;**
- **однородная модель управления объектами позволяет легко регулировать квоты ресурсов.**

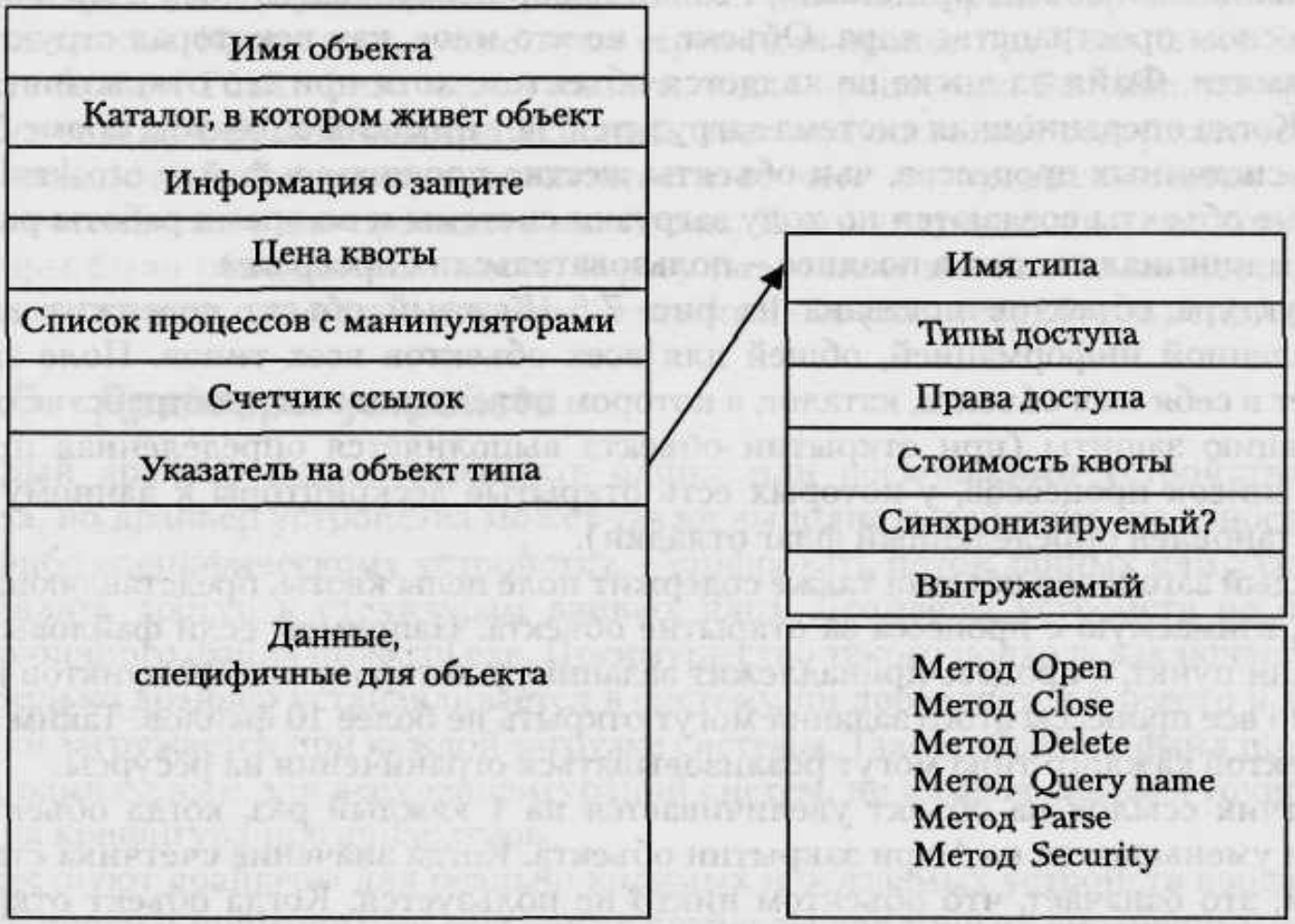

7.7.6. Реализация объектов

Исполняемый объект представляет собой набор последовательных слов в виртуальном адресном пространстве ядра. **Объект - не что иное, как некоторая структура данных в памяти.**

Когда операционная система загрузится, нет никаких объектов, кроме бездействующих системных процессов, чьи объекты жестко прошиты в файле ntoskml.exe.

Все остальные объекты создаются по ходу загрузки системы и во время работы различных программ инициализации, а позднее - пользовательских программ.

Структура объектов



7.7.6. Реализация объектов

Каждый объект содержит заголовок с определенной информацией, общей для всех объектов всех типов.

Поле заголовка включает в себя имя объекта, каталог, в котором объект живет в пространстве объектов, информацию защиты (при открытии объекта выполняется определенная проверка), а также список процессов, у которых есть открытые дескрипторы к данному объекту (если установлен определенный флаг отладки).

Каждый **заголовок объекта** также содержит поле цены квоты, представляющей собой «плату», взимаемую с процесса за открытие объекта. Например, если файловый объект стоит один пункт, а процесс принадлежит заданию, у которого есть 10 пунктов квоты, то суммарно все процессы этого задания могут открыть не более 10 файлов.

Счетчик ссылок на объект увеличивается на 1 каждый раз, когда объект открывается, и уменьшается на 1 при закрытии объекта. Когда значение счетчика становится нулевым, это означает, что объектом никто не пользуется.

7.7.6. Реализация объектов

Когда объект открывается или освобождается компонентом исполняющей системы, используется **второй счетчик**, даже если настоящий дескриптор при этом не создается.

Когда оба счетчика равны нулю, это означает, что объект больше не используется ни одним пользовательским, ни одним исполняющим процессом. В этом случае объект может быть удален, а его адресное пространство возвращено системе.

Доступ к объектам бывает необходим не только менеджеру объектов, но и другим частям исполняющей системы. Для этого исполняющая система содержит два пула в адресном пространстве ядра: для объектов и для других динамических структур данных. Один пул является **выгруженным**, а другой - **невыгруженным** (фиксированным в памяти).

Объекты, обращения к которым часты, хранятся в невыгруженном пуле; объекты, к которым обращения редки, хранятся в выгружаемом пуле. Когда памяти не хватает - этот пул может быть выгружен на диск и загружен обратно по страничному прерыванию в случае возникновения в нем необходимости.

7.7.6. Реализация объектов

Объекты, которые могут понадобиться, когда система выполняет критический участок программы (в этом случае подкачка не разрешается), должны храниться в невыгруженном пуле.

Объекты подразделяются на типы. Это означает, что у каждого объекта есть свойства, общие для всех объектов этого типа.

Тип объекта определяется указателем на объект типа. **Информация о типе** включает такие пункты, как название типа, данные о том, может ли поток ждать изменения состояния этого объекта (да - для мьютексов, нет - для открытых файлов) и должен ли объект этого типа храниться в выгружаемом или невыгружаемом пуле.

Наконец, самая важная часть объекта - **указатели на программы для определения стандартных операций**, таких как open, close, delete. Когда вызывается одна из этих операций, используется указатель на типовой объект, в котором выбирается и выполняется соответствующая процедура.

7.7.6. Реализация объектов
Наиболее употребительные типы объектов

Тип	Описание
Процесс	Процесс пользователя
Поток	Поток внутри процесса
Семафор	Семафор со счетчиком, используемый для синхронизации процессов
Мьютекс	Двоичный семафор, используемый для входа в критическую секцию
Событие	Объект синхронизации с перманентным состоянием
Порт	Механизм для передачи сообщений между процессами
Таймер	Объект, позволяющий потоку спать в течение фиксированного интервала времени
Очередь	Объект, используемый для уведомления о завершении асинхронного ввода-вывода
Открытый файл	Объект, ассоциированный с открытым файлом
Маркер доступа	Описатель защиты для некоторого объекта
Профиль	Структура данных, используемая для анализа загрузки центрального процессора
Секция	Структура, используемая для отображения файла на виртуальное адресное пространство
Ключ	Ключ реестра
Каталог объектов	Каталог для группировки объектов в менеджере объектов
Символьная ссылка	Указатель на другой объект по имени
Устройство	Объект устройства ввода-вывода
Драйвер устройства	У каждого загруженного драйвера есть свой объект

7.7.6. Реализация объектов

Для каждого процесса и потока существует соответствующий объект, хранящий основные данные, необходимые для управления этим процессом или потоком.

Объекты семафор, мьютекс и события имеют отношение к синхронизации процессов.

Объекты порт, таймер и очередь также имеют отношение к связи и синхронизации. **Порты** представляют собой каналы между процессами, использующиеся для обмена сообщениями. **Таймеры** предоставляют способ блокировать процесс или поток на определенное время. **Очереди** применяются для уведомления потоков о том, что начатая ранее синхронная операция ввода-вывода завершена.

Объекты открытых файлов создаются при открытии файла.

Менеджеры доступа представляют собой объекты безопасности. Они идентифицируют пользователя и сообщают, какие привилегии имеет данный пользователь.

Профили представляют собой структуры, используемые для хранения периодически фиксируемых значений счетчика команд работающего потока, которые позволяют определить, на что тратит свое время данная программа.

7.7.6. Реализация объектов

Секции являются объектами, используемыми системой памяти для управления отображаемыми на память файлами. Они хранят сведения о том, какой файл на какие адреса памяти отображается.

Ключи представляют собой ключи реестра и применяются для установки связи между именем и значением.

Каталоги объектов позволяют объединять связанные объекты тем же способом, каким обычные каталоги объединяют файлы в файловой системе.

Символьные ссылки также подобны своим двойникам в файловой системе: они позволяют имени в одной части пространства имен объектов ссылаться на объект в другой части этого пространства имен.

У каждого известного системе устройства есть **объект устройства**, содержащий информацию о нем и использующийся для ссылки на устройство в системе.

У каждого **загруженного драйвера** есть объект в пространстве объектов.

Пользователи могут создавать или открывать уже существующие объекты при помощи вызовов Win 32, таких как CreateSemaphere или OpenSemaphere. Эти вызовы являются библиотечными, которые в конечном итоге обращаются к настоящим системным вызовам.

7.7.6. Реализация объектов

Для выполнения задачи слежения за объектами менеджер объектов использует пространство имен объектов

Во время загрузки различные части исполняющей системы создают каталоги и заполняют их объектами. Например, когда менеджер plug-and-play обнаруживает новые устройства, он создает по объекту для каждого устройства и помещает эти объекты в пространство имен. Когда система полностью загружена, все устройства ввода-вывода, дисковые разделы и другие устройства (модемы, сканеры и др.) оказываются представленными в пространстве имен объектов.

Некоторые компоненты исполняющей системы просматривают реестр, чтобы определить, что им делать. Пример - драйверы устройств. При загрузке система просматривает реестр, чтобы определить, какие драйверы нужны. При загрузке каждого драйвера создается объект, а его имя добавляется в пространство имен объектов. В системе обращение к драйверу осуществляется по указателю на его объект.

Без использования специальных средств просмотра пространства имен объектов невидимо для пользователей. Одним из таких средств является программа winobj, которую можно бесплатно получить на сайте www.sysinternals.com.

7.7.6. Реализация объектов
Некоторые каталоги

Каталог	Описание
\??	Начальное место для поиска устройств MS DOS
Device	Все обнаруженные устройства ввода-вывода
Driver	Объекты, соответствующие каждому загруженному драйверу устройства
ObjectTypes	Объекты типов, например поток, мьютекс, семафор и т.д.
Windows	Объекты для отправки сообщений всем окнам
BaseNameObjs	Объекты, создаваемые пользователем (семафоры и т.д.)
Arcname	Имена разделов, обнаруженные загрузчиком
NLS	Объекты языковой поддержки.
FileSystem	Объекты драйверов файловой системы и объекты распознавателя файловой системы.
Security	Объекты системы безопасности.
KnownDLLs	Совместно используемые библиотеки, находящиеся в открытом состоянии.

7.7.7. Подсистемы окружения

В Windows 2000 существует три типа компонентов, работающих в режиме пользователя: динамические библиотеки DLL, подсистемы окружения и служебные процессы.Эти компоненты работают вместе, предоставляя пользовательским процессам три различных документированных интерфейса прикладного программирования API (Win32, POSIX, OS/2).

У каждого интерфейса есть список библиотечных вызовов, которые используются программистами. Работа библиотек DLL и подсистем окружения заключается в том, чтобы реализовать функциональные возможности опубликованного интерфейса, скрывая истинный интерфейс системных вызовов от прикладных программ.

Программы, пользующиеся интерфейсом Win32, содержит, как правило, большое количество обращений к функциям Win32 API. Поэтому, если работает одновременно несколько программ, то в памяти будут находиться многочисленные копии одних и тех же библиотечных процедур. Чтобы избежать подобной проблемы Windows поддерживает динамически присоединяемые библиотеки DLL. При этом при запуске нескольких приложений, использующих одну и ту же DLL, в памяти будет находиться только одна копия текста DLL.

В каталоге \winnt\system32 есть более 800 отдельных файлов DLL (130 Мбайт при числе вызовов API более13000).

7.7.7. Подсистемы окружения

Каждая динамическая библиотека содержит набор тесно связанных библиотечных процедур и все их структуры данных в одном файле, как правило (но не всегда), с расширением .dll.

Когда приложение компонуется, компоновщик видит, что некоторые библиотечные процедуры принадлежат к динамичным библиотекам, и записывает эту информацию в заголовок исполняемого файла.

Обращения к процедурам динамических библиотек производятся не напрямую, а при помощи **вектора передачи** в адресном пространстве вызывающего процесса. Изначально этот вектор заполнен нулями, так как адреса вызываемых процедур еще не известны.

При запуске прикладного процесса все требуемые динамические библиотеки обнаруживаются (на диске или в памяти) и отображаются на виртуальное адресное пространство процесса.

Затем вектор передачи заполняется верными адресами, что позволяет вызвать библиотечные процедуры через этот вектор с незначительной потерей производительности. Выигрыш этой схемы заключается в том, что при запуске нескольких приложений, использующих одну и ту же библиотеку, в физической памяти требуется только одна копия текста DLL (но каждый процесс получает собственную копию приватных статических данных в DLL).

7.7.7. Подсистемы окружения

Каждый пользовательский процесс, как правило, связан с несколькими динамическими библиотеками, совместно реализующими интерфейс Win 32. Чтобы обратиться к вызову API, вызывается одна из процедур в DLL.

Дальнейшие действия зависят от вызова Win32 API.

Разные вызовы реализуются по-разному.

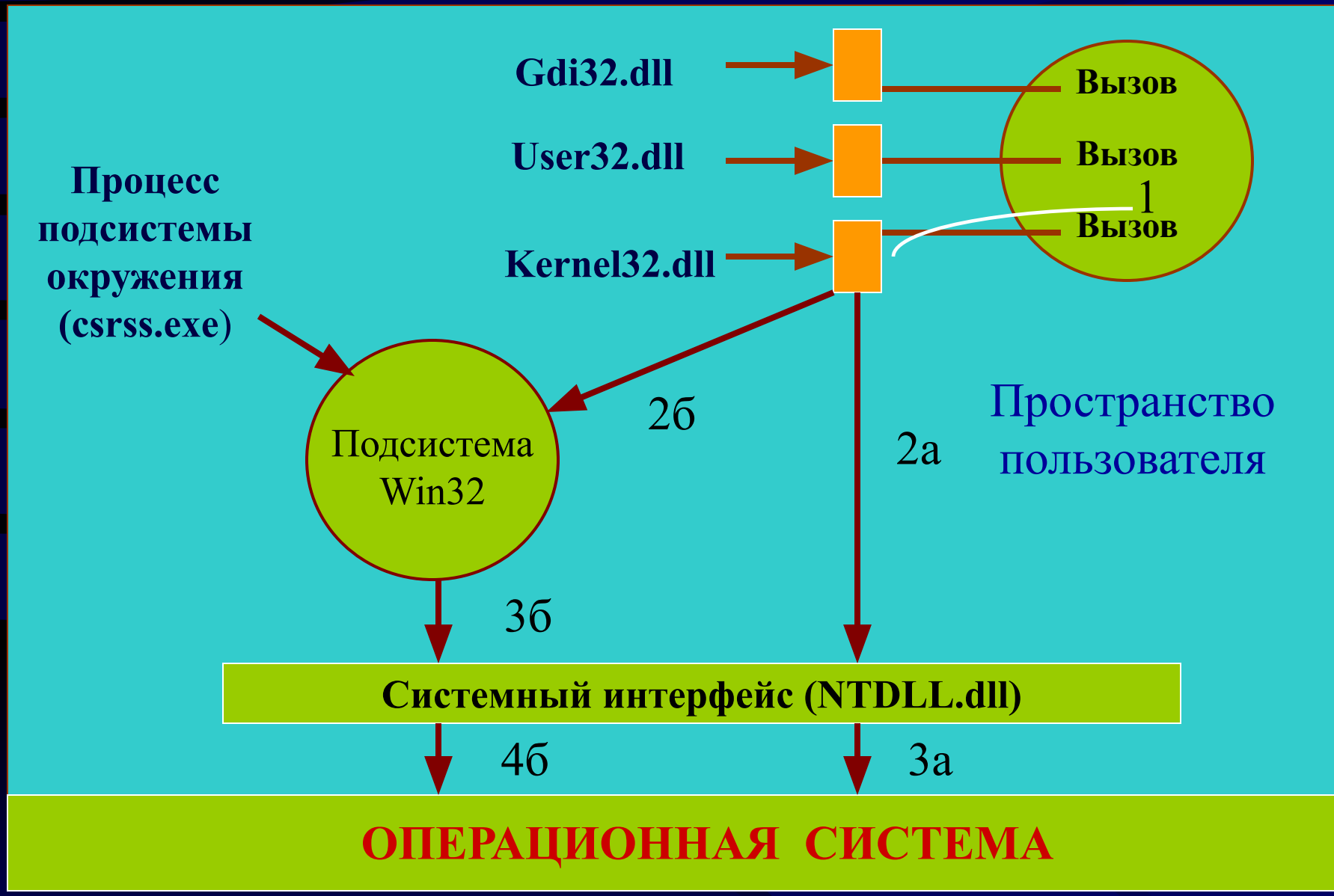
В некоторых случаях динамические библиотеки обращаются к другой динамической библиотеке (ntdll.dll), которая, в свою очередь, обращается к ядру операционной системы.

Динамическая библиотека может также выполнить всю работу самостоятельно, совсем не обращаясь к системным вызовам.

Для других вызовов Win32 API выбирает другой маршрут: сначала к процессу подсистемы Win32 (csrss.exe) посылается сообщение, выполняющее некоторую работу и обращающееся к системному вызову.

При этом в некоторых случаях подсистема также выполняет всю работу в пространстве пользователя и немедленно возвращает управление.

Передача сообщения между прикладным процессом и процессом подсистемы Win32 оптимизирована по времени, для чего использован механизм вызова локальной процедуры, реализованный в исполняющей системе.


Gdi32.dll
User32.dll
Kernel32.dll
Вызов
Вызов
1
Вызов
Процесс подсистемы окружения (csrss.exe)
Подсистема Win32
2б
2а
Пространство пользователя
3б
Системный интерфейс (NTDLL.dll)
4б
3а
ОПЕРАЦИОННАЯ СИСТЕМА

7.7.8. Загрузка Windows

Самотестирование при включении (Power-Jn Self-Test. POST)

Системная BIOS ищет загрузочный диск (в порядке, заданном пользователем)

Считывание главной загрузочной записи MBR загружаемого диска

Анализ таблицы разделов и определение раздела, в котором находится загружаемая ОС

Передача управления загрузочному сектору 0 раздела, в котором находится ОС

Программа загрузочного сектора находит в корневом каталоге раздела файл ntldr

Программа ntldr считывает файл Boot.ini, в котором хранятся списки файлов hal.dll, ntoskerrnl.exe (в Boot.ini указано количество ЦПУ, ОП, F часов реального времени). Загружает и выполняет программу ntdetect.com.

Загружаются файлы hal.dll, ntoskerrnl.exe и bootvid.dll. Программа ntldr считывает реестр, чтобы найти драйверы, необходимые для завершения загрузки (для микросхем мат. платы, клавиатуры, мыши и т. д.).

Загрузчик считывает драйверы и передает управление программе ntoskernl.exe

Общие процедуры инициализации и инициализация компонентов исполняющей системы. Загрузка и инициализация драйверов устройств и сервисов, создание файлов подкачки

Создание сеансового менеджера smss.exe

Запуск подсистемы окружения Win32 (csrss.exe), считывание реестра и выполнение указанных команд, создание файлов подкачки и открытие нужных DLL

Создание демона регистрации **winlogon.exe**

Демон регистрации

Менеджер аутентификации

Создание менеджера аутентификации **lsass.exe**

Запуск родительского процесса всех служебных процессов **services.exe.** По информации, хранящейся в реестре определяет, какие демоны в пространстве пользователя надо запустить (сервер принтера, файловый сервер, обработчик входящей электронной почты, факсов и т. д.)..

Выбор из реестра профиля пользователя и запуск требуемей оболочки.

7.7.9. Файловая система Windows 2000

Основные свойства файловой системы NTFS:

1. Поддержка больших файлов и больших дисков (объем до 2^{64} байт).
2. Восстанавливаемость после сбоев и отказов программ и аппаратуры управления дисками.
3. Высокая скорость операций, в том числе для больших дисков.
4. Низкий уровень фрагментации, в том числе для больших дисков.
5. Гибкая структура, допускающая развитие за счет добавления новых типов записей и атрибутов файлов с сохранением совместимости с предыдущими версиями ФС.
6. Устойчивость к отказам дисковых накопителей.
7. Поддержка длинных символьных имен.
8. Контроль доступа к каталогам и отдельным файлам.

Структура тома NTFS

Основой структуры тома является *главная таблица файлов* (***Master File Table, MFT***), которая содержит одну или несколько записей для каждого файла (каталога) тома и одну запись для самой себя (размер записи – 1, 2 или 4 Кбайт).

Каждый том состоит из линейной последовательности блоков (кластеров). Размер кластера фиксирован для каждого тома и варьируется от 512 байт до 64 Кбайт в зависимости от размера тома. Обращение к блокам осуществляется по их смещению от начала тома. Порядковый номер кластера в томе – ***логический номер кластера*** *(Logical Cluster Number, LCN)* – 64-х разрядное число.

Все файлы в томе NTFS идентифицируются номером файла, который определяется позицией в файла в MFT (Master File Table) – главной таблице файлов. Файл состоит из последовательности кластеров, порядковый номер кластера внутри файла называется ***виртуальным номером кластера*** (*Virtual Cluster Number, VCN*).

Базовая единица распределения дискового пространства – ***отрезок*** – непрерывная область кластеров.

Структура тома NTFS

Адрес отрезка – пара (LCN, k), где LCN – логический номер первого кластера, k –количество кластеров в отрезке.

Адрес файла (или его части) – (LCN, VCN, k).

Загрузочный блок тома NTFS располагается в начале тома. Загрузочный блок содержит стандартный блок параметров BIOS, количество блоков в томе, начальный логический номер кластера основной и зеркальной копии MFT.

Главной структурой данных в каждом томе является *главная файловая таблица* MFT (Master File Table), представляющая собой линейную последовательность записей 2-Кбайт размера. Каждая запись MFT описывает один файл или каталог.

Сама главная файловая таблица представляет собой файл и может располагаться в любом месте тома. Кроме того, этот файл может расти до максимального размера 2^{48} записей.

Первые 16 записей MFT зарезервированы для файлов метаданных NTFS. Каждая запись описывает файл, у которого есть атрибуты и блоки данных, как у любого файла. У каждого такого файла есть имя, начинающееся с символа доллара, указывающего на то, что это файл метаданных.

Структура тома NTFS

0. Описание MFT, в том числе адреса всех ее отрезков.

Первая запись (нулевая) описывает сам файл MFT. В частности, она содержит информацию о расположении блоков файла MFT, что позволяет системе найти файл MFT. Чтобы найти всю остальную информацию о файловой системе, у операционной системы должен быть некий способ нахождения первого блока файла MFT. Номер первого блока файла MFT содержится в загрузочном блоке, куда он помещается при установке системы.

1. Зеркальная копия 3-х первых записей MFT.

Эта информация является настолько ценной, что наличие второй копии может быть необходимо на случай, если один из первых блоков главной файловой таблицы вдруг станет дефектным

2. Журнал для восстановления файловой системы.

В нем фиксируются все изменения происходящие в файловой системе.

3. Информация о томе (имя, версия и др. информация).

4. Таблица определения атрибутов (ссылка на файл $AttrDef).


Структура тома NTFS

5. Индекс корневого каталога.

Содержит данные о корневом каталоге. Последний представляет собой файл, который может увеличиваться в размерах.

6. Битовая карта кластеров.

Учитывает свободное место на диске.

7. Загрузочный сектор раздела.

Указывает на файл начальной загрузки.

8. Список дефектных кластеров.

Используется для того, чтобы связать вместе все дефектные блоки и гарантировать, что они никогда не встретятся в файлах.

9. Описатели защиты файлов.

10. Таблица преобразования регистра символов (для Unicode).

11. Таблица квот, точек повторного анализа и др.

12 – 15 – зарезервировано.


Структура тома NTFS

Каждая запись MFT состоит из заголовка записи, за которым идет последовательность пар (заголовок атрибута, значение).

Заголовок записи содержит магическое число, используемое для проверки действительности записи; порядковый номер, обновляемый каждый раз, когда запись используется для нового файла; счетчик обращений к файлу; действительное количество байт, используемых в записи; идентификатор (индекс, порядковый номер) базовой записи (используемый только для записи расширения), а также другие различные поля.

Следом за заголовком записи располагаются **пары атрибут, значение**. Каждый атрибут начинается с заголовка, идентифицирующего этот атрибут и сообщающего длину значения. В файловой системе NTFS определено 13 атрибутов, которые могут появляться в записях MFT.

Структура файлов NTFS

Файлы и каталоги состоят из набора атрибутов. Атрибут содержит следующие поля: тип, длина, имя (образуют заголовок) и значение.

Системные атрибуты:

1. Стандартная информация (сведения о владельце, флаговые биты, время создания, время обновления и др.).
2. Имя файла в кодировке Unicode, м.б. повторено для имени MS DOS.
3. Список атрибутов (содержит ссылки на номера записей MFT, где расположены атрибуты), используется для больших файлов.
4. Версия – номер последней версии файла.
5. Дескриптор безопасности (описатель защиты) – список прав доступа ACL.
6. Версия тома –используется в системных файлах тома.
7. Имя тома.
8. Битовая карта MFT – карта использования блоков тома.
9. Корневой индекс – используется для поиска файлов в каталоге.
10. Размещение индекса – нерезидентная часть индексного списка (для очень больших каталогов).
11. Идентификатор объекта – 64-разрядный идентификатор файла, уникальный для данного тома.
12. Данные – поточные данные файла.
13. Точка повторного анализа - используется для монтирования и симв. ссылок

Структура файлов NTFS

Как правило, значения атрибутов располагаются непосредственно за заголовками, но если длина значения слишком велика, чтобы поместиться в запись таблицы MFT, она может быть помещена в отдельный блок диска. Такой атрибут называется *нерезидентным атрибутом*. Например, таким атрибутом является атрибут данных.

Длина заголовков **резидентных атрибутов** 24 байт, заголовки для нерезидентных атрибутов длиннее, так как они содержат информацию о месте расположения атрибута.

Стандартное информационное поле содержит сведения о владельце файла, информацию о защите, временные штампы, необходимые для стандарта POSIX, счетчик жестких связей, бит «только чтение», «архивный» бит и т. д. Это поле имеет фиксированную длину и всегда присутствует.

Имя файла хранится в кодировке Unicode в поле переменной длины.

В ОС NT/4.0 **информация о защите файла** могла содержаться в атрибуте файла, но в Windows 2000/2003 эти данные хранятся в отдельном файле, что позволяет нескольким файлам совместно пользоваться общими описателями защиты.

Структура файлов NTFS

Список атрибутов нужен на случай, если атрибуты не помещаются в запись MFT.

Атрибут **идентификатор объекта** задает файлу уникальный номер.

Точка повторного анализа велит процедуре, анализирующей имя файла, выполнить специальные действия. Этот механизм применяется для монтирования устройств и символьных ссылок.

Атрибуты **имя тома** и **версия тома** используются только для идентификации тома.

Еще три атрибута используются для реализации каталогов.

Поток данных утилиты регистрации используется шифрующей файловой системой.

Имя потока данных, если оно присутствует, располагается в заголовке атрибута «Данные». Следом за этим заголовком располагается либо список дисковых адресов, определяющий положение файла на диске, либо - для файлов длиной всего в несколько сотен байтов (а таких файлов довольно много) - сам файл.

Метод помещения самого содержимого файла в запись MFT (если позволяет размер) называется ***непосредственным файлом.***

В противном случае в записи MFT хранится резидентная часть файла (некоторые его атрибуты), а остальная часть файла хранится в отдельном отрезке тома или нескольких отрезках.

Структура файлов NTFS

Конечно, в большинстве случаев все данные файла не помещаются в запись MFT, поэтому атрибут «данные», как правило, является нерезидентным.

Для увеличения эффективности дисковые блоки файлам назначаются по возможности в виде серий последовательных блоков (сегментов файла).

Блоки в файле описываются последовательностью записей, каждая из которых описывает последовательность логически непрерывных блоков.

Непрерывный файл описывается всего одной записью. Каждая запись начинается с заголовка, определяющего смещение первого блока в файле. За заголовком располагаются пары, в которых содержатся дисковые адреса и длины серий блоков.

Файлы NTFS в зависимости от способа размещения делятся на небольшие, большие, очень большие и сверхбольшие.

Пример небольшого файла NTFS

Блоки диска 20 – 23, 64 – 65, 80 - 82

Пример большого файла NTFS

Структура файлов NTFS

Если файл очень большой, то иногда бывает необходимо использовать две или более записей главной файловой таблицы, чтобы вместить список всех блоков файла.

В этом случае первая запись MFT называется *базовой записью* и указывает на другие записи MFT. Какие из элементов главной файловой таблицы свободны, учитывается в битовом массиве.

Если данные файла не помещаются в одну запись MFT, то этот факт отражается в заголовке атрибута Data, который содержит признак того, что этот атрибут является нерезидентным, т. е. находится в отрезках вне таблицы MFT. В этом случае атрибут Data содержит адресную информацию (LCN, VCN, К) каждого отрезка данных.

Если файл настолько велик, что его атрибут данных, хранящий адреса нерезидентных отрезков данных, не помещается в одной записи, то этот атрибут помещается в базовую и дополнительные записи MFT, а ссылка на такой атрибут помещается в базовую запись файла. Эта ссылка содержится в атрибуте Attribute List.

Структура файлов NTFS

Каждый каталог NTFS представляет собой один вход в таблицу MFT, который содержит атрибут **Index Roof**.

Индекс содержит список файлов, входящих в каталог. Индексы позволяют сортировать файлы для ускорения поиска, основанного на значении определенного атрибута. NTFS позволяет использовать для сортировки любой атрибут, если он хранится в резидентной форме. Имеются две формы хранения списка файлов.

Если количество файлов в каталоге невелико, то список файлов может быть резидентным в записи MFT, являющейся каталогом. Для хранения списка используется единственный атрибут - **Index Roof**. Список файлов содержит значения атрибутов файла. По умолчанию это имя файла и номер записи MFT, содержащий начальную запись файла.

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. Имена файлов резидентной части списка файлов являются узлами так называемого B-дерева (двоичного дерева).

Структура файлов NTFS

Остальные части списка файлов размещаются вне MFT. Для их поиска используется специальный атрибут Index Allocation, представляющий собой адреса отрезков, хранящих остальные части списка файлов каталога.

Одни части списков являются листьями дерева, а другие являются промежуточными узлами, т. е. содержат наряду с именами файлов атрибут Index Location, указывающий на списки файлов более низких уровней.

7.7.10. Сжатие файлов и шифрующая файловая система

Файловая система NTFS поддерживает прозрачное сжатие файлов. Файл может быть создан в сжатом режиме. Это значит, что файловая система NTFS будет автоматически пытаться сжать блоки этого файла при записи их на диск и автоматически распаковывать их при чтении.

Сжатие данных файла происходит следующим образом. Когда файловая система NTFS записывает на диск файл, помеченный для сжатия, она изучает первые 16 (логических) блоков файла, независимо оттого, сколько сегментов на диске они занимают. Затем к этим блокам применяется алгоритм сжатия. Если полученные на выходе блоки могут поместиться в 15 или менее блоков, то сжатые данные записываются на диск, предпочтительно в виде одного сегмента. Если получить выигрыш хотя бы в один блок не удается, то данные 16 блоков так и записываются в несжатом виде. Затем весь алгоритм повторяется для следующих 16 блоков и т. д.

При чтении сжатого файла NTFS должна знать, какие из сегментов файла сжаты, а какие нет. Она видит это по дисковым адресам. Дисковый адрес 0 указывает на то, что предыдущий сегмент сжат. (Заметим, что дисковый блок 0 не может использоваться для хранения данных, поскольку в этом блоке содержится загрузочный сектор.)

7.7.10. Сжатие файлов и шифрующая файловая система

Произвольный доступ к сжатому файлу также возможен. Для получения доступа к данному блоку сначала потребуется прочитать и распаковать весь сегмент файла. После этого система может определить, где находится данный блок, и передать его читающему процессу.

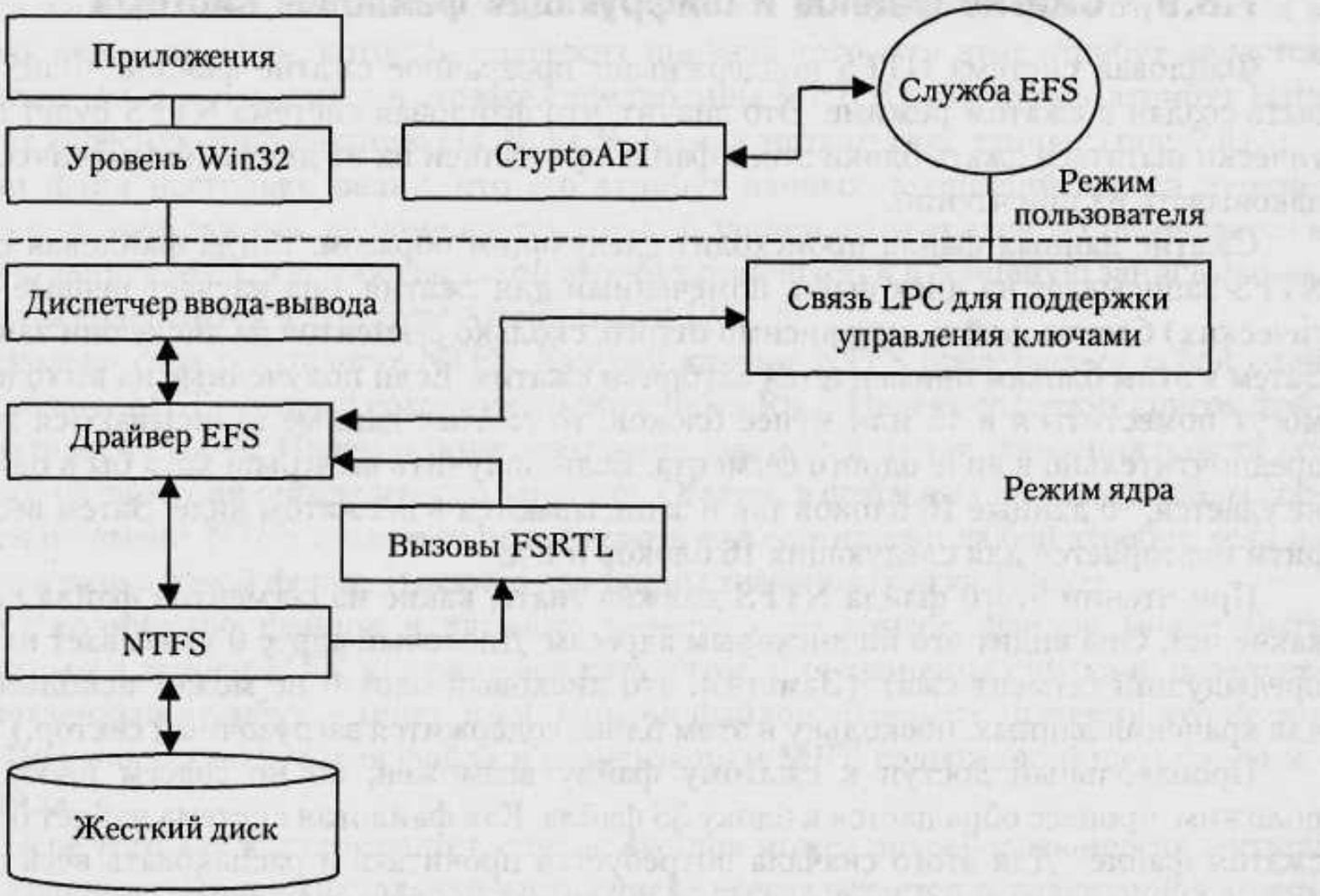

7.7.10. Сжатие файлов и шифрующая файловая система

На персональном компьютере ОС можно загружать с гибкого диска. Это позволяет обойти проблемы, связанные с отказом жесткого диска и разрушением загрузочных разделов. Однако это позволяет обойти встроенную систему управления доступом файловой системы NTFS и с помощью определенных инструментов прочесть информацию жесткого диска. Наконец, жесткий диск можно просто вынуть из одного компьютера и установить в другой.

Единственный надежный способ защиты информации - это шифрующая файловая система (Encrypting File System, EFS), реализованная в Windows 2000/2003 и работающая только на NTFS 5.0.

7.7.10. Сжатие файлов и шифрующая файловая система

Структура EFS ОС Windows 2000



7.7.10. Сжатие файлов и шифрующая файловая система

Шифрующая файловая система содержит следующие компоненты:

1. **Драйвер EFS, являющийся надстройкой над файловой системой NTFS. Он обменивается данными со службой EFS - запрашивает ключи шифрования, наборы DDF (Data Decryption Field) и DRF (Data Recovery Field), - а также с другими службами управления ключами. Полученную информацию драйвер EFS передает библиотеке реального времени файловой системы EFS (File System Run-Time Library, FSRTL), которая прозрачно для операционной системы выполняет различные операции, характерные для файловой системы.**
2. **Библиотека реального времени файловой системы EFS - это модуль, находящийся внутри драйвера EFS, реализующий вызовы NTFS, выполняющие такие операции, как чтение, запись и открытие зашифрованных файлов и каталогов, а также операции, связанные с шифрованием, дешифрованием и восстановлением файлов при их чтении или записи на диск. Для передачи сообщений друг другу драйверы EFS и FSRTL используют механизм вызовов (callouts) NTFS, предназначенный для управления файлами, что гарантирует, что вся работа с файлами происходит при непосредственном участии NTFS.**

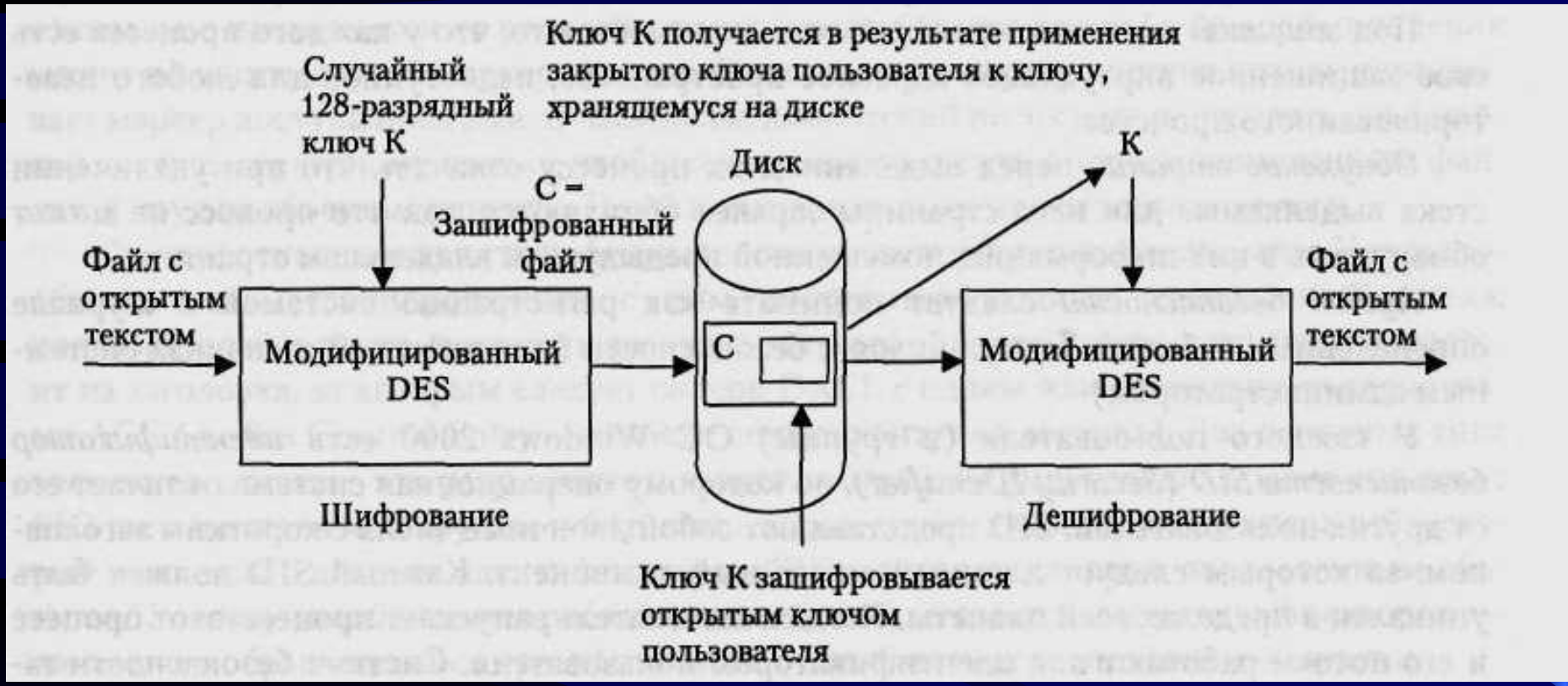
7.7.10. Сжатие файлов и шифрующая файловая система

С помощью механизма управления файлами операции записи значений атрибутов EFS (DDF и DRF) реализованы как обычная модификация атрибутов файла. Кроме того, передача ключа шифрования файла FEK, полученного службой EFS, в FSRTL выполняется так, чтобы он мог быть установлен в контексте открытого файла. Затем контекст файла используется для автоматического выполнения операций шифрования и дешифрования при записи и чтении информации файла.

- Служба EFS является частью системы безопасности ОС Windows 2000. Для обмена данными с драйвером EFS она использует порт связи LPC, существующий между локальным администратором безопасности (Local Security Authority, LSA) и монитором безопасности, работающим в привилегированном режиме. В режиме пользователя для создания ключей шифрования файлов и генерирования данных для DDF и DRF служба EFS использует Crypto API. Она также поддерживает набор API для Win 32.
- Набор API для Win 32. Этот набор интерфейсов прикладного программирования позволяет выполнять шифрование файлов, дешифрование и восстановление зашифрованных файлов, а также их импорт и экспорт (без предварительного дешифрования).

7.7.10. Сжатие файлов и шифрующая файловая система

Схема шифрования



7.7.10. Сжатие файлов и шифрующая файловая система

Когда пользователь сообщает системе, что хочет зашифровать определенный файл, формируется случайный 128-разрядный ключ (File Encryption Key, FEK). Ключ используется для шифрования файла с помощью симметричного алгоритма, параметром в котором является этот ключ. Каждый новый шифруемый файл получает новый ключ FEK, так что никакие два файла не используют один и тот же ключ, что улучшает степень защиты данных.

Ключи FEK хранятся на диске в зашифрованном виде. Для этого используется шифрование с открытым ключом или несколькими открытыми ключами в том случае, если нужно организовать доступ к файлу со стороны нескольких пользователей.

Список зашифрованных FEK хранится вместе с зашифрованным файлом в специальном атрибуте EFS, называемом полем дешифрования данных (Data Decryption Field, DDF).

Чтобы расшифровать файл, с диска считывается зашифрованный ключ, хранящийся в атрибуте DDE. Для его расшифровки необходим закрытый ключ.

7.7.10. Сжатие файлов и шифрующая файловая система

Когда пользователь в первый раз зашифровывает файл с помощью EFS, ОС формирует пару ключей (закрытый и открытый) и сохраняет закрытый ключ, зашифрованный с помощью симметричного алгоритма шифрования, на диске. Ключ для этого симметричного алгоритма формируется либо из пароля пользователя для регистрации в системе, либо из ключа, хранящегося на смарт-карте, если регистрация при помощи смарт-карты разрешена.

Таким образом, система EFS расшифровывает закрытый ключ во время регистрации пользователя в системе и хранить его в своем виртуальном адресном пространстве во время работы, чтобы иметь возможность расшифровать ключи без дополнительного обращения к диску. Когда компьютер выключается, ключ стирается из виртуального адресного пространства EFS, так что никто, даже украв компьютер, не получит доступа к закрытому ключу.

7.7.11. Безопасность в Windows 2000

ОС Windows NT была разработана так, чтобы соответствовать уровню C2 требований безопасности Министерства обороны США (DoD 5200.28 - STD).

Этот стандарт требует наличия у операционных систем определенных свойств по обеспечению надежности для выполнения военных задач определенного рода.

Windows 2000 унаследовала от NT множество свойств безопасности, включая следующие:

1. **Безопасная регистрация в системе с мерами предосторожности против попыток применения фальшивой программы регистрации.**
2. **Дискреционное управление доступом.**
3. **Управление привилегированным доступом.**
4. **Защита адресного пространства для каждого процесса.**
5. **Обнуление страниц перед выделением их процессу.**
6. **Аудит безопасности.**

7.7.11. Безопасность в Windows 2000

Безопасная регистрация **означает, что системный администратор может потребовать от всех пользователей наличия пароля для входа в систему.**

Программа, имитирующая регистрацию в системе, использовалась ранее на некоторых системах злоумышленниками с целью выведать пароль пользователя.

В ОС Windows 2000 подобный обман пользователя невозможен, так как пользователь для входа в систему должен нажать комбинацию клавиш CTRL + ALT + DEL. Эта комбинация всегда перехватывается драйвером клавиатуры, который вызывает законную программу регистрации.

7.7.11. Безопасность в Windows 2000

Дискреционное управление доступом **позволяет владельцу файла или другого объекта указать, кто может пользоваться объектом и каким образом.**

Средства управления привилегированным доступом **позволяют системному администратору получать доступ к объекту, несмотря на установленные его владельцем разрешения доступа.**

Под *защитой адресного пространства* понимается то, что у каждого процесса есть свое защищенное виртуальное адресное пространство, недоступное для любого неавторизованного процесса.

Обнуление страниц **перед выделением их процессу означает, что при увеличении стека выделяемые для него страницы заранее обнуляются, так что процесс не может обнаружить в них информации, помещенной предыдущим владельцем страницы.**

Аудит безопасности **следует понимать как регистрацию системой в журнале определенных событий, относящихся к безопасности (для действий и анализа системным администратором).**

7.7.11. Безопасность в Windows 2000

У каждого пользователя (и группы) ОС Windows 2000 есть *идентификатор безопасности SID (Security IDentifier)*, по которому операционная система отличает его от других пользователей.

SID представляют собой двоичные числа с коротким заголовком, за которым следует длинный случайный компонент. Каждый SID должен быть уникален в пределах всей планеты.

Когда пользователь запускает процесс, этот процесс и его потоки работают под идентификатором пользователя. Система безопасности гарантирует предоставление доступа к каждому объекту только потокам с авторизованными идентификаторами безопасности.

У каждого процесса есть *маркер доступа*, в котором указываются SID и другие свойства. Как правило, он назначается при регистрации в системе процедурой Winlogon.

Структура маркера

Заголовок	Срок действия	Группы	DACL	Ограниченные идентификаторы SID	SID пользователя	SID группы	Привилегии

7.7.11. Безопасность в Windows 2000

Чтобы получить эту информацию, процесс должен вызвать функцию **GetTokenInformation**, так как информация может измениться со временем.

Заголовок содержит некоторую административную информацию.

По значению **срока действия** можно определить, когда маркер перестанет быть действительным.

Поле **Группы (Groups)** указывает группы, к которым принадлежит процесс. Это поле необходимо для соответствия требованиям стандарта POSIX.

Поле **Default DACL (DACL по умолчанию, Discretionary Access Control List - список разграничительного контроля доступа)** представляет собой список управления доступом, назначаемый объектам, созданным процессом, если не определены другие списки ACL.

Идентификатор безопасности пользователя указывает пользователя, владеющего процессом.

Ограниченные идентификаторы SID позволяют ненадежным процессам принимать участие в заданиях вместе с надежными процессами, но с меньшими полномочиями и меньшими возможностями причинения ущерба.

7.7.11. Безопасность в Windows 2000

Перечисленные в маркере привилегии дают процессу особые полномочия, такие как право выключить компьютер или получить доступ к файлам, к которым в противном случае в этом доступе процессам было бы отказано. Привилегии позволяют развить полномочия системного администратора на отдельные права, которые могут предоставляться процессам по отдельности.

Когда пользователь регистрируется в системе, процесс winlogon назначает маркер доступа начальному процессу. Последующие процессы наследуют этот маркер. Он же изначально применяется ко всем потокам процесса. Однако поток во время выполнения может получить другой маркер доступа. В этом случае маркер доступа потока перекрывает маркер доступа процесса. В частности, клиентский поток может передать свой маркер доступа серверному потоку, чтобы сервер мог получить доступ к защищенным файлам и другим объектам клиента (*перевоплощение*).

7.7.11. Безопасность в Windows 2000

У каждого объекта есть ассоциированный с ним дескриптор защиты, содержащий список пользователей и групп, имеющих доступ к данному объекту.

Дескриптор защиты состоит из заголовка, за которым следует список DACL с одним или несколькими элементами ACE (Access Control Entry - элемент списка контроля доступа).

Два основных типа элементов списка - это разрешение и запрет доступа.

***Разрешающий элемент* содержит SID пользователя или группы и битовый массив, определяющий набор операций, который процессы с данным идентификатором SID могут выполнять с определенным объектом.**

***Запрещающий элемент* работает аналогично, но совпадение идентификаторов означает, что обращающийся процесс не может выполнять перечисленные операции.**

Кроме списка DACL у дескриптора защиты есть список SACL (System Access Control List - системный список контроля доступа), который похож на DACL, только вместо пользователей и групп, имеющих доступ к объекту, в нем перечислены операции с этим объектом, регистрируемые в специальном журнале.

7.7.11. Безопасность в Windows 2000

В основе большей части механизмов управления доступом в Windows 2000 лежат **дескрипторы защиты**.

Как правило, когда процесс создает объект, он передает функции CreateProcess, CreateFile или другой функции в качестве одного из параметров **дескриптор защиты**. Этот дескриптор защиты затем становится дескриптором защиты, присоединенным к объекту. Если при создании объекта не предоставляется дескриптор защиты, используется дескриптор защиты вызывающего процесса по умолчанию.

Для управления дескрипторами защиты существует множество вызовов Win32 API. **Чтобы создать дескриптор защиты**, для него выделяется место хранения, после чего он инициализируется с помощью вызова InitializeSecurityDescriptor. Этот вызов заполняет заголовок.

Если SID владельца неизвестен, он может быть найден по имени при помощи вызова LookupAccountSid. Затем он может быть вставлен в дескриптор защиты. То же самое справедливо для SID группы, если группа существует. Как правило, используется SID вызывающего процесса, а также SID одной из групп вызывающего процесса, но системный администратор может записать в дескриптор защиты любой SID.

7.7.11. Безопасность в Windows 2000

Список DACL или SACL дескриптора защиты может быть проинициализирован при помощи функции InitializeAcl. Элементы списка ACL могут быть добавлены с помощью функций AddAccessAllowedAce и AddAccessDeniedAce.

К этим вызовам можно обращаться многократно, чтобы добавить столько записей ACL, сколько необходимо. Удаление записи происходит при помощи функции DeleteAce.

Когда список ACL готов, его можно присоединить к дескриптору защиты с помощью функции SetSekurityDescriptor Dacl. Наконец, когда объект создан, к нему можно присоединить дескриптор защиты. Защита в автономной системе Windows 2000 реализуется при помощи нескольких компонентов.

Регистрацией в системе управляет программа winlogon, а аутентификацией - lsass и msgina.dll. Результатом успешной регистрации в системе является новая оболочка с ассоциированным с ней маркером доступа. Этот процесс использует в реестре ключи SECURITY и SAM. Первый ключ определяет общую политику безопасности, а второй ключ содержит информацию о защите для индивидуальных пользователей.

7.7.11. Безопасность в Windows 2000

Как только пользователь регистрируется в системе, выполняется операция защиты при открытии объекта.

Для каждого вызова OpenXXX требуются имя открываемого объекта и набор прав доступа к нему. Во время обработки процедуры открытия объекта менеджер безопасности проверяет наличие у вызывающего процесса соответствующих прав доступа.

Для этого он просматривает все маркеры доступа вызывающего процесса, а также список DACL, ассоциированный с объектом. Он просматривает по очереди элементы списка ACL.

Как только он находит запись, соответствующую идентификатору SID вызывающего процесса или одной из его групп, поиск прав доступа считается законченным.

Если вызывающий процесс обладает необходимыми правами, объект открывается, в противном случае в открытии объекта отказывается.

Помимо разрешающих записей списки D ACL могут также содержать запрещающие записи.

7.7.11. Безопасность в Windows 2000

Поскольку менеджер безопасности прекращает поиск, найдя первую запись с указанным идентификатором, запрещающие записи помещаются в начало списка DACL, чтобы пользователь, которому строго запрещен доступ к какому-либо объекту, не смог получить его как член какой-либо группы, которой этот доступ предоставлен.

После того как объект открыт, дескриптор объекта возвращается вызывающему процессу. При последующих обращениях проверяется только, входит ли данная операция в число операций, разрешенных в момент открытия объекта.

7.8. Сетевая операционная система реального времени QNX

7.8.1. Принципы построения ОС реального времени

Многозадачность в истинном смысле этого слова. Невытесняющая многозадачность неприемлема, поскольку допускает возможность блокировки или даже полного развала системы одним неправильно работающим процессом. Для предотвращения блокировок вычислений ОСРВ должна использовать вытесняющую, а не кооперативную многозадачность.

Организация *надежных вычислений* — может быть эффективно решена за счет специальных аппаратных возможностей процессора. При построении системы для работы на компьютерах IBM PC для этого необходимы процессоры типа Intel 80386 и выше, чтобы иметь возможность организовать функционирование операционной системы в защищенном (32-разрядном) режиме работы процессора.

Эффективное обслуживания прерываний. Для этого ОС должна использовать алгоритм диспетчеризации, обеспечивающий *вытесняющее планирование, основанное на приоритетах*.

7.8. Сетевая операционная система реального времени QNX

7.8.1. Принципы построения ОС реального времени

Желательна эффективная поддержка *сетевых коммуникаций* и наличие развитых механизмов *взаимодействия между процессами,* поскольку реальные технологические системы обычно управляются целым комплексом компьютеров и/или контроллеров.

Желательна поддержка многопоточности (не только мультипрограммный, но и мультизадачный режимы) и симметричной мультипроцессорности.

Способность работать на ограниченных аппаратных ресурсах, поскольку одна из ее основных областей применения — встроенные системы. К сожалению, данное условие обычно реализуется путем простого урезания стандартных сервисных средств.

процессора.

7.8.2. Краткая характеристика QNX

Мощная ОС, разработанная для процессоров с архитектурой ia32.

Позволяет проектировать сложные программные комплексы, работающие в реальном времени как на отдельном компьютере, так и в локальной вычислительной сети.

Встроенные средства QNX обеспечивают поддержку многозадачного режима на одном компьютере и взаимодействие параллельно выполняемых задач на разных компьютерах, работающих в среде локальной вычислительной сети (хорошо подходит для построения распределенных систем).

Основным языком программирования в системе является С. Основная операционная среда соответствует стандарту POSIX, что позволяет с небольшими доработками переносить ранее разработанное программное обеспечение в QNX для организации их работы в среде распределенной обработки.

Является многопользовательской (многотерминальной) ОС.

Масштабируемая ОС.

Пользовательский интерфейс и интерфейс прикладного программирования очень похожи на таковые в UNIX, поскольку выполняются требования стандарта POSIX.

7.8.2. Краткая характеристика QNX

QNX — это не версия UNIX, хотя почему-то многие так считают. Система QNX была разработана, что называется, «с нуля» канадской фирмой QNX Software Systems Limited в 1989 году по заказу Министерства обороны США, причем на совершенно иных архитектурных принципах, нежели использовались при создании операционной системы UNIX.

Построена на принципах микроядра и обмена сообщениями. Реализована в виде совокупности независимых (но взаимодействующих путем обмена сообщениями) процессов различного уровня (менеджеры и драйверы), каждый из которых реализует определенный вид услуг.

Предсказуемость означает применимость системы к задачам жесткого реального времени. QNX является операционной системой, которая дает полную гарантию того, что процесс с наивысшим приоритетом начнет выполняться практически немедленно, и критически важное событие (например, сигнал тревоги) никогда не будет потеряно.

7.8.2. Краткая характеристика QNX

Масштабируемость и *эффективность* достигаются оптимальным использованием ресурсов и означают применимость QNX для встроенных (embedded) систем. В каталоге /dev нет файлов, соответствующих ненужным драйверам, что характерно для UNIX-систем. Драйверы и менеджеры можно запускать и удалять (кроме файловой системы, что очевидно) динамически, просто из командной строки. Мы можем иметь только те услуги, которые нам реально нужны.

Расширяемость и *надежность* обеспечиваются одновременно, поскольку написанный драйвер не нужно компилировать в ядро, рискуя вызвать нестабильность системы. Менеджеры ресурсов (служба логического уровня) работают в третьем кольце защиты, и можно добавлять свои менеджеры, не опасаясь за систему. Драйверы работают в первом кольце и могут вызвать проблемы, но не фатального характера. Кроме того, их достаточно просто писать и отлаживать.

Быстрый сетевой протокол FLEET прозрачен для обмена сообщениями, автоматически обеспечивает отказоустойчивость, балансирование нагрузки и маршрутизацию между альтернативными путями доступа.

7.8.2. Краткая характеристика QNX

Компактная графическая подсистема Photon, построенная на тех же принципах модульности, что и сама операционная система, позволяет получить полнофункциональный интерфейс GUI (расширенный интерфейс Motif), работающий вместе с POSIX-совместимой операционной системой всего в 4 Мбайт памяти, начиная с i80386 процессора.

7.8.2. Архитектура системы QNX

QNX — это операционная система реального времени для персональных компьютеров, позволяющая эффективно организовать распределенные вычисления.

В системе реализована концепция связи между задачами на основе сообщений, посылаемых от одной задачи к другой, причем задачи эти могут решаться как на одном и том же компьютере, так и на разных, но связанных между собой локальной вычислительной сетью.

Реальное время и концепция связи между процессами посредством сообщений оказывают решающее влияние и на разрабатываемое для операционной системы QNX программное обеспечение, и на программиста, стремящегося с максимальной выгодой использовать преимущества системы.

Микроядро операционной системы QNX имеет объем всего в несколько десятков килобайтов (в одной из версий — 10 Кбайт, в другой — менее 32 Кбайт, хотя есть вариант и на 46 Кбайт), то есть это одно из самых маленьких ядер среди всех существующих операционных систем.

7.8.2. Архитектура системы QNX

В микроядре помещаются :

- **механизм передачи сообщений между процессами IPC (Inter Process Communication — взаимодействие между процессами);**
- **редиректор (redirector) прерываний;**
- **блок планирования выполнения задач (иначе говоря, диспетчер задач);**
- **сетевой интерфейс для перенаправления сообщений (менеджер Net).**

Механизм IPC обеспечивает **пересылку сообщений между процессами. Все взаимодействие между процессами, в том числе и системными, происходит через сообщения. Сообщение в операционной системе QNX — это последовательность байтов произвольной длины (0-65 535 байт) произвольного формата.**

Протокол обмена сообщениями может выглядеть, например, таким образом. Задача блокируется для ожидания сообщения. Другая задача посылает первой сообщение и при этом блокируется сама, ожидая ответа. Первая задача деблокируется, обрабатывает сообщение и отвечает, деблокируя вторую задачу.

7.8.2. Архитектура системы QNX

Сообщения и ответы, пересылаемые между процессами при их взаимодействии, находятся в теле отправляющего их процесса до того момента, когда они могут быть приняты.

Благодаря этому: снижается вероятность повреждения сообщения в процессе передачи; уменьшается объем оперативной памяти, необходимый для работы ядра; становится меньше пересылок из памяти в память, что разгружает процессор.

В сети, состоящей из нескольких компьютеров, работающих под управлением QNX, сообщения могут прозрачно передаваться процессам, выполняющимся на любом из узлов.

Определены в QNX еще и два дополнительных метода передачи сообщений — *метод представителей* (proxy) и *метод сигналов* (signal).

Представители используются в случаях, когда процесс должен передать сообщение, но не должен при этом блокироваться на передачу. Тогда вызывается функция qnx_proxy__attach() и создается представитель. Он накапливает в себе сообщения, которые должны быть доставлены другим процессам.

7.8.2. Архитектура системы QNX

Любой процесс, знающий идентификатор представителя, может вызвать функцию Trigger(), после чего будет доставлено первое в очереди сообщение. Функция Trigger() может вызываться несколько раз, и каждый раз представитель будет доставлять следующее сообщение. При этом представитель содержит буфер, в котором может храниться до 65 535 сообщений.

Операционная система QNX также поддерживает множество сигналов, совместимых с POSIX, большое количество сигналов, традиционно использовавшихся в UNIX, а также несколько сигналов, специфичных для самой системы QNX.

По умолчанию любой сигнал, полученный процессом, приводит к завершению процесса (кроме нескольких сигналов, которые по умолчанию игнорируются). Но процесс с приоритетом уровня суперпользователя может защититься от нежелательных сигналов. В любом случае процесс может содержать обработчик для каждого возможного сигнала. Сигналы удобно рассматривать как разновидность программных прерываний.

7.8.2. Архитектура системы QNX

Редиректор прерываний **занимается перенаправлением аппаратных прерываний в связанные с ними процессы. С аппаратной частью компьютера работает ядро, оно перенаправляет прерывания процессам — обработчикам прерываний.**

Обработчики прерываний обычно встроены в процессы, хотя каждый из них исполняется асинхронно с процессом, в который он встроен. Обработчик исполняется в контексте процесса и имеет доступ ко всем глобальным переменным процесса.

При работе обработчика прерываний прерывания разрешены, но обработчик приостанавливается только в том случае, если произошло более высокоприоритетное прерывание.

Если это позволяется аппаратной частью, к одному прерыванию может быть подключено несколько обработчиков, каждый из которых получит управление при возникновении прерывания.

7.8.2. Архитектура системы QNX

Блок планирования выполнения задач **обеспечивает многозадачность. ОС QNX предоставляет разработчику огромный простор для выбора той дисциплины выделения ресурсов процессора задаче, которая обеспечит наиболее подходящие условия для выполнения критически важных приложений, а обычным приложениям обеспечит такие условия, при которых они будут выполняться за разумное время, не мешая работе критически важных приложений.**

7.8.2. Архитектура системы QNX

К выполнению своих функций **как диспетчера ядро приступает** в следующих случаях:

- какой-либо процесс вышел из блокированного состояния;
- истек квант времени для процесса, владеющего центральным процессором;
- работающий процесс прерван каким-либо событием.

Диспетчер выбирает процесс для запуска среди неблокированных процессов в порядке значений их приоритетов в диапазоне от 0 (наименьший) до 31 (наибольший).

Обслуживание каждого из процессов зависит от метода его диспетчеризации (приоритет и метод диспетчеризации могут динамически меняться во время работы).

В QNX существуют **три метода диспетчеризации**:

- **очередь** (First In First Out, FIFO) — раньше пришедший процесс раньше обслуживается;
- **карусель** (Round Robin, RR) — процессу выделяется определенный квант времени для работы, после чего процессор предоставляется следующему процессу;
- **адаптивный метод** (используется чаще других).

7.8.2. Архитектура системы QNX

Метод FIFO наиболее близок к невытесняющей многозадачности. Процесс выполняется до тех пор, пока он не перейдет в состояние ожидания сообщения, в состояние ожидания ответа на сообщение или не отдаст управление ядру. При переходе в одно из таких состояний процесс помещается последним в очередь процессов с таким же уровнем приоритета, а управление передается процессу с наибольшим приоритетом.

В **методе RR** все происходит так же, как и в предыдущем, с той разницей, что период, в течение которого процесс может работать без перерыва, ограничивается неким квантом времени.

Процесс, работающий в соответствии с **адаптивным методом**, ведет себя следующим образом:

- если процесс полностью использует выделенный ему квант времени, а в системе есть готовые к исполнению процессы с тем же уровнем приоритета, его приоритет снижается на 1;
- если процесс с пониженным приоритетом остается необслуженным в течение секунды, его приоритет увеличивается на 1;
- если процесс блокируется, ему возвращается исходное значение приоритета.

7.8.2. Архитектура системы QNX

По умолчанию процессы запускаются в режиме адаптивной многозадачности. В этом же режиме работают все системные утилиты QNX. Процессы, работающие в разных режимах многозадачности, могут одновременно находиться в памяти и исполняться.

Обычно приоритет процесса устанавливается при его запуске. Но есть дополнительная возможность, называемая *вызываемым клиентом приоритетом*. Как правило, она реализуется для серверных процессов (исполняющих запросы на какое-либо обслуживание). При этом приоритет процесса-сервера устанавливается только на время обработки запроса и становится равным приоритету процесса-клиента.

7.8.2. Архитектура системы QNX

Сетевой интерфейс **в операционной системе QNX является неотъемлемой частью ядра. Взаимодействует с сетевым адаптером через сетевой драйвер. Базовые сетевые службы реализованы на уровне ядра.**

Передача сообщения процессу, находящемуся на другом компьютере, ничем не отличается с точки зрения приложения от передачи сообщения процессу, выполняющемуся на том же компьютере. Благодаря такой организации сеть превращается в однородную вычислительную среду. Для большинства приложений не имеет значения, с какого компьютера они были запущены, на каком исполняются и куда поступают результаты их работы.

Все службы операционной системы QNX, не реализованные непосредственно в ядре, работают как обычные стандартные процессы в полном соответствии с основными концепциями микроядерной архитектуры.

7.8.2. Архитектура системы QNX

Драйверы устройств также работают как стандартные процессы. Чтобы новый драйвер устройства стал частью операционной системы, необходимо изменить конфигурационный файл системы так, чтобы драйвер запускался при загрузке.

7.8.3. Основные механизмы организации распределенных вычислений

QNX является сетевой операционной системой, которая позволяет организовать эффективные распределенные вычисления.

Для этого на каждой машине, называемой узлом, помимо ядра и менеджера процессов должен быть запущен уже менеджер Net.

Менеджер Net не зависит от аппаратной реализации сети, что обеспечивается за счет сетевых драйверов. В ОС QNX имеются драйверы для сетей с различными технологиями: Ethernet и FastEthernet, Arcnet, IBM Token Ring и др. Имеется возможность организации сети через последовательный канал или модем.

В QNX версии 4 полностью реализовано встроенное сетевое взаимодействие типа «точка-точка». Например, сидя за машиной *A,* вы можете скопировать файл с гибкого диска, подключенного к машине *B,* на жесткий диск, подключенный к машине *C*.

Сеть из машин с операционными системами QNX действует как один мощный компьютер. Любые ресурсы (модемы, диски, принтеры) могут быть добавлены к системе простым их подключением к любой машине в сети.

7.8.3. Основные механизмы организации распределенных вычислений

QNX обеспечивает возможность одновременной работы в сетях Ethernet, Arcnet, Serial и Token Ring, более одного пути для связи и балансировку нагрузки в сетях. Если кабель или сетевая плата выходит из строя и связь через эту сеть прекращается, система автоматически перенаправит данные через другую сеть.

Каждому узлу в сети соответствует уникальный целочисленный идентификатор — логический номер узла. Любой поток выполнения в сети QNX имеет прозрачный доступ (при наличии достаточных привилегий) ко всем ресурсам сети; то же самое относится и к взаимодействию потоков.

Для взаимодействия потоков, находящихся на разных узлах сети, используются те же самые вызовы ядра, что и для потоков, выполняемых на одном узле.

Если потоки находятся на разных узлах сети, ядро переадресует запрос менеджеру сети. Для обмена в сети используется надежный и эффективный протокол транспортного уровня FLEET. Каждый из узлов может принадлежать одновременно нескольким QNX-сетям. Если сетевое взаимодействие может быть реализовано несколькими путями, для передачи выбирается менее загруженная и более скоростная сеть.

7.8.3. Основные механизмы организации распределенных вычислений

Разработчики операционной системы QNX создали собственную специальную сетевую технологию FLEET и соответствующий уникальный протокол транспортного уровня FTL (FLEET Transport Layer).

Основные его качества зашифрованы в аббревиатуре FLEET, которая расшифровывается следующим образом:

- Fault-Tolerant Networking — QNX может одновременно использовать несколько физических сетей, при выходе из строя любой из них данные будут «на лету» перенаправлены через другую сеть;
- Load-Balancing on the Fly — при наличии нескольких физических соединений QNX автоматически распараллеливает передачу пакетов по соответствующим сетям;
- Efficient Performance — специальные драйверы, разрабатываемые фирмой QSSL для широкого спектра оборудования, позволяют использовать это оборудование с максимальной эффективностью;
- Extensable Architecture — любые новые типы сетей могут быть поддержаны путем добавления соответствующих драйверов;
- Transparent Distributed Processing — благодаря отсутствию разницы между передачей сообщений в пределах одного узла и между узлами нет необходимости вносить какие-либо изменения в приложения, для того чтобы они могли взаимодействовать через сеть.

7.8.3. Основные механизмы организации распределенных вычислений

Благодаря технологии FLEET сеть компьютеров с операционными системами QNX фактически можно представлять как один виртуальный суперкомпьютер. Все ресурсы любого из узлов сети автоматически доступны другим. Любая программа может быть запущена на любом узле, причем ее входные и выходные потоки могут быть направлены на любое устройство на любых других узлах.

Поддержка сети обеспечивается и микроядром (специальный код в его составе позволяет операционной системе QNX фактически объединять все микроядра в сети в одно ядро).

Когда ядро получает запрос на передачу данных процессу, находящемуся на удаленном узле, он переадресовывает этот запрос менеджеру Net, в подчинении которого находятся драйверы всех сетевых карт. Имея перед собой полную картину состояния всего сетевого оборудования, Net может отслеживать состояние каждой сети и динамически перераспределять нагрузку между ними. В случае, когда одна из сетей выходит из строя, поток данных автоматически перенаправляется в другую доступную сеть, что очень важно при построении высоконадежных систем.

7.8.3. Основные механизмы организации распределенных вычислений

Кроме поддержки собственного протокола, Net обеспечивает передачу пакетов TCP/IP, SMB (Server Message Block) и многих других, используя то же сетевое оборудование. При этом производительность компьютеров с операционной системой QNX в сети приближается к производительности аппаратного обеспечения — настолько малы задержки, вносимые операционной системой.

В QNX/Neutrino[1] параллельность выполнения достигается за счет использования *потоковой модели POSIX*, в которой процессы в системе представляются в виде совокупности потоков выполнения.

Поток является минимальной единицей выполнения и диспетчеризации для ядра Neutrino; процесс определяет адресное пространство для потоков. Каждый процесс состоит минимум из одного потока. Операционная система QNX предоставляет богатый набор функций для синхронизации потоков. В отличие от потоков, само ядро не подлежит диспетчеризации. Код ядра исполняется только в том случае, когда какой-нибудь поток вызывает функцию ядра, или при обработке аппаратного прерывания.

7.8.3. Основные механизмы организации распределенных вычислений

Передачу и диспетчеризацию сообщений в QNX осуществляет ядро системы. Кроме того, ядро управляет временными прерываниями. Выполнение остальных функций обеспечивается задачами-администраторами.

Программа, желающая создать задачу, посылает сообщение администратору задач (модуль task) и блокируется для ожидания ответа. Если новая задача должна выполняться одновременно с порождающей ее задачей, администратор задач task создает ее и, отвечая, выдает порождающей задаче идентификатор созданной задачи. В противном случае никакого сообщения не посылается до тех пор, пока новая задача не закончится сама по себе. Тогда в ответе администратора задач будут содержаться конечные характеристики закончившейся задачи.

Сообщения различаются количеством данных, которые передаются от одной задачи точно к другой задаче. Данные копируются из адресного пространства первой задачи в адресное пространство второй, и выполнение первой задачи приостанавливается до тех пор, пока вторая задача не вернет ответное сообщение. В действительности обе задачи кратковременно взаимодействуют во время выполнения передачи. Максимальная длина сообщения может достигать 64 Кбайт). Существует несколько протоколов, которые могут быть использованы для передачи сообщений.

7.8.3. Основные механизмы организации распределенных вычислений

Основные операции над сообщениями: *послать, получить* и *ответить,* а также несколько их вариантов для обработки специальных ситуаций.

Получатель всегда идентифицируется своим идентификатором задачи, хотя существуют способы ассоциировать имена с идентификатором задачи.

Варианты операций включают в себя возможность получать (копировать) только первую часть сообщения, а затем получать оставшуюся часть такими кусками, какие потребуются. Это может быть полезным, поскольку позволяет сначала узнать длину сообщения, а затем динамически распределить принимающий буфер.

Если необходимо задержать ответное сообщение до тех пор, пока не будет получено и обработано другое сообщение, то чтение первых нескольких байтов дает вам компактный «обработчик», через который позже можно получить доступ ко всему сообщению. Таким образом, задача оказывается избавленной от необходимости хранить в себе большое количество буферов.

Другие функции позволяют программе получать сообщения только тогда, когда она уже ожидает их приема, а не блокироваться до тех пор, пока не прибудет сообщение. Можно также транслировать сообщение другой задаче без изменения идентификатора передатчика.

7.8.3. Основные механизмы организации распределенных вычислений

ОС QNX обеспечивает объединение сообщений в структуру данных, называемую очередью. *Очередь сообщений* — это просто область данных в третьей, отдельной задаче, которая временно принимает передаваемое сообщение и немедленно отвечает передатчику.

В отличие от стандартной передачи сообщений, передатчик немедленно освобождается для того, чтобы продолжить свою работу. Задача администратора очереди — хранить в себе сообщение до тех пор, пока приемник не будет готов прочитать его; делает он это, запрашивая сообщение у администратора-очереди.

Любое количество сообщений (ограничено только возможностью памяти) может храниться в очереди. Сообщения хранятся и передаются в том порядке, в котором они были приняты. Может быть создано любое количество очередей. Каждая очередь идентифицируется своим именем.

7.8.3. Основные механизмы организации распределенных вычислений

Помимо сообщений и очередей в операционной системе QNX для взаимодействия задач и организации распределенных вычислений имеются так называемые *порты*, которые позволяют формировать сигнал одного конкретного условия и механизм исключений.

Порт подобен флагу, известному всем задачам на одном и том же узле (но не на разных узлах). Он имеет только два состояния, которые могут трактоваться как «присоединить» и «освободить», хотя пользователь может интерпретировать их по-своему, например «занят» и «доступен».

Порты используются для быстрой простой синхронизации между задачей и обработчиком прерываний устройства. Они нумеруются от нуля до 32 максимум (на некоторых типах узлов возможно и больше). Первые 20 номеров зарезервированы для операционной системы.

С портом может быть выполнено три операции:

- присоединить порт,
- отсоединить порт,
- послать сигнал в порт.

7.8.3. Основные механизмы организации распределенных вычислений

Одновременно к порту может быть присоединена только одна задача. Если другая задача попытается «отсоединиться» от того же самого порта, то произойдет отказ при вызове функции, и управление вернется к задаче, которая в настоящий момент присоединена к этому порту.

Любая задача может посылать сигнал в любой порт независимо от того, была она присоединена к нему или нет. Сигнал подобен неблокирующей передаче пустого сообщения. То есть передатчик не приостанавливается, а приемник не получает какие-либо данные; он только отмечает, что конкретный порт изменил свое состояние.

Задача, присоединенная к порту, может ожидать прибытия сигнала или может периодически читать порт. Система QNX хранит информацию о сигналах, передаваемых в каждый порт, и уменьшает счетчик после каждой операции «приема» сигнала («чтение» возвращает счетчик и устанавливает его в нуль).

Сигналы всегда принимают перед сообщениями, давая им тем самым больший приоритет над сообщениями. В этом смысле сигналы часто используются обработчиками прерываний для того, чтобы оповестить задачу о внешних (аппаратных) событиях. Обработчики прерываний не имеют возможности посылать сообщения и должны использовать сигналы.

7.8.3. Основные механизмы организации распределенных вычислений

Исключения обеспечивают асинхронное взаимодействие, то есть исключение может прервать нормальное выполнение потока задачи. ОС QNX резервирует для себя 16 исключений, чтобы оповещать задачи о прерываниях с клавиатуры, нарушении памяти и подобных необычных ситуациях. Остальные 16 исключений могут быть определены и использованы прикладными задачами.

Системная функция может быть вызвана для того, чтобы позволить задаче реализовать собственный механизм обработки исключений и во время возникновения исключения выполнять свою внутреннюю функцию.

Функция исключения задачи вызывается асинхронно операционной системой, а не самой задачей. Поэтому исключения могут негативно повлиять на операции (например, передачу сообщений), которые выполняются в это же время.

Одна задача может установить одно или несколько исключений для другой задачи. Эти исключения могут быть комбинацией системных исключений и исключений, определяемых приложениями, обеспечивая другие возможности для межзадачного взаимодействия.

7.8.3. Основные механизмы организации распределенных вычислений

Благодаря возможности обмена посланиями между задачами и узлами сети, программы не заботятся о конкретном размещении ресурсов в сети. Это свойство придает системе необычную гибкость. Так, узлы могут произвольно добавляться в систему и изыматься из системы, не затрагивая системные программы.

QNX имеет эту конфигурационную независимость благодаря концепции *виртуальных задач*. У виртуальных задач непосредственный код и данные, будучи на одном из удаленных узлов, возникают и ведут себя так, как если бы они были локальными задачами какого-то узла со всеми их атрибутами и привилегиями.

Программа, посылающая сообщение в сеть, никогда не направляет его точно. Сначала она открывает *виртуальный канал*. *Виртуальный канал* связывает между собой все виртуальные задачи. На обоих концах такой связи имеются буферы, которые позволяют хранить самое большое послание из тех, которые канал может нести в данном сеансе связи.

Сетевой администратор помещает в эти буферы все сообщения для соединенных задач. Виртуальная задача, таким образом, занимает всего лишь пространство, необходимое для буфера и входа в таблице задач.

7.8.3. Основные механизмы организации распределенных вычислений

Чтобы открыть виртуальный канал, необходимо знать идентификатор узла и задачи, с которой устанавливается связь. Для этого требуется идентификатор задачи-администратора, ответственного за данную функцию, или глобальное имя сервера. Задача может вообще выполняться на другом узле, где, допустим, имеется более совершенный процессор.