

Министерство науки и высшего образования Российской Федерации

Калужский филиал  
федерального государственного бюджетного образовательного  
учреждения высшего образования  
**«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»**  
(КФ МГТУ им. Н.Э. Баумана)

**Е.В. Красавин**

## **ТЕХНОЛОГИИ СИСТЕМНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

*Конспект лекций*

Калуга - 2024

Конспект лекций по курсу «Компьютерные сети» включает основные стандарты, концепции эволюционного развития, теоретические основы архитектурной и программной реализации различных информационно - коммуникационных сетей. Предназначен для студентов 3-го курса бакалавриата КФ МГТУ им. Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

© Калужский филиал МГТУ им. Н.Э. Баумана, 2024 г.  
© Е.В. Красавин, 2012г.

## Оглавление

ТЕХНОЛОГИИ СИСТЕМНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ	1
Модуль 1. БАЗОВЫЕ ПОНЯТИЯ О СИСТЕМНОМ ПРОГРАММНОМ ОБЕСПЕЧЕНИИ	4
Лекция 1.1. Введение в системное программное обеспечение. Особенности распределенных систем .....	4
Лекция 1.2. Обзор системы UNIX.....	11
Лекция 1.3. Процессы в системе UNIX .....	21
Лекция 1.4. Управление памятью в UNIX .....	38
Лекция 1.5. Ввод-вывод в системе UNIX .....	51
Лекция 1.6. Файловая система UNIX .....	60
Лекция 1.7. Безопасность в UNIX.....	85
Модуль 2. ПРОГРАММИРОВАНИЕ В WINDOWS .....	89
Лекция 2.1 Уровни программирования в Windows и структура ядра	89
Лекция 2.2 Процессы и потоки в Windows .....	125
Лекция 2.3 Управление памятью в Windows .....	138

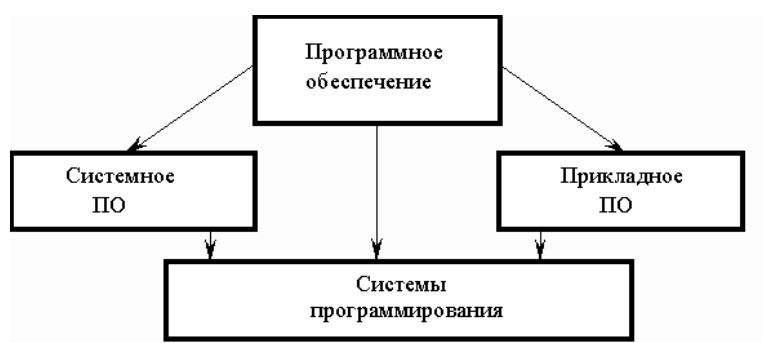
Лекция 2.4 Ввод-вывод в Windows .....	142
Лекция 2.5 Файловая система NTFS .....	152
Лекция 2.6 Безопасность в Windows .....	163
ОСНОВНАЯ ЛИТЕРАТУРА .....	170
ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА .....	170

# Модуль 1. БАЗОВЫЕ ПОНЯТИЯ О СИСТЕМНОМ ПРОГРАММНОМ ОБЕСПЕЧЕНИИ

## Лекция 1.1. Введение в системное программное обеспечение. Особенности распределенных систем

Программное обеспечение (ПО) - это совокупность всех программ и соответствующей документации, обеспечивающая использование ЭВМ в интересах каждого ее пользователя.

Различают системное и прикладное ПО. Схематически программное обеспечение можно представить так:



**Рис.1.** Классификация программного обеспечения

Системное ПО – это совокупность программ для обеспечения работы компьютера. Системное ПО подразделяется на базовое и сервисное. Системные программы предназначены для управления работой вычислительной системы, выполняют различные вспомогательные функции (копирования, выдачи справок, тестирования, форматирования и т. д).

Базовое ПО включает в себя:

- операционные системы;
- оболочки;
- сетевые операционные системы.

Сервисное ПО включает в себя программы (утилиты):

- диагностики;
- антивирусные;
- обслуживания носителей;
- архивирования;
- обслуживания сети.

Прикладное ПО – это комплекс программ для решения задач определённого класса конкретной предметной области. Прикладное ПО работает только при наличии системного ПО.

Прикладные программы называют приложениями. Они включают в себя:

- текстовые процессоры;
- табличные процессоры;
- базы данных;
- интегрированные пакеты;
- системы иллюстративной и деловой графики (графические процессоры);
- экспертные системы;
- обучающие программы;
- программы математических расчетов, моделирования и анализа;
- игры;
- коммуникационные программы.

Особую группу составляют системы программирования (инструментальные системы), которые являются частью системного ПО, но носят прикладной характер. Системы программирования – это совокупность программ для разработки, отладки и внедрения новых программных продуктов. Системы программирования обычно содержат:

- трансляторы;
- среду разработки программ;
- библиотеки справочных программ (функций, процедур);
- отладчики;
- редакторы связей и др.

Распределенная организация операционной системы позволяет упростить работу пользователей и программистов в сетевых средах. В распределенной ОС реализованы механизмы, которые дают возможность пользователю представлять и воспринимать сеть в виде традиционного однопроцессорного компьютера. Характерными признаками распределенной организации ОС являются: наличие единой справочной службы разделяемых ресурсов, единой службы времени, использование механизма вызова удаленных процедур (RPC) для прозрачного распределения программных процедур по машинам, многократной обработки, позволяющей распараллеливать вычисления в рамках одной задачи и выполнять эту задачу сразу на нескольких компьютерах сети, а также наличие других распределенных служб.

Сетевая операционная система составляет основу любой вычислительной сети. Каждый компьютер в сети в значительной степени автономен, поэтому под сетевой операционной системой в широком смысле понимается совокупность операционных систем отдельных компьютеров, взаимодействующих с целью обмена сообщениями и разделения ресурсов по единым правилам — протоколам. В узком смысле сетевая ОС — это операционная система отдельного компьютера, обеспечивающая ему возможность работать в сети.

В сетевой операционной системе отдельной машины можно выделить несколько частей (рис. 2):

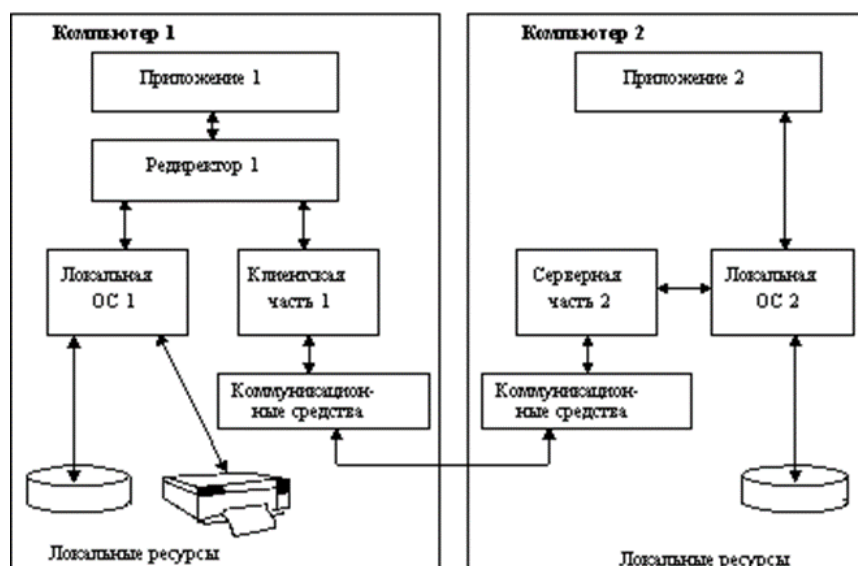
- средства управления локальными ресурсами компьютера: функции распределения оперативной памяти между процессами, планирования и диспетчеризации процессов, управления процессорами в мультипроцессорных машинах, управления периферийными устройствами и другие функции управления ресурсами локальных ОС;
- средства предоставления собственных ресурсов и услуг в общее пользование — серверная часть ОС (сервер). Эти средства обеспечивают, например, блокировку файлов и записей, что необходимо для их совместного использования; ведение справочников имен сетевых ресурсов; обработку запросов удаленного доступа к собственной файловой системе и базе данных; управление очередями запросов удаленных пользователей к своим периферийным устройствам;
- средства запроса доступа к удаленным ресурсам и услугам и их использования — клиентская часть ОС (редиректор). Эта часть выполняет распознавание и перенаправление в сеть запросов к удаленным ресурсам от приложений и пользователей, при этом запрос поступает от приложения в локальной форме, а передается в сеть в другой форме, соответствующей требованиям сервера. Клиентская часть также осуществляет прием ответов от серверов и преобразование их в локальный формат, так что для приложения выполнение локальных и удаленных запросов неразличимо;
- коммуникационные средства ОС, с помощью которых происходит обмен сообщениями в сети. Эта часть обеспечивает адресацию и буферизацию сообщений, выбор маршрута передачи сообщения по сети, надежность передачи и т.п., то есть является средством транспортировки сообщений.



**Рис.2** Части операционной системы

В зависимости от функций, возлагаемых на конкретный компьютер, в его операционной системе может отсутствовать либо клиентская, либо серверная часть.

На рис. 3 показано взаимодействие сетевых компонентов. Здесь компьютер 1 выполняет роль "чистого" клиента, а компьютер 2 — роль "чистого" сервера, соответственно на первой машине отсутствует серверная часть, а на второй — клиентская. На рисунке отдельно показан компонент клиентской части — редиректор. Именно редиректор перехватывает все запросы, поступающие от приложений, и анализирует их. Если выдан запрос к ресурсу данного компьютера, то он переадресовывается соответствующей подсистеме локальной ОС, если же это запрос к удаленному ресурсу, то он переправляется в сеть. При этом клиентская часть преобразует запрос из локальной формы в сетевой формат и передает его транспортной подсистеме, которая отвечает за доставку сообщений указанному серверу. Серверная часть операционной системы компьютера 2 принимает запрос, преобразует его и передает для выполнения своей локальной ОС. После того как результат получен, сервер обращается к транспортной подсистеме и направляет ответ клиенту, выдавшему запрос. Клиентская часть преобразует результат в соответствующий формат и адресует его тому приложению, которое выдало запрос.



Единственным по-настоящему важным отличием распределенных систем от централизованных является межпроцессная взаимосвязь. В централизованных системах связь между процессами, как правило, предполагает наличие разделяемой памяти. Типичный пример - проблема "поставщик-потребитель", в этом случае один процесс пишет в разделяемый буфер, а другой - читает из него. Даже наиболее простая форма синхронизации - семафор - требует, чтобы хотя бы одно слово (переменная самого семафора) было разделяемым. В распределенных системах нет какой бы то ни было разделяемой памяти, таким образом вся природа межпроцессных коммуникаций должна быть продумана заново. Основой этого взаимодействия может служить только передача по сети сообщений. В самом простом случае системные средства обеспечения связи могут быть сведены к двум основным системным вызовам (примитивам), один - для отправки сообщения (*send*), другой - для получения сообщения (*receive*). В дальнейшем на их базе могут быть построены более мощные средства сетевых коммуникаций, такие как распределенная файловая система или вызов удаленных процедур, которые, в свою очередь, также могут служить основой для построения других сетевых сервисов.

### **Способы адресации**

Для того, чтобы послать сообщение, необходимо указать адрес получателя. В очень простой сети адрес может задаваться в виде константы, но в более сложных сетях нужен и более изощренный способ адресации. Одним из вариантов адресации на верхнем уровне является использование физических адресов сетевых адаптеров. Если в получающем компьютере выполняется только один процесс, то ядро будет знать, что делать с поступившим сообщением - передать его этому процессу. Однако, если на машине выполняется несколько процессов, то ядру не известно, какому из них предназначено сообщение, поэтому использование сетевого адреса адаптера в качестве адреса получателя приводит к очень серьезному ограничению - на каждой машине должен выполняться только один процесс.

Альтернативная адресная система использует имена назначения, состоящие из двух частей, определяющие номер машины и номер процесса. Однако адресация типа "машина-процесс" далека от идеала, в частности, она не гибка и не прозрачна, так как пользователь должен явно задавать адрес машины-получателя. В этом случае, если в один прекрасный день машина, на которой работает сервер, отказывает, то программа, в которой жестко используется адрес сервера, не сможет работать с другим сервером, установленным на другой машине.

### **Вызов удаленных процедур (RPC) Концепция удаленного вызова процедур**

Хотя модель передачи сообщений предоставляет удобный способ структурирования операционной системы многомашинной системы, у нее есть один существенный недостаток: связь между процессами построена на парадигме ввода-вывода. Процедуры *send* и *receive* занимаются Идея вызова удаленных процедур (*Remote Procedure Call - RPC*) состоит в расширении хорошо известного и понятного механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. Средства удаленного вызова процедур предназначены для облегчения организации распределенных вычислений. Наибольшая эффективность использования RPC достигается в тех приложениях, в которых существует интерактивная связь между удаленными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Такие приложения называются RPC-ориентированными. Характерными чертами вызова локальных процедур являются:

- Асимметричность, то есть одна из взаимодействующих сторон является инициатором;
- Синхронность, то есть выполнение вызываемой процедуры приостанавливается с момента выдачи запроса и возобновляется только после возврата из вызываемой процедуры.

Реализация удаленных вызовов существенно сложнее реализации вызовов локальных процедур. Начнем с того, что поскольку вызывающая и вызываемая процедуры выполняются на разных машинах, то они имеют разные адресные пространства, и это создает проблемы при передаче параметров и результатов, особенно если машины не идентичны. Так как RPC не может рассчитывать на разделяемую память, то это означает, что параметры RPC не должны содержать указателей на ячейки нестековой памяти и что значения параметров должны копироваться с одного компьютера на другой. Следующим отличием RPC от локального вызова является то, что он обязательно использует нижележащую систему связи, однако это не должно быть явно

видно ни в определении процедур, ни в самих процедурах. Удаленность вносит дополнительные проблемы. Выполнение вызывающей программы и вызываемой локальной процедуры в одной машине реализуется в рамках единого процесса. Но в реализации RPC участвуют как минимум два процесса - по одному в каждой машине. В случае, если один из них аварийно завершится, могут возникнуть следующие ситуации: при аварии вызывающей процедуры удаленно вызванные процедуры станут "осиротевшими", а при аварийном завершении удаленных процедур станут "обездоленными родителями" вызывающие процедуры, которые будут безрезультатно ожидать ответа от удаленных процедур.

**Базовые операции RPC**

Чтобы понять работу [RPC](#), рассмотрим вначале выполнение вызова локальной процедуры в обычной машине, работающей автономно. Пусть это, например, будет системный вызов `count=read (fd, buf, nbytes);` где `fd` - целое число, `buf` - массив символов, `nbytes` - целое число. Чтобы осуществить вызов, вызывающая процедура заталкивает параметры в стек в обратном порядке (рисунок 4). После того, как вызов `read` выполнен, он помещает возвращаемое значение в регистр, перемещает адрес возврата и возвращает управление вызывающей процедуре, которая выбирает параметры из стека, возвращая его в исходное состояние. Заметим, что в языке C параметры могут вызываться или по ссылке (*by name*), или по значению (*by value*). По отношению к вызываемой процедуре параметры-значения являются инициализируемыми локальными переменными. Вызываемая процедура может изменить их, и это не повлияет на значение оригиналов этих переменных в вызывающей процедуре. Если в вызываемую процедуру передается указатель на переменную, то изменение значения этой переменной вызываемой процедурой влечет изменение значения этой переменной и для вызывающей процедуры. Этот факт весьма существенен для RPC. Существует также другой механизм передачи параметров, который не используется в языке C. Он называется *call-by-copy/restore* и состоит в необходимости копирования вызывающей программой переменных в стек в виде значений, а затем копирования назад после выполнения вызова поверх оригинальных значений вызывающей процедуры. Решение о том, какой механизм передачи параметров использовать, принимается разработчиками языка. Иногда это зависит от типа передаваемых данных. В языке C, например, целые и другие скалярные данные всегда передаются по значению, а массивы - по ссылке.

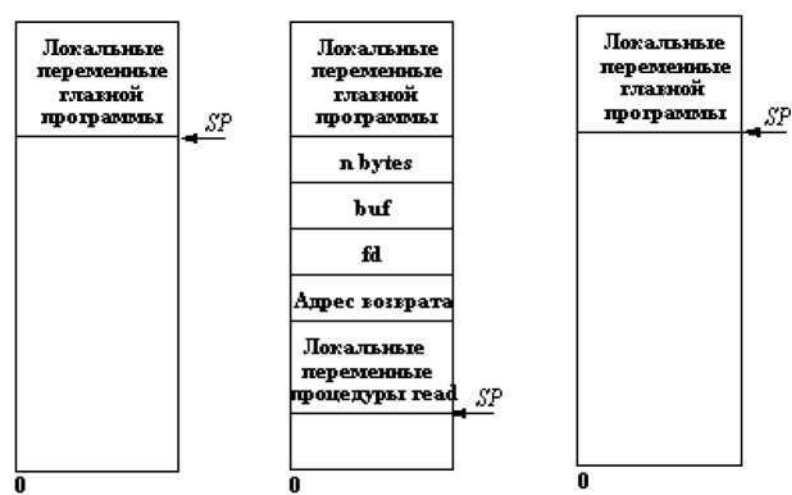


Рис. 4 Выполнение вызова процедуры на локальной машине

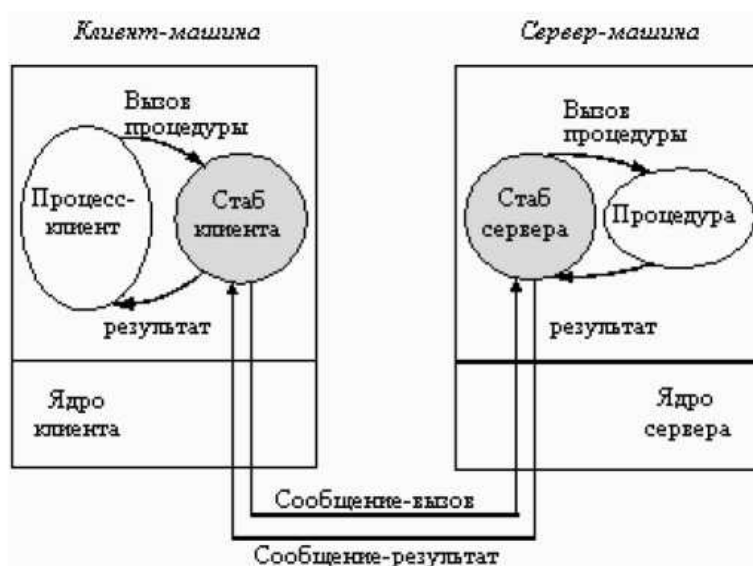
Идея, положенная в основу RPC, состоит в том, чтобы сделать вызов удаленной процедуры выглядящим по возможности также, как и вызов локальной процедуры.

**Этапы выполнения RPC**

Взаимодействие программных компонентов при выполнении удаленного вызова процедуры иллюстрируется рисунком 5. После того, как клиентский стаб был вызван программой клиентом, его первой задачей является заполнение буфера отправляемым сообщением. В некоторых системах клиентский стаб имеет единственный буфер фиксированной длины, заполняемый каждый раз с самого начала при поступлении каждого нового запроса. В других системах буфер сообщения представляет собой пул буферов



для отдельных полей сообщения, причем некоторые из этих буферов уже заполнены. Этот метод особенно подходит для тех случаев, когда пакет имеет формат, состоящий из большого числа полей, но значения многих из этих полей не меняются от вызова к вызову. Затем параметры должны быть преобразованы в соответствующий формат и вставлены в буфер сообщения. К этому моменту сообщение готово к передаче, поэтому выполняется прерывание по вызову ядра.



**Рис.5** Взаимодействие программных компонентов при выполнении удаленного вызова процедуры

Когда ядро получает управление, оно переключает контексты, сохраняет регистры процессора и карту памяти (дескрипторы страниц), устанавливает новую карту памяти, которая будет использоваться для работы в режиме ядра. Поскольку контексты ядра и пользователя различаются, ядро должно точно скопировать сообщение в свое собственное адресное пространство, так, чтобы иметь к нему доступ, запомнить адрес назначения (а, возможно, и другие поля заголовка), а также оно должно передать его сетевому интерфейсу. На этом завершается работа на клиентской стороне. Включается таймер передачи, и ядро может либо выполнять циклический опрос наличия ответа, либо передать управление планировщику, который выберет какой-либо другой процесс на выполнение. В первом случае ускорится выполнение запроса, но отсутствует мультипрограммирование. На стороне сервера поступающие биты помещаются принимающей аппаратурой либо во встроенный буфер, либо в оперативную память. Когда вся информация будет получена, генерируется прерывание. Обработчик прерывания проверяет правильность данных пакета и определяет, какому стабу следует их передать. Если ни один из стабов не ожидает этот пакет, обработчик должен либо поместить его в буфер, либо вообще отказаться от него. Если имеется ожидающий стаб, то сообщение копируется ему. Наконец, выполняется переключение контекстов, в результате чего восстанавливаются регистры и карта памяти, принимая те значения, которые они имели в момент, когда стаб сделал вызов receive. Теперь начинает работу серверный стаб. Он распаковывает параметры и помещает их соответствующим образом в стек. Когда все готово, выполняется вызов сервера. После выполнения процедуры сервер передает результаты клиенту. Для этого выполняются все описанные выше этапы, только в обратном порядке. Рисунок 6 показывает последовательность команд, которую необходимо выполнить для каждого RPC-вызова



**Рис. 6** Последовательность команд, которую необходимо выполнить для каждого RPC-вызова

### Динамическое связывание

Рассмотрим вопрос о том, как клиент задает месторасположение сервера. Одним из методов решения этой проблемы является непосредственное использование сетевого адреса сервера в клиентской программе. Недостаток такого подхода - его чрезвычайная негибкость: при перемещении сервера, или при увеличении числа серверов, или при изменении интерфейса во всех этих и многих других случаях необходимо перекомпилировать все программы, которые использовали жесткое задание адреса сервера. Для того, чтобы избежать всех этих проблем, в некоторых распределенных системах используется так называемое динамическое связывание.

Начальным моментом для динамического связывания является формальное определение (спецификация) сервера. Спецификация содержит имя файл-сервера, номер версии и список процедур-услуг, предоставляемых данным сервером для клиентов. Для каждой процедуры дается описание ее параметров с указанием того, является ли данный параметр входным или выходным относительно сервера.

Некоторые параметры могут быть одновременно входными и выходными - например, некоторый массив, который посылается клиентом на сервер, модифицируется там, а затем возвращается обратно клиенту (операция copy/ restore).

### Семантика RPC в случае отказов

В идеале RPC должен функционировать правильно и в случае отказов. Рассмотрим следующие классы отказов:

1. Клиент не может определить местонахождения сервера, например, в случае отказа нужного сервера, или из-за того, что программа клиента была скомпилирована давно и использовала старую версию интерфейса сервера. В этом случае в ответ на запрос клиента поступает сообщение, содержащее код ошибки.
2. Потерян запрос от клиента к серверу. Самое простое решение - через определенное время повторить запрос.
3. Потеряно ответное сообщение от сервера клиенту. Этот вариант сложнее предыдущего, так как некоторые процедуры не являются идемпотентными. Идемпотентной называется процедура, запрос на выполнение которой можно повторить несколько раз, и результат при этом не изменится. Примером такой процедуры может служить чтение файла. Но вот процедура снятия некоторой суммы с банковского счета не является идемпотентной, и в случае потери ответа повторный запрос может существенно изменить состояние счета клиента. Одним из возможных решений является приведение всех процедур к идемпотентному виду. Однако на практике это не всегда удается, поэтому может быть использован другой метод - последовательная нумерация всех запросов клиентским ядром. Ядро сервера запоминает номер самого последнего запроса от каждого из клиентов, и при получении каждого запроса выполняет анализ - является ли этот запрос первичным или повторным.
4. Сервер потерпел аварию после получения запроса. Здесь также важно свойство идемпотентности, но, к сожалению, не может быть применен подход с нумерацией запросов. В данном случае имеет значение, когда произошел отказ - до или после выполнения операции. Но клиентское ядро не может распознать эти ситуации,

для него известно только то, что время ответа истекло.

5. Клиент потерпел аварию после отсылки запроса. В этом случае выполняются вычисления результатов, которых никто не ожидает. Такие вычисления называют "сиротами". Наличие сирот может вызвать различные проблемы: непроизводительные затраты процессорного времени, блокирование ресурсов, подмена ответа на текущий запрос ответом на запрос, который был выдан клиентской машиной еще до перезапуска системы. Как поступать с сиротами? Рассмотрим 4 возможных решения:

- Уничтожение
- Перевоплощение.
- Мягкое перевоплощение
- Истечение срока
- 

#### **Синхронизация в распределенных системах**

К вопросам связи процессов, реализуемой путем передачи сообщений или вызовов RPC, тесно примыкают и вопросы синхронизации процессов. Синхронизация необходима процессам для организации совместного использования ресурсов, таких как файлы или устройства, а также для обмена данными. В однопроцессорных системах решение задач взаимного исключения, критических областей и других проблем синхронизации осуществлялось с использованием общих методов, таких как семафоры и мониторы. Однако эти методы не совсем подходят для распределенных систем, так как все они базируются на использовании разделяемой оперативной памяти. Например, два процесса, которые взаимодействуют, используя семафор, должны иметь доступ к нему. Если оба процесса выполняются на одной и той же машине, они могут иметь совместный доступ к семафору, хранящемуся, например, в ядре, делая системные вызовы. Однако, если процессы выполняются на разных машинах, то этот метод не применим, для распределенных систем нужны новые подходы.

## **Лекция 1.2. Обзор системы UNIX**

В 40-е и 50-е годы все компьютеры были персональными компьютерами, по крайней мере, в следующем смысле: в те времена обычное использование компьютера заключалось в том, что пользователь записывался на определенный час, и вся машина на этот период оказывалась в его распоряжении. Физические размеры этих компьютеров были огромными, но работать на таком компьютере в каждый момент времени мог тогда только один пользователь (программист).

Чтобы как-то усовершенствовать существовавшую схему, которую практически все считали неудовлетворительной и непродуктивной, в Дартмутском колледже и Массачусетском технологическом институте были изобретены системы разделения времени. Дартмутская система, в которой работал только BASIC, имела кратковременный коммерческий успех, после чего она полностью исчезла. Система Массачусетского технологического института, CTSS, была универсальной системой, получившей колоссальный успех в научных кругах. За короткое время исследователи из Массачусетского технологического института объединили усилия с лабораторией Bell Labs и корпорацией General Electric (в те времена General Electric производила компьютеры) и начали разработку системы второго поколения MULTICS

Хотя лаборатория Bell Labs была одним из основополагающих партнеров проекта MULTICS, она вскоре вышла из него, в результате чего один из исследователей этой лаборатории, Кен Томпсон, оказался в ситуации поиска новых интересных систем MULTICS, для чего у него был списанный мини-компьютер PDP-7. Не смотря на крошечные размеры PDP-7, система Томпсона работала и позволяла Томпсону продолжать разработки новой операционной системы. Впоследствии еще один исследователь лаборатории Bell Labs, Брайан Керниган, как-то в шутку назвал эту систему UNICS (UNiplexed Information and Computing Service — примитивная информационная и вычислительная служба). Несмотря на все каламбуры и шутки о системе MULTICS (кое-кто предлагал назвать систему Томпсона EUNUCHS, то есть евнухом), кличка, данная Керниганом, прочно пристала к новой системе, хотя написание этого слова стало слегка короче, при том же произношении, превратившись в UNIX.

### **PDP-11 UNIX**

Работа Томпсона произвела на его коллег из лаборатории Bell Labs столь сильное впечатление, что вскоре к нему присоединился Деннис Ритчи, а чуть позднее и весь его отдел. На это время приходится два технологических усовершенствования. Во-первых, система UNIX была перенесена с устаревшей машины PDP-7 на более современные компьютеры PDP-11/20, а позднее на PDP-11/45 и PDP-11/70. Последние две машины доминировали в мире мини-компьютеров в течение большей части 70-х годов. Компьютеры PDP-11/45 и PDP-11/70 представляли собой мощные по тем временам машины с большой физической памятью (256 Кбайт и 2 Мбайт соответственно). Кроме того, они обладали аппаратной защитой памяти, что позволяло поддерживать одновременную работу нескольких пользователей. Однако это были 16-разрядные машины, и это ограничивало адресное пространство процессов 64 Кбайт для команд и 64 Кбайт для данных, несмотря на то, что у этих машин физической памяти было значительно больше 64 Кбайт (Точнее, у этих машин было одно общее 64-килобайтное адресное пространство и для команд, и для данных. Аналоги этих машин в СССР назывались СМ-4 и СМ-1420.) Второе усовершенствование касалось языка, на котором писалась операционная система UNIX. Уже давно стало очевидно, что необходимость переписывать всю систему заново для каждой новой машины — занятие отнюдь не веселое, поэтому Томпсон решил переписать UNIX на языке высокого уровня, который он сам специально разработал и назвал языком В. Язык В представлял собой упрощенную форму языка BCPL (который, в свою очередь, был упрощенным языком CPL, подобно PL/1, никогда не работавшим). Эта попытка оказалась неудачной из-за слабостей языка В, в первую очередь, из-за отсутствия в нем структур данных. Тогда Ритчи разработал следующий язык, явившийся преемником языка В, который, естественно, получил название С, и написал для него прекрасный компилятор. Вместе Томпсон и Ритчи переписали UNIX на С. Язык С оказался как раз тем языком, который и был нужен в то время, и он сохраняет лидирующие позиции в области системного программирования до сих пор.

В 1974 г. Ритчи и Томпсон опубликовали ставшую важной вехой в истории компьютеров статью об операционной системе UNIX. За работу, описанную в данной статье, им позднее ассоциацией по вычислительной технике ACM была присуждена престижная премия Тьюринга. Публикация этой статьи привела к тому, что многие университеты выстроились в очередь в лабораторию Bell Labs за копией системы UNIX. Корпорация AT&T, являвшаяся учредителем лаборатории Bell Labs, была в то время регулируемой монополией и ей не разрешалось заниматься компьютерным бизнесом, поэтому она не возражала против того, чтобы университеты получали лицензии на право использования системы UNIX за умеренную плату.

По случайному стечению обстоятельств, которые часто формируют историю, машина PDP-11 использовалась на факультетах кибернетики практически каждого университета, а операционные системы, с которыми поставлялись эти компьютеры, профессора и студенты считали ужасными. Операционная система UNIX быстро заполнила имевшийся вакуум, не в последнюю очередь также благодаря тому, что система поставлялась с полным комплектом исходных текстов, поэтому новые владельцы системы могли без конца подправлять и совершенствовать ее. Операционной системе UNIX было посвящено множество научных симпозиумов, на них докладчики рассказывали о скрытых ошибках в ядре, которые им удалось обнаружить и исправить. Австралийский профессор Джон Лайонс написал к исходному тексту системы UNIX комментарий стилем, обычно использующимся в трактатах О Джеффри Чосере или Вильяме Шекспире. Книга описывала систему UNIX Version 6, названную так потому, что эта версия операционной системы была описана в шестом издании руководства программиста UNIX Programmer's Manual. Исходный текст системы состоял всего из 8200 строк на С и 900 строк ассемблера. В результате новые идеи и усовершенствования системы распространялись с огромной скоростью.

Через несколько лет Version 6 сменила Version 7, которая стала первой переносимой версией операционной системы UNIX (она работала как на машинах PDP-11, так и на Interdata 8/32). Эта версия системы уже состояла из 18 800 строк на С и 2100 ассемблерных строк. На Version 7 выросло целое поколение студентов, которые, закончив свои учебные заведения и начав работу в промышленности, в значительной степени содействовали дальнейшему распространению системы UNIX. К середине 80-х операционная система UNIX широко применялась на мини-компьютерах и инженерных рабочих станциях самых различных производителей. Многие компании даже приобрели лицензии на исходные тексты, чтобы производить свои версии системы UNIX. Одной из таких компаний была небольшая начинающая фирма Microsoft, в течение нескольких лет продававшая Version 7 под именем XENIX, пока ее интересы не повернулись в другую сторону.

## Переносимая система UNIX

После того как система UNIX была переписана на языке высокого уровня C, задача переноса ее на новые машины стала значительно более простым делом. Для переноса системы сначала требуется написать для новой машины компилятор с языка C. Затем для устройств ввода-вывода новой машины, таких как терминалы, принтеры и диски, нужно написать драйверы устройств. Хотя драйвер может быть написан и на C, его нельзя просто перекомпилировать на новой машине и запустить на ней, поскольку устройство и организация дисков на разных платформах сильно отличаются. Наконец, требуется переписать заново, как правило, на ассемблере, небольшое количество машиннозависимого кода, например, обработчики прерываний и процедуры управления памятью.

Процесс переноса операционной системы UNIX на мини-компьютер Interdata 8/32 шел медленно, так как вся работа должна была производиться на единственной в лаборатории машине, на которой работала система UNIX, на [PDP-11](#). Однако случилось так, что компьютер PDP-11 оказался на пятом этаже лаборатории, тогда как Interdata 8/32 был установлен на первом этаже. Создание новой версии системы означало ее компиляцию на пятом этаже и запись на магнитную ленту, которая физически переносилась на первый этаж, чтобы посмотреть, работает ли она. Уже через несколько месяцев подобной работы у разработчиков появился сильный интерес к возможности соединения этих двух машин электронным способом. Вся работа с сетью в системе UNIX началась именно с соединения этих двух машин.

После Interdata система UNIX была перенесена на VAX, а затем и на другие компьютеры. После того как в 1984 году правительство США разделило корпорацию AT&T, компания получила законную возможность инвестировать средства в компьютерную промышленность, что она и сделала. Вскоре после этого компания AT&T выпустила на рынок первый коммерческий вариант системы UNIX, System III. Ее выход на рынок был не очень успешным, поэтому через год она была заменена улучшенной версией, System V. Что случилось с System IV, до сих пор остается одной из неразгаданных тайн компьютерного мира. Оригинальную систему System V впоследствии сменили выпуски 2,3 и 4 все той же System V, каждый последующий выпуск более сложный и громоздкий, чем предшествующий. В процессе усовершенствований оригинальная идея, лежащая в основе системы UNIX и заключающаяся в простоте и элегантности системы, была в значительной мере утрачена. Хотя Ритчи и Томпсон позднее выпустили 8-ю, 9-ю, и 10-ю редакции системы UNIX, они не получили широкого распространения, так как компания AT&T все свои усилия на рынке вкладывала в продажу версии System V. Однако некоторые идеи из 8-й, 9-й и 10-й редакций системы, в конце концов, были включены в System V. Наконец, компания AT&T решила, что хочет быть телефонной компанией, а не компьютерной фирмой и в 1993 году продала весь свой бизнес, связанный с системой UNIX, корпорации Novell, которая, в свою очередь, в 1995 году перепродала его компании Santa Cruz Operation. К этому времени стало практически неважным, кому принадлежит этот бизнес, так как почти у всех основных компьютерных компаний уже были лицензии.

## Berkeley UNIX

Калифорнийский университет в Беркли был одним из многих университетов, приобретших UNIX Version 6 практически с момента ее выхода. Поскольку с системой поставлялся полный комплект исходных текстов, университет мог существенно модифицировать систему. При финансовой поддержке управления перспективного планирования научно-исследовательских работ ARPA (Advanced Research Projects Agency) при Министерстве обороны США университет в Беркли разработал и выпустил улучшенную версию операционной системы UNIX для мини-компьютера PDP-11, названную 1BSD (First Berkeley Software Distribution - программное изделие Калифорнийского университета, 1-я версия). Вслед за этой магнитной лентой вскоре появилась 2BSD, также для PDP-11.

Более важным событием был выпуск версии 3BSD и ее преемника 4BSD, уже рассчитанных на 32-разрядные машины VAX. Хотя компания AT&T распространяла свою собственную версию для VAX, называвшуюся 32V, по существу, это была Version 7. В противоположность ей система 4BSD (включая 4.1BSD, 4.2BSD, 4.3BSD и 4.4BSD) содержала большое количество усовершенствований. Важнейшими из них были использование виртуальной памяти и страничная подкачка файлов, что позволяло создавать программы, большие по размеру, чем физическая память. Другое изменение заключалось в поддержке имен файлов длиной более 14 символов. Реализация файловой системы также была изменена, благодаря чему работа с

файловой системой стала существенно быстрее. Более надежной стала обработка сигналов. В 4-й версии Berkeley UNIX появилась поддержка сетей, в результате чего используемый в 4BSD протокол TCP/IP стал стандартом де-факто в мире UNIX, а позднее и в Интернете, в котором преобладают серверы на базе системы UNIX.

Университет в Беркли также добавил значительное количество утилит для системы UNIX, включая новый редактор vi и новую оболочку ssh, компиляторы языков Pascal и Lisp и многое другое. Все эти усовершенствования привели к тому, что многие производители компьютеров (Sun Microsystems, DEC и другие) стали основывать свои версии системы UNIX на Berkeley UNIX, а не на «официальной» версии компании AT&T, System V. В результате Berkeley UNIX получила широкое распространение в академических и исследовательских кругах.

### Стандартная система UNIX

К концу 80-х широкое распространение получили две различные и в чем-то несовместимые версии системы UNIX: 4.3 BSD и System V Release 3. Кроме того, практически каждый производитель добавлял нестандартные усовершенствования. Этот раскол в мире UNIX, вместе с тем фактом, что стандарта на формат двоичных программ не было, сильно замедлил коммерческое признание операционной системы UNIX. Производители программного обеспечения не могли написать пакет программ для системы UNIX так, чтобы он гарантированно мог быть запущен на любой системе UNIX (как, например, это делалось в системе MS-DOS). Различные попытки стандартизации системы UNIX провалились с самого начала. Например, корпорация AT&T выпустила стандарт **SVID** (System V Interface Definition — описание интерфейса UNIX System V), в котором определялись все системные вызовы, форматы файлов и т. д. Этот документ был попыткой построить в одну шеренгу всех производителей UNIX System V, но он не оказал никакого влияния на вражеский лагерь (BSD), который просто проигнорировал его.

Первая попытка примирить два варианта системы UNIX была предпринята при содействии Совета по стандартам при Институте инженеров по электротехнике и электронике IEEE Standard Boards, глубокоуважаемой и, что еще важнее, нейтральной организации. В этой работе приняли участие сотни людей из промышленности, академических и правительственных организаций. Коллективное название данного проекта — **POSIX**. Первые три буквы этого сокращения означали Portable Operating System — переносимая операционная система. Буквы IX в конце слова были добавлены, чтобы имя проекта выглядело юниксообразно.

После большого количества высказанных аргументов и контраргументов, опровержений и опровергнутых опровержений, комитет POSIX выработал стандарт, известный как **1003.1**. Этот стандарт определяет набор библиотечных процедур, которые должна предоставлять каждая соответствующая данному стандарту система UNIX. Большая часть этих процедур обращается к системному вызову, но некоторые из них могут быть реализованы вне ядра. Типичными процедурами являются open, read и fork. Идея стандарта POSIX заключается в том, что производитель программного обеспечения при написании программы использует только процедуры, описанные в стандарте 1003.1, таким образом, гарантируя, что эта программа будет работать на любой версии системы UNIX, поддерживающей данный стандарт.

Хотя большинство комитетов по стандартам, как правило, создают нечто ужасное, сплошь состоящее из компромиссов, стандарт 1003.1 заметно отличается от общего правила в лучшую сторону, особенно если учитывать большое число заинтересованных сторон, принимавших участие в его разработке. Вместо того чтобы принять за точку отсчета объединение множеств всех свойств System V и BSD (норма для большинства комитетов по стандартам), комитет IEEE взял за основу пересечение множеств. То есть в первом приближении дело обстоит так: если какое-либо свойство присутствовало как в System V, так и в BSD, то оно включалось в стандарт. В противном случае это свойство в стандарт не включалось. В результате применения такого алгоритма стандарт 1003.1 сильно напоминает прямого общего предка систем System V и BSD, а именно Version 7. От Version 7 стандарт сильнее всего отличается в двух областях: обработке сигналов (что по большей части взято из BSD) и управлению терминалом, что представляет собой нововведение. Документ 1003.1 написан так, чтобы как разработчики операционной системы, так и создатели программного обеспечения были способны его понять, что также ново в мире стандартов, хотя в настоящее время уже полным ходом ведется работа по исправлению этого нестандартного для стандартов свойства.

Стандарт 1003.1 описывает только системные вызовы. Принят также ряд документов, стандартизирующих

потоки, утилиты, сетевое программное обеспечение и многие другие особенности системы UNIX. Кроме того, язык C также был стандартизирован Национальным институтом стандартизации США ANSI и Международной организацией по стандартизации ISO.

## Linux

В ранние годы развития системы MINIX и обсуждений этой системы в Интернете многие люди просили (а часто требовали) все больше новых и более сложных функций, и на эти просьбы автор часто отвечал отказом (сохраняя небольшие размеры системы, чтобы студенты могли полностью понять ее за один семестр). Эти отказы раздражали многих пользователей. В те времена бесплатной системы FreeBSD еще не было. Наконец через несколько лет финский студент Линус Торвалдс решил сам написать ещё один клон системы UNIX, который он называл Linux. Это должна была быть полноценная операционная система, со многими функциями, отсутствующими (по намерению авторов) в системе MINIX. Первая версия операционной системы Linux 0.01 была выпущена в 1991 году. Она была разработана и собрана в системе MINIX и заимствовала некоторые идеи системы MINIX, начиная со структуры дерева исходных текстов и кончая структурой файловой системы. Однако, в отличие от микроядерной системы MINIX, Linux была монолитной системой, то есть вся операционная система помещалась в ядре. Размер исходного текста составил 9300 строк на C и 950 строк на ассемблере, что приблизительно совпадало с версией MINIX. Функционально первая версия Linux также практически почти не отличалась от MINIX.

Операционная система Linux быстро росла в размерах и впоследствии развилась в полноценный клон UNIX с виртуальной памятью, более сложной файловой системой и многими другими добавленными функциями. Хотя изначально система Linux работала только на процессоре Intel 386 (и даже содержала ассемблерный код 386-й машины посреди процедур на языке C), она была довольно быстро перенесена на другие платформы и теперь работает на широком спектре машин, как и UNIX. Однако одно из основных отличий системы Linux от других клонов системы UNIX заключается в использовании многих специальных особенностей компилятора gcc, поэтому, чтобы откомпилировать ее стандартным ANSI C компилятором, потребуется приложить немало усилий.

Следующим основным выпуском системы Linux была версия 1.0, появившаяся в 1994 году. Она состояла из 165 000 строк кода и включала новую файловую систему, отображение файлов на адресное пространство памяти и совместимое с BSD сетевое программное обеспечение с сокетами и TCP/IP. Она также включала многие новые драйверы устройств. Следующие два года выходили версии с незначительными исправлениями.

К этому времени операционная система Linux стала достаточно совместимой с UNIX, поэтому в нее было перенесено большое количество программного обеспечения UNIX, что значительно увеличило полезность рассматриваемой системы. Кроме того, операционная система Linux привлекла большое количество людей, которые начали работу над ее совершенствованием и расширением под общим руководством Торвалдса.

Следующий главный выпуск, версия 2.0, вышел в свет в 1996 году. Эта версия системы Linux состояла из 470 000 строк на C и 8000 строк ассемблерного текста. Она включала в себя поддержку 64-разрядной архитектуры, симметричной многозадачности, новых сетевых протоколов и прочих многочисленных функций. Значительную часть от общей массы исходного текста составляла внушительная коллекция различных драйверов устройств. Следом за версией 2.0 довольно часто выходили дополнительные выпуски.

В систему Linux была перенесена внушительная часть стандартного программного обеспечения UNIX, включая более 1000 утилит, оконную систему X Windows и большую часть сетевого программного обеспечения. Кроме того, специально для Linux было написано два различных графических интерфейса пользователя: GNOME и KOE. В общем, система Linux выросла в полноценный клон UNIX со всеми погрешностями, какие только могут понадобиться фанату UNIX.

## Задачи UNIX

Операционная система [UNIX](#) представляет собой интерактивную систему, разработанную для одновременной поддержки нескольких процессов и нескольких пользователей. Она была разработана программистами и для программистов, чтобы использовать ее в окружении, в котором большинство пользователей являются относительно опытными и занимаются проектами (часто довольно сложными) разработки программного обеспечения. Во многих случаях большое количество программистов активно

сотрудничают в деле создания единой системы, поэтому в операционной системе UNIX есть достаточное количество средств, позволяющих программистам работать вместе и управлять совместным использованием общей информации. Очевидно, что модель группы опытных программистов, совместно работающих над созданием передового программного обеспечения, существенно отличается от модели одиночного начинающего пользователя, сидящего за персональным компьютером в текстовом процессоре, и это отличие отражено в операционной системе UNIX от начала до конца. Чего хотят от операционной системы хорошие программисты? Во-первых, большинство хотело бы, чтобы их система была простой, элегантной и непротиворечивой. Например, на нижнем уровне файл должен представлять собой просто набор байтов. Наличие различных классов файлов для последовательного и произвольного доступа, доступа по ключу, удаленного доступа и т. д. (как это реализовать на мейнфреймах) просто является помехой. А если команда `ls A*` означает вывод списка всех файлов, имя которых начинается с буквы «А», то команда `rm A*` должна означать удаление всех файлов, имя которых начинается с буквы «А», а не одного файла, имя которого состоит из буквы «А» и звездочки. Эта характеристика иногда называется принципом наименьшей неожиданности.

Другое свойство, которое, как правило, опытные программисты желают видеть в операционной системе, — это мощь и гибкость. Это означает, что в системе должно быть небольшое количество базовых элементов, которые можно комбинировать бесконечным числом способов, чтобы приспособить их для конкретного приложения. Одно из основных правил системы UNIX заключается в том, что каждая программа должна выполнять всего одну функцию, но делать это хорошо. Таким образом, компиляторы не занимаются созданием листингов, так как другие программы могут лучше справиться с этой задачей.

Наконец, у большинства программистов сильная неприязнь к бесполезной избыточности. Зачем писать `сору`, когда достаточно `ср`? Чтобы получить список всех строк, содержащих строку «ard» из файла `f`, программист в операционной системе UNIX вводит команду `grep ard f`

Противоположный подход состоит в том, что программист сначала запускает программу `grep` (без аргументов), после чего программа `grep`



приветствует программиста фразой: «Здравствуйте, я gper. Я ищу символные строки в файлах. Введите, пожалуйста, искомую строку». Получив строку, программатор просит задать имя файла. Затем она спрашивает, есть ли еще какие-либо файлы. Наконец она выводит листинг итогового задания и спрашивает, все ли верно. Хотя такой тип пользовательского интерфейса, возможно, удобен для начинающих пользователей, он бесконечно раздражает опытных программистов. То, что им нужно— это слуга, а не нянька.

## Интерфейсы системы UNIX

Операционную систему UNIX можно рассматривать в виде пирамиды (рисунок 7). У основания пирамиды располагается аппаратное обеспечение, состоящее из центрального процессора, памяти, дисков, терминалов и других устройств. На голом «железе» работает операционная система 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

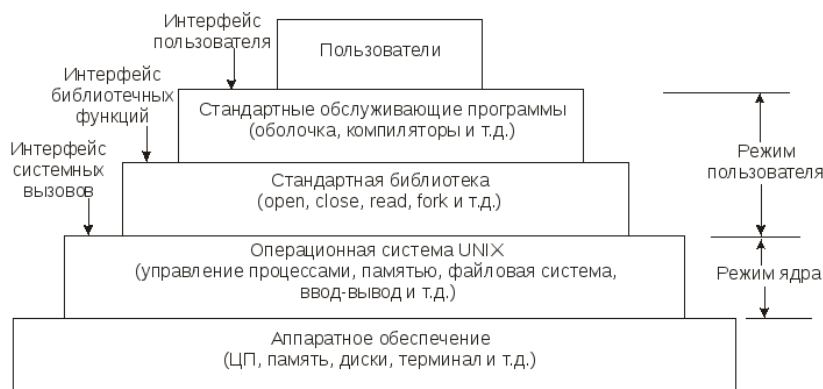


Рис. 7 Операционная система UNIX

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 операционной системе UNIX. Поскольку на языке C невозможно написать команду эмулированного прерывания, этим занимаются библиотечные функции, по одной на системный вызов. Эти процедуры написаны на ассемблере, но они могут вызываться из программ, написанных на C. Каждая такая процедура помещает аргументы в нужное место и выполняет команду эмулированного прерывания TRAP. Таким образом, чтобы обратиться к системному вызову read, программа на C должна вызвать библиотечную процедуру read. Кстати, в стандарте POSIX определен именно интерфейс библиотечных функций, а не интерфейс системных вызовов. Другими словами, 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.

Помимо операционной системы и библиотеки системных вызовов, все версии UNIX содержат большое количество стандартных программ, некоторые из них описываются стандартом POSIX 1003.2, тогда как другие могут различаться в разных версиях системы UNIX. К этим программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами. Именно эти программы и запускаются пользователем с терминала.

Таким образом, мы можем говорить о трех интерфейсах в операционной системе UNIX: интерфейсе системы вызовов, интерфейсе библиотечных функций и интерфейсе, образованном набором стандартных обслуживающих программ. Хотя именно последний интерфейс большинство пользователей считает системой UNIX, в действительности он не имеет практически никакого отношения к самой операционной системе и легко может быть заменен.

## Оболочка UNIX

У многих версий системы UNIX имеется графический интерфейс пользователя, схожий с популярными интерфейсами, примененными на компьютере Macintosh и впоследствии в системе Windows. Однако истинные программисты до сих пор предпочитают интерфейс командной строки, называемый оболочкой

(shell). Подобный интерфейс значительно быстрее в использовании, существенно мощнее, проще расширяется и не раздражает пользователя необходимостью постоянно хвататься за мышь. Ниже будет кратко описана оболочка Бурна (sh). С тех пор было написано много других оболочек (ksh, bash и т. д.). Хотя система UNIX полностью поддерживает графическое окружение (X Windows), даже в этом мире многие программисты просто создают множество консольных окон и действуют так, как если бы у них была дюжина алфавитно-цифровых терминалов, на каждом из которых работала бы оболочка.

Когда оболочка запускается, она инициализируется, а затем печатает на экране символ приглашения к вводу (обычно это знак доллара или процента) и ждет, когда пользователь введет командную строку.

После того как пользователь введет командную строку, оболочка извлекает из нее первое слово и ищет файл с таким именем. Если такой файл удастся 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. Здесь важно подчеркнуть, что оболочка представляет собой обычную пользовательскую программу. Все, что ей нужно, — это способность ввода с терминала и вывода на терминал, а также возможность запускать другие программы.

У команд оболочки могут быть аргументы, которые передаются запускаемой программе в виде текстовых строк. Например, командная строка

```
cp src dest
```

запускает программу cp с двумя аргументами src и dest. Эта программа интерпретирует первый аргумент как имя существующего файла. Она копирует этот файл и называет его копию dest.

Не все аргументы обязательно должны быть именами файлов. В строке

```
head -20 file
```

первый аргумент -20 велит программе head напечатать первые 20 строк file вместо принятых по умолчанию 10 строк. Аргументы, управляющие работой команды или указывающие дополнительные значения, называются флагами или ключами и по соглашению обозначаются знаком тире. Тире требуется, чтобы избежать двусмысленности. Так, например, команда

```
head 20 file
```

вполне законна. Она велит программе head напечатать первые 10 строк файла 20, а затем первые 10 строк файла file. Большинство команд системы UNIX могут принимать несколько флагов и аргументов.

Чтобы было легче указывать группы файлов, оболочка принимает так называемые волшебные символы, иногда называемые также джокерами. Например, символ звездочки означает все возможные варианты текстовой строки, так что строка

```
ls *.c
```

велит программе ls вывести список всех файлов, имя которых оканчивается на .c. Если файлы x.c, y.c и z.c существуют, то данная команда эквивалентна команде

```
ls x.c y.c z.c
```

Другим джокером является вопросительный знак, который заменяет один любой символ. Кроме того, в квадратных скобках можно указать множество символов, из которых программа должна будет выбрать один. Например, команда

```
ls [a-z]*
```

велит программе ls вывести список всех файлов, имя которых начинается с символов «a», «p» или «e».

Программа вроде 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 standard error (стандартное устройство для вывода сообщений об ошибках). По умолчанию всем трем устройствам соответствует терминал, то есть клавиатура для ввода и экран для вывода. Многие программы в системе UNIX читают данные со стандартного устройства ввода и пишут на стандартное устройство вывода. Например, команда sort

Стандартные ввод и вывод также можно перенаправить, что является очень полезным свойством. Для этого используются символы «<» и «>» соответственно. Разрешается их одновременное использование в одной командной строке. Например, команда:

```
sort <in >out
```

Рассмотрим следующую командную строку, состоящую из трех отдельных команд:

```
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 свою работу, оболочка запускает программу `head`, веля ей распечатать первые 30 строк из файла `temp` на стандартном устройстве вывода, которым по умолчанию является терминал. Наконец, временный файл `temp` удаляется.

В системе UNIX часто используются командные строки, в которых первая программа в командной строке формирует вывод, используемый второй программой в качестве входа. В приведенном выше примере для этого использовался временный файл `temp`. Однако система UNIX предоставляет более простой способ для этого. В командной строке

```
sort <in | head -30
```

для этого используется вертикальная черта, называемая символом канала. Этот символ означает, что вывод программы `sort` должен использоваться в качестве входа для программы `head`, что позволяет обойтись без явного указания оболочке создать временный файл, а потом удалить его. Набор команд, соединенных символом канала, называется конвейером и может содержать произвольное количество команд.

### Утилиты UNIX

Пользовательский интерфейс UNIX состоит не только из оболочки, но также из большого числа стандартных обслуживающих программ, называемых также утилитами. Грубо говоря, эти программы можно разделить на шесть следующих категорий:

1. Команды управления файлами и каталогами.
2. Фильтры.
3. Средства разработки программ, такие как текстовые редакторы и компиляторы.
4. Текстовые процессоры.
5. Системное администрирование.
6. Разное.

Компиляторы и программные средства включают в себя компилятор с языка C `cc` и программу `ar`, собирающую библиотечные процедуры в архивные файлы.

Еще одной важной инструментальной программой является команда `make`, используемая для сборки больших программ, исходный текст которых состоит из нескольких файлов. Как правило, некоторые из этих файлов представляют собой заголовочные файлы, содержащие определения типов, переменных, макросов и т. д. Исходные файлы обычно ссылаются на эти файлы с помощью специальной директивы компилятора `include`. Таким образом, два и более исходных файла могут совместно использовать одни и те же определения. Однако если файл заголовков изменен, необходимо найти все исходные файлы, зависящие от него, и перекомпилировать их. Задача команды `make` заключается в том, чтобы отслеживать, какой файл от какого зависит, и автоматически запускать компилятор для тех файлов, которые требуется перекомпилировать. Почти все программы в системе UNIX, кроме самых маленьких, компилируются с помощью команды `make`.

### Структура ядра

Давайте подробнее рассмотрим ядро системы. Обзор структуры ядра системы UNIX представляет собой довольно непростое дело, так как существует множество различных версий этой системы. Однако, хотя диаграмма на рисунке 8 описывает UNIX 4.4 BSD, она также применима ко многим другим версиям, возможно, с небольшими изменениями в тех или иных местах.

Системные вызовы				Аппаратные и эмулированные прерывания		
Управление терминалом	Сокеты	Именованье файла	Отображение адресов	Страничные прерывания	Обработка сигналов	Создание и завершение процессов
Обработанный	Сетевые протоколы	Файловые системы	Виртуальная память			
Дисциплины	Маршрутизация	Буферный КЭШ	Страничный КЭШ		Планирование процесса	
Символьные устройства	Драйверы сетевых устройств	Драйверы дисковых устройств			Диспетчеризация процессов	
Аппаратура						

**Рис. 8 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 устройств и драйверы 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 их, вероятно, правильнее выделить в отдельных класс, как это и было сделано на схеме. Диспетчеризация процессов производится при возникновении прерывания. При этом низкоуровневая программа останавливает выполнение работающего процесса, сохраняет его состояние в таблице процессов ядра и запускает соответствующий драйвер. Кроме того, диспетчеризация процессов производится также, 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.

В более высоких уровнях программы отличаются в каждом из четырех «столбцов» диаграммы. Слева располагаются символьные устройства. Они могут использоваться двумя способами. Некоторым программам, таким как текстовые редакторы vi и emacs, требуется каждая нажатая клавиша без какой-либо обработки. Для этого служит ввод-вывод с необработанного терминала (телетайпа). Другое программное обеспечение, например, оболочка (sh), принимает на входе уже готовую текстовую строку, позволяя пользователю редактировать ее, пока не будет нажата клавиша ENTER. Такое программное обеспечение используется вводом с терминала в обработанном виде и дисциплинами линии связи.

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 содержат 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 протоколы IP и TCP, но также иногда и некоторые дополнительные протоколы. 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 пользователя. Во многих современных системах UNIX этой фиксированной границы уже не существует, и любая страница памяти может быть схвачена для выполнения любой задачи, в зависимости от того, что требуется в данный момент.

Над буферным кэшем располагаются файловые системы. Большинство систем UNIX поддерживаются несколько файловых систем, включая быструю файловую систему System V. Все эти файловые системы совместно используют общий буферный кэш. Выше файловые системы совместно используют общий буферный кэш. Выше файловых систем помещается именование файлов, управление каталогами, управление

жесткими и символьными связями, а также другие свойства файловой системы, одинаковые для всех файловых систем.

Над страничным кэшем располагается система виртуальной памяти. В нем вся логика работы со страницами, например, алгоритм замещения страниц. Поверх него находится программа отображения файлов на виртуальную память и высокоуровневая программа управления страничными прерываниями. Эта программа решает, что нужно делать при возникновении страничного прерывания. Сначала она проверяет допустимость обращения к памяти и, если все в порядке, определяет местонахождение требуемой страницы и то, как она может быть получена.

Последний столбец имеет отношение к управлению процессами. Над диспетчером располагается планировщик процессов, выбирающий процесс, который должен быть запущен следующим. Если потоками управляет ядро, то управление потоками также помещается здесь, хотя в некоторых системах UNIX управление потоками вынесено в пространство пользователя. Над планировщиком расположена программа для обработки сигналов и отправки их в требуемом направлении, а также программа, занимающаяся созданием и завершением процессов.

Верхний уровень представляет собой [интерфейс системы](#). Слева располагается интерфейс системных вызовов. Все системные вызовы поступают сюда и направляются одному из модулей низших уровней в зависимости от природы системного вызова. Правая часть верхнего уровня представляет собой вход для аппаратных и эмулированных прерываний, включая сигналы, страничные прерывания, разнообразные исключительные ситуации процессора и прерывания ввода-вывода.

### Лекция 1.3. Процессы в системе UNIX

Единственными активными сущностями в системе UNIX являются процессы. Каждый процесс запускает одну программу и изначально получает один поток управления. Другими словами, у процесса есть один счетчик команд, указывающий на следующую исполняемую команду процессора, большинство версий UNIX позволяют процессу после того, как он запущен, создавать дополнительные потоки.

UNIX представляет собой многозадачную систему, так что несколько независимых процессов могут работать одновременно. У каждого пользователя может быть одновременно несколько активных процессов, так что в большой системе могут одновременно работать сотни и даже тысячи процессов. Действительно, в большинстве однопользовательских рабочих станций, даже когда пользователь куда-либо отлучается, работают десятки фоновых процессов, называемых демонами. Они запускаются автоматически при загрузке системы.

Типичным демоном является `cron daemon`. Он просыпается раз в минуту, проверяя, не нужно ли чего сделать. Если у него есть работа, он ее выполняет и отправляется спать дальше.

Этот демон позволяет планировать в системе UNIX активность на минуты, часы, дни и даже месяцы вперед. Например, представьте, что пользователю назначено явиться к зубному врачу в 3 часа дня в следующий вторник. Он может создать запись в базе данных демона `cron`, чтобы тот библикнул ему, скажем, в 2:30. Когда наступает назначенный день, `cron daemon` видит, что у него есть работа, и запускает в назначенное время пишущую программу в виде нового процесса.

Демон `cron` также используется для периодического запуска задач, например, ежедневной архивации диска в 4 часа ночи или напоминания забывчивым пользователям каждый год 31 октября купить новые страшненькие товары для веселого празднования Хэллоуина. Другие демоны управляют входящей и исходящей электронной почтой, очередями на принтер, проверяют, достаточно ли еще осталось свободных страниц памяти и т. д. Демоны реализуются в системе UNIX довольно просто, так как каждый из них представляет собой отдельный процесс, независимый от всех остальных процессов.

[Процессы](#) создаются в операционной системе UNIX чрезвычайно просто. Системный вызов `fork` создает точную копию исходного процесса, называемого родительским процессом. Новый процесс называется дочерним процессом. У родительского и у дочернего процессов есть свои собственные образы памяти. Если родительский процесс впоследствии изменяет какие-либо свои переменные, изменения остаются невидимыми для дочернего процесса, и наоборот.

Открытые файлы совместно используются родительским и дочерним процессами. Это значит, что, если

какой-либо файл был открыт до выполнения системного вызова `fork`, он останется открытым в обоих процессах и в дальнейшем. Изменения, произведенные с этим файлом, будут видимы каждому процессу. Такое поведение является единственно разумным, так как эти изменения будут также видны любому другому процессу, который тоже откроет этот файл.

Тот факт, что образы памяти, переменные, регистры и все остальное у родительского процесса и у дочернего идентично, приводит к небольшому затруднению: Как процессам, узнать, который из них должен исполнять родительскую программу, а который дочернюю? Эта проблема решается просто: системный вызов `fork` 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.

Процессы распознаются по своим PID-идентификаторам. Как уже говорилось выше, при создании процесса его PID выдается родителю нового процесса. 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 также выдается его родителю. Это может быть важно, так как у родительского процесса может быть много дочерних процессов. Поскольку у дочерних процессов также могут быть дочерние процессы, исходный процесс может создать целое дерево детей, внуков, правнуков и т.д.

В системе UNIX процессы могут общаться друг с другом с помощью разновидности обмена сообщениями. Можно создать канал между двумя процессами, в который один процесс может писать поток байтов, а другой процесс может его читать. Эти каналы иногда называют трубами. Синхронизация процессов достигается путем блокирования процесса при попытке прочесть данные из пустого канала. Когда данные появляются в канале, процесс разблокируется.

Процессы также могут общаться другим способом: при помощи программных прерываний. 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 (действие по умолчанию для большинства сигналов). Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуры обработки сигналов. Когда сигнал прибывает, управление внезапно передается обработчику. Когда процедура обработки сигнала завершает свою работу, управление снова возвращается в то место процесса, в котором оно находилось, 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 прадедов, братьев и сестер, а также детей (внуков и правнуков). Процесс может также послать сигнал сразу всей своей группе за один системный вызов.

## **Системные вызовы управления процессами в UNIX**

Рассмотрим теперь системные вызовы UNIX, предназначенные для управления процессами. Обсуждение системных вызовов проще всего начать с системного вызова `fork`. Этот системный вызов представляет собой единственный способ создания новых процессов в системах UNIX. Он создает точную копию оригинального процесса, включая все описатели файлов, регистры и все остальное. После выполнения системного вызова `fork` исходный процесс и его копия (родительский процесс и дочерний) идут каждый своим путем. Сразу 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 идентификатору (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. Рассмотрим работу оболочки. 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`. У этого системного вызова три параметра. В первом параметре указывается PID процесса, завершение которого ожидается. Если вместо идентификатора процесса указать число -1, то в этом случае системный вызов

ожидает 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 определяет, будет ли обращающийся к системному вызову `waitpid` 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.

Рассмотрим случай выполнения оболочкой команды

```
cp file1 file2
```

используемой для копирования файла `file1` в файл `file2`. После того как оболочка создает дочерний процесс, тот обнаруживает и исполняет файл `cp` и передает ему информацию о копируемых файлах.

Головной модуль файла `cp` (как и многие другие программы) содержит определение функции

```
main (argc, argv, envp)
```

где `argc` — счетчик слов (последовательностей символов, ограниченных пробелами) в командной строке, включая имя программы. Для вышеприведенного примера значение `argc` равно 3.

Второй параметр `argv` представляет собой указатель на массив, 1-й элемент этого массива является указателем на 1-е слово командной строки. В нашем примере элемент `argv[0]` указывает на строку «`cp`». Соответственно, элемент `argv[1]` указывает на строку «`file1`», а элемент `argv[2]` указывает на строку «`file2`».

Третий параметр процедуры `main`, `envp`, представляет собой указатель на переменные среды и является массивом, содержащим строки вида `имя = значение`, используемые для передачи программе такой информации, как тип терминала и имя рабочего каталога. В листинге 10.2 дочернему процессу переменные среды не передаются, поэтому третий параметр `envp` в данном случае равен нулю.

В качестве примера простого системного вызова рассмотрим `exit`, который процессы должны использовать, заканчивая исполнение. У него есть один параметр, статус выхода (от 0 до 255), возвращаемый родительскому процессу в переменной `status` системного вызова `waitpid`. Младший байт переменной `status` содержит статус завершения, равный 0 при нормальном завершении или коду ошибки при аварийном завершении. Например, если родительский процесс выполняет оператор

```
n = waitpid (-1, &status, 0);
```

он будет приостановлен до тех пор, пока не завершится какой-либо дочерний процесс. Если дочерний процесс завершится со, скажем, значением статуса, равным 4, в качестве параметра библиотечной процедуры `exit`, то родительский процесс получит PID дочернего процесса и значение статуса, равное 0x0400. Младший байт переменной `status` относится к сигналам, старший байт представляет собой значение, за даваемое дочерним процессом в виде параметра при обращении к системному вызову `exit`.

Если процесс уже завершил свою работу, а родительский процесс не ожидает этого события, то дочерний процесс переводится в так называемое состояние зомби, то есть приостанавливается. Когда родительский процесс наконец обращается к библиотечной процедуре `waitpid`, дочерний процесс завершается.

Обработчик сигнала может выполняться сколь угодно долго. Однако на практике обработка сигналов не занимает много времени. Когда процедура обработки сигнала завершает свою работу, она возвращается к той точке, в которой её прервали.

Системный вызов `sigaction` может также использоваться для игнорирования сигнала или чтобы восстановить действие по умолчанию, заключающееся в уничтожении процесса.

Нажатие на клавишу `DEL` не является единственным способом послать сигнал. Системный вызов `kill` позволяет процессу послать сигнал любому родственному процессу. Выбор названия для данного системного вызова (`kill` — убить, уничтожить) не особенно удачен, так как по большей части он используется процессами не для уничтожения других процессов, а, наоборот, в надежде, что этот сигнал будет перехвачен и обработан соответствующим образом.

### **Системные вызовы управления потоками**

В первой версии системы не было потоков. Это свойство было добавлено много лет спустя. Изначально применялось множество различных пакетов поддержки потоков, однако распространение этих различных пакетов привело к тому, что написать переносимую программу стало очень сложно. В конце концов, системные вызовы, используемые для управления потоков, были стандартизированы в виде части стандарта POSIX (P1003.1c).



В стандарте POSIX не указывается, должны ли потоки реализовываться в пространстве ядра или в пространстве пользователя. Преимущество потоков в пользовательском пространстве состоит в том, что они легко реализуются без необходимости изменения ядра, а переключение потоков осуществляется очень эффективно. Недостаток потоков в пространстве пользователя заключается в том, что если один из потоков заблокируется (например, на операции ввода-вывода, семафоре или страничном прерывании), все потоки процесса блокируются. Ядро полагает, что существует только один поток, и не передает управление процессу потока, пока блокировка не снимется. Таким образом, системные вызовы, определенные в стандарте P1003.1c, были тщательно отобраны так, чтобы потоки могли быть реализованы любым способом.

Поток, выполнивший свою работу и желающий прекратить свое существование, обращается к системному вызову `pthread_exit`. Поток может подождать, пока не завершится процесс, обратившись к системному вызову `pthread_join`. Если ожидаемый поток уже завершил свою работу, системный вызов `pthread_join` выполняется мгновенно. В противном случае обратившийся к нему поток блокируется.

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) мьютекс перед обращением к ресурсу и разблокируют (отпускают) его, когда ресурс им более не нужен. До тех пор, пока потоки соблюдают данный протокол, состояния состязания можно избежать. Мьютексы подобны двоичным семафорам, то есть семафорам, способным принимать только значения 0 и 1. Название мьютекс (`mutex`) образовано от английских слов `mutual exclusion`— взаимное исключение.

Мьютексы могут создаваться вызовом `pthread_mutex_init` и уничтожаться при помощи вызова `pthread_mutex_destroy`. Мьютекс может находиться в одном из двух состояний: блокирован и разблокирован. Поток может заблокировать мьютекс с помощью вызова `pthread_mutex_lock`. Если мьютекс уже заблокирован, то поток, обратившийся к этому вызову, блокируется. Когда поток, захвативший мьютекс, выполнил свою работу в критической области, он должен освободить мьютекс, обратившись к вызову `pthread_mutex_unlock`. Мьютексы предназначены для кратковременной блокировки, например, для защиты совместно используемой переменной. Они не предназначены для долговременной синхронизации, например, для ожидания, когда освободится накопитель на магнитной ленте. Для долговременной синхронизации предоставляются переменные состояния. Эти переменные создаются с помощью вызова `pthread_cond_init` и уничтожаются вызовом `pthread_cond_destroy`.

Переменные состояния используются следующим образом: один поток ждет, когда переменная примет определенное значение, а другой поток сигнализирует ему изменением этой переменной. Например, обнаружив, что нужный ему накопитель на магнитной ленте занят, поток может обратиться к вызову `pthread_cond_wait`, задав в качестве параметра адрес переменной, которую все потоки согласились связать с накопителем на магнитной ленте. Когда поток, использующий накопитель на магнитной ленте, наконец, освободит это устройство (возможно, через несколько часов), он обращается к вызову `pthread_cond_signal`, чтобы изменить переменную состояния и тем самым сообщить ожидающим потокам, что магнитофон свободен. Если ни один поток в этот момент не ждет, когда освободится накопитель на магнитной ленте, этот сигнал просто теряется. Другими словами, переменные состояния не считаются семафорами. С потоками, мьютексами и переменными состояния также определены несколько других операций.

## Реализация процессов в UNIX

Ядро поддерживает две ключевые структуры данных, относящиеся к процессам: таблицу процессов (содержит дескрипторы) и структуру пользователя (контекст процесса). Таблица процессов является резидентной. В ней содержится информация, необходимая для всех процессов, даже для тех процессов, которых в данный момент нет в памяти. Структура пользователя выгружается на диск, освобождая место в памяти, когда относящийся к ней процесс отсутствует в памяти, чтобы не тратить память на ненужную в данный момент информацию.

Информация в таблице дескрипторов процессов подразделяется на следующие категории:

1. Параметры планирования. Приоритеты процессов, процессорное время, потребленное за последний учитываемый период, количество времени, проведенное процессом в режиме ожидания. Вся эта информация используется для выбора процесса, которому будет передано управление следующим.



2. Образ памяти. Указатели на сегменты кода, данных и стека, или, если используется страничная организация памяти, указатели на соответствующие им таблицы страниц. Если сегмент кода используется совместно, то указатель ссылается на разделяемую область памяти. Когда процесса нет в памяти, то здесь также содержится информация о том, как найти части процесса на диске.

3. Сигналы. Маски, указывающие, какие сигналы игнорируются, какие перехватываются, какие временно заблокированы, а какие находятся в процессе доставки.

4. Разное. Текущее состояние процесса, события, ожидаемые процессом (если таковые есть), время до истечения интервала будильника, PID процесса, PID родительского процесса, идентификаторы пользователя и группы.

В структуре пользователя (контексте процесса) содержится информация, которая не требуется, когда процесса физически нет в памяти, и он не выполняется. Например, хотя процессу, выгруженному на диск, можно послать сигнал, выгруженный процесс не может прочитать файл. По этой причине информация о сигналах должна храниться в таблице процессов, постоянно находящейся в памяти, даже когда процесс не присутствует в памяти. С другой стороны, сведения об описателях файлов могут храниться в структуре пользователя и загружаться в память вместе с процессом.

Данные, хранящиеся в структуре пользователя, включают в себя следующие пункты:

1. Машинные регистры. Когда происходит прерывание с переключением в режим ядра, машинные регистры (включая регистры с плавающей точкой) сохраняются здесь.

2. Состояние системного вызова. Информация о текущем системном вызове, включая параметры и результаты.

3. Таблица дескрипторов файлов. Когда происходит обращение к системному вызову, работающему с файлом, дескриптор файла используется в качестве индекса в данной таблице, что позволяет найти структуру данных (i-node), соответствующую данному файлу.

4. Учетная информация. Указатель на таблицу, учитывающую процессорное время, использованное процессом в пользовательском и системном режимах. В некоторых системах здесь также ограничивается процессорное время, которое может использовать процесс, максимальный размер стека, количество страниц памяти и т. д.

5. Стек ядра. Фиксированный стек для использования процессом в режиме ядра.

Когда пользователь вводит с терминала команду, например, `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`. Эти действия показаны на рисунке 9

Механизм создания нового процесса довольно прост. Для дочернего процесса создается новая ячейка в таблице процессов, которая заполняется по большей мере из соответствующей ячейки родительского процесса. Дочерний процесс получает PID, затем настраивается его карта памяти. Кроме того, дочернему процессу предоставляется совместный доступ к файлам родительского процесса. Затем настраиваются регистры дочернего процесса, после чего он готов к запуску.

В принципе, следует создать полную копию адресного пространства, так как семантика системного вызова `fork` говорит, что никакая область памяти не используется совместно родительским и дочерним процессами. Однако копирование памяти является дорогим удовольствием, поэтому все системы UNIX слегка жульничают. Они выделяют дочернему процессу новые таблицы страниц, но эти таблицы указывают на страницы родительского процесса, помеченные как доступные только для чтения. Когда дочерний процесс пытается писать в такую страницу, происходит прерывание. При этом ядро выделяет дочернему процессу новую копию этой страницы, к которой этот процесс получает также и доступ записи. Таким образом, копируются только те страницы, в которые дочерний процесс пишет новые данные. Такой механизм называется копированием при записи. При этом сохраняется память, так как страницы с кодом не копируются.

После того как дочерний процесс начинает работу, его программа (копия оболочки) выполняет системный вызов `exec`, задавая имя команды в качестве параметра. При этом ядро находит и проверяет используемый файл, копирует в ядро аргументы и строки окружения, а также освобождает старое адресное пространство и его таблицы страниц.

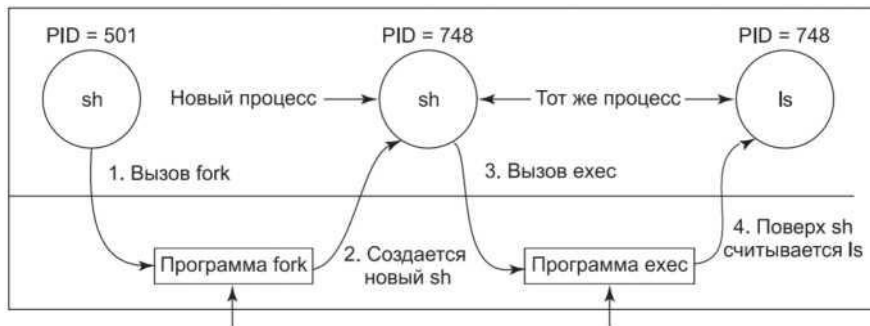


Рис. 9 Ввод с терминала команды ls

После этого следует создать и заполнить новое адресное пространство. Если системой поддерживается отображение файлов на адресное пространство памяти, как, например, в System V, BSD и в большинстве других версий UNIX, то таблицы страниц настраиваются следующим образом: в них указывается, что страниц в памяти нет, кроме, возможно, одной страницы со стеком, а содержимое адресного пространства может подгружаться из исполняемого файла на диске. Когда новый процесс начинает работу, он немедленно вызывает страничное прерывание, в результате которого первая страница программы подгружается с диска. Таким образом, ничего не нужно загружать заранее, что позволяет быстро запускать программы, а в память загружать только те страницы, которые действительно нужны программам. Наконец, в стек копируются аргументы и строки окружения, сигналы сбрасываются, а все регистры устанавливаются на ноль. С этого момента новая команда начинает исполнение.

### Потоки в UNIX

Реализация потоков зависит от того, поддерживаются, они ядром или нет. Если потоки ядром не поддерживаются, как, например, в 4BSD, реализация потоков целиком осуществляется в библиотеке, загружающейся в пространство пользователя. Если ядро поддерживает потоки, как в системах SystemV и Solaris, то у ядра есть чем заняться. Сделаем несколько замечаний по поводу реализации потоков в ядре в системе UNIX.

Основная проблема реализации потоков заключается в поддержке корректной традиционной семантики UNIX. Рассмотрим сначала системный вызов `fork`. Предположим, что процесс с несколькими потоками (реализуемыми в ядре) выполняет системный вызов `fork`. Следует ли при этом в новом процессе создать все потоки оригинального процесса? Предположим, что мы ответили на этот вопрос утвердительно. Допустим также, что один из потоков был блокирован, ожидая ввода с клавиатуры. Должна ли в этом случае копия этого потока также быть блокирована ожиданием ввода с клавиатуры? Если да, то какому потоку достанется следующая, набранная на клавиатуре строка? Если нет, то что должен делать этот поток в новом процессе? Проблема касается и других аспектов. В однопоточном процессе такой проблемы не возникает, так как единственный поток не может быть блокирован при обращении к системному вызову `fork`. Теперь рассмотрим случай, при котором в дочернем процессе другие потоки не создаются. Предположим, что один из не копируемых потоков удерживает мьютекс, который пытается получить единственный созданный новый поток после выполнения системного вызова `fork`. В этом случае мьютекс в новом процессе некому освободить и единственный новый поток будет «висеть». Существует еще множество подобных проблем. И простого решения у этих проблем нет.

Файловый ввод-вывод представляет собой еще одну проблемную область. Предположим, что один поток блокирован при чтении из файла, а другой поток закрывает файл или обращается к системному вызову, чтобы изменить текущий указатель файла. Что произойдет в результате этих действий? Кто знает?

### Потоки в системе Linux

Операционная система Linux поддерживает потоки в ядре довольно интересным способом, с которым следует познакомиться. В основе реализации системы Linux лежат идеи из системы 4.4BSD, но в 4.4BSD потоки на уровне ядра реализованы не были, так как у университета Калифорнии в Беркли кончились деньги прежде, чем библиотеки языка C могли быть переписаны так, чтобы решить описанные выше проблемы.

Сердцем реализации потоков в системе Linux является новый системный вызов `clone`, отсутствующий во всех остальных версиях системы UNIX. Формат обращения к нему выглядит следующим образом:

```
pid = clone (function, stack_ptr, sharing_flags, arg);
```

Системный вызов `clone` создает новый поток либо в текущем процессе, либо в новом процессе, в зависимости от флага `sharing_flags`. Если новый поток находится в текущем процессе, он совместно использует с остальными потоками адресное пространство и любое изменение каждого байта в адресном пространстве любым потоком тут же становится видимым всем остальным потокам процесса. С другой стороны, если адресное пространство не используется совместно, тогда новый поток получает точную копию адресного пространства, но последующие изменения в памяти уже не видны остальным потокам. Таким образом, здесь используется та же семантика, что и у системного вызова `fork`.

В обоих случаях новый поток начинает выполнение функции `function` с аргументом `arg` в качестве параметра. Также в обоих случаях новый поток получает свой собственный стек, при этом указатель стека инициализируется параметром `stack_ptr`.

Параметр `sharing_flags` представляет собой битовый массив, обеспечивающий существенно более тонкую настройку совместного использования, нежели используется в традиционных системах UNIX. У этого флага определены пять битов, показаны в таблице 4.1. Каждый бит управляет одним из аспектов совместного использования, и каждый из битов может быть установлен независимо от остальных битов. Бит `CLONE_VM` определяет, будет ли виртуальная память (то есть адресное пространство) использоваться совместно со старыми потоками или будет копироваться. Если этот бит установлен, новый поток просто помещается вместе со старыми потоками, так что системный вызов `clone` создает новый поток в существующем процессе. Если бит сброшен, новый поток получает свое собственное адресное пространство. Это означает, что результат команды процессора `STORE` не виден остальным потокам. Такое поведение подобно поведению системного вызова `fork`. Создание нового адресного пространства равнозначно определению нового процесса.

Таблица 4.1

Флаг	Значение в 1	Значение в 0
<code>CLONE_VM</code>	Создать новый поток	Создать новый процесс
<code>CLONE_FS</code>	Общий рабочий каталог, <code>root</code> и <code>umask</code>	Не использовать их совместно
<code>CLONE_FILES</code>	Общие дескрипторы файлов	Копировать дескрипторы файлов
<code>CLONE_SIGHAND</code>	Общая таблица обработчика сигналов	Копировать таблицу
<code>CLONE_PID</code>	Новый поток получает старый PID	Новый поток получает новый PID

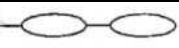


Бит `CLONE_FS` управляет совместным использованием рабочего каталога и каталога `root`, а также флага `umask`. Даже если у нового потока свое собственное адресное пространство, при установленном бите `CLONE_FS` старый и новый потоки будут совместно использовать рабочие каталоги. Это означает, что обращение к системному вызову `chdir` одним из потоков изменит рабочий каталог другого потока, несмотря на то что у другого потока есть свое собственное адресное пространство. В системе UNIX обращение к системному вызову `chdir` потоком всегда изменяет рабочий каталог всех остальных потоков этого процесса, но никогда не меняет рабочих каталогов других процессов. Таким образом, этот бит обеспечивает разновидность совместного использования, недоступную в UNIX.

Бит `CLONE_FILES` аналогичен биту `CLONE_FS`. Если он установлен, то новый поток пользуется теми же дескрипторами файлов, что и старые потоки. Таким образом, обращение к системному вызову `lseek` одним потоком становится видимым для других потоков, что также обычно справедливо для потоков одного процесса, но не для потоков различных процессов. Аналогично бит `CLONE_SIGHAND` разрешает или запрещает совместное использование таблицы обработчиков сигналов старым и новым потоками. Если таблица общая даже у потоков в различных адресных пространствах, тогда изменение обработчика в одном потоке повлияет и на другой поток. Наконец, бит `CLONE_PID` указывает, получит ли новый поток свой собственный PID или будет использовать PID своего родительского потока. Это свойство нужно при загрузке системы. Процессам пользователя не разрешается использовать этот бит.

## Планирование в системе UNIX

Давайте теперь изучим алгоритм планирования системы UNIX. Поскольку UNIX всегда была многозадачной системой, ее алгоритм планирования с самого начала развития системы разрабатывался так, чтобы обеспечить хорошую реакцию в интерактивных процессах. У этого алгоритма два уровня. Низкоуровневый алгоритм выбирает следующий процесс из набора процессов в памяти и готовых к работе. Высокоуровневый алгоритм перемещает процессы из памяти на диск и обратно, что предоставляет всем процессам возможность попасть в память и быть запущенными. У каждой версии UNIX свой слегка отличающийся низкоуровневый алгоритм планирования, но у большинства этих алгоритмов есть много общих черт, которые мы разберем. В низкоуровневом алгоритме используется несколько очередей. С каждой очередью связан диапазон непересекающихся значений приоритетов. Процессы, выполняющиеся в режиме пользователя (нижняя часть айсберга), имеют положительные значения приоритетов. У процессов, выполняющихся в режиме ядра (обращающихся к системным вызовам), значения приоритетов отрицательные. Отрицательные значения приоритетов считаются наивысшими, а положительные — наоборот, минимальными, как показано на рисунке 4.12. В очередях располагаются только процессы, находящиеся в памяти и готовые к работе.

Максимальный  
приоритет

1_	
Ожидание дискового ввода-вывода	—C
Ожидание дискового буфера	Ожидание
Ожидание терминального ввода	
Ожидание	
терминального вывода	
Ожидание завершения дочернего процесса	-----
Приоритет пользователя 0	
Приоритет Пользователя 1	 Ожидание в режиме
Приоритет пользователя 2	
Приоритет пользователя 3	
	

На основе сосчитанного нового приоритета каждый процесс прикрепляется к соответствующей очереди на рис.10.4. Для получения номера очереди приоритет, как правило, делится на некую константу. Изучим вкратце каждый из трех компонентов этой формулы приоритета.

Параметр `CPU_usage` (использование центрального процессора) представляет собой среднее значение тиков таймера в секунду, которые процесс работал в течение последних нескольких секунд. При каждом тике (прерывании) таймера счетчик использования центрального процессора в таблице процессов увеличивается, на единицу. Этот счетчик в конце концов добавляется к приоритету процесса, увеличивая тем самым числовое значение его приоритета (что соответствует более низкому приоритету), в результате чего процесс попадает в менее приоритетную очередь.

Однако операционная система UNIX не наказывает процесс за использование центрального процессора навечно, и величина `CPU_usage` со временем уменьшается. В различных версиях UNIX это уменьшение выполняется по-разному. Один из способов состоит в том, что к `CPU_usage` прибавляется полученное число тиков, после чего сумма делится на два. Такой алгоритм учитывает самое последнее значение времени использования ЦП с весовым коэффициентом  $1/2$ , предшествующее ему — с весовым коэффициентом  $1/4$  и т. д. Алгоритм взвешивания очень быстр, так как состоит из всего одной операции сложения и одного сдвига, но также применяются и другие схемы взвешивания.

Когда процесс эмулирует прерывание для выполнения системного вызова в ядре, он, вероятно, должен быть заблокирован, пока системный вызов не будет выполнен. Процесс может обратиться к системному вызову `waitpid`, ожидая, пока один из его дочерних процессов не закончит работу. Он может также ожидать ввода с терминала или завершения дисковой операции ввода-вывода и т.д. Когда процесс блокируется, он удаляется из структуры очереди, пока этот процесс снова не будет готов работать.

Однако, когда происходит событие, которого ждал процесс, он снова помещается в очередь с отрицательным значением приоритета.

Выбор очереди определяется событием, которого ждал процесс. На рис. 4.15 дисковый ввод-вывод показан как событие с наивысшим приоритетом, так что процесс, только что прочитавший или записавший блок диска, вероятно, получит центральный процессор в течение 100 мс. Отрицательные значения приоритета для дискового ввода-вывода, терминального ввода-вывода и т. д. жестко прошиты в операционной системе и могут быть изменены только путем перекомпиляции самой системы. Эти (отрицательные) значения представлены в приведенной выше формуле слагаемым *base* (база), и их величина достаточно отличается от нуля, чтобы перезапущенный процесс наверняка попадал в другую очередь.

### **Планирование в системе Linux**

Планирование представляет собой одну из немногих областей, в которых операционная система Linux использует алгоритм, отличный от применяющегося в UNIX. Мы только что рассмотрели алгоритм [планирования системы UNIX](#), теперь познакомимся с алгоритмом планирования системы Linux. Начнем с того, что потоки в системе Linux реализованы в ядре, поэтому планирование основано на потоках, а не на процессах. В операционной системе Linux алгоритмом планирования различаются три класса потоков:

1. Потоки реального времени, обслуживаемые по алгоритму FIFO.
2. Потоки реального времени, обслуживаемые в порядке циклической очереди.
3. Потоки разделения времени

Потоки реального времени, обслуживаемые по алгоритму FIFO, имеют наивысшие приоритеты и не могут прерываться другими потоками, за исключением такого же потока реального времени FIFO, перешедшего в состояние готовности. Потоки реального времени, обслуживаемые в порядке циклической очереди, представляют собой то же самое, что и потоки реального времени FIFO, но с тем отличием, что они могут прерываться таймером. Находящиеся в состоянии готовности потоки реального времени, обслуживаемые в порядке циклической очереди, выполняются в течение определенного кванта времени, после чего поток помещается в конец своей очереди. Ни один из этих классов

на самом деле не является классом реального времени.

Здесь нельзя задать предельный срок выполнения задания и предоставить гарантий его выполнения. Эти классы просто имеют более высокий приоритет, чем у потоков стандартного класса разделения времени. Причина, по которой в системе Linux эти классы называются классами реального времени, в том, что операционная система Linux совместима со стандартом P1003.4 (расширение «реального времени» для UNIX), в котором они носят эти имена.

У каждого потока есть приоритет планирования. Значение приоритета по умолчанию равно 20, но оно может быть изменено при помощи системного вызова `nice (value)`, вычитающего значения `value` из 20. Поскольку `value` должно находиться в диапазоне от -20 до +19, приоритеты всегда попадают в промежуток от 1 до 40. Цель алгоритма планирования состоит в том, чтобы обеспечить грубое пропорциональное соответствие качества обслуживания приоритету, то есть чем выше приоритет, тем меньше должно быть время отклика и тем большая доля процессорного времени достанется процессу.

Помимо приоритета с каждым процессом связан квант времени, то есть количество тиков таймера, в течение которых процесс может выполняться. По умолчанию системные часы тикают с частотой 100 Гц, так что каждый тик равен 10 мс. Этот интервал в системе Linux называют «джиффи» (`jiffy` — мгновение, миг, момент). Планировщик использует приоритет и квант следующим образом. Сначала он вычисляет для каждого готового процесса величину называемую в системе Linux «добродетелью» (`goodness`) по следующему алгоритму:

```
if (class == RT_CLASS_REALTIME) goodness = 1000 +
priority:
if (class == RT_CLASS_TIMESHARING && quantum > 0) goodness
= quantum + priority:
if (class == RT_CLASS_TIMESHARING && quantum == 0)
goodness = 0:
```

Для обоих классов реального времени выполняется первое условие. Все, что дает пометка процесса, как процесса реального времени, — это



гарантия, что этот процесс получит более высокое значение goodness, чем все процессы разделения времени. У алгоритма есть еще одно дополнительное свойство: если у процесса, который запускался последним, осталось неиспользованное процессорное время, он получает бонус, позволяющий выиграть в спорных ситуациях. Идея состоит в том, что при прочих равных условиях более эффективным представляется запустить предыдущий процесс, так как его страницы и кэш с большой вероятностью еще находятся на своих местах.

В остальном алгоритм планирования очень прост: когда нужно принять решение, выбирается поток с максимальным значением «добродетели». Во время работы процесса его квант (переменная quantum) уменьшается на единицу при каждом тике. Центральный процессор отнимается у потока при выполнении одного из следующих условий:

1. Квант потока уменьшился до 0.
2. Поток блокируется на операции ввода-вывода, семафоре и т. д.
3. В состояние готовности перешел ранее заблокированный поток с более высокой «добродетелью».

Так как кванты постоянно уменьшаются, рано или поздно у всех работающих потоков квант станет нулевым. Однако у потока, заблокированного вводом-выводом, может остаться некая ненулевая величина кванта. В этот момент планировщик пересчитывает значения квантов для всех потоков, как готовых, так и заблокированных, по следующей формуле:

$$\text{quantum} = (\text{quantum}/2) + \text{priority}$$

где квант измеряется в «джиффи», то есть в тиках. Поток, ограниченный производительностью центрального процессора, как правило, быстро истратит свой квант, и при пересчёте кванта его новое значение будет равно приоритету потока. В то же время у потока, ограниченного вводом-выводом, может остаться значительное количество неистраченного процессорного времени, поэтому в следующий раз значение его нового кванта будет больше, чем у потока, ограниченного производительностью процессора. Если системный вызов `nice` не используется, приоритет потока будет равен 20 и квант станет равным 20 тикам или 200 мс. С другой стороны, у потока, сильно

ограниченного вводом-выводом, к моменту пересчета квантов может остаться квант, равный 20. Поэтому если его приоритет также равен 20, то новое значение его кванта будет равно  $20/2 + 20 = 30$  тиков. Если он опять заблокируется вводом-выводом, прежде чем успеет истратить один тик, то в следующий раз его квант будет равен  $30/2 + 20 = 35$  тиков. Эта величина стремится снизу к удвоенному значению приоритета. В результате применения данного алгоритма потоки, ограниченные вводом-выводом, получают большие кванты времени и, следовательно, считаются более «добродетельными», чем потоки, ограниченные производительностью процессора. Таким образом, потоки, ограниченные вводом-выводом, получают преимущество при [планировании](#).

### **Загрузка UNIX**

Точные детали процесса загрузки операционной системы UNIX варьируются от системы к системе. Рассмотрим, как загружается 4.4BSD, но в своей основе это описание применимо и для других версий. Когда компьютер включается, в память считывается и исполняется первый сектор (MBR-главная загрузочная запись) загружаемого диска. Этот сектор содержит небольшую (512- байтовую) программу, позволяющую найти и запустить boot (загрузчик ОС) с загрузочного устройства, как правило, с IDE или SCSI-диска. Программа boot сначала копирует саму себя в фиксированный адрес памяти в старших адресах, чтобы освободить нижнюю память для операционной системы. Загрузившись, программа boot считывает корневой каталог с загрузочного устройства. Чтобы сделать это, она должна понимать формат файловой системы и каталога. Затем она считывает ядро операционной системы и передает ему управление. На этом программа boot завершает свою работу, после чего уже работает ядро системы.

Начальная программа ядра написана на ассемблере и является в значительной мере машинно-зависимой. Как правило, эта программа устанавливает указатель стека, определяют тип центрального процессора, вычисляет количество имеющегося в наличии ОЗУ, разрешает работу диспетчеру памяти и, наконец, вызывает процедуру main, написанную на С, чтобы запустить основную часть операционной системы.

Программа на языке С также должна проделать значительную работу

по инициализации, но эта инициализация скорее логическая, нежели физическая. Она начинается с того, что выделяет память под буфер сообщений, что должно помочь решению проблем с загрузкой системы. По мере выполнения инициализации в этот буфер записываются сообщения, информирующие о том, что происходит в системе. В случае неудачной загрузки их можно выудить оттуда с помощью специальной программы диагностики. Этот буфер подобен черному ящику, который обычно пытаются найти на месте крушения самолета.

Затем выделяется память для структур данных ядра. Большинство этих структур имеют фиксированный размер, но размер некоторых из них, например, размер буферного кэша и некоторых структур таблиц управления страницами памяти, зависит от доступного объема оперативной памяти.

Затем операционная система начинает определение конфигурации компьютера. Операционная система считывает файлы конфигурации, в которых сообщается, какие типы устройств ввода-вывода могут присутствовать, и проверяет, какие из устройств действительно присутствуют. Если проверяемое устройство отвечает, оно добавляется к таблице подключенных устройств. Если устройство не отвечает, оно считается отсутствующим и в дальнейшем игнорируется.

Как только список устройств определен, операционная система должна найти драйверы устройств. В этом месте версии UNIX несколько различаются между собой. В частности, система 4.4BSD не могла динамически загружать драйверы устройств, поэтому любое устройство ввода-вывода, чей драйвер не был статически скомпонован с ядром, не мог использоваться. Некоторые другие версии UNIX, как, например, Linux, напротив, могут динамически загружать драйверы (как это могут делать все eерсуuWindows).

Итак, когда загружающаяся операционная система определила конфигурацию аппаратного обеспечения, она должна аккуратно загрузить процесс 0, установить его стек и запустить этот процесс. Процесс 0 продолжает инициализацию, выполняя такие задачи, как программирование таймера реального времени, монтирование корневой файловой системы и создание процесса 1 (init) и страничного демона (процесс 2).

Процесс `init` проверяет свои флаги, в зависимости от которых он запускает операционную систему либо в однопользовательском, либо в многопользовательском режиме. В первом случае он создает процесс, выполняющий оболочку, и ждет, когда тот завершит свою работу. Во втором

случае процесс `init` создает процесс, исполняющий сценарий оболочки инициализации системы `/etc/rc`, который может выполнять проверку непротиворечивости файловой системы, монтировать дополнительные файловые системы, запускать демонов и т. д. Затем он считывает файл `/etc/tty`, в котором перечисляются терминалы и некоторые их свойства. Для каждого разрешенного терминала он создает копию самого себя, которая затем исполняет программу `getty`.

Программа `getty` устанавливает для каждой линии (некоторые из них могут быть, например, модемами) скорость линии, после чего выводит на терминале приглашение к входу в систему:

`login:`

После этого программа `getty` пытается прочитать имя пользователя, введенное с клавиатуры. Когда пользователь садится за терминал и вводит свое имя, программа `getty` завершает свою работу выполнением программы регистрации `/bin/login`. После этого 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` просто еще раз спрашивает имя пользователя. Этот механизм проиллюстрирован на рисунке 4.13 для системы с тремя терминалами.

На рисунке 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 `ср`. Процесс оболочки блокируется в ожидании завершения дочернего процесса, после чего оболочка снова напечатает приглашение к вводу и будет ждать ввода с клавиатуры следующей команды. Если бы пользователь на терминале 2 вместо `ср` ввел `сс`, то запустилась бы главная программа компилятора `С`, который, в свою очередь, запустил бы несколько дочерних процессов для выполнения различных проходов компилятора.

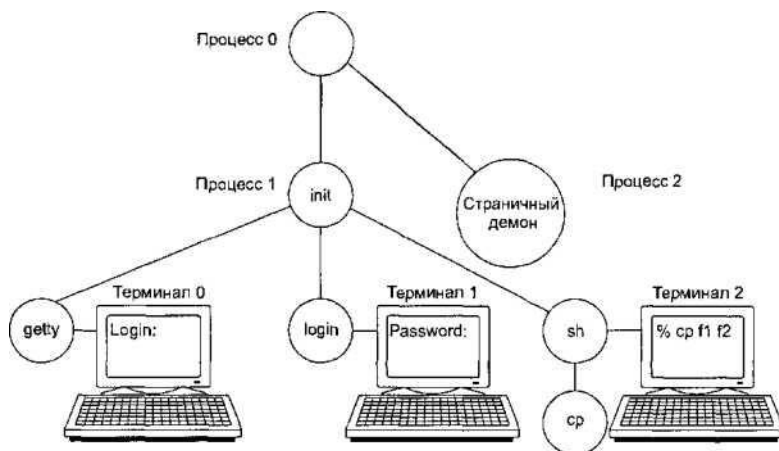


Рис. 4.13

## Лекция 1.4. Управление памятью в UNIX

У каждого процесса в системе UNIX есть адресное пространство, состоящее из трех сегментов: текста, данных и стека. Текстовый (программный) сегмент содержит машинные команды, образующие исполняемый код программы. Он создается компилятором и ассемблером при трансляции программы, написанной на языке высокого уровня (например, C или C++) в машинный код. Как правило, текстовый сегмент разрешен только для чтения. Самомодифицирующиеся программы вышли из моды примерно в 1950 году, так как их было слишком сложно понимать и отлаживать. Таким образом, текстовый сегмент не изменяется ни в размерах, ни по своему содержанию.

Сегмент данных содержит переменные, строки, массивы и другие данные программы. Он состоит из двух частей: инициализированных данных и неинициализированных данных. По историческим причинам вторая часть называется BSS (Bulk Storage System — запоминающее устройство большой емкости, массовое ЗУ). Инициализированная часть сегмента данных содержит переменные и константы компилятора, значения которых должны быть заданы при запуске программы.

Например, на языке C можно объявить символьную строку и в то же время задать ее значение, то есть проинициализировать ее. Когда программа запускается, она предполагает, что в этой строке уже содержится некий осмысленный текст. Чтобы реализовать это, компилятор назначает строке определенное место в адресном пространстве и гарантирует, что в момент запуска программы по этому адресу будет располагаться соответствующая строка. С точки зрения операционной системы, инициализированные данные не отличаются от текста программы — и тот и другой сегменты содержат сформированные компилятором последовательности битов, загружаемые в память при запуске программы.

Однако из экономии места на диске этого не делается. Файл содержит только те переменные, начальные значения которых явно заданы. Вместо неинициализированных переменных компилятор помещает в исполняемый файл просто одно слово, содержащее размер

области неинициализированных данных в байтах. При запуске программы операционная система считывает это слово, выделяет нужное число байтов и обнуляет их.

Когда программа запускается, ее стек не пуст. Напротив, 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. Например, когда вводится команда

```
cp src dest
```

запускается программа `cp` со строкой «`cp src dest`» в стеке, что позволяет ей определить имена файлов, с которыми ей предстоит работать. Строка представляется в виде массива указателей на отдельные аргументы командной строки, что облегчает ее обработку.

Когда два пользователя запускают одну и ту же программу, например, текстовый редактор, в памяти можно хранить две копии программы редактора. Однако такой подход является неэффективным. Вместо этого большинством систем UNIX поддерживаются текстовые сегменты совместного использования.

На некоторых компьютерах аппаратное обеспечение поддерживает отдельные адресные пространства для команд и для данных. Если такая возможность есть, система UNIX может ею воспользоваться. Например, на компьютере с 32-разрядными адресами при возможности использования отдельных адресных пространств можно получить 232 бит адресного пространства для команд и еще 232 бит (Точнее, 232 байт, что равно 4 Гбайт, так как к отдельным битам процессоры, как правило, адресуются внутри байта или слова) адресного пространства для данных. Передача управления по адресу 0 будет восприниматься как передача управления по адресу 0 в текстовом пространстве, тогда как при обращении к данным по адресу 0 будет использоваться адрес 0 в пространстве данных. Таким образом, это свойство удваивает доступное адресное пространство.

Многими версиями UNIX поддерживается отображение файлов на адресное пространство памяти. Это свойство позволяет отображать файл на часть адресного пространства процесса, так чтобы можно было читать из файла и писать в файл, как если бы это был массив, хранящийся в памяти. Отображение файла на адресное пространство

памяти делает произвольный доступ к нему существенно более легким, нежели при использовании системных вызовов, таких как read и write. Совместный доступ к библиотекам предоставляется именно при помощи этого механизма. На рисунке 4.14 показан файл, одновременно отображенный на адресные пространства двух процессов по различным виртуальным адресам.

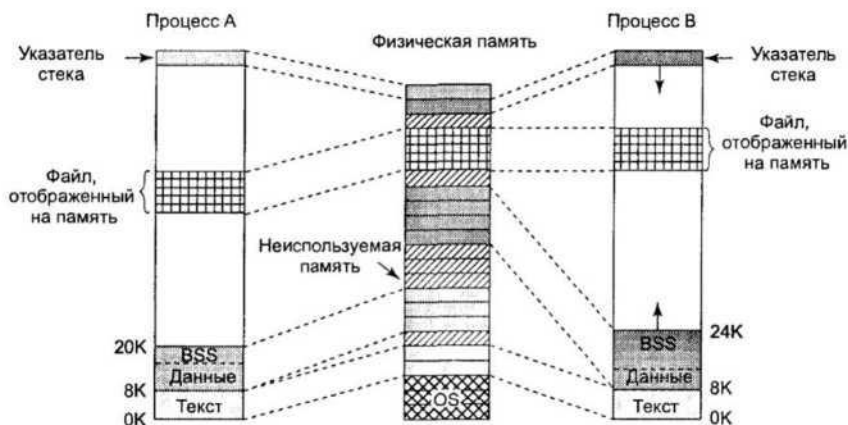


Рис. 4.14

Дополнительное 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. Таким образом, отображение на адресное пространство памяти временного файла (который будет удален после завершения работы процессов) представляет собой механизм реализации общей памяти для нескольких процессов, причем у такого механизма будет высокая пропускная способность. В предельном случае два или более процессов могут отобразить на память файл, покрывающий все адресное пространство, получая, таким образом, форму совместного использования памяти, что-то среднее между процессами и потоками. В этом случае, как и у потоков, все адресное пространство используется совместно, но каждый процесс может управлять собственными файлами и сигналами, что отличает этот вариант от потоков. Однако на практике



такое никогда не применяется.

## **Системные вызовы управления памятью в UNIX**

Стандартом POSIX системные вызовы для управления памятью не определяются. Эту область посчитали слишком машинно-зависимой, чтобы ее стандартизировать. Вместо этого просто сделали вид, что проблемы не существует, и заявили, что программы, которым требуется динамическое управление памятью, могут использовать библиотечную процедуру `malloc` (определенную стандартом ANSI C). Таким образом, вопрос реализации процедуры `malloc` был вынесен за пределы рассмотрения стандарта POSIX. В некоторых кругах такой подход был расценен как перекладывание бремени решения проблемы на чужие плечи.

Системные вызовы `mmap` и `munmap` управляют отображением файлов на адресное пространство памяти. Первый параметр системного вызова `mmap`, `addr`, указывает адрес, по которому будет отображаться файл (или его часть). Он должен быть кратен размеру страницы. Если этот параметр равен 0, тогда операционная система определяет этот адрес сама и возвращает его в `a`. Второй параметр, `len`, задает количество отображаемых байтов. Он также должен быть кратен размеру страницы. Третий параметр, `prot`, задает режим защиты для отображаемого файла. Файл может быть помечен как доступный для чтения, записи, исполнения или любой комбинации этих трех битов. Четвертый параметр, `flags`, определяет, является ли отображаемый файл приватным или доступным для совместного использования, а также содержит ли параметр `addr` жесткое требование или это всего лишь намек. Пятый параметр, `fd`, представляет собой дескриптор отображаемого файла. Отображаться могут только открытые файлы. Наконец, параметр `offset` сообщает, с какого места должен отображаться файл. Файл может быть отображен, начиная с любого байта.

Второй системный вызов, `munmap`, отменяет отображения файла на память. Если отменяется отображение только части файла, то остальная часть файла продолжает отображаться на память.

## **Реализация управления памятью в UNIX**

До версии 3BSD большинство систем UNIX основывались на свопинге (подкачке), работавшем следующим образом. Когда загружалось больше процессов, чем могло поместиться в памяти,

некоторые из них выгружались на диск. Выгружаемый процесс всегда выгружался на диск целиком (исключение представляли только совместно используемые текстовые сегменты). Таким образом, процесс мог быть либо в памяти, либо на диске.

### **Свопинг**

Перемещением данных между памятью и диском управлял верхний уровень двухуровневого планировщика, называвшийся свопером (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 бывало необходимо удалить из памяти другой процесс, чтобы освободить место для запускаемого процесса.

Выбирая жертву, свопер сначала рассматривал заблокированные (например, ожиданием ввода с терминала) процессы. Лучше удалить из памяти процесс, который не может работать, чем работоспособный процесс. Если такие процессы находились, из них выбирался процесс с наивысшим значением суммы приоритета и времени пребывания в памяти.

Каждые несколько секунд свопер исследовал список выгруженных процессов, проверяя, не готов ли какой-либо из этих процессов к работе. Если процессы в состоянии готовности обнаруживались, из них выбирался процесс, дольше всех находящийся на диске. Затем свопер проверял, будет ли это легкий свопинг или тяжелый. Легким свопингом считался тот, для которого не требовалось дополнительное высвобождение памяти. При этом нужно было всего лишь загрузить выгруженный на диск процесс. Тяжелым свопингом назывался свопинг, при котором для загрузки в память выгруженного на диск процесса из нее требовалось удалить один или несколько других процессов.

Затем весь этот алгоритм повторялся до тех пор, пока не выполнялось одно из следующих двух условий: (1) на диске не оставалось процессов, готовых к работе, или (2) в памяти не оставалось

места для новых процессов. Чтобы не терять большую часть производительности системы на свопинг, ни один процесс не выгружался на диск, если он пробыл в памяти менее 2 с.

Свободное место в памяти и на устройстве подкачки учитывалось при помощи связанного списка свободных пространств. Когда требовалось свободное пространство в памяти или на диске, из списка выбиралось первое подходящее свободное пространство. После этого в список возвращался остаток от свободного пространства.

### **Постраничная подкачка в системе UNIX**

Все версии операционной системы UNIX для компьютеров PDP-11 и Interdata, а также начальная версия для машины VAX были основаны на [свопинге](#), о котором только что рассказывалось. Однако, начиная с версии 3BSD, университет в Беркли добавил к системе страничную подкачку, чтобы предоставить возможность работать с программами самых больших размеров. Практически во всех версиях системы UNIX теперь есть страничная подкачка по требованию, появившаяся впервые в версии 3BSD. Ниже мы рассмотрим строение версии 4BSD, но SystemV основана на 4BSD и во многом схожа с нею.

Идея, лежащая в основе страничной подкачки в системе 4BSD, проста: чтобы работать, процессу не нужно целиком находиться в памяти. Все, что в действительности требуется, — это структура пользователя и таблицы страниц. Если они загружены, процесс считается находящимся в памяти и может быть запущен планировщиком. Страницы с сегментами текста, данных и стека загружаются в память динамически, по мере обращения к ним.

В системе Berkeley UNIX не используется модель рабочего набора или любая другая форма опережающей подкачки страниц, так как для этого требуется знать, какие страницы используются в данный момент, а какие нет. Поскольку в машине VAX не было битов обращений к памяти, эту информацию было получить непросто (хотя это можно было поддерживать программно за счет значительных дополнительных накладных расходов).

Страничная подкачка реализуется частично ядром и частично новым процессом, называемым страничным демоном. Страничный демон — это процесс 2 (процесс 0 — это свопер, а процесс 1 — init, как показано на рис. 10.5). Как и все демоны, страничный демон периодически

запускается и смотрит, есть ли для него работа. Если он обнаруживает, что количество страниц в списке свободных страниц слишком мало, страничный демон инициирует действия по освобождению дополнительных страниц.

При организации B4BSD память делится на три части. Первые две части, ядро операционной системы и карта памяти, фиксированы в физической памяти (то есть никогда не выгружаются). Остальная память компьютера 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 страничного блока в карте памяти есть запись фиксированной длины. У каждой страницы в памяти есть фиксированное место хранения на диске, в которое она помещается, когда выгружается из памяти. Еще три поля содержат ссылку на запись в таблице процессов, тип хранящегося в странице сегмента и смещение в сегменте процесса. Последнее поле содержит некоторые флаги, нужные для алгоритма страничной подкачки.

При запуске процесс может вызвать страничное прерывание, если одной или нескольких его страниц не окажется в памяти. При страничном прерывании операционная система берет первый страничный блок из списка свободных страниц, 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.

### **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 количество свободных страничных блоков не станет равно `lotsfree`. Если же количество свободных страничных блоков больше или равно `lotsfree`, тогда страничный демон, ничего не предпринимая, отправляется спать дальше. Если у компьютера много памяти и мало активных процессов,

страничный демон спит практически все время.

Страничный демон использует модифицированную версию алгоритма часов. Это глобальный алгоритм, то есть при удалении страницы он не учитывает, чья это страница. Таким образом, количество страниц, выделяемых каждому процессу, меняется со временем.

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 останется 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.

На машинах типа VAX, у которых не было битов использования, когда стрелка часов указывала на страничный блок на первом проходе, сбрасывался программный бит, а страница помечалась в таблице страниц как недействительная. При следующем доступе к странице происходило страничное прерывание, что позволяло операционной системе установить программный бит использования. Эффект достигался тот же самый, что и при использовании аппаратного бита использования, но реализация была значительно более сложной и медленной. Таким образом, программное обеспечение расплачивалось за недостаточно развитую схему аппаратного обеспечения.

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 (см. рис. 10.8). 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. После чего перемещает обе стрелки. Если две стрелки находятся близко друг от друга, то только у очень активно используемых страниц появляется шанс, что к ним будет обращение между проходами двух стрелок. Если же стрелки разнесены на 359 градусов (то есть задняя стрелка находится

слегка впереди передней), мы снова получаем исходный алгоритм часов. При каждом запуске страничного демона стрелки проходят не полный оборот, а столько, сколько необходимо, чтобы количество страниц в списке свободных страниц было не менее `lotsfree`.

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 бездействовавший в течение максимального срока и выгружается на диск. Если таких процессов нет, намечаются четыре самых больших процесса, из которых выбирается тот, который находился 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 `pipe` (если такое обращение было), и от того, как долго этот процесс спал, прежде чем был выгружен на диск. Эта функция обычно взвешивается так, чтобы загружать в память процесс, дольше всех находящийся в выгруженном состоянии, если только он не крайне большой. Теория утверждает, что загружать большие процессы дорого, потому их не следует перемещать туда-сюда слишком часто. Загрузка процесса производится только при условии наличия достаточного количества свободных страниц, чтобы, когда случится неизбежное страничное прерывание, для него нашлись свободные страничные блоки. Сwoпер загружает в память только структуру пользователя и таблицы страниц. Страницы с текстом, данными и стеком подгружаются при помощи обычной [страничной подкачки](#).

У каждого сегмента каждого активного процесса есть место на диске, где он располагается, когда его страницы удаляются из памяти. Сегменты данных и стека сохраняются на временном устройстве, но

текст программы подгружается из самого исполняемого двоичного файла. Для текста программы временная копия не используется.

Страничная подкачка в System V во многом схожа с применяемой в системе 4BSD, что не удивительно, так как версия UNIX университета в Беркли уже в течение многих лет стабильно работала, прежде чем страничная подкачка была добавлена к System V. Тем не менее между этими версиями операционной системы есть два интересных 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 страниц на втором проходе, страница помещается туда только в случае, если она не использовалась в течении нескольких последовательных проходов. Хотя при таком решении страницы не освобождаются так быстро, как это делается алгоритмом в Berkeley UNIX, оно значительно увеличивает вероятность того, что освобожденная страница не потребуется тут же снова.

Во-вторых, вместо единственной переменной `lotsfree` в System V используются две переменные, `min` и `max`. Когда количество свободных страниц опускается ниже `min`, страничный демон начинает освобождать страницы. Демон продолжает работать до тех пор, пока число свободных страниц не сравняется со значением `max`. Такой подход позволяет избежать неустойчивости, возможной в системе 4DSD. Рассмотрим ситуацию, в которой количество свободных страниц на единицу меньше, чем `lostfree`, поэтому страничный демон освобождает одну страницу, чтобы привести количество свободных страниц в соответствие со значением `lostfree`. Затем происходит еще одно страничное прерывание, и количество свободных страниц опять становится на единицу меньше `lotsfree`, в результате страничный демон снова начинает работать. Если установить значение `max` существенно большим, чем `min`, страничный демон, завершив освобождение страниц, создает достаточный запас, чтобы иметь возможность отдохнуть некоторое время.

### **Управление памятью в Linux**

Каждый процесс системы Linux на 32-разрядной машине получает 3 Гбайт виртуального адресного пространства для себя, с оставшимся 1 Гбайт памяти для страничных таблиц и других данных ядра. Один

гигабайт ядра не виден в пользовательском режиме, но становится доступным, когда процесс переключается в режим ядра. Адресное пространство создается при создании процесса и перезаписывается системным вызовом `exec`.

Виртуальное адресное пространство делится на однородные непрерывные области, выровненные по границам страниц. Таким образом, каждая область состоит из набора соседних страниц с одинаковым режимом защиты и одинаковыми свойствами подкачки. Примерами областей являются текстовый сегмент и файлы, отображенные на память (см. рис. 10.7). Между областями в виртуальном адресном пространстве могут быть свободные участки. Любое обращение процесса к памяти в этих свободных участках приводит к фатальному страничному прерыванию. Размер страницы фиксирован, например, 4 Кбайт для процессора Pentium и 8 Кбайт для процессора Alpha.

Каждая область описывается в ядре записью `vm_area_struct`. Все структуры `vm_area_struct` одного процесса связаны вместе в список, отсортированный по виртуальным адресам, что позволяет быстро находить все страницы. Когда список становится слишком длинным (более 32 записей), создается дерево для ускорения поиска. Запись `vm_area_struct` перечисляет свойства области. К ним относятся режим защиты (например, только чтение или чтение/запись), является ли данная область фиксированной в памяти (невыгружаемой), и направление, в котором область может расти (вверх для сегментов данных, вниз для сегмента стека).

Структура `vm_area_struct` также содержит данные о том, является ли данная область приватной областью процесса или ее совместно используют несколько процессов. После системного вызова `fork` система Linux создает копию списка областей для дочернего процесса, но у дочернего и родительского процессов оказываются указатели на одни и те же таблицы страниц. Области помечаются как доступные для чтения/записи, но страницы доступны только для чтения. Если любой из процессов пытается записать данные в такую страницу, происходит прерывание, ядро видит, что область логически доступна для записи, а страница недоступна, поэтому оно дает процессу копию страницы, которую помечает как доступную для чтения/записи. Таким образом



реализован механизм копирования при записи.

Кроме того, в структуре `vm_area_struct` записано, есть ли у этой области памяти место хранения на диске, и если да, то где оно расположено. Текстовые сегменты в качестве резервного хранения используют двоичные файлы, а отображаемые на адресное пространство памяти файлы выгружаются на диск в соответствующие им файлы. Всем остальным областям, таким как область стека, не назначаются области резервного хранения, пока не потребуется их выгрузка на диск.

В системе Linux используется трехуровневая схема страничной подкачки. Хотя эта схема была реализована в системе для процессора Alpha, она также используется (в упрощенном виде) для всех архитектур. Каждый виртуальный адрес разбивается на четыре поля, как показано на рисунке 4.19. Каталогное поле используется как индекс в глобальном каталоге, в котором есть личный каталог для каждого процесса. Содержание элемента каталога является указателем на одну из средних страничных таблиц, которые тоже проиндексированы полем виртуального адреса. Наконец, элемент средней таблицы указывает на таблицу страниц, также проиндексированную полем страницы виртуального адреса. Элемент в последней таблице содержит указатель на нужную страницу. На компьютерах с процессором Pentium используется только двухуровневая организация страниц. В этом случае каждый средний страничный каталог содержит только одну запись. Таким образом, глобальный каталог фактически указывает на таблицу страниц.

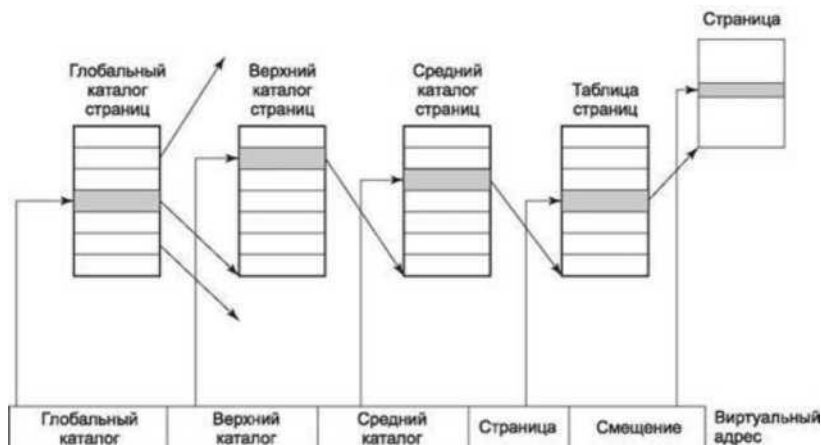


Рис. 4.15

Физическая память используется для различных целей. Само ядро жестко фиксировано. Ни одна его часть не выгружается на диск. Остальная часть памяти доступна для страниц пользователей, буферного кэша, используемого файловой системой, страничного кэша и других задач. Буферный кэш содержит блоки файлов, которые были недавно считаны или были считаны заранее в надежде на то, что они скоро могут понадобиться. Его размер динамически меняется. Буферный кэш состязается за место в памяти со страницами пользователей. Страничный кэш в действительности не является настоящим отдельным кэшем, а представляет собой просто набор страниц пользователя, которые более не нужны и ожидают выгрузки на диск. Если страница, находящаяся в страничном кэше, потребуется снова, прежде чем она будет удалена из памяти, ее можно быстро объявить находящейся в памяти.

Кроме этого, операционная система Linux поддерживает динамически загружаемые модули, в основном драйверы устройств. Они могут быть произвольного размера и каждому из них должен быть выделен непрерывный участок в памяти ядра. Для выполнения всех этих требований система Linux управляет памятью таким образом, что она может получить по желанию участок памяти произвольного размера. Для этого используется алгоритм, известный как «дружественный».

Операционная система Linux управляет памятью при помощи

данного алгоритма. К нему добавляется массив, в котором первый элемент представляет собой начало списка блоков размером в 1 единицу, второй элемент является началом списка блоков размером в 2 единицы, третий элемент— началом списка блоков размером в 4 единицы и т. д. Таким образом, можно быстро найти любой блок с размером кратным степени 2.

Этот алгоритм приводит к существенной внутренней фрагментации, так как, если вам нужен 65-страничный участок, вы получите 128-страничный блок.

Чтобы как-то решить эту проблему, в системе Linux есть второй алгоритм выделения памяти, выбирающий блоки памяти при помощи «дружественного» алгоритма, а затем нарезающий из этих блоков более мелкие фрагменты и управляющий этими фрагментами отдельно. Кроме того, существует третий алгоритм выделения памяти, использующийся, когда выделяемая память должна быть непрерывна только в виртуальном адресном пространстве, но не в физической памяти. Все эти алгоритмы выделения памяти были взяты из системы System V.

## Лекция 1.5. Ввод-вывод в системе UNIX

Как и у всех компьютеров, у машин, работающих под управлением операционной системы UNIX, есть устройства ввода-вывода, такие как диски, принтеры и сети, соединенные с ними. Требуется некий способ предоставления программам доступа к этим устройствам. Хотя возможны различные варианты решений данного вопроса, подход, применяемый в операционной системе UNIX, 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/hd1, у принтера может быть путь /dev/lp, а у сети— /dev/net.

Доступ к этим специальным файлам осуществляется так же, как и к обычным файлам. Для этого не требуется никаких специальных команд или системных вызовов. Вполне подойдут обычные системные вызовы

read и write. Например, команда `cp file /dev/lp` скопирует файл `file` на принтер, в результате чего этот файл будет распечатан (при условии, что у пользователя есть разрешение доступа к `/dev/lp`). Программы могут открывать, читать специальные файлы, а также писать в них тем же способом, что и в обычные файлы. На самом деле программа `cp` в приведенном выше примере даже не знает, что она занимается печатью файла на принтере. Таким образом, для выполнения ввода-вывода не требуется специального механизма.

Специальные файлы подразделяются на две категории: блочные и символьные. **Блочный специальный файл** — это специальный файл, состоящий из последовательности нумерованных блоков. Основное свойство блочного специального файла заключается в том, что к каждому его блоку можно адресоваться и получить доступ отдельно. Другими словами, программа может открыть блочный специальный файл и прочитать, скажем, 124-й блок, не читая сначала блоки с 0 по 123. Блочные специальные файлы обычно используются для дисков.

**Символьные специальные файлы**, как правило, используются для устройств ввода или вывода символьного потока. Символьные специальные файлы используются такими устройствами, как клавиатуры, принтеры, сети, мыши, плоттеры и т. д. Невозможно (и даже бессмысленно) искать на мыши 124-й блок.

С каждым специальным файлом связан драйвер устройства, осуществляющий управление соответствующим устройством. У каждого драйвера есть так называемый номер **старшего устройства**, служащий для его идентификации. Если драйвер одновременно поддерживает несколько устройств, например, два диска одного типа, то каждому диску присваивается номер **младшего устройства**, идентифицирующий это устройство. Вместе номера главного устройства и младшего устройства однозначно обозначают каждое устройство ввода-вывода. В некоторых случаях один драйвер может управлять двумя связанными устройствами. Например, драйвер, соответствующий символьному специальному файлу `/dev/tty`, управляет и клавиатурой, и экраном, которые часто воспринимаются как единое устройство, терминал.

В операционной системе UNIX эти символы не заданы жестко, а

могут быть настроены пользователем. Для установки этих параметров обычно предоставляется специальный системный вызов. Этот системный вызов также управляет преобразованием табулятора в пробелы, включением и выключением эха при вводе, преобразованиями символов перевода каретки в перенос строки и т. д. Для обычных файлов и блочных специальных файлов этот системный вызов недоступен.

### Работа с сетью

Другим примером ввода-вывода является работа с сетью, впервые появившаяся в Berkeley UNIX. Ключевым понятием в схеме Berkeley UNIX 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 позволяют абоненту подключить телефон и соединиться с телефонной системой. Схематично расположение сокетов показано на рисунке 5.1

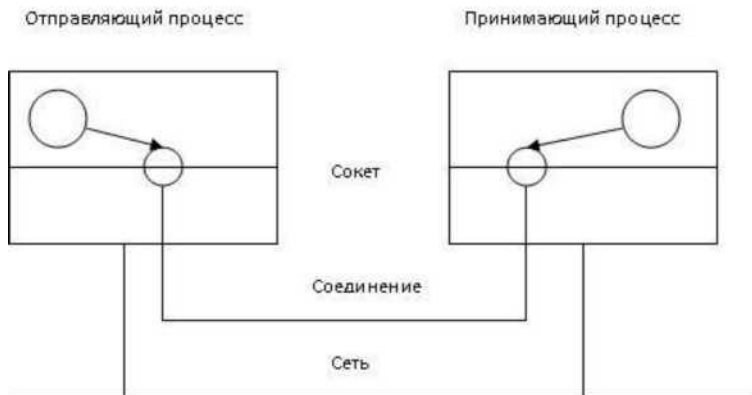


Рис. 5.1

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.

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

1. Надежный, ориентированный на соединение байтовый поток.

2. Надежный, ориентированный на соединение поток пакетов.
3. Ненадежная передача пакетов.

Первый тип сокетов позволяет двум процессам на различных машинах установить между собой эквивалент «трубы» (канала между процессами на одной машине). Байты подаются в канал с одного конца и в том же порядке выходят с другого. Такая система гарантирует, что все посланные байты придут на другой конец канала и придут именно в том порядке, в котором были отправлены.

Второй тип сокетов отличается от первого тем, что он сохраняет границы между пакетами. Если отправитель пять раз отдельно обращается к системному вызову `write`, каждый раз отправляя по 512 байт, а получатель запрашивает 2560 байт по сокету типа 1, он получит все 2560 байт сразу. При использовании сокета типа 2 ему будут выданы только первые 512 байт. Чтобы получить остальные байты получателю придется выполнить системный вызов `read` еще четыре раза. Третий тип сокета предоставляет пользователю доступ к «голой» сети. Этот тип сокета особенно полезен для приложений реального времени и ситуаций, в которых пользователь хочет реализовать специальную схему обработки ошибок. Сеть может терять пакеты или доставлять их в неверном порядке. В отличие от сокетов первых двух типов, сокет типа 3 не предоставляет никаких гарантий доставки. Преимущество этого режима заключается в более высокой производительности, которая в некоторых ситуациях оказывается важнее надежности (например, для доставки мультимедиа, при которой скорость ценится существенно выше, нежели сохранность данных по дороге).

При создании сокета один из параметров указывает протокол, используемый для него. Для надежных байтовых потоков, как правило, используется протокол **TCP** (Transmission Control Protocol — протокол управления передачей). Для ненадежной передачи пакетов обычно применяется протокол **UDP** (User Datagram Protocol — пользовательский протокол данных). Для надежного потока пакетов специального протокола нет.

Как только сокеты созданы на компьютере-источнике и компьютере-приемнике, между ними может быть установлено соединение (для ориентированной на соединение связи). 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 придут 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, тогда система устанавливает соединение между двумя сокетами.

Функции установленного соединения аналогичны функциям канала. Процесс может читать из канала и писать в него, используя дескриптор файла для локального сокета. Когда соединение более не нужно, оно может быть закрыто обычным способом, при помощи системного вызова `close`.

### **Системные вызовы ввода-вывода системы UNIX**

С каждым устройством ввода-вывода в операционной системе UNIX обычно связан специальный файл. Большую часть операций ввода-вывода можно выполнить при помощи соответствующего файла, что позволяет избежать необходимости использования специальных системных вызовов. Тем не менее иногда возникает необходимость в обращении к неким специфическим устройствам. До принятия стандарта POSIX в большинстве версий системы UNIX был системный вызов `ioctl`, выполнявший со специальными файлами большое количество операций, специфических для различных устройств. С годами все это привело к путанице. В стандарте POSIX этот вопрос был приведен в порядок, для чего функции системного вызова `Ioctl` были разбиты на отдельные функциональные вызовы, главным образом для управления терминалом. Является ли каждый из них отдельным системным вызовом или все они вместе используют один системный вызов, зависит от конкретной реализации.

Первые четыре вызова показаны в таблице 5.1, используются для задания скорости терминала. Для управления вводом и выводом предоставлены различные вызовы, так как некоторые модемы работают в несимметричном режиме. Например, старые системы `videotext` предоставляли пользователям доступ к открытым базам данных с короткими запросами от дома до сервера на скорости 75 бит/с и ответами, посылаемыми со скоростью 1200 бит/с. Этот стандарт был принят в то время, когда скорость 1200 бит/с в обоих направлениях была

слишком дорогой для домашнего использования. Такая асимметрия все еще сохраняется на многих линиях связи. Например, некоторые телефонные компании предоставляют услуги цифровой связи ADSL (Asymmetric Digital Subscriber Line — асимметричная цифровая абонентская линия) с входящим потоком на скорости 1,5 Мбит/с и исходящим потоком в 384 кбит/с.

Таблица 5.1

Вызов	Описание
<code>s=cfsetospeed(&amp;termois, speed)</code>	Задать исходящую скорость
<code>s=cfsettispeed(&amp;termois, speed)</code>	Задать входящую скорость
<code>s=cfgetospeed(&amp;termois, speed)</code>	Получить исходящую скорость
<code>s=cfgettispeed(&amp;termois, speed)</code>	Получить входящую скорость
<code>s=tsetattr(fd, opt, &amp;termois)</code>	Задать атрибуты
<code>s=tsetattr(fd, &amp;termois)</code>	Получить атрибуты

Последние два системных вызова в списке предназначены для задания и считывания всех специальных символов, используемых для удаления символов и строк, прерывания процессов и т. д. Помимо этого они позволяют разрешить и запретить вывод эха, осуществлять управление потоком, а также выполнять другие связанные с символьным вводом-выводом функции. Также существуют дополнительные функциональные вызовы ввода-вывода, но они в некотором роде специализированные, поэтому мы не станем рассматривать их в дальнейшем. Следует также отметить, что системный вызов `Ioctl` по сию пору существует во многих системах UNIX.

### Реализация ввода-вывода в системе UNIX

Ввод-вывод в операционной системе UNIX реализуется набором драйверов устройств, по одному для каждого типа устройств. Функция драйвера заключается в изолировании остальной части системы от индивидуальных отличительных особенностей аппаратного обеспечения. При помощи стандартных интерфейсов между драйверами и остальной операционной системой большая часть системы ввода-вывода может быть помещена в машинно-независимую часть ядра.

Когда пользователь получает доступ к специальному файлу,



файловая система определяет номера старшего и младшего устройств, а также выясняет, является ли файл блочным специальным файлом или символьным специальным файлом. Номер старшего устройства используется в качестве индекса для одного из двух внутренних массивов структур —bdevswuna блочных специальных файлов или cdevsw для символьных специальных файлов.

Найденная таким образом структура содержит указатели на процедуры открытия устройства, чтения из устройства, записи на устройство и т. д. Номер младшего устройства передается в виде параметра. Добавление нового типа устройства к системе UNIX означает добавление нового элемента к одной из этих таблиц, а также предоставление соответствующих процедур выполнения различных операций с устройством.

Система ввода-вывода разделена на два основных компонента: обработку блочных специальных файлов и обработку символьных специальных файлов. Мы по очереди рассмотрим эти компоненты.

Цель той части системы, которая занимается операциями ввода-вывода с блочными специальными файлами (например, дисковым вводом-выводом), заключается в минимизации количества операций переноса данных. Для достижения данной цели в системах UNIX между дисковыми драйверами и файловой системой помещается **буферный кэш** (рисунок 5.2)

Буферный кэш представляет собой таблицу в ядре, в которой хранятся тысячи недавно использованных блоков. Когда файловой системе требуется блок диска (например, блок i-узла, каталога или данных), сначала проверяется буферный кэш. Если нужный блок есть в кэше, он получается оттуда, при этом обращения к диску удастся избежать. Буферный кэш значительно улучшает производительность системы.

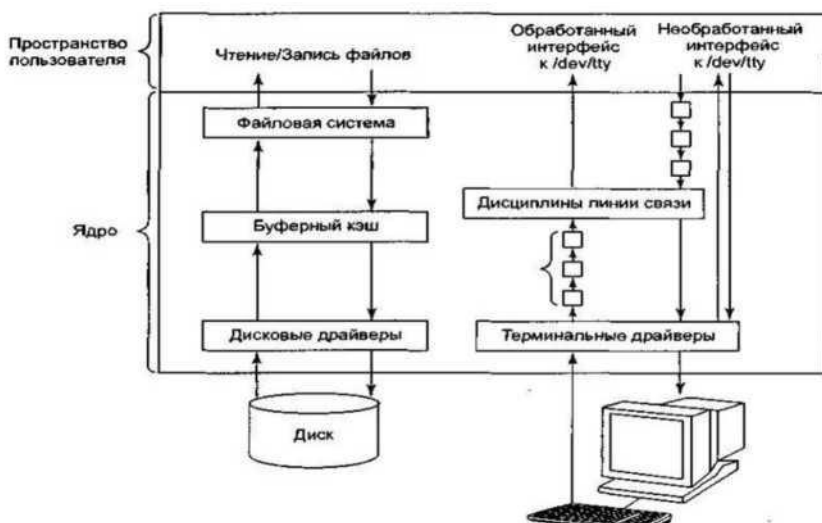


Рис. 5.2

Если же блока нет в буферном кэше, он считывается с диска в кэш, а оттуда копируется туда, куда нужно. Поскольку в буферном кэше есть место только для фиксированного количества блоков, требуется некий алгоритм управления кэшем. Обычно блоки в кэше организуются в связный список. При каждом обращении к блоку он перемещается в начало списка. Если в кэше не хватает места для нового блока, то из него удаляется самый старый блок, находящийся в конце списка.

В течение десятилетий драйверы устройств системы UNIX статически компоновались вместе с ядром, так что все они постоянно находились в памяти при каждой загрузке системы. Такая схема хорошо работала в условиях мало меняющихся конфигураций факультетских мини-компьютеров, а затем сложных рабочих станции, то есть в тех условиях, в которых росла и развивалась операционная система UNIX. Как правило, компьютерный центр формировал ядро операционной системы, содержащее драйверы для всех необходимых в данной конфигурации устройств ввода-вывода, которыми потом и пользовался. Если на следующий год покупался новый диск, компьютерный центр перекомпоновывал ядро, что было не так уж и сложно.

В операционной системе Linux подобные проблемы были решены при помощи концепции **подгружаемых модулей**. Это куски кода,

которые могут быть загружены в ядро во время работы операционной системы. Как правило, это драйверы символьных или блочных устройств, но подгружаемым модулем также могут быть целая файловая система, сетевые протоколы, программы для отслеживания производительности системы и т. д.

### Потоки данных

Так как символьные специальные файлы имеют дело с символьными потоками, а не перемещают блоки данных между памятью и диском, они не пользуются буферным кэшем. Вместо этого в первых версиях системы UNIX каждый драйвер символьного устройства выполнял всю работу, требуемую для данного устройства. Однако с течением времени стало ясно, что многие драйверы, например, программы буферизации, управления потоком и сетевые протоколы, дублировали процедуры друг друга. Поэтому для структурирования драйверов символьных устройств и придания им модульности было разработано два решения. Мы кратко рассмотрим их по очереди.

Решение, реализованное в системе BSD, основано на структурах данных, присутствующих в классических системах UNIX, называемых **С-списками** (они показаны в виде квадратиков на рис. 10.12). Каждый С-список представляет собой блок размером до 64 символов плюс счетчик и указатель на следующий блок. Символы, поступающие с терминала или любого другого символьного устройства, буферизируются в цепочках таких блоков.

Когда пользовательский процесс считывает данные из `/dev/tty` (то есть из стандартного входного потока), символы не передаются процессу напрямую из С-списков. Вместо этого они пропускаются через процедуру, расположенную в ядре и называющуюся **дисциплиной линии связи**. Дисциплина линии связи работает как фильтр, принимая необработанный поток символов от драйвера терминала, обрабатывая его и формируя то, что называется **обработанным символьным потоком**. В обработанном потоке выполняются операции локального строкового редактирования (например, удаляются отмененные пользователем символы и строки), символы возврата каретки заменяются символами переноса строки, а также выполняются другие специальные операции обработки. Обработанный поток передается процессу. Однако если процесс желает воспринимать каждый символ,

введенный пользователем, он может принимать необработанный поток, минуя дисциплину линии связи.

Вывод работает аналогично, заменяя табуляторы пробелами, добавляя перед символами переноса строки символы возврата каретки, добавляя для медленных механических терминалов символы-заполнители, за которыми следует символ возврата каретки и т. д. Как и входной поток, выходной символьный поток может быть пропущен через дисциплину линии связи (обработанный режим) или миновать ее (необработанный режим). Необработанный режим особенно полезен при отправке двоичных данных на другие компьютеры по линии последовательной передачи или для графических интерфейсов пользователя. Здесь не требуется никакого преобразования.

Решение, реализованное в системе System V под названием **потоков данных**, было разработано Деннисом Ритчи. Это общий подход. (в System V также есть буферный кэш для блочных специальных файлов, но поскольку он, по сути, не отличается от схемы, применяемой в BSD, кэш не показан здесь.) Потоки данных основаны на возможности динамически соединять процесс пользователя с драйвером, а также динамически, во время исполнения, вставлять модули обработки в поток данных. В некотором смысле поток представляет собой работающий в ядре аналог каналов в пространстве пользователя

## Лекция 1.6. Файловая система UNIX

Файл в системе UNIX — это последовательность байтов произвольной длины (от 0 до некоторого максимума), содержащая произвольную информацию. Не делается принципиального различия между текстовыми (ASCII) файлами, двоичными файлами и любыми другими файлами. Значение битов в файле целиком определяется владельцем файла. Системе это безразлично. Изначально размер имен файлов был ограничен 14 символами, но в системе Berkeley UNIX этот предел был расширен до 255 символов, что впоследствии было принято в System V, а также в большинстве других версий. В именах файлов

разрешается использовать все ASCII-символы, кроме символа NUL, поэтому допустимы даже имена файлов, состоящие, например, из трех символов возврата каретки (хотя такое имя и не слишком удобно в использовании).

Существует два способа задания имени файла в системе UNIX, как в оболочке, так и при открытии файла из программы. Первый способ заключается в использовании абсолютного пути, указывающего, как найти файл от корневого каталога. Пример абсолютного пути: `/usr/ast/books/mos2/chap-10`. Он сообщает системе, что в корневом каталоге следует найти каталог `usr`, затем в нем найти каталог `ast`, который содержит каталог `books`, в котором содержится каталог `mos2`, а в нем, наконец, расположен файл `chap-10`.

Абсолютные имена путей часто бывают длинными и неудобными. По этой причине операционная система UNIX позволяет пользователям и процессам обозначить каталог, в котором они работают в данный момент, как рабочий каталог (также называемый текущим каталогом). Имена путей также могут указываться относительно рабочего каталога. Путь файла, заданный относительно рабочего каталога, называется относительным путем. Например, если каталог `/usr/ast/books/mos2` является рабочим каталогом, тогда команда оболочки

```
ср chap-10 backupl
```

возымеет тот же самый эффект, что и более длинная команда

```
ср /usr/ast/books/mos2/chap-10 /usr/ast/books/mos2/backupl
```

Пользователям часто бывает необходимо обратиться к файлам, принадлежащим другим пользователям, или к своим файлам, расположенным в другом месте дерева файлов. Например, если два пользователя совместно используют один файл, он будет находиться в каталоге, принадлежащем одному из них, поэтому другому пользователю понадобится для обращения к этому файлу использовать абсолютное имя пути (или менять свой рабочий каталог). Если абсолютный путь достаточно длинен, то необходимость вводить его каждый раз может весьма сильно раздражать. В системе UNIX эта проблема решается при помощи так называемых связей, представляющих собой записи каталога, указывающие на другие файлы.

Кроме обычных файлов, системой UNIX также поддерживаются

символьные специальные файлы и блочные специальные файлы. Символьные специальные файлы используются для моделирования последовательных устройств ввода-вывода, таких как клавиатуры и принтеры. Если процесс откроет файл `/dev/tty` и прочитает из него, он получит символы, введенные с клавиатуры. Если открыть файл `/dev/lp` и записать в него данные, то эти данные будут распечатаны на принтере. Блочные специальные файлы, часто с такими именами, как `/dev/hd1`, могут использоваться для чтения и записи необработанных дисковых разделов, минуя файловую систему. Таким образом, поиск байта номер  $k$ , за которым последует чтение, приведет к чтению  $k$ -го байта на соответствующем дисковом разделе, игнорируя  $i$ -узел и файловую структуру. Необработанные блочные устройства используются для страничной подкачки и свопинга программами установки файловой системы (например, `mkfs`) и программами, исправляющими ломаные файловые системы (например, `fsck`).

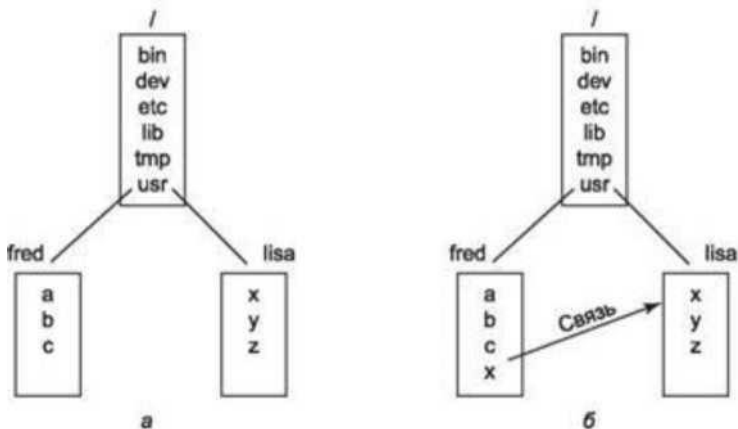


Рис. 5.3

На многих компьютерах установлено по два и более дисков. Например, на мэйнфреймах в банках часто бывает необходимо подключать по 100 и более дисков к одной машине, чтобы хранить большие базы данных. Даже у персональных компьютеров есть по меньшей мере два диска— жесткий диск и дисковод для гибких дисков. При наличии у компьютера нескольких дисков возникает вопрос, как ими управлять.

Одно из решений заключается в том, чтобы установить самостоятельную файловую систему на каждый отдельный диск и управлять ими как отдельными файловыми системами. Рассмотрим, например, ситуацию, изображенную на рисунке 5.3. Здесь показан жесткий диск, который мы будем называть C:, а также гибкий диск, который мы будем называть A:. У каждого есть собственный корневой каталог и файлы. При таком решении пользователь должен помимо каталогов указывать также и устройство, если оно отличается от используемого по умолчанию. Например, чтобы скопировать файл x в каталог d (предполагая, что по умолчанию выбирается диск C:), следует ввести команду

```
ср a:/x /a/d/x
```

Такой подход применяется в операционных системах MS-DOS, Windows 98 и VMS.

Решение, применяемое в операционной системе UNIX, заключается в том, чтобы позволить монтировать один диск в дерево файлов другого диска. В нашем примере мы можем смонтировать дискету в каталог /b, получая в результате файловую систему, показанную на рисунке 5.4. Теперь пользователь видит единое дерево файлов и уже не должен думать о том, какой файл на каком устройстве хранится. В результате приведенная выше команда примет вид

```
ср /b/x /a/d/x
```

то есть все будет выглядеть так, как если бы файл копировался из одного каталога жесткого диска в другой каталог того же диска.

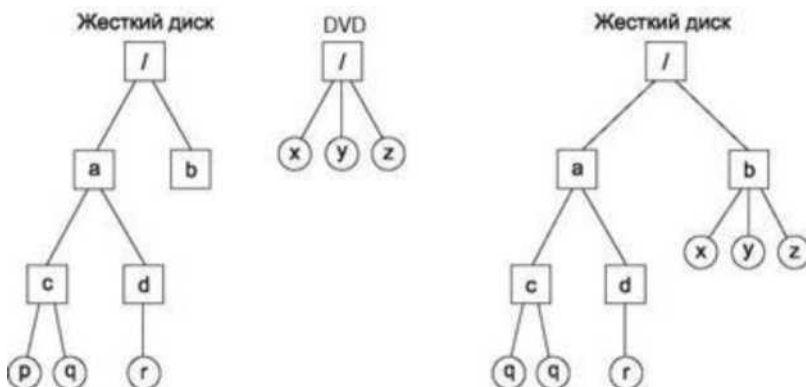


Рис. 5.4

Другое интересное свойство файловой системы UNIX представляет собой блокировка. В некоторых приложениях два и более процессов могут одновременно использовать один и тот же файл, что может привести к конфликту. Одно из решений данной проблемы заключается в том, чтобы создать в приложении критические области. Однако если эти процессы принадлежат независимым пользователям, даже не знакомым друг с другом, такой способ координации действий, как правило, очень неудобен.

Рассмотрим, например, базу данных, состоящую из многих файлов в одном или нескольких каталогах, доступ к которым могут получить никак не связанные между собой пользователи. С каждым каталогом или файлом можно связать семафор и достичь взаимного исключения, заставляя процессы выполнять операцию down на соответствующем семафоре, прежде чем читать или писать определенные данные. Недостаток этого решения заключается в том, что при этом недоступным становится весь каталог или файл, даже если процессам нужна всего одна запись.

Стандартом определены два типа блокировки: блокировка с монополизацией и блокировка без монополизации. Если часть файла уже содержит блокировку без монополизации, то повторная установка блокировки без монополизации на это место файла разрешается, но попытка установки блокировку с монополизацией будет отвергнута. Если же какая-либо область файла содержит блокировку с монополизацией, то любые попытки заблокировать любую часть этой области файла будут отвергаться, пока не будет снята монополярная блокировка. Для успешной установки блокировки необходимо, чтобы каждый байт в данной области был доступен.

При установке блокировки процесс должен указать, хочет ли он сразу получить управление или будет ждать, пока не будет установлена блокировка. Если процесс выбрал вызов с ожиданием, то он блокируется до тех пор, пока с запрашиваемой области файла не будет снята блокировка, установленная другим процессом, после чего процесс активизируется, и ему сообщается, что блокировка установлена. Если процесс решил воспользоваться системным вызовом без ожидания, он немедленно получает ответ об успехе или неудаче операции.

Заблокированные области могут перекрываться. На рисунке 5.5, а



показано, что процесс А установил блокировку без монополизации на байты с 4 по 7 в некотором файле. Затем процесс В устанавливает блокировку без монополизации на байты с 6 по 9. Наконец, процесс С устанавливает блокировку без монополизации на байты со 2 по 11. Пока это блокировки без монополизации, они могут устанавливаться одновременно. Теперь посмотрим, что произойдет, если какой-либо процесс попытается получить блокировку с монополизацией к байту 9, используя системный вызов с ожиданием, когда блокировка данного участка файла сразу невозможна. Две предыдущие блокировки перекрываются с этой блокировкой. Поэтому обращающийся с новым

запросом процесс будет заблокирован и останется заблокированным, пока оба процесса В и С не снимут свою блокировку.

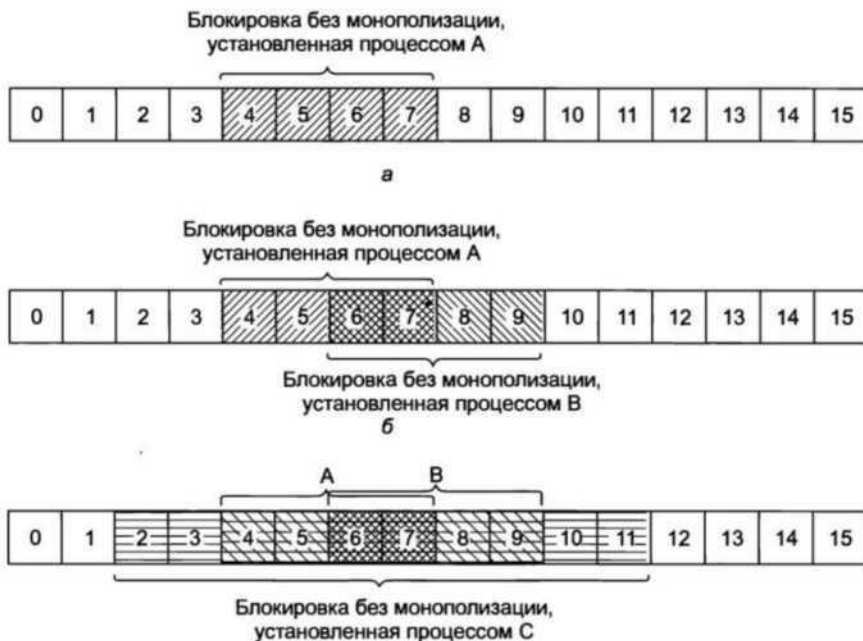


Рис. 5.5

## Вызовы файловой системы в UNIX

Многие системные вызовы относятся к файлам и файловой системе. Сначала мы рассмотрим системные вызовы, работающие с отдельными файлами. Затем мы изучим те системные вызовы, которые оперируют каталогами или всей файловой системой в целом. Для создания нового файла можно использовать системный вызов `creat`. (Когда Кена Томпсона однажды спросили, что бы он поменял, если бы у него была возможность во второй раз

разработать операционную систему UNIX, он ответил, что на этот раз вместо `creat` он назвал бы этот системный вызов `create`.) В качестве параметров этому системному вызову следует задать имя файла и режим защиты. Так, команда

```
fd = creat("abc". mode) :
```

создает файл `abc` с режимом защиты, указанном в переменной (или константе) `mode`. Биты `mode` определяют круг пользователей, которые могут получить доступ к файлу, а также уровень предоставляемого им доступа.

Системный вызов `creat` не только создает новый файл, но также и открывает его для записи. Чтобы последующие системные вызовы могли получить доступ к файлу, успешный системный вызов `creat` возвращает небольшое неотрицательное целое число, называемое дескриптором файла (`fd` в приведенном выше примере). Если системный вызов выполняется с уже существующим файлом, длина этого файла уменьшается до 0, а все содержимое теряется.

Хотя большинство программ читают и записывают файлы последовательно, некоторым программам бывает необходимо иметь доступ к произвольной части файла. С каждым открытым файлом связан указатель на текущую позицию в файле. При последовательном чтении или записи он указывает на следующий байт, который будет прочитан или записан. Например, если указатель установлен на 4096-й байт, то после успешного чтения 1024 байт из этого файла при помощи системного вызова `read` он будет указывать на 5120-й байт. Указатель в файле можно переместить с помощью системного вызова `lseek`, что позволяет при следующих обращениях к системным вызовам `read` и `write` читать данные из файла или писать их в файл в произвольной позиции в файле и даже за концом файла. Этот системный вызов назван `lseek`, чтобы не путать его с теперь уже устаревшим, использовавшимся ранее на 16-разрядных компьютерах системным вызовом `seek`.

У системного вызова `lseek` три параметра: первый — это дескриптор файла, второй — новая позиция в файле, а третий сообщает, указывается ли эта позиция относительно начала файла, конца файла или относительно текущей позиции. Значение, возвращаемое системным вызовом `lseek`, представляет собой абсолютную позицию в файле после того, как указатель был перемещен. Забавно, что системный вызов `lseek` (`seek` означает поиск, термин, также используемый для перемещения блока головок диска) никогда не вызывает перемещения блока головок диска, так как все, что он делает, — это обновление текущей позиции в файле, представляющей собой просто число в памяти.

## **Реализация файловой системы UNIX**

Сначала опишем реализацию традиционной файловой системы UNIX. Затем мы обсудим усовершенствования, реализованные в версии Berkeley. Также используются и другие файловые системы. Все системы UNIX могут поддерживать несколько дисковых разделов, каждый со своей файловой системой.

В классической системе UNIX раздел диска содержит файловую систему, расположение которой изображено на рисунке 5.6. Блок 0 не используется системой и часто содержит программу загрузки компьютера. Блок 1 представляет собой суперблок. В нем хранится критическая информация о размещении файловой системы, включая количество *i*-узлов, количество дисковых блоков, а также начало списка свободных блоков диска (обычно несколько сот записей). При уничтожении суперблока файловая система окажется нечитаемой.

Следом за суперблоком располагаются *i*-узлы (*i*-nodes, сокращение от *index-nodes* — индекс узлы). Они нумеруются от 1 до некоторого максимального числа. Каждый *i*-узел имеет 64 байт в длину и описывает ровно один файл, *i*-узел содержит учетную информацию (включая всю информацию, возвращаемую системным вызовом *stat*, который ее просто берет в *i*-узле), а также достаточное количество информации, чтобы найти все блоки файла на диске.



Рис. 5.6

Следом за *i*-узлами располагаются блоки с данными. Здесь хранятся все файлы и каталоги. Если файл или каталог состоит более чем из одного блока, блоки файла не обязаны располагаться на диске подряд. В действительности блоки большого файла, как правило, оказываются

разбросанными по всему диску. Именно эту проблему должны были решить усовершенствования версии Berkeley.

Каталог в традиционной файловой системе (то есть V7) представляет

собой несортированный набор 16-байтовых записей. Каждая запись состоит из 14-байтного имени файла и номера i-узла (см. рис. 6.33). Чтобы открыть файл в рабочем каталоге, система просто считывает каталог, сравнивая имя искомого файла с каждой записью, пока не найдет нужную запись или пока не закончится каталог.

Если искомый файл присутствует в каталоге, система извлекает его i-узел и использует его в качестве индекса в таблице i-узлов (на диске), чтобы найти соответствующий i-узел и считать его в память. Этот i-узел помещается в таблицу i-узлов, структуру данных в ядре, содержащую все i-узлы открытых в данный момент файлов и каталогов. Формат i-узлов варьируется от одной версии UNIX к другой. Как минимум i-узел должен содержать все поля, возвращаемые системным вызовом `stat`.

Рассмотрим теперь, как система считывает файл. Вспомним, что типичное обращение к библиотечной процедуре для запуска системного вызова `read` выглядит следующим образом:

```
n = read(fd, buffer, nbytes);
```

Когда ядро получает управление, ему подаются только эти три параметра. Все остальные необходимые данные оно может получить из внутренних таблиц, относящихся к пользователю. Одной из таких таблиц является массив дескрипторов файла. Он проиндексирован по номерам дескрипторов файла и содержит по одной записи для каждого открытого файла (до некоторого максимума, как правило около 20 файлов).

По дескриптору файла файловая система должна найти i-узел соответствующего файла. Рассмотрим одно из возможных решений: просто поместим в таблицу дескрипторов файла указатель на i-узел. Несмотря на простоту, данный метод, увы, не работает. Проблема заключается в следующем. С каждым дескриптором файла должен быть связан указатель в файле, определяющий байт в файле, который будет считан или записан при следующем обращении к файлу. Где следует хранить этот указатель? Один вариант состоит в помещении его в таблице i-узлов. Однако такой подход не сможет работать, если несколько не связанных друг с другом процессов одновременно откроют один и тот же файл, так как у каждого процесса должен быть свой собственный указатель.

Второй вариант решения заключается в помещении указателя в

таблицу дескрипторов файла. При этом каждый процесс, открывающий файл, получает собственную позицию в файле. К сожалению, такая схема также не работает, но причина неудачи в данном случае не столь очевидна и имеет отношение к природе совместного использования файлов в системе UNIX. Рассмотрим сценарий оболочки *s*, состоящий из двух команд, *p1* и *p2*, которые должны работать по очереди. Если сценарий вызывается командной строкой

```
s >x
```

то ожидается, что команда *p1* будет писать свои выходные данные в файл *x*, а команда *p2* будет также писать свои выходные данные в файл *x*, начиная с того места, на котором остановилась команда *p1*.

Когда оболочка запустит процесс *p1*, файл *x*, будет изначально пустым, поэтому команда *p1* просто начнет запись в файл в позиции 0. Однако когда она закончит свою работу, требуется некий механизм передачи указателя в файле *x* от процесса *p1* процессу *p2*. Если же позицию в файле хранить просто в таблице дескрипторов файла, процесс *p2* начнет запись в файл с позиции 0.

### **Файловая система Berkeley Fast**

Приведенное выше описание объясняет принципы работы классической файловой системы UNIX. Теперь познакомимся с усовершенствованиями этой системы, реализованными в версии Berkeley. Во-первых, были реорганизованы каталоги. Длина имен файлов была увеличена до 255 символов. Конечно, изменение структуры всех каталогов означало, что программы, продолжающие наивно читать напрямую содержимое каталогов (что было и остается допустимым) и ожидающие обнаружить в них последовательность 16-байтовых записей, более не работали. Для обеспечения совместимости двух систем в системе Berkeley были разработаны системные вызовы *opendir*, *closed^*, *readdir* и *rewinddir*, чтобы программы могли читать каталоги, не зная их внутренней структуры. Позднее длинные имена файлов и эти системные вызовы были добавлены ко всем другим версиям UNIX и к стандарту POSIX.

Структура каталогов BSD, поддерживающая имена файлов длиной до 255 символов, показана на рис. 5.10. Каждый каталог состоит из некоторого целого количества дисковых блоков, так что каталоги могут записываться на диск как единое целое. Внутри каталога записи файлов

и каталогов никак не отсортированы, при этом каждая запись сразу следует за предыдущей записью. В конце каждого блока может оказаться несколько неиспользованных байтов, так как записи могут быть различного размера.

Каждая каталоговая запись на рис. 5.7 состоит из четырех полей фиксированной длины и одного поля переменной длины. Первое поле представляет собой номер *i*-узла, равный 19 для файла colossal, 42 для файла voluminous и 88 для каталога bigdir. Следом за номером *i*-узла идет поле, сообщающее размер всей каталоговой записи в байтах, возможно, вместе с дополнительными байтами-заполнителями в конце записи. Это поле необходимо, чтобы найти следующую запись. На рисунке это поле обозначено стрелкой, указывающей на начало следующей записи. Затем располагается поле типа файла, определяющее, является ли этот файл каталогом и т. д. Последнее поле содержит длину имени файла в байтах (8, 10 и 6 для данного примера). Наконец, идет само имя файла, заканчивающееся нулевым байтом и дополненное до 32-байтовой границы. За ним могут следовать дополнительные байты-заполнители

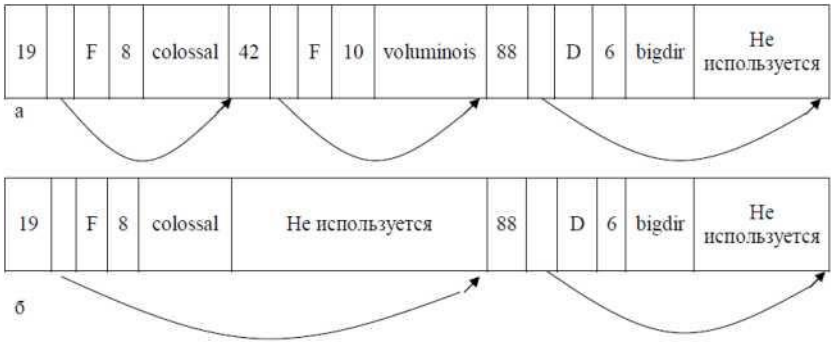


Рис. 5.7

На рис 5.10 показан тот же самый каталог после того, как файл voluminous был удален. Все, что при этом делается в каталоге, — увеличивается размер записи предыдущего файла colossal, а байты каталоговой записи удаленного файла voluminous превращаются в заполнители первой записи. Впоследствии эти байты могут использоваться для записи при создании нового файла.

Поскольку поиск в каталогах производится линейно, он может занять

много времени, пока не будет найдена запись у конца большого каталога. Для увеличения производительности в BSD было добавлено кэширование имен. Прежде чем искать имя в каталоге, система проверяет кэш. Если имя файла есть в кэше, то в каталоге его уже можно не искать.

Вторым существенным изменением, введенным в Berkeley, было разбиение диска на группы цилиндров, у каждой из которых был собственный суперблок, i-узлы и блоки данных. Идея такой организации диска заключается в том, чтобы хранить i-узел и блоки данных файла ближе друг к другу. Тогда при обращении к файлам снижается время, затрачиваемое жестким диском на перемещение блоков головок. По мере возможности блоки для файла выделяются в группе цилиндров, в которой содержится i-узел.

Третье изменение заключалось в использовании блоков не одного, а двух размеров. Для хранения больших файлов значительно эффективнее использовать небольшое количество крупных блоков, чем много маленьких блоков. С другой стороны, размер многих файлов в системе UNIX невелик, поэтому при использовании только блоков большого размера расходовалось бы слишком много дискового пространства. Наличие блоков двух размеров обеспечивает эффективное чтение/запись для больших файлов и эффективное использование дискового пространства для небольших файлов. Платой за эффективность является значительная дополнительная сложность программы.

### **Файловая система Linux**

Изначально в операционной системе Linux использовалась файловая система операционной системы MINIX. Однако в системе MINIX длина имен файлов ограничивалась 14 символами (для совместимости с UNIX Version 7), а максимальный размер файла был равен 64 Мбайт. Поэтому у разработчиков операционной системы Linux практически сразу появился интерес к усовершенствованию файловой системы. Первым шагом вперед стала файловая система Ext, в которой длина имен файлов была увеличена до 255 символов, а размер файлов — до 2 Гбайт. Однако эта система была медленнее файловой системы MINIX, поэтому исследования некоторое время продолжались. Наконец, была разработана файловая система Ext2, с длинными именами файлов,



длинными файлами и высокой производительностью. Эта файловая система и стала основной файловой системой Linux. Однако операционная система Linux также поддерживает еще более десятка файловых систем, используя для этого файловую систему NFS. При компоновке операционной системы Linux предлагается сделать выбор файловой системы, которая будет встроена в ядро. Другие файловые системы при необходимости могут динамически подгружаться во время исполнения в виде модулей. Файловая система Ext2 очень похожа на файловую систему Berkeley Fast File system с небольшими изменениями. Размещение файловой системы Ext2 на жестком диске показано на рисунке 5.8. Вместо того чтобы использовать группы цилиндров, что практически ничего не значит при современных дисках с виртуальной геометрией, она делит диск на группы блоков, независимо от того, где располагаются границы между цилиндрами. Каждая группа блоков начинается с суперблока, в котором хранится информация о том, сколько блоков и i-узлов находятся в данной группе, о размере группы блоков и т. д. Затем следует описатель группы, содержащий информацию о расположении битовых массивов, количестве свободных блоков и i-узлов в группе, а также количестве каталогов в группе. Эта информация важна, так как файловая система Ext2 пытается распространить каталоги равномерно по всему диску.



Рис 5.8

В двух битовых массивах ведется учет свободных блоков и свободных i-узлов. Размер каждого битового массива равен одному блоку. При размере блоков в 1 Кбайт такая схема ограничивает размер группы блоков 8192 блоками и 8192 i-узлами. (На практике ограничение числа i-узлов никогда не встречается, так как блоки заканчиваются раньше.) Затем располагаются сами i-узлы. Размер каждого i-узла -128 байт, что в два раза больше размера стандартных i-узлов в UNIX.

Дополнительные байты в *i*-узле используются следующим образом. Вместо 10 прямых и 3 косвенных дисковых адресов файловая система Linux позволяет 12 прямых и 3 косвенных дисковых адреса. Кроме того, длина адресов увеличена с 3 до 4 байт, и это позволяет поддерживать дисковые разделы размером более 224 блоков (16 Гбайт), что уже стало проблемой для UNIX. Помимо этого зарезервированы поля для указателей на списки управления доступом, что должно обеспечить более высокую степень защиты, но это пока находится на стадии проектирования. Остаток *i*-узла зарезервирован на будущее. Как показывает история, неиспользуемые биты недолго остаются таковыми.

Работа файловой системы похожа на функционирование быстрой файловой системы Berkeley. Однако в отличие от BSD, в системе Linux используются дисковые блоки только одного размера, 1 Кбайт. Быстрая файловая система Berkeley использует 8-килобайтные блоки, которые затем разбиваются при необходимости на килобайтные фрагменты. Файловая система Ext2 делает примерно то же самое, но более простым способом. Как и система Berkeley, когда файл увеличивается в размерах, файловая система Ext2 пытается поместить новый блок файла в ту же группу блоков, что и остальные блоки, желательно сразу после предыдущих блоков. Кроме того, при создании нового файла в каталоге файловая система Ext2 старается выделить ему блоки в той же группе блоков, в которой располагается каталог. Новые каталоги, наоборот, равномерно распределяются по всему диску.

### **Особенности Ext3 и Ext4.**

Для предотвращения потерь данных после сбоев системы и отказов электропитания файловой системе Ext2 пришлось бы записывать каждый блок данных на диск немедленно после его создания. Вызванные перемещением головок записи задержки были бы такими значительными, что производительность стала бы недопустимо низкой. Поэтому записи откладываются, и изменения могут находиться в незафиксированном на диске состоянии до 30 с — что является очень длинным (для современного компьютерного оборудования) промежутком времени.

Для повышения живучести файловой системы Linux использует журналируемые файловые системы (journaling file systems). Примером такой системы является Ext3 — последовательница файловой системы

Ext2.

Основная идея такого типа файловой системы состоит в поддержке журнала, который в последовательном порядке описывает все операции файловой системы. При такой последовательной записи изменений в данных файловой системы или ее метаданных (i-узлах, суперблоке и т. д.) операции записи не страдают от издержек перемещения дисковых головок (во время случайных обращений к диску). В конечном итоге изменения записываются (фиксируются) в соответствующее место на диске — и соответствующие им записи журнала можно удалить. Если же до фиксации изменений происходит системный сбой или отказ электропитания, то при последующем запуске система обнаружит, что файловая система не была должным образом размонтирована, просмотрит журнал и выполнит все (описанные в журнале) изменения в файловой системе.

Ext3 спроектирована таким образом, чтобы быть в значительной степени совместимой с Ext2, и фактически все основные структуры данных и компоновка диска в обеих системах одинаковы. Более того, размонтированная файловая система Ext2 может быть затем смонтирована как система Ext3 и обеспечивать журналирование.

Журнал — это файл, с которым работают как с кольцевым буфером. Журнал может храниться как на том же устройстве, что и основная файловая система, так и на другом. Поскольку операции с журналом не журналируются, файловая система Ext3 с ними не работает. Для выполнения операций чтения/записи в журнал используется отдельное блочное устройство журналирования JBD (Journaling Block Device).

JBD поддерживает три основные структуры данных: запись журнала (log record), описатель атомарной операции (atomic operation handle), транзакция (transaction). Запись журнала описывает операцию низкого уровня в файловой системе (которая обычно приводит к изменениям внутри блока). Поскольку системный вызов (такой, как write) обычно приводит к изменениям во многих местах—i-узлах, блоках существующих файлов, новых блоках файлов, списке свободных блоков и т.д., — соответствующие записи журнала группируются в атомарные операции. Ext3 уведомляет JBD о начале и конце обработки системного вызова (чтобы устройство JBD могло обеспечить фиксацию либо всех записей журнала данной атомарной операции, либо никаких). И

наконец, в основном из соображений эффективности, JBD обрабатывает коллекции атомарных операций как транзакции. В транзакции записи журнала хранятся последовательно. JBD позволяет удалять фрагменты файла журнала только после того, как все принадлежащие к транзакции записи журнала надежно зафиксированы на диске.

Поскольку запись элемента журнала для каждого изменения диска может быть дорогой, то Ext3 можно настроить таким образом, чтобы она хранила журнал либо всех изменений на диске, либо только тех изменений, которые относятся к метаданным файловой системы (узлы, суперблоки, битовые массивы и т. д.). Журналирование только метаданных снижает издержки системы и повышает производительность, но не гарантирует от повреждения данных в файлах. Некоторые другие журнальные файловые системы ведут журналы только для операций с метаданными (например, XFS в SGI).

Основные достоинства файловой системы ext4 - увеличение максимального объема раздела до одного экзбайта и прирост производительности за счет реализации механизма пространственной записи файлов (новая информация добавляется в конец заранее выделенной по соседству области файла), что приводит к значительному снижению фрагментации.

Другой файловой системой Linux является файловая система /proc (process -процесс). Идея этой файловой системы изначально была реализована в 8-й редакции операционной системы UNIX, созданной лабораторией Bell Labs, а позднее скопированной в 4.4BSD и System V. Однако в операционной системе Linux данная идея получила дальнейшее развитие. Основная концепция этой файловой системы заключается в том, что для каждого процесса системы создается подкаталог в каталоге /proc. Имя подкаталога формируется из PID процесса в десятичном формате. Например, /proc619 — это каталог, соответствующий процессу с PID 619. В этом каталоге располагаются файлы, которые хранят информацию о процессе - его командную строку, строки окружения и маски сигналов. В действительности этих файлов на диске нет. Когда они считываются, система получает информацию от фактического процесса и возвращает ее в стандартном формате.

Многие расширения, реализованные в операционной системе Linux,

относятся к файлам и каталогам, расположенным в каталоге /rproc. Они содержат информацию о центральном процессоре, дисковых разделах, векторах прерывания, счетчиках ядра, файловых системах, подгружаемых модулях и о многом другом. Непривилегированные программы пользователя могут читать большую часть этой информации, что позволяет им узнать о поведении системы безопасным способом. Некоторые из этих файлов могут записываться в каталог /rproc, чтобы изменить параметры системы.

### **Файловая система NFS**

Сети играют главную роль в операционной системе UNIX с самого начала (первая сеть в системе UNIX была построена для переноса нового ядра с машины PDP-11/70 на компьютер Interdata 8/32). Рассмотрим файловую систему NFS (Network File System— сетевая файловая система) корпорации Sun Microsystems, использующуюся на всех современных системах UNIX (а также на некоторых не-UNIX системах) для объединения на логическом уровне файловых систем отдельных компьютеров в единое целое. Версия 3 реализации NFS введена в 1995 году. Версия NFSv4 была введена в 2000 году. Она предоставляет несколько улучшений по сравнению с архитектурой предыдущих версий. Интересны три аспекта файловой системы NFS: архитектура, протокол и реализация. Мы рассмотрим их все по очереди — сначала в контексте более простой версии 3, а затем кратко обсудим улучшения версии 4.

### **Архитектура файловой системы NFS**

В основе файловой системы NFS лежит представление о том, что пользоваться общей файловой системой может произвольный набор клиентов и серверов. Во многих случаях все клиенты и серверы располагаются на одной и той же локальной сети, хотя этого не требуется. Файловая система NFS может также работать в глобальной сети, если сервер находится далеко от клиента. Для простоты мы будем говорить о клиентах и серверах, как если бы они работали на различных компьютерах, хотя файловая система NFS позволяет каждой машине одновременно быть клиентом и сервером.

Каждый сервер файловой системы NFS экспортирует один или несколько ее каталогов, предоставляя доступ к ним удаленным клиентам. Как правило, доступ к каталогу предоставляется вместе со

всеми его подкаталогами, то есть все дерево каталогов экспортируется как единое целое. Список экспортируемых сервером каталогов хранится в файле `/etc/exports`, таким образом, эти каталоги экспортируются автоматически при загрузке сервера. Клиенты получают доступ к экспортируемым каталогам, монтируя эти каталоги. Когда клиент монтирует (удаленный) каталог, этот каталог становится частью иерархии каталогов клиента, как показано на рисунке 5.9. (Каталоги показаны на рисунке в виде квадратов, а файлы — в виде кружков.)

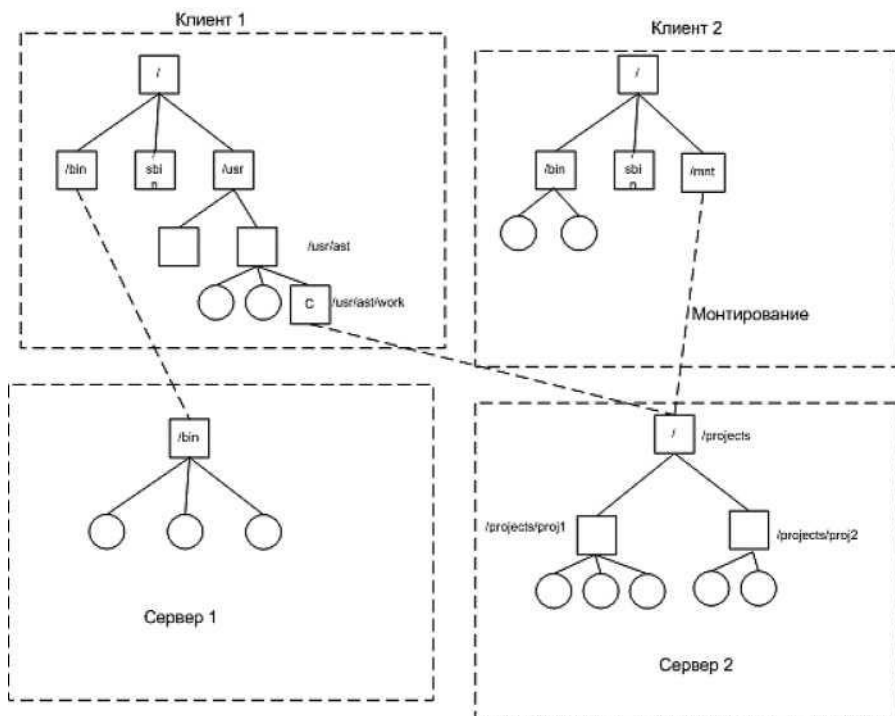


Рис. 5.9

В этом примере клиент 1 смонтировал каталог `bin` сервера 1 в своем собственном каталоге `bin`, поэтому он теперь может обращаться к оболочке сервера 1 как к `bin/sh`. У бездисковых рабочих станций часто есть только скелет файловой системы (в ОЗУ), а все свои файлы они получают с удаленных серверов, как в данном примере. Аналогично клиент 1 смонтировал каталог `projects` сервера 1 в своем каталоге `usr/ast/work`, поэтому он теперь может получать доступ к файлу, а как к `usr/ast/work/proj1/a`. Как видно из этого примера, у одного и того же файла могут быть различные имена на различных клиентах, так как их каталоги могут монтироваться в различных узлах каталоговых деревьев. Выбор узла, в котором монтируется удаленный каталог, целиком зависит от клиента. Сервер не знает, где клиент монтирует его каталог.

## Протоколы файловой системы NFS

Так как одна из целей файловой системы NFS заключается в

поддержке разнородных систем, в которых клиенты и серверы могут работать под управлением различных операционных систем и на различном оборудовании, существенно, чтобы интерфейс между клиентами и серверами был тщательно определен. Только в этом случае можно ожидать, что новый написанный клиент будет корректно работать с существующими серверами, и наоборот.

В файловой системе NFS эта задача выполняется при помощи двух протоколов клиент- сервер. Протокол— это набор запросов, посылаемых клиентами серверам, и ответов серверов, посылаемых клиентам.

Первый протокол NFS управляет монтированием каталогов. Клиент может послать серверу путь к каталогу и запросить разрешение смонтировать этот каталог где-либо в своей иерархии каталогов. Данные о месте, в котором клиент намеревается смонтировать удаленный каталог, серверу не посылаются, так как серверу это безразлично. Если путь указан верно, и указанный каталог был экспортирован, тогда сервер возвращает клиенту дескриптор файла, содержащий поля, однозначно идентифицирующие тип файловой системы, диск, i-узел каталога и информацию о правах доступа. Этот дескриптор файла используется последующими обращениями чтения и записи к файлам в монтированном каталоге или в любом из его подкаталогов.

Во время загрузки операционная система UNIX, прежде чем перейти в многопользовательский режим, запускает сценарий оболочки /etc/re. В этом сценарии можно разместить команды монтировки файловых систем. Таким образом, все необходимые удаленные файловые системы будут автоматически смонтированы прежде, чем будет разрешена регистрация в системе. В качестве альтернативы в большинстве версий системы UNIX также поддерживается автомонтировка. Это свойство позволяет ассоциировать с локальным каталогом несколько удаленных каталогов. Ни один из этих удаленных каталогов не монтируется во время загрузки операционной системы (не происходит даже контакта с сервером). Вместо этого при первом обращении к удаленному файлу (когда файл открывается) операционная система посылает каждому серверу сообщение. Побеждает ответивший первым сервер, чей каталог и монтируется.



У автомонтировки есть два принципиальных преимущества перед статической монтировкой с использованием файла /etc/rc. Во-первых, если один из серверов, перечисленных в файле /etc/re, окажется выключенным, запустить клиента будет невозможно, по крайней мере без определенных трудностей, задержки и большого количества сообщений об ошибках. Если пользователю в данный момент этот сервер не нужен, вся работа просто окажется напрасной. Во-вторых, предоставление клиенту возможности связаться с несколькими серверами параллельно позволяет значительно повысить устойчивость системы к сбоям (так как для работы достаточно всего одного работающего сервера) и улучшить показатели производительности (так как первый ответивший сервер скорее всего окажется наименее загруженным).

Второй протокол NFS предназначен для доступа к каталогам и файлам. Клиенты могут посылать серверам сообщения, содержащие команды управления каталогами и файлами, что позволяет им создавать, удалять, читать и писать файлы. Кроме того, у клиентов есть доступ к атрибутам файла, таким как режим, размер и время последнего изменения файла. Файловой системой NFS поддерживается большинство системных вызовов операционной системы UNIX, за исключением, как ни странно, системных вызовов open и close.

Файловая система NFS использует стандартный механизм защиты UNIX с битами gwx для владельца, группы и всех прочих пользователей. Изначально каждое сообщение с запросом просто содержало идентификаторы пользователя и группы вызывающего процесса, которые сервер NFS использовал для проверки прав доступа. Сервер NFS предполагал, что клиенты не будут его обманывать. Несколько лет опыта работы с файловой системой NFS показали, что такое предположение было (как бы это помягче назвать?) наивным. Сегодня для установки надежного ключа для аутентификации клиента и сервера при каждом запросе и каждом ответе можно использовать шифрование с открытым ключом. При этом злоумышленник не сможет выдать себя за другого клиента (или другой сервер), так как ему неизвестен секретный ключ этого клиента (или сервера).

### **Реализация файловой системы NFS**

Хотя реализация программ клиента и сервера не зависит от

протоколов NFS, в большинстве систем UNIX используется трехуровневая реализация, сходная с изображенной на рисунке 5.10. Верхний уровень представляет собой уровень системных вызовов. Он управляет такими системными вызовами, как `open`, `read` и `close`. После анализа системного вызова и проверки его параметров он вызывает второй уровень, уровень VFS (Virtual File System — виртуальная файловая система).

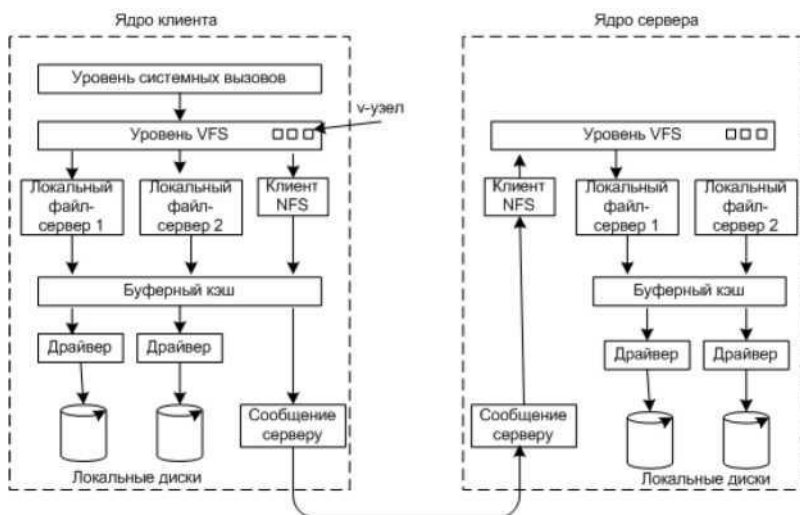


Рис. 5.10

Задача уровня VFS заключается в управлении таблицей, содержащей по одной записи для каждого открытого файла, аналогичной таблице `i`-узлов для открытых файлов в системе UNIX. В обычной системе UNIX `i`-узел однозначно указывается парой (устройство, номер `i`-узла). Вместо этого уровень VFS содержит для каждого открытого файла записи, называемые `v`-узлами (virtual `i`-node — виртуальный `i`-узел). `V`-узлы используются, чтобы отличать локальные файлы от удаленных. Для удаленных файлов предоставляется информация, достаточная для доступа к ним. Для локальных файлов записываются сведения о файловой системе и `i`-узле, так как современные системы UNIX могут поддерживать несколько файловых систем (например, V7, Berkeley FFS, `ext2fs`, `/proc`, FAT и т. д.). Хотя уровень VFS был создан для поддержки

файловой системы NFS, сегодня он поддерживается большинством современных систем UNIX как составная часть операционной системы, даже если NFS не используется.

Чтобы понять, как используются v-узлы, рассмотрим выполнение последовательности системных вызовов mount, open и read. Чтобы смонтировать файловую систему, системный администратор (или сценарий /etc/re) вызывает программу mount, указывая ей удаленный каталог, локальный каталог, в котором следует смонтировать удаленный каталог, и прочую информацию. Программа mount анализирует имя удаленного каталога и обнаруживает имя сервера NFS, на котором располагается удаленный каталог. Затем она соединяется с этой машиной, запрашивая у нее дескриптор удаленного каталога. Если этот каталог существует и его удаленное монтирование разрешено, сервер возвращает его дескриптор. Наконец, программа mount обращается к системному вызову mount, передавая ядру полученный от сервера дескриптор каталога.

Когда дескриптор файла используется в последующем системном вызове, например, read, уровень VFS находит соответствующий v-узел и по нему определяет, является ли он локальным или удаленным, а также какой i-узел или r-узел его описывает. Затем он посылает серверу сообщение, содержащее дескриптор, смещение в файле (хранящееся на стороне клиента, а не сервера) и количество байтов. Для повышения эффективности обмен информацией между клиентом и сервером выполняется большими порциями, как правило, по 8192 байт, даже если запрашивается меньшее количество байтов.

Когда сообщение с запросом прибывает на сервер, оно передается там уровню VFS, который определяет файловую систему, содержащую файл. Затем уровень VFS обращается к этой файловой системе, чтобы прочитать и вернуть байты. Эти данные передаются клиенту. После того как уровень VFS клиента получает 8-килобайтную порцию данных, которую запрашивал, он автоматически посылает запрос на следующую порцию, чтобы она была под рукой, когда понадобится. Такая функция, называемая опережающим чтением, позволяет значительно увеличить производительность.

При записи в удаленный файл проходит аналогичный путь от клиента к серверу. Данные также передаются 8-килобайтными

порциями. Если системному вызову `write` подается менее 8 Кбайт данных, данные просто накапливаются локально. Только когда порция в 8 Кбайт готова, она посылается серверу. Если файл закрывается, то весь остаток немедленно посылается серверу.

Кроме того, для увеличения производительности применяется кэширование, как в обычной системе UNIX. Серверы кэшируют данные, чтобы снизить количество обращений к дискам, но это происходит незаметно для клиентов. Клиенты управляют двумя кэшами, одним для атрибутов файлов (*i*-узлов) и одним для данных. Когда требуется либо *i*-узел, либо блок файла, проверяется, нельзя ли получить эту информацию из кэша. Если да, то обращения к сети можно избежать.

Хотя кэширование на стороне клиента во много раз повышает производительность, оно также приводит к появлению новых непростых проблем. Предположим, что два клиента сохранили в своих кэшах один и тот же блок файла и что один из них его модифицировал. Когда другой клиент считывает этот блок, он получает из кэша старое значение блока.

## Лекция 1.7. Безопасность в UNIX

Сообщество пользователей операционной системы UNIX состоит из некоторого количества зарегистрированных пользователей, у каждого из которых есть свой уникальный UID (User ID — идентификатор пользователя). UID представляет собой целое число в пределах от 0 до 65 535. Идентификатором владельца помечаются файлы, процессы и другие ресурсы. По умолчанию владельцем файла является пользователь, создавший этот файл, хотя владельца можно сменить.

Пользователи могут организовываться в группы, которые также нумеруются 16-разрядными целыми числами, называемыми GID (Group ID — идентификатор группы). Назначение пользователя к группе выполняется вручную системным администратором и заключается в создании нескольких записей в системной базе данных, в которой содержится информация о том, какой пользователь к какой группе принадлежит. Вначале пользователь мог принадлежать только к одной группе, но теперь в некоторых версиях системы UNIX пользователь может одновременно принадлежать к нескольким группам.

Основной механизм безопасности в операционной системе UNIX прост. Каждый процесс несет на себе UID и GID своего владельца. Когда создается файл, он получает UID и GID создающего его процесса. Файл также получает набор разрешений доступа, определяемых создающим процессом. Эти разрешения определяют доступ к этому файлу для владельца файла, для других членов группы владельца файла и для всех прочих пользователей. Для каждой из этих трех категорий определяется три вида доступа: чтение, запись и исполнение файла, что обозначается соответственно буквами r, w, x (read, write, execute). Возможность исполнять файл, конечно, имеет смысл только в том случае, если этот файл является исполняемой двоичной (Для возможности запуска сценария оболочки, представляющего собой, по сути не двоичный, а текстовый файл, у этого файла также нужно установить командой `chmod` биты r и x.) программой. Попытка запустить файл, у которого есть разрешение на исполнение, но который не является исполняемым (то есть не начинается с соответствующего заголовка), закончится ошибкой. Поскольку существует три категории пользователей и три

вида доступа для каждой категории, все режимы доступа к файлу можно закодировать 9 битами. Некоторые примеры этих 9-разрядных чисел и их значения показаны в таблице 5.2.

Таблица 5.2

Двоичное	Символьное	Разрешенный доступ
111000000	rwX-----	Владелец может читать, писать и исполнять
111111000	rwXrwX---	Владелец и группа могут читать, писать и исполнять
110100000	rw-r-----	Владелец может читать и писать, группа может читать
110100100	rw-r--r--	Владелец может читать и писать, все остальные могут читать
111101101	rwXr-xr-x	Владелец имеет все права, все остальные могут читать и исполнять
000000000		Ни у кого нет доступа
000000111	-----rwX	Только у посторонних есть доступ

Первые два примера в таблице понятны. В них к файлу предоставляется полный доступ для владельца файла и для его группы соответственно. В третьем примере группе владельца разрешается читать файл, но не разрешается его изменять, а всем посторонним запрещается всякий доступ. Вариант из четвертого примера часто применяется в тех случаях, когда владелец файла желает сделать файл с данными публичным. Пятый пример показывает режим защиты файла, представляющего собой опубликованную программу. В шестом примере доступ запрещен всем. Такой режим иногда используется для файлов-пустышек, применяемых для реализации взаимных исключений, так как любая попытка создания такого файла приведет к ошибке, если такой файл уже существует. Если несколько программ одновременно попытаются создать такой файл в качестве блокировки, только первой из них это удастся. Режим, показанный в последнем примере, довольно странный, так как он предоставляет всем посторонним пользователям больше привилегий, чем владельцу файла.

Тем не менее такой режим допустим. К счастью, у владельца файла всегда есть способ изменить в дальнейшем режим доступа к файлу, даже если ему будет запрещен всякий доступ к самому файлу.

Пользователь, UID которого равен 0, является особым пользователем и называется суперпользователем (superuser или root). Суперпользователь может читать и писать все файлы в системе, независимо от того, кто ими владеет и как они защищены. Процессы с UID=0 также обладают возможностью обращаться к небольшой группе системных вызовов, доступ к которым запрещен для обычных пользователей. Как правило, пароль суперпользователя известен только системному администратору.

Каталоги представляют собой файлы и обладают теми же самыми режимами защиты, что и обычные файлы. Отличие состоит в том, что бит x интерпретируется в отношении каталогов как разрешение не исполнения, а поиска в каталоге. Таким образом, каталог с режимом `gwxg-xg-x` позволяет своему владельцу читать, изменять каталог, а также искать в нем файлы, а всем остальным пользователям разрешаем только читать каталоги и искать файлы в них, но не создавать в них новые файлы или удалять файлы из этого каталога.

У специальных файлов, соответствующих устройствам ввода-вывода, есть те же самые биты защиты. Благодаря этому может использоваться тот же самый механизм для ограничения доступа к устройствам ввода-вывода. Например, владельцем специального файла принтера `/dev/lp` может быть суперпользователь (root) или специальный пользователь, демон принтера.

В операционной системе UNIX есть множество программ, владельцем которых является системный администратор, но у них установлен бит SETUID. Например, программе `passwd`, позволяющей пользователям менять свои пароли, требуется доступ записи к файлу паролей. Если разрешить изменять этот файл кому угодно, то ничего хорошего из этой затеи не получится. Вместо этого есть программа, владельцем которой выступает root, и у файла этой программы установлен бит SETUID. Хотя у этой программы есть полный доступ к файлу паролей, она изменит только пароль вызывавшего ее пользователя и не затронет остального содержимого файла.

Помимо бита SETUID, есть также еще и бит SETGID, работающий аналогично и временно предоставляющий пользователю рабочий GID программы. Однако на практике этот бит почти не используется.

### Системные вызовы безопасности в UNIX

Лишь небольшое число системных вызовов относятся к безопасности. Наиболее важные системные показаны в таблице 5.3. Чаще всего используется системный вызов `chmod`. С его помощью можно изменить режим защиты файла. Например, оператор `s = chmod("/usr/ast/newgame", 0755)` устанавливает для файла `newgame` режим доступа `rwxr-xr-x`, что позволяет запускать эту программу всем пользователям (обратите внимание, что `0755` представляет собой восьмеричную константу, что удобно в данном случае, так как биты защиты группируются тройками). Изменять биты защиты могут только владелец файла и суперпользователь.

Таблица 5.3

Системный вызов	Описание
<code>s=chmod(path, mode)</code>	Изменить режим защиты файла
<code>s=access(path, mode)</code>	Проверить разрешение доступа к файлу, используя действительные UID и GID
<code>uid=getuid()</code>	Получить действительный UID
<code>uid=geteuid()</code>	Получить рабочий UID
<code>gid=getgid</code>	Получить действительный GID
<code>gid=getegid()</code>	Получить рабочий GID
<code>s=chown(path, owner, group)</code>	Изменить владельца и группу
<code>s=setuid(uid)</code>	Установить UID
<code>s=setgid(gid)</code>	Установить GID

Системный вызов `access` проверяет, будет ли разрешен определенный тип доступа при заданных UID и GID. Этот системный вызов нужен, чтобы избежать появления брешей в системе безопасности. Он используется в программах с установленным битом SETUID, владельцем которых является `root`. Такие программы могут выполнять любые действия, поэтому им иногда бывает необходимо



определить, уполномочен ли вызвавший их пользователь на выполнение определенных действий. Программа не может просто попытаться получить требуемый доступ, так как любой доступ ей будет обязательно предоставлен.

Следующие четыре системных вызова возвращают значения реального и рабочего UID и GID. К последним трем системным вызовам разрешено обращаться только суперпользователю. Они изменяют владельца файла, а также UID и GID процесса.

### Реализация безопасности в UNIX

Когда пользователь входит в систему, программа регистрации login (которая является SETUID root) запрашивает у пользователя его имя и пароль. Затем она хэширует пароль и ищет его в файле паролей /etc/passwd, чтобы определить, соответствует ли хэш-код содержащимся в нем значениям (сетевые системы работают несколько по-иному). Хеширование uc2 п вызовы setuid и setgid, чтобы установить для себя UID и GID (как мы помним, она была запущена как SETUID root). После этого программа регистрации открывает клавиатуру для стандартного ввода (файл с дескриптором 0) и экран для стандартного вывода (файл с дескриптором 1), а также экран для вывода стандартного потока сообщений об ошибках (файл с дескриптором 2). Наконец, она выполняет оболочку, которую указал пользователь, таким образом, завершая свою работу.

## Модуль 2. ПРОГРАММИРОВАНИЕ В WINDOWS

### Лекция 2.1 Уровни программирования в Windows и структура ядра

#### История Windows

Операционные системы корпорации Microsoft для настольных и переносных компьютеров можно разделить на три семейства: MS-DOS, ConsumerWindows (Windows 95/98/Me) и Windows на основе NT. Ниже мы кратко опишем каждое из этих семейств.

#### MS-DOS

В начале 80-х годов компания IBM (которая на тот момент являлась самой большой и могущественной компьютерной компанией в мире) разрабатывала **персональный компьютер** на базе микропроцессора Intel 8088. С середины 70-х годов компания Microsoft стала лидирующим поставщиком языка программирования BASIC для 8-битных микрокомпьютеров на базе микропроцессоров 8080 и Z-80. Когда компания IBM обратилась в компанию Microsoft с предложением лицензировать BASIC для нового компьютера IBM PC, то компания Microsoft дала свое согласие и предложила, чтобы IBM связалась с компанией Digital Research для лицензирования ее операционной системы CP/M (поскольку в то время компания Microsoft операционными системами не занималась). Компания IBM так и сделала, но Гари Килдал (Gary Kildall), президент Digital Research, не нашел времени для встречи с компанией IBM, так что IBM опять вернулась в Microsoft. Очень скоро компания Microsoft купила клон операционной системы CP/M у местной компании Seattle Computer Products, перенесла ее на IBM PC и предоставила компании IBM лицензию на нее. Затем она была переименована в MS-DOS 1.0 (Microsoft Disk Operating System) и начала поставляться вместе с первыми IBMPC в 1981 году.

MS-DOS была 16-битной однопользовательской операционной системой с интерфейсом командной строки, которая работала в реальном режиме процессора и код которой занимал в памяти всего 8 Кбайт. В течение следующих десяти лет и персональные компьютеры, и MS-DOS продолжали развиваться, приобретая все больше функциональных возможностей. К 1986 году компания IBM создала компьютер PC/AT на базе процессора Intel 286, и система MS-DOS выросла до 36 Кбайт, однако она продолжала оставаться однозадачной операционной системой с интерфейсом командной строки.

### **Windows 95/98/Me**

Выход в августе 1995 года Windows 95 до сих пор не привел к полному вытеснению системы MS-DOS, хотя почти все функции MS-DOS были перенесены в Windows. Как Windows 95, так и новая версия MS-DOS 7.0 содержали большинство особенностей монолитной

операционной системы, включая виртуальную память и управление процессами. Однако операционная система Windows 95 не была полностью 32-разрядной программой. Она содержала большие куски 16-разрядного ассемблерного кода (а также немного 32-разрядного) и продолжала использовать файловую систему MS-DOS, практически со всеми ее ограничениями. Единственное значительное изменение файловой системы заключалось в добавлении длинных имен файлов к именам из 8 + 3 символа, разрешенным в MS-DOS.

Даже в выпуске Windows 98 в июне 1998 года MS-DOS все еще присутствовала (теперь она называлась версией 7.1) и состояла из 16-разрядного кода. Хотя теперь еще больше функций было переведено из MS-DOS-части системы в часть Windows, а поддержка больших дисковых разделов стала стандартом, по своему строению операционная система Windows 98 не сильно отличалась от

Windows 95. Основное отличие заключалось в интерфейсе пользователя, в большей степени интегрировавшем в себе Интернет и рабочий стол пользователя. Именно эта интеграция и привлекла внимание Министерства юстиции США, которое затем выдвинула против корпорации Microsoft иск, обвиняя корпорацию Microsoft в нарушении закона о монополиях. Корпорация

Microsoft яростно отрицала свою вину. В апреле 2000 года Федеральный суд США согласился с правительством.

Кроме того, что в ядре операционной системы Windows 98 содержался большой кусок 16-разрядного ассемблерного кода, у этой системы были еще две серьезные проблемы. Во-первых, хотя эта система была многозадачной, само ядро не было реентерабельным.

Если процесс был занят управлением какой-либо структурой данных в ядре, а затем его квант времени заканчивался и начинал работу другой процесс, новый процесс мог получить структуру данных в противоречивом состоянии. Чтобы предотвратить возникновение подобной проблемы, большинство процессов, зайдя в ядро, первым делом получали гигантский мьютекс, покрывающий всю систему, прежде чем приступить к каким-либо действиям. Хотя такой подход и устранял потенциальную угрозу противоречивости структур данных, он также уничтожал большую часть преимуществ многозадачности, так

как процессам, чтобы войти в ядро, часто приходилось ждать, пока другой процесс ядро покинет.

Во-вторых, у каждого процесса было 4-гигабайтное адресное пространство, в котором первые 2 Гбайт полностью принадлежали процессу. Однако следующий 1 Гбайт совместно использовался (с возможностью записи) всеми процессами системы. Нижний 1 Мбайт также совместно использовался всеми процессами, чтобы все они могли получать доступ к векторам прерывания MS-DOS. Эта возможность в свою очередь использовалась большинством приложений Windows 98. В результате ошибка в одной программе могла повредить ключевые структуры данных, используемые посторонними процессами, вследствие чего все эти процессы рушились. Что еще хуже, последний 1 Гбайт совместно использовался (с возможностью записи) процессами и ядром и содержал некоторые критические структуры данных. Любая программа, записав поверх этих структур какой-либо мусор (преднамеренно или нет), могла вывести из строя всю систему. Очевидное решение, заключающееся в том, чтобы не помещать структуры данных ядра в пространство пользователя, было неприменимо, так как старые программы, написанные для MSDOS, не смогли бы тогда работать в Windows 98.

### **Windows на базе NT (от 3 до 5)**

К концу 80-х годов прошлого века компания Microsoft поняла, что продолжать разрабатывать операционную систему на базе MS-DOS — это не лучший путь. Аппаратные возможности персональных компьютеров продолжали возрастать, и в конечном итоге рынок персональных компьютеров должен был столкнуться с рынком рабочих станций и серверов предприятий, где главенствующей операционной системой была UNIX. Компанию Microsoft также беспокоила конкурентоспособность микропроцессоров Intel, поскольку уже появилась архитектура RISC. Для решения этих проблем Microsoft наняла группу инженеров из компании DEC во главе с Дэйвом Катлером (Dave Cutler) — одним из главных разработчиков операционной системы VMS компании DEC. Ему было поручено разработать новую 32-битную операционную систему, в которой должен был быть

реализован интерфейс прикладного программирования OS/2 (который компания Microsoft разрабатывала в это время совместно с компанией IBM). В рабочих документах команды Катлера эта система первоначально называлась NT OS/2.

Система Катлера была названа NT— от слов New Technology («новая технология»), а также потому, что изначально она разрабатывалась под новый процессор Intel 860 с кодовым названием N10. NT проектировалась как система, способная к переносу на разные процессоры. Упор делался на безопасность и надежность, а также на совместимость с версиями Windows на базе MS-DOS. OnbiT работы Катлера в компании DEC дает о себе знать в самых разных аспектах, так что сходство между NT и другими разработанными Катлером операционными системами не является случайным.

Когда инженеры компании DEC (а затем и ее юристы) увидели, насколько похожа была NT на VMS (а также на ее никогда не выпускавшегося в свет последователя — MICA), между компаниями DEC и Microsoft возник спор относительно использования компанией Microsoft интеллектуальной собственности компании DEC. В итоге этот спор был урегулирован во внесудебном порядке. Кроме того, компания Microsoft согласилась на некоторое время предоставить поддержку NT для процессора Alpha компании DEC. Однако все это не помогло компании DEC преодолеть свою неприязнь к персональным компьютерам, которую очень характерно выразил в 1977 году основатель компании DEC мистер Кен Олсен. Он сказал буквально следующее: «Нет никаких причин, по которым кто-то может захотеть иметь дома компьютер». В 1998 году все то, что осталось от DEC, было продано компании Compaq, которая в свою очередь была позже куплена компанией Hewlett-Packard.

Знакомые с системами UNIX программисты считают, что архитектура NT совершенно другая. Это не только из-за влияния VMS, но и вследствие имевшихся (на момент разработки) отличий между компьютерными системами. UNIX была первоначально спроектирована в 70-х годах для 16-битных систем с одним процессором, малым количеством памяти и механизмом подкачки. В этих системах процесс был единицей параллелизма и структуры, а fork и exec были

малозатратными операциями (поскольку системы с подкачкой часто копируют процессы на диск). КТ была разработана в начале 90-х годов, когда обычными стали многопроцессорные 32битные компьютерные системы с большим количеством памяти и виртуальной памятью. В системе NT единицей параллелизма является поток, единицей структуры — динамическая библиотека, *a fork* и *exec* реализованы в виде одной операции (создание нового процесса и выполнение другой программы без предварительного копирования).

Первая версия Windows на базе NT (Windows NT 3.1) была выпущена в 1993 году. Она получила название 3.1 для того, чтобы соответствовать тогдашней текущей версии Windows 3.1 для домашних пользователей. Совместный проект с компанией IBM к этому времени развалился, так что, несмотря на по-прежнему имевшуюся поддержку интерфейсов OS/2, основными интерфейсами были 32-битные расширения интерфейсов прикладного программирования Windows под названием Win32. В период между началом разработки и первым выпуском в свет системы NT была выпущена Windows 3.0, которая имела исключительный коммерческий успех. Она также могла выполнять программы Win32, но для этого нужно было использовать библиотеку совместимости Win32s. Подобно первой версии Windows на базе MS-DOS, Windows на базе NT сначала также не имела успеха. Для NT требовалось больше памяти, 32-битных приложений было мало, а проблемы совместимости драйверов устройств и приложений привели к тому, что многие покупатели остались верны Windows на базе MS-DOS (которую компания Microsoft продолжала совершенствовать, выпустив в 1995 году Windows 95). Как и NT, Windows 95 предоставила настоящие 32-битные интерфейсы программирования, но она имела лучшую совместимость с уже существовавшим 16-битным программным обеспечением и приложениями. Неудивительно, что сначала NT добилась успеха на рынке серверов, где она конкурировала с VMS и NetWare. NT достигла заявленных целей по переносимости — в дополнительных выпусках 1994 и 1995 годов была добавлена поддержка архитектур MIPS и PowerPC. Первым существенным обновлением NT стала Windows NT 4.0 в 1996 году. Эта система имела мощь, безопасность и надежность NT и обеспечивала тот же самый

интерфейс пользователя, что и чрезвычайно популярная к тому моменту Windows 95.

## Программирование в Windows

Рассмотрим интерфейс прикладного программирования NT для системных вызовов (а затем и подсистему окружения Win32). Несмотря на наличие POSIX, практически весь написанный для Windows код использует либо напрямую Win32, либо .NET — которая сама работает поверх Win32.

На рисунке 6.1 показаны уровни операционной системы Windows. Под уровнями апплетов и графического интерфейса пользователя находятся интерфейсы программирования, на которых построены приложения. Как и в других операционных системах, они состоят в основном из библиотек кода (DLL), которые программы динамически используют для доступа к функциональным возможностям операционной системы. В Windows имеется также несколько интерфейсов программирования, которые реализованы как работающие в виде отдельных процессов службы. Приложения ведут обмен со службами пользовательского режима при помощи вызовов удаленных процедур (RPC).

Ядром операционной системы NT является программа режима ядра NTOS (ntoskrnl.exe), которая обеспечивает традиционные интерфейсы системных вызовов (на которых построена остальная часть операционной системы). В Windows только программисты компании *Microsoft* пишут на уровне системных вызовов. Опубликованные интерфейсы пользовательского режима реализованы при помощи работающих поверх уровней NTOS подсистем «API».

Первоначально NT поддерживала три «API»: OS/2, POSIX и Win32. От OS/2 отказались в Windows XP. POSIX также убрали, однако клиенты могут получить улучшенную систему POSIX (под названием Interix) как часть служб Services For UNIX (SFU) компании *Microsoft*, поэтому вся инфраструктура для поддержки POSIX в системе имеется. Большинство приложений Windows написано с использованием Win32 (хотя компания *Microsoft* поддерживает и другие интерфейсы прикладного программирования).



Рис. 6.1

*Client/Server Runtime Subsystem*, (CSRSS) или `csrss.exe`, входит в состав операционной системы Windows NT, и предоставляет собой часть пользовательского режима подсистемы окружения Win32. Когда процесс пользовательского режима вызывает функцию с участием консольных окон, создания процесса/потока, или поддержки Side-by-Side, библиотеки Win32 (`kernel32.dll`, `user32.dll`, `gdi32.dll`) вместо запроса системного вызова обращаются к процессу CSRSS путем быстрого локального межпроцессного вызова (Local Procedure Call), а затем CSRSS делает большую часть реальной работы, без того, чтобы подвергать опасности (компрометировать) ядро. Однако вызовы к оконному менеджеру и сервисам GDI обрабатываются драйверами режима ядра (`win32k.sys`) напрямую. Этот процесс взаимодействия элементов, реализующих подсистему окружения, показан на рисунке 6.2. Набор библиотек реализует как специфичные для подсистемы окружения функции операционной системы высокого уровня, так и заглушки процедур, которые ведут обмен между использующими подсистему процессами



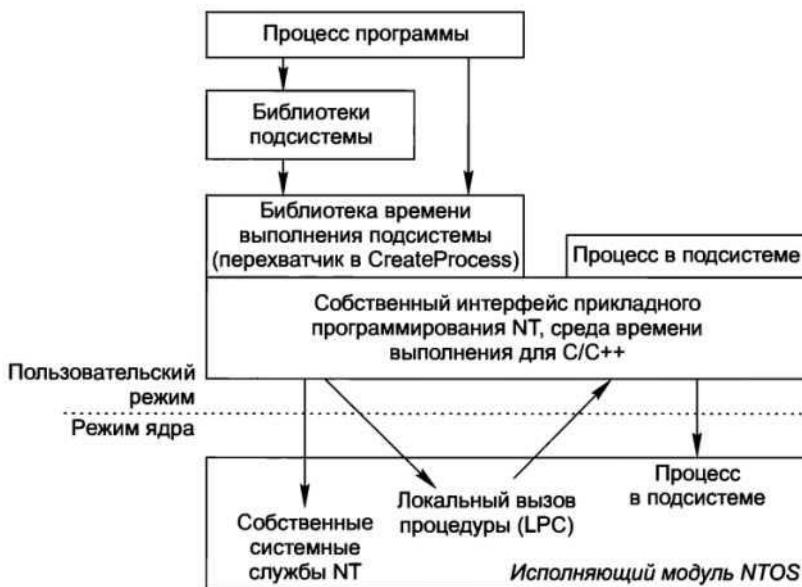


Рис. 6.2

Перехватчик в Win32 (в *CreateProcess*) определяет подсистему, которая требуется программе (для этого он изучает двоичный образ). Затем он делает запрос к *smss.exe* (*Session Manager Subsystem Service* — подсистеме управления сеансами) на запуск процесса подсистемы *csrss.exe* (если он еще не запущен). После этого за загрузку программы отвечает процесс подсистемы. Реализации других подсистем имеют аналогичные перехватчики (например, в системном вызове *exec* подсистемы POSIX). Ядро NT спроектировано таким образом, что оно имеет множество средств общего назначения, которые можно использовать для написания специфичных подсистем. Однако для правильной реализации любой подсистемы окружения необходим еще и специальный код. Например, собственный системный вызов *NtCreateProcess* реализует дублирование процесса для поддержки системного вызова *fork* подсистемы POSIX, а в ядре реализована для Win32 таблица строк специального типа (называемых элементами — *atoms*), которая эффективно позволяет процессам совместно использовать предназначенные только для чтения строки.

Собственные системные службы NT используют собственные

системные вызовы ядра NT и ее основных служб, таких как *smss.exe* и *lsass.exe* (локальное администрирование безопасности). Собственные системные вызовы содержат межпроцессные средства для управления виртуальными адресами, потоками, описателями, а также исключениями, созданными для запуска программ, написанных под конкретную подсистему окружения.

Работа библиотек DLL и подсистем окружения заключается в том, чтобы реализовать функциональные возможности опубликованного интерфейса, тем самым, скрывая истинный интерфейс системных вызовов от прикладных программ.

В отличие от Win32, система .NET не реализована как официальная подсистема окружения на собственных интерфейсах ядра NT. Вместо этого .NET построена поверх модели Win32.

### **Собственный интерфейс прикладного программирования NT**

Windows имеет набор системных вызовов, которые она может выполнять. Они реализованы на уровне исполняющего модуля NTOS, который работает в режиме ядра. Компания *Microsoft* почти не публиковала никаких подробностей об этих собственных системных вызовах. Они используются программами низкого уровня (в основном это службы и подсистемы), которые поставляются как часть операционной системы, а также драйверами устройств, работающими в режиме ядра. Собственные системные вызовы NT мало изменяются от версии к версии, однако компания *Microsoft* решила не предавать их огласке — чтобы приложения для Windows делались на основе Win32 и таким образом имели бы больше шансов работать и в Windows на базе MS-DOS, и в Windows на базе NT (поскольку Win32 API есть в них обеих).

Каждый объект режима ядра имеет определяемый системой тип, имеет строго заданные для него операции и занимает область в памяти ядра. Несмотря на то что программы пользовательского режима могут выполнять эти операции (делая системные вызовы), они не могут получить данные напрямую.

*NtCreateThread* принимает *ProcHandle*, поскольку он может создать поток в любом процессе, для которого у вызывающего процесса есть

описатель (при наличии достаточных прав доступа). Аналогичным же образом, *NtAllocateVirtualMemory*, *NtMapViewOfSection*, *NtReadVirtualMemory* и *NtWriteVirtualMemory* позволяют процессу работать не только с его собственным адресным пространством, но и выделять виртуальные адреса, отображать сегменты, а также читать или писать в виртуальную память других процессов. *NtCreateFile*—это собственный вызов API для создания нового файла или для открытия уже существующего *NtDuplicateObject*— это вызов API для дублирования описателей из одного процесса в другой.

Диспетчер объектов позволяет давать объектам имена (при их создании), а затем открывать их по имени (чтобы получить описатели для объектов). Для представления названий в **пространстве имен NT (NT namespace)** диспетчер объектов использует **Unicode**. В отличие от UNIX, система NT обычно не делает различия между верхним и нижним регистрами (она *сохраняет регистр, но не чувствительна к нему*). Пространство имен NT является иерархической коллекцией каталогов, символических ссылок и объектов.

Диспетчер объектов предоставляет также унифицированные средства для синхронизации, обеспечения безопасности и управления существованием объекта. Будут ли доступны пользователям данного конкретного объекта те средства общего назначения, которые предоставляет диспетчер объектов, — это определяется компонентами исполняющей подсистемы, поскольку именно они предоставляют те интерфейсы прикладного программирования, которые манипулируют этими типами объектов.

Не только приложения используют объекты, которыми управляет диспетчер объектов. Сама операционная система также может создавать и использовать объекты — и делает это очень интенсивно. Большинство этих объектов создается для того, чтобы позволить одному компоненту системы сохранить на значительный период времени некоторую информацию или передать некоторую структуру данных другому компоненту (и при этом использовать имеющуюся в диспетчере объектов поддержку именования и времени существования). Например, при обнаружении устройства для его представления и для логического описания подключения устройства к системе создается один или

несколько **объектов устройств (device objects)**. Для управления устройством загружается драйвер устройства и создается **объект драйвера (driver object)**, в котором представлены свойства устройства и даны указатели на реализованные в нем функции для обработки запросов ввода-вывода. После этого ссылка на драйвер внутри операционной системы делается при помощи использования его объекта. К драйверу можно также обратиться непосредственно (по имени), а не косвенно (через управляемые им устройства)— например, для установки параметров (которые определяют его работу) из режима пользователя.

Именованный объект может быть помечен как *постоянный (permanent)* — это означает, что он продолжает существовать до явного его удаления или до перезагрузки системы (даже если ни один процесс не имеет описателя для этого объекта). Такие объекты могут даже расширять пространство имен NT, предоставляя процедуры для анализа, вроде точек монтирования в UNIX. Файловые системы и реестр используют это средство для монтирования томов и «ульев» в пространство имен NT. Обращение к объекту устройства тома дает доступ к самому тому, но объект устройства также предоставляет неявное монтирование тома в пространство имен NT. К файлам тома можно обращаться путем присоединения имени файла на томе в конец имени объекта устройства (для этого тома).

Постоянные имена также могут использоваться для представления объектов синхронизации и совместно используемой памяти (чтобы их можно было совместно использовать процессам без постоянного их воссоздания при запуске и останове процессов). Объектам устройств (и часто — объектам драйверов) даются постоянные имена, что придает им некоторые свойства сохраняемости специальных узлов i-nodes, содержащихся в каталоге */dev* систем UNIX.

## **Интерфейс прикладного программирования Win32**

Вызовы функций Win32 называются **интерфейсом прикладного программирования Win32 (Win32 API)**. Эти интерфейсы раскрыты и полностью документированы. Они реализованы как библиотечные процедуры, которые заключают в оболочку собственные системные

вызовы NT (используемые для выполнения задачи), либо (в некоторых случаях) выполняют эту задачу прямо в пользовательском режиме. Несмотря на то, что собственные интерфейсы прикладного программирования NT не опубликованы, большая часть предоставляемой ими функциональности доступна через Win32 API. При выходе новых версий Windows уже существующие вызовы Win32 API меняются редко (хотя в API добавляется много новых функций).

Философия интерфейсов прикладного программирования Windows сильно отличается от философии UNIX. В системах UNIX функции операционной системы просты, имеют мало параметров и мало таких мест, где одну и ту же операцию можно выполнить несколькими путями. Win32 предоставляет очень подробные интерфейсы со множеством параметров (часто есть три-четыре способа выполнения одного и того же), причем используется смесь функций низкого и высокого уровней (вроде *CreateFile* и *CopyFile*).

Это означает, что Win32 предоставляет очень богатый набор интерфейсов, который, однако, является очень сложным из-за плохого разбиения на уровни этой системы, где в одном API смешаны функции высокого и низкого уровня. Для нашего изучения операционных систем существенны только те функции низкого уровня Win32 API, которые являются оболочками собственного NT API, поэтому именно на них мы и сосредоточимся.

Win32 имеет вызовы для создания и управления процессами и потоками. Существует также много таких вызовов, которые относятся к межпроцессному обмену (такие, как создание, уничтожение и использование мьютексов, семафоров, событий, портов обмена и прочих объектов IPC).

Windows реализует отображение файлов в память как объекты режима ядра, представленные описателем. Подобно большинству описателей, отображения файлов могут быть сдублированы в другие процессы. Каждый из этих процессов может отобразить отображение файла в свое адресное пространство (так, как считает нужным). Это полезно для совместного использования памяти разными процессами (без необходимости создавать для

этого файлы). На уровне NT отображения файлов (сегменты) могут

быть сделаны постоянными в пространстве имен NT, после чего к ним можно обращаться по имени.

Для многих программ очень важным является файловый ввод-вывод. В основном представлении Win32 файл является просто линейной последовательностью байтов. Win32 предоставляет более 60 вызовов для создания и уничтожения файлов и каталогов, открытия и закрытия файлов, чтения и записи в них, запроса и установки атрибутов файлов, блокировки диапазона байтов, а также для многих других основных операций по организации файловой системы и обращению к файлам.

Модель ввода-вывода низкого уровня в Windows принципиально асинхронная. После начала операции ввода-вывода системный вызов может сделать возврат и позволить потоку (который инициировал ввод-вывод) продолжить работу параллельно с операцией ввода-вывода. Windows также позволяет программам указать при открытии файла, что ввод-вывод должен быть синхронным, многие библиотечные функции (библиотеки C и многие вызовы Win32) также задают синхронный ввод-вывод для совместимости или для упрощения модели программирования. В этих случаях исполняющая подсистема будет явно синхронизироваться с завершением ввода-вывода (перед возвращением в пользовательский режим).

Еще одна область, для которой в Win32 есть вызовы, — это безопасность. Каждый поток ассоциируется с объектом режима ядра, называемым **маркером (token)**, который предоставляет информацию об идентификаторе и привилегиях потока. Каждый объект может иметь **ACL (Access Control List — список управления доступом)**, который очень подробно указывает, какие пользователи могут к объекту обращаться и какие операции они могут с ним

выполнять. Такой подход обеспечивает тонкую настройку безопасности, при которой конкретным пользователям может быть предоставлен конкретный доступ к любому объекту.

Модель безопасности способна к расширению, что позволяет приложениям добавлять новые правила безопасности (такие, как ограничение часов доступа).

В дополнение к описанным нами системным интерфейсам низкого

уровня Win32 API поддерживает также многие функции по работе с графическим интерфейсом пользователя (и в том числе все вызовы для управления графическим интерфейсом системы). Имеются вызовы для создания, уничтожения, управления и использования окон, меню, панелей инструментов, строк состояния, полос прокрутки, диалоговых окон, значков (и множества других имеющихся на экране элементов). Есть вызовы для рисования геометрических фигур, их закраски, управления палитрами используемых ими цветов, работы со шрифтами, размещения значков на экране. И наконец, есть вызовы для работы с клавиатурой, мышью и другими устройствами ввода, а также для аудио, печати и прочих устройств вывода.

Операции графического интерфейса пользователя работают напрямую с драйвером win32k.sys (при помощи специальных интерфейсов для доступа к этим функциям режима ядра из библиотек пользовательского режима). Поскольку эти вызовы не влекут за собой выполнения базовых системных вызовов исполняющей подсистемы NTOS, говорить о них мы не будем.

## **Реестр**

Операционной системе Windows приходится управлять большими объемами информации об оборудовании, программном обеспечении и пользователях. В Windows 3.x эта информация хранилась в сотнях файлов с расширением .ini (initialization — инициализация), разбросанных по всему диску. Начиная с Windows 95, почти вся информация, необходимая для загрузки и конфигурирования системы и настройки ее под конкретного пользователя, была собрана в одной большой центральной базе данных, называемой реестром.

Для начала отметим, что, хотя многие части системы Windows сложны и запутанны, реестр является одним из самых запутанных мест в Windows, и похожие на шифр условные обозначения отнюдь не помогают в нем разобраться. К счастью, этой теме были посвящены целые книги. Сама же идея реестра очень проста. Он состоит из набора каталогов, каждый из которых содержит либо подкаталоги, либо записи. В этом он похож на файловую систему, содержащую очень маленькие файлы.

Реестр организован в отдельные тома, называемые **ульями (hives)**, однако, фактически эти каталоги верхнего уровня начинаются со строки HKEY, что означает «дескриптор ключа». У подкаталогов, как правило, лучшие имена, хотя и не всегда. Каждый улей хранится в отдельном файле (находящемся в каталоге C:\Windows\system32\config\загрузочного тома). Когда Windows загружается, улей SYSTEM грузится в память той же самой программой загрузки, которая загружает ядро и прочие файлы загрузки (такие, как загрузочные драйверы) с загрузочного тома.

В нижней части этой иерархической структуры располагаются записи, называемые **значениями**, содержащие информацию. У каждого значения три части: имя, тип и данные. Имя представляет собой просто строку формата Unicode, часто default, если каталог содержит всего одно значение. Тип может быть одним из 11 стандартных типов. Среди наиболее часто используемых типов строка формата Unicode, 32-разрядное целое число, двоичное число произвольной длины и символьная ссылка на каталог или запись реестра. Символьные имена полностью аналогичны символьным ссылкам в файловых системах или значкам на рабочем столе Windows: они позволяют одной записи указывать на другую запись или каталог. Символьная ссылка также может использоваться как ключ, то есть то, что кажется каталогом, может на самом деле оказаться ссылкой на другой каталог.

Первый улей (то есть каталог), HKEY\_LOCAL\_MACHINE, является, вероятно, наиболее важным, так как в нем содержится вся информация о локальной системе. У этого улья есть пять подключей (подкаталогов).

Подключ HARDWARE содержит множество подключей, в которых хранится вся информация об аппаратном обеспечении, например, какой драйвер какой частью аппаратуры управляет. Эта информация формируется на лету менеджером устройств plug-and-play во время загрузки системы. В отличие от других подключей этот подключ не хранится на диске.

Подключ SAM (Security Account Manager — администратор учетных данных в системе безопасности) хранит имена пользователей, групп, пароли, а также другую информацию об учетных записях пользователей, необходимую для регистрации в системе.



Подключ SECURITY содержит данные об общей политике безопасности, например, минимальную длину паролей, допустимое количество неудачных попыток регистрации и т. д.

Подключ SOFTWARE — это то место, в котором производители программного обеспечения хранят настройки программ. Если у пользователя в системе установлены программы Acrobat, Photoshop и Premiere компании Adobe, там будет содержаться подключ Adobe, под которым будут располагаться подключи для хранения Acrobat; Photoshop, Premiere и других программных продуктов компании Adobe. Записи, в этих подкаталогах могут хранить все, что программистам компании Adobe понадобится поместить туда, — это номера версии и сборки, а также параметры установки пакета, сведения о драйверах и т. д. Наличие системного реестра избавляет их от хлопот по выдумыванию собственного метода хранения подобной информации.

Подключ SYSTEM содержит главным образом информацию о загрузке системы, например, список драйверов, которые требуется загрузить. Здесь также хранится список служб (демонов), которые должны быть запущены после загрузки, и сведения об их конфигурации.

Улей HKEY\_USERS содержит профили для каждого пользователя. Все выбираемые пользователем настройки и параметры хранятся здесь. Когда пользователь изменяет какой-либо параметр при помощи панели управления, скажем, цветовую схему рабочего стола, новые установки записываются сюда. Многие программы в панели управления главным образом занимаются сбором информации, получая ее от пользователя, и сохраняют полученные сведения в реестре. Некоторые из подключей ключа HKEY\_USERS показаны в табл. 11.4 и практически не требуют дополнительных комментариев. Другие подключи, например, Software, содержат удивительно большое количество подключей, даже если в системе не установлено никаких пакетов программного обеспечения.

Улей HKEY\_PERFORMANCE\_DATA не содержит ни данных, считываемых с диска, ни данных, собираемых менеджером plug-and-play. Вместо этого данный ключ предоставляет окно в операционную систему. Сама система содержит сотни счетчиков для мониторинга производительности системы. К таким счетчикам можно получить доступ через этот ключ реестра.

При обращении к подключу запускается специальная процедура, собирающая и возвращающая информацию (возможно, считывающая один или несколько счетчиков и объединяющая их определенным способом). В редакторе regedit этот ключ не виден. Вместо редактора нужно воспользоваться одной из утилит измерения производительности. Существует множество подобных программ, они либо поставляются на диске с Windows, либо входят в пакет Resource Kit, либо производятся другими компаниями.

Остальных трех ульев верхнего уровня на самом деле не существует. Каждый из них представляет собой символьную ссылку на определенный подключ реестра. Самым интересным является HKEY\_CLASSES\_ROOT. Он указывает на каталог, управляющий объектами COM (Component Object Model— модель компонентных объектов), а также занимающийся установкой соответствий между расширениями файлов и программами. Когда пользователь дважды щелкает мышью на файле, имя которого оканчивается на, скажем, .doc, программа, перехватывающая щелчок мыши, смотрит в это место реестра, чтобы определить, которую программу следует запустить (вероятно, Microsoft Word). Т.о. улей HKEY\_CLASSES\_ROOT содержит целую базу данных распознаваемых расширений и соответствующих им программ.

Улей HKEY\_CURRENT\_CONFIG представляет собой ссылку на подключ, содержащий информацию о текущей конфигурации аппаратного обеспечения. Пользователь может сформировать несколько конфигураций аппаратуры, например, отключая различные устройства, чтобы проверить, не они ли служили причиной странного поведения системы. Этот улей указывает на текущую конфигурацию.

Улей HKEY\_CURRENT\_USER указывает на настройки текущего пользователя, что позволяет быстро находить их.

Ни один из последних трех ключей в действительности ничего не добавляет, так как эта информация уже была доступна (хотя и не в столь удобном виде). Таким образом, хотя редактор regedit перечисляет пять ульев верхнего уровня, на самом деле существуют только три каталога верхнего уровня, один из которых не отображается.

Когда система выключается, большая часть информации реестра

сохраняется на диске в файлах ульев (COMPONENTS, DEFAULT, SAM, SECURITY, SOFTWARE, SYSTEM). Поскольку их целостность критична для правильного функционирования системы, то автоматически выполняется резервное копирование и сделанные в метаданных записи сбрасываются на диск (во избежание повреждения в случае отказа системы). Потеря реестра приводит к необходимости повторной установки всего программного обеспечения системы.

### **Структура операционной системы**

Погрузимся в самые нижние уровни операционной системы — те, которые работают в режиме ядра. Центральный уровень — это само ядро, которое загружается из файла `ntoskrnl.exe` (при загрузке Windows). Часть NT, работающая в режиме ядра (называемая NTOS), имеет два уровня: исполняющий (`executive`), в котором содержится большая часть служб (иногда его называют исполняющей системой NTOS), и меньший по размеру уровень, который (собственно) называется ядром (ядро внутри ядра это сильно), и реализует планирование потоков, абстракции синхронизации, обработчики ловушек, прерывания и прочие аспекты управления процессором.

Уровни режима ядра системы NT показаны на рисунке 6.3. Уровень ядра NTOS показан над уровнем исполняющей системы, поскольку он реализует механизмы ловушек и прерываний, которые используются для перехода из пользовательского режима в режим ядра. Верхний уровень на рис. 6.3 это системная библиотека `ntdll.dll`, которая фактически работает в пользовательском режиме. Эта системная библиотека содержит некоторые вспомогательные функции для библиотек компилятора (времени исполнения и низкого уровня) — аналогично библиотеке `libc` системы UNIX. `Ntdll.dll` содержит также и специальные точки входа, используемые ядром для инициализации потоков, а также диспетчеризации исключений и вызовов APC (Asynchronous Procedure Calls — асинхронные вызовы процедур) пользовательского режима. Поскольку системная библиотека так тесно интегрирована в работу ядра, то каждый созданный NTOS процесс пользовательского режима имеет отображение на `ntdll` по одному и тому

же фиксированному адресу. Когда NTOS инициализирует систему, он создает объектсегмент, который будет использоваться для отображения ntdll, а также записывает адреса точек входа ntdll, используемых ядром. Ниже уровня ядра и исполняющего уровня NTOS находится программное обеспечение под названием HAL (Hardware Abstraction Layer — уровень абстрагирования оборудования), который абстрагирует низкоуровневые детали оборудования (вроде доступа к регистрам устройств и работы в режиме DMA), а также и то, как BIOS представляет конфигурационную информацию и работает с различными чипсетами (например, с контроллерами прерываний). BIOS поставляется несколькими компаниями, он интегрируется в энергонезависимую память EEPROM (которая находится на системной плате компьютера). Другой важный компонент режима ядра — это драйверы устройств. Windows использует драйверы устройств для всех тех компонентов режима ядра, которые не являются частью NTOS или HAL. Сюда входят: файловые системы и стеки сетевых протоколов, а также расширения ядра, такие как антивирусы и программное обеспечение DRM (Digital Rights Management — управление цифровыми правами), а также драйверы для управления физическими устройствами, для интерфейсов с системными шинами и т. Д.



Рис 6.3

Компоненты ввода-вывода и виртуальной памяти совместно загружают (и выгружают) драйверы устройств в память ядра и связывают их с уровнями NTOS и HAL. Диспетчер ввода-вывода обеспечивает интерфейсы, которые позволяют обнаруживать устройства, организовывать их и работать с ними — и в том числе загружать соответствующий драйвер устройства. Большая часть конфигурационной информации для управления устройствами и драйверами находится в улье SYSTEM реестра. Компонент Plug-and-Play диспетчера ввода-вывода поддерживает информацию по обнаруженному оборудованию в улье HARDWARE, который поддерживается в памяти (а не на диске) — поскольку он полностью создается заново при каждой загрузке системы.

### **Уровень аппаратных абстракций**

Аппаратные подробности организации памяти на больших серверах (или наличие аппаратных примитивов синхронизации) могут повлиять также и на более высокие уровни системы. Например, диспетчер виртуальной памяти NT и уровень ядра знают об аппаратных подробностях кэширования и локальности памяти. NT везде использует примитивы синхронизации compare&swap, и ее будет очень трудно перенести на систему, в которой их нет. И наконец, в системе есть очень много зависимостей от порядка байтов в словах. На всех системах, куда была портирована NT, оборудование имеет прямой порядок байтов. Кроме этих серьезных проблем переносимости, имеется также большое количество более мелких проблем (даже между системными платами разных производителей). Разница в процессорах влияет на реализацию примитивов синхронизации (таких, как спин-блокировка). Есть несколько семейств чипсетов, которые имеют разные приоритеты аппаратных прерываний, способы доступа к регистрам устройств ввода-вывода, управление передачей в режиме DMA, управление таймерами и часами реального времени, синхронизацию многопроцессорности, работу с BIOS, например с ACPI (Advanced Configuration and Power Interface — усовершенствованный интерфейс управления конфигурированием и энергопотреблением) и т. д. Компания Microsoft

предприняла серьезную попытку сокрытия этих особенностей компьютерных систем внутри тонкого уровня в самом низу — он называется HAL (как уже упоминалось ранее). Задача HAL — представить остальной части операционной системы некое абстрактное оборудование, которое скрывает специфичные подробности (версию процессора, чипсет и прочие особенности конфигурации). Эти абстракции уровня HAL представлены в форме независимых от компьютера служб (вызовов процедур и макросов), которые могут использоваться драйверами и NTOS. При использовании служб HAL и отсутствии прямых обращений к оборудованию для драйверов и ядра требуется меньше изменений при переносе на новые процессоры — и почти во всех случаях они могут работать без всякой модификации на всех системах с одинаковой архитектурой процессора (несмотря на разные их версии и разные чипсеты). HAL не обеспечивает абстракций или служб для конкретных устройств ввода-вывода, таких как клавиатура, мышь, диски или устройство управления памятью. Эти средства разбросаны по компонентам режима ядра, и без использования HAL при переносе пришлось бы модифицировать большое количество кода (даже при небольших отличиях в оборудовании). Перенос самого HAL прост, поскольку весь машинозависимый код сконцентрирован в одном месте и цели переноса хорошо определены: реализация всех служб HAL. Для многих версий компания Microsoft поддерживала набор HAL Development Kit, который позволял производителям систем создавать свой собственный HAL, чтобы прочие компоненты ядра могли работать на новых системах без всякой модификации (при условии, что изменения в оборудовании не были слишком значительными). В качестве примера того, что делает HAL, рассмотрим сравнение ввода-вывода с отображением в память и портов ввода-вывода.

Компоненты ядра иногда должны синхронизироваться на очень низком уровне, особенно для предотвращения «состояния гонки» в многопроцессорных системах. HAL обеспечивает примитивы для управления такой синхронизацией (такие, как спин-блокировки), когда один процессор просто ждет освобождения ресурса, удерживаемого другим процессором, — особенно в таких ситуациях, когда ресурс

удерживается всего на несколько машинных команд. И наконец, после загрузки системы HAL опрашивает BIOS и изучает конфигурацию системы, чтобы определить, какие шины и устройства ввода-вывода содержит система и как они были сконфигурированы. Эта информация затем помещается в реестр. На рис. 11.5 дана сводка некоторых вещей, которые делает HAL.

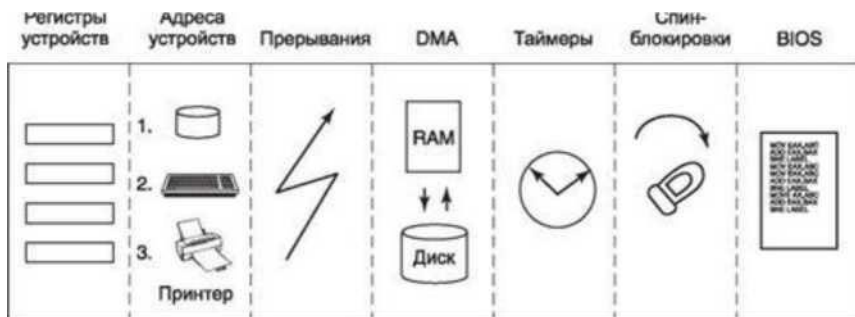


Рис. 6.4

## Уровень ядра

Над уровнем HAL находится NTOS, состоящий из двух уровней: ядро и исполняющая система. «Ядро» в Windows является сбивающим с толку термином. Этот термин может относиться ко всему коду, который работает в режиме ядра процессора. Он может также относиться к файлу `ntoskrnl.exe`, который содержит NTOS — основную часть операционной системы Windows. Может также относиться к уровню ядра внутри NTOS (именно так мы используем этот термин). Он даже используется для именования библиотеки `kernel32.dll` (которая обеспечивает оболочки для собственных системных вызовов) пользовательского режима Win32. В операционной системе Windows уровень ядра (показан на рис. 11.4 над исполняющим уровнем) предоставляет набор абстракций для управления процессором. Центральной абстракцией является поток, однако ядро реализует также обработку исключений, ловушки и несколько видов прерываний. Создание и уничтожение структур данных (которые поддерживают потоки) реализовано в исполнительном уровне. Уровень ядра отвечает за планирование и синхронизацию потоков. Поддержка потоков на отдельном уровне

позволяет исполнительному уровню реализовываться при помощи той же самой модели многопоточности с вытеснением, которая использована для написания параллельного кода пользовательского режима (хотя примитивы синхронизации в исполнительном уровне узкоспециализированные). Планировщик потоков ядра отвечает за то, какие потоки выполняются на процессорах системы. Каждый поток выполняется до тех пор, пока прерывание таймера не сигнализирует о том, что пора переключаться на другой поток (квант закончился), или до тех пор, когда потоку нужно ждать какого-то события (завершения ввода-вывода или снятия блокировки), либо до тех пор, пока работоспособным не станет поток с более высоким приоритетом (которому требуется процессор). При переключении с одного потока на другой планировщик обеспечивает сохранение регистров и прочего состояния оборудования. Затем планировщик выбирает для выполнения на процессоре другой поток и восстанавливает ранее сохраненное состояние (для выбранного потока). Если следующий подлежащий выполнению поток находится в другом адресном пространстве (то есть принадлежит другому процессу) — не в том, где находился поток, с которого произошло переключение, — то планировщик должен также изменить адресное пространство. Подробности алгоритма планировщика мы будем обсуждать далее, когда перейдем к процессам и потокам. Кроме обеспечения абстракции оборудования и работы с переключениями потоков, уровень ядра имеет еще одну ключевую функцию: предоставление поддержки низкого уровня для двух классов механизмов синхронизации: объектов управления и диспетчеризации. Объекты управления (control objects) — это структуры данных для управления процессором, которые уровень ядра предоставляет как абстракции для уровня исполняющей системы NTOS. Объекты диспетчеризации (dispatcher objects) — это класс обычных объектов исполняющей системы, который использует общую структуру данных для синхронизации.

### **Отложенные вызовы процедур**

Объекты управления включают: объекты-примитивы для потоков, прерываний, таймеров, синхронизации, профилирования, а также два



специальных объекта для реализации DPC и APC. Объекты DPC (Deferred Procedure Call — отложенный вызов процедуры) используются для уменьшения времени выполнения ISR (Interrupt Service Routines — процедура обслуживания прерываний), которая запускается по прерыванию от устройства. Оборудование системы присваивает прерываниям аппаратный уровень приоритета. Процессор также связывает уровень приоритета с выполняемой им работой. Процессор реагирует только на те прерывания, которые имеют более высокий приоритет, чем используемый им в данный момент. Нормальный уровень приоритета (и в том числе уровень приоритета всего пользовательского режима) — это 0. Прерывания устройств происходят с уровнем 3 или более высоким, а ISR для прерывания устройства обычно выполняется с тем же уровнем приоритета, что и прерывание (чтобы другие менее важные прерывания не происходили при обработке более важного прерывания). Если ISR выполняется слишком долго, то обслуживание прерываний более низкого приоритета будет отложено, что, возможно, приведет к потере данных или замедлит ввод-вывод системы. В любой момент времени может выполняться несколько ISR, при этом каждая последующая ISR будет возникать от прерываний с все более высоким уровнем приоритета. Для уменьшения времени обработки ISR выполняются только критические операции, такие как запись результатов операций ввода-вывода и повторная инициализация устройства. Дальнейшая обработка прерывания откладывается до тех пор, пока уровень приоритета процессора не снизится и не перестанет блокировать обслуживание других прерываний. Объект DPC используется для представления подлежащей выполнению работы, а ISR вызывает уровень ядра для того, чтобы поставить DPC в список DPC конкретного процессора. Если DPC является первым в списке, то ядро регистрирует специальный аппаратный запрос на прерывание процессора с уровнем 2 (на котором NT вызывает уровень DISPATCH). Когда завершается последняя из существующих ISR, то уровень прерывания процессора падает ниже 2 и это разблокирует прерывание для обработки DPC. ISR для прерывания DPC обработает каждый из объектов DPC (которые ядро поставило в очередь). Методика использования программных прерываний для откладывания обработки

прерываний является признанным методом уменьшения латентности ISR. UNIX и другие системы начали использовать отложенную обработку в 1970-х годах

(для того, чтобы справиться с медленным оборудованием и ограниченным размером буферов последовательных подключений к терминалам). ISR получала от оборудования символы и ставила их в очередь. После того как вся обработка прерываний высшего уровня была закончена, программное прерывание запускало ISR с низким приоритетом для обработки символов (например, для реализации возврата курсора на одну позицию — для этого на терминал посылался управляющий символ для стирания последнего отображенного символа, и курсор перемещался назад). Аналогичным примером в современной системе Windows может служить клавиатура. После нажатия клавиши клавиатурная ISR читает из регистра код клавиши, а затем опять разрешает клавиатурное прерывание, но не делает обработку клавиши немедленно.

APC пользовательского режима вызывает назначенную приложением процедуру пользовательского режима, но только тогда, когда целевой поток заблокирован в ядре и помечен как готовый принимать APC. Ядро прерывает ожидание потока и делает возврат в пользовательский режим, но уже со стеком пользовательского режима и регистрами, модифицированными для выполнения процедуры диспетчеризации APC из системной библиотеки ntdll.dll. Процедура диспетчеризации APC вызывает процедуру пользовательского режима, которую приложение связало с операцией ввода-вывода. Исполняющая система NTOS использует APC и для других операций (помимо завершения вводавывода). Поскольку механизм APC тщательно спроектирован для того, чтобы поставлять APC только тогда, когда это можно сделать безопасно, то его можно использовать для безопасного завершения потоков. Если время для завершения потока не подходящее, то поток объявляет, что он вошел в критическую область, и откладывает доставку APC до момента выхода из нее. Потоки ядра помечают себя в качестве входящих в критическую область (для откладывания доставки APC) перед получением блокировок или других ресурсов, чтобы их нельзя было завершить во время удержания ими ресурсов.

## Объекты диспетчеризации

Еще один тип объектов синхронизации — объекты диспетчеризации. Это любой из обычных объектов режима ядра (на которые пользователи могут ссылаться при помощи описателей), который содержит структуру данных под названием «заголовок диспетчеризации» (рисунок 6.5)

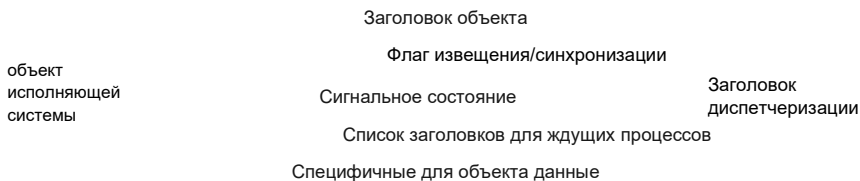


Рис. 6.5

Это могут быть семафоры, мьютексы, события, таймеры ожидания и прочие объекты, которых могут ожидать потоки для синхронизации своего выполнения с другими потоками. Сюда также входят объекты, представляющие открытые файлы, процессы, потоки и порты IPC. Структура данных диспетчеризации содержит флаг, представляющий сигнальное состояние объекта, а также очередь потоков, ожидающих сигнализации объекта. Прimitives сигнализации (такие, как семафоры) являются естественными диспетчерскими объектами. Таймеры, файлы, порты, потоки и процессы также используют механизм диспетчерских объектов для уведомлений. При срабатывании таймера, завершении ввода-вывода в файл, появлении в порту данных, завершении потока или процесса соответствующий диспетчерский объект сигнализирует, пробуждая все потоки, которые ждут этого события. Поскольку Windows использует для синхронизации с объектами режима ядра единый механизм, то специализированные API (такие, как `wait` в UNIX — для ожидания дочерних процессов) для ожидания событий не нужны. Часто потоки хотят ждать сразу многих событий. В UNIX процесс может ждать (при помощи системного вызова `select`) появления данных в любом из 64 сетевых сокетов. В Windows есть аналогичный API с названием `WaitForMultipleObjects`, но он позволяет потоку ждать диспетчерский объект любого типа (для которого у него есть описатель). Для `WaitForMultipleObjects` можно

указать до 64 описателей, а также необязательное значение тайм-аута. Поток становится готовым к выполнению тогда, когда сигнализирует любой из связанных с описателями объектов (или происходит тайм-аут).

Фактически существуют две разные процедуры, которые используются ядром для перевода в работоспособное состояние ожидающих потоков. Сигнализирование объекта уведомления (notification object) делает готовыми к работе все ожидающие потоки. Объекты синхронизации (synchronization objects) делают работоспособным только первый ждущий поток и используются для тех диспетчерских объектов, которые реализуют примитивы блокировки (вроде мьютексов). Когда ждущий блокировки поток опять начинает выполняться, то прежде всего он опять пытается получить блокировку. Если блокировку может удерживать одновременно только один поток, то все остальные ставшие работоспособными потоки могут немедленно заблокироваться (что может привести к большому количеству ненужных переключений контекста). Разница между использующими синхронизацию и уведомление (извещение) диспетчерскими объектами состоит во флаге в соответствующем поле dispatcher\_header (заголовка диспетчеризации). В качестве небольшого отступления: мьютексы в Windows называют «мутантами» кода, поскольку они требовались для реализации семантики OS/2, где они автоматически не разблокировали сами себя при выходе удерживающего их потока, что Катлер считал неестественным.

### **Исполняющая система (Исполняющий уровень)**

Диспетчер процессов (process manager) управляет созданием и завершением процессов и потоков, включая настройку управляющих ими политик и параметров. Однако операционные аспекты потоков определяются уровнем ядра, который управляет планированием и синхронизацией потоков, а также их взаимодействием с управляющими объектами (такими, как APC). Процессы содержат потоки, адресное пространство, а также таблицу описателей, содержащую те описатели, которые процесс может использовать для ссылки на объекты режима ядра. Процессы также содержат информацию, которая нужна планировщику для переключения между адресными пространствами и

для управления специфичной для процессов информацией относительно оборудования (такой, как дескрипторы сегментов).

Диспетчер памяти (memory manager) реализует архитектуру виртуальной памяти с подкачкой по требованию. Он управляет отображением виртуальных страниц на физические фреймы страниц, управляет имеющимися физическими фреймами, а также файлом подкачки на диске, используемым для хранения приватных экземпляров виртуальных страниц, которые уже не загружены в память. Диспетчер памяти предоставляет также специальные средства для больших серверных приложений (таких, как базы данных) и компоненты времени исполнения для языков программирования (такие, как сборщики мусора)

Диспетчер кэширования (cache manger) оптимизирует производительность вводы-вывода в файловой системе (поддерживая кэш страниц файловой системы в виртуальном адресном пространстве ядра). Диспетчер кэширования использует виртуально адресуемое кэширование, то есть организует кэшированные страницы по их расположению в их файлах. Это существенно отличается от кэширования физических блоков — как это делается в UNIX, где система поддерживает кэш физически адресуемых блоков дискового тома. Управление кэшированием реализовано при помощи отображения файлов в память. Реальное кэширование выполняется диспетчером памяти. Диспетчер кэширования должен заботиться только о принятии решения, какую часть какого файла кэшировать, обеспечивая своевременный сброс на диск кэшированных данных и управляя виртуальными адресами ядра (используемыми для отображения кэшированных страниц файлов). Если нужная для ввода-вывода в файл страница в кэше отсутствует, то при использовании диспетчера памяти будет получена ошибка отсутствия страницы. Монитор безопасности (security reference monitor) обеспечивает работу сложных механизмов безопасности Windows, которые поддерживают международные стандарты по компьютерной безопасности.

### **Реализация объектов**

Диспетчер объектов — это, вероятно, самый важный компонент

исполняющей системы Windows, и именно поэтому мы уже рассмотрели многие его концепции. Как уже описывалось ранее, он предоставляет унифицированный интерфейс для управления ресурсами системы и структурами данных, такими как открытые файлы, процессы, потоки, сегменты памяти, таймеры, устройства и семафоры. Даже более специализированные объекты (такие, как транзакции ядра, профили, маркеры безопасности и рабочие столы Win32) управляются диспетчером объектов. Объекты устройств связывают описания системы ввода-вывода (включая связь между пространством имен NT и томами файловой системы). Диспетчер конфигурации использует объект типа Key для связи с улями реестра. Сам диспетчер объектов имеет такие объекты, которые он использует для управления пространством имен NT и реализации объектов при помощи обычных средств. Это каталоги, символические ссылки, а также объекты «объект-тип». Обеспечиваемое диспетчером объектов единообразие имеет различные аспекты. Все эти объекты используют один и тот же механизм для создания, уничтожения и учета в системе квот. Ко всем этим объектам можно обращаться из процессов пользовательского режима при помощи описателей. Существует унифицированное соглашение для управления указателями, ссылающимися на объекты из ядра. Объектам можно давать имена в пространстве имен NT (которое управляется диспетчером объектов). Объекты диспетчеризации (которые начинаются с обычной структуры данных для сигнализации событий) могут использовать обычные интерфейсы синхронизации и уведомления (вроде WaitForMultipleObjects). Существует обычная система безопасности с использованием списков управления доступом (ACL), обязательная для открываемых по имени объектов, а также проверки доступа при каждом использовании описателя. Есть даже средства (для трассировки использования объектов) для помощи разработчикам режима ядра при отладке программ. Чтобы понять объекты, надо усвоить, что объект в исполняющей системе — это просто структура данных в виртуальной памяти, доступная режиму ядра. Эти структуры данных обычно используются для представления более абстрактных концепций. Например, объекты файлов исполняющего уровня создаются для каждого экземпляра открытого файла из файловой

<u>Имя объекта</u>	
<u>Каталог, в котором находится объект</u>	Информация по безопасности (кто может <b>использовать</b>
<b>объект)</b>	
<b>Квоты (стоимость использования объекта)</b>	
<u>Список процессов с описателями</u>	
<u>Количество ссылок</u>	
<u>Указатель на объект типа</u>	

Выделенная для объектов память берется из одной из двух куч (или пулов) памяти, поддерживаемых исполняющей



системой. Это служебные функции (типа malloc), которые позволяют компонентам режима ядра выделять либо страничную память ядра, либо бесстраничную память ядра. Безстраничная память требуется для любой

структуры данных или объекта режима ядра, к которому может понадобиться обратиться с уровня приоритета процессора номер 2 (или более высокого). Это может быть ISR или DPC (но не APC), а также и сам планировщик потоков. Описателю страничной ошибки также требуются свои структуры данных, которые должны выделяться в бесстраничной памяти ядра (во избежание возникновения рекурсии). Большая часть выделений памяти диспетчером кучи ядра производится при помощи справочных списков, которые содержат списки (типа LIFO — стек) выделенных блоков одинакового размера. Эти списки оптимизируются для операций без блокировок, что улучшает производительность и масштабируемость системы.

На рисунке 6.7 показана структура данных таблицы дескрипторов, используемой для трансляции описателей в указатели на объекты. Таблица описателей расширяется путем добавления дополнительных уровней косвенного обращения. Каждый процесс имеет свою таблицу, включая и системный процесс, который содержит все потоки ядра, не связанные с процессом пользовательского режима.

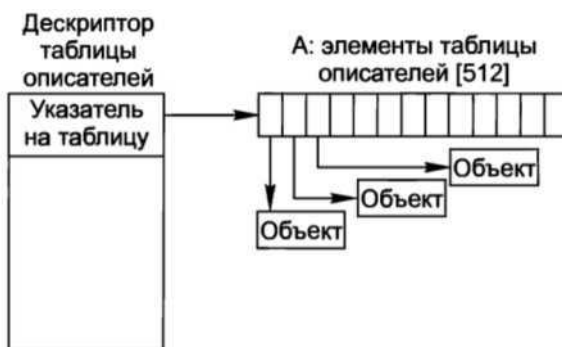


Рис 6.7

На рисунке 6.8 показана таблица описателей с двумя дополнительными уровнями косвенного обращения (это максимум). Для выполняющегося в режиме ядра кода иногда бывает удобно иметь возможность использовать описатели (а не указатели со ссылками). Они называются описателями ядра и специальным образом кодируются, чтобы их можно было отличить от описателей пользовательского режима. Описатели ядра хранятся в таблице описателей системных процессов, и к ним нельзя получить доступ из пользовательского



режима. Точно так же, как и большая часть виртуального адресного пространства ядра, системная таблица описателей совместно используется всеми компонентами ядра (вне зависимости от того, какой процесс пользовательского режима является текущим)

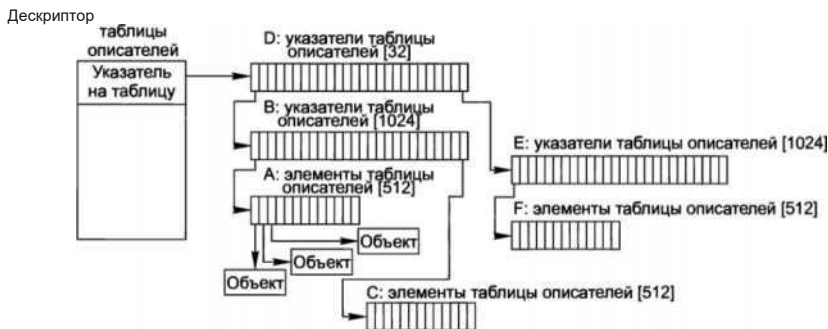


Рис 6.8

Пользователи могут создавать новые объекты или открывать уже существующие объекты (при помощи выполнения вызовов Win32, таких как CreateSemaphore или OpenSemaphore). Это вызовы библиотечных процедур, которые в итоге приводят к выполнению соответствующих системных вызовов. Результатом любого успешного вызова, который создает или открывает объект, является 64- битный элемент таблицы описателей, который записывается в частную таблицу описателей процесса (впамяти ядра). Пользователю возвращается 32-битный индекс логического положения описателя в таблице (для использования при последующих вызовах). 64-битный элемент таблицы описателей в ядре состоит из двух 32-битных слов. Одно слово содержит 29-битный указатель на заголовок объекта. Младшие 3 бита используются как флаги (например, наследуется ли описатель создаваемыми им процессами). Эти 3 бита маскируются перед переходом по указателю. Второе слово содержит 32-битную маску привилегий. Она нужна, поскольку проверка прав производится только в момент создания или открывания объекта. Если процесс имеет на объект только право чтения, то все прочие биты привилегий в маске будут равны 0, что даст операционной системе возможность отвергнуть любую другую операцию с объектом (кроме чтения).

1. Когда компонент исполнительного уровня (такой, как реализующий собственный системный вызов *NtCreateFile* диспетчер ввода-вывода) вызывает *ObOpenObjectByName* в диспетчере объектов, то он передает маршрут (в кодировке Unicode) для пространства имен NT (например, `\??\C:\foo\bar`).

2. Диспетчер объектов делает поиск по каталогам и символическим ссылкам и в итоге обнаруживает, что `\??\C:` относится к объекту устройства (типу, определенному диспетчером ввода-вывода). Объект устройства — это листовой узел в той части пространства имен, которой управляет диспетчер объектов.

3. Затем диспетчер объектов вызывает процедуру *Parse* для этого типа объектов — это *IopParseDevice*, реализованная диспетчером ввода-вывода. Она передает не только указатель на найденный объект устройства (для C), но также и остаток строки (`\foo\bar`).

4. Диспетчер ввода-вывода создает IRP (I/O Request Packet — пакет запроса ввода-вывода), выделяет файловый объект и отправит запрос в стек устройств ввода-вывода (определенный объектом устройства, который был найден диспетчером объектов).

5. IRP передается вниз по стеку ввода-вывода, пока не достигнет того объекта устройства, который представляет экземпляр файловой системы для C:. На всех стадиях управление передается точке входа в объект драйвера, связанный с объектом устройства на этом уровне. В данном случае используется точка входа для операций CREATE, поскольку запрос дан на создание или открытие файла с названием `\foo\bar` данного тома).

6. Обнаруженные по мере продвижения IRP к файловой системе объекты устройств представляют драйверы фильтров файловой системы, которые могут модифицировать операции ввода-вывода до того, как они достигнут объекта устройства файловой системы.

7. Объект устройства файловой системы имеет ссылку на объект драйвера файловой системы (например, NTFS). Таким образом, объект драйвера содержит адрес операции CREATE в NTFS.

8. NTFS заполняет файловый объект и возвращает его диспетчеру ввода-вывода, который осуществляет возврат вверх по всему стеку устройств (до тех пор, пока *IopParseDevice* не вернется в диспетчер

объектов).

9. Диспетчер объектов закончил поиск в пространстве имен. Он получил обратно инициализированный объект из процедуры *Parse* (который является файловым объектом, а не исходным объектом устройства, который он обнаружил). Таким образом, диспетчер объектов создает описатель для файлового объекта в таблице описателей текущего процесса и возвращает описатель вызывавшей стороне.

10. Последний шаг — возврат к вызывавшей стороне (в пользовательский режим), который в данном случае является процедурой *CreateFile* интерфейса Win32 API. Она вернет описатель в приложение.

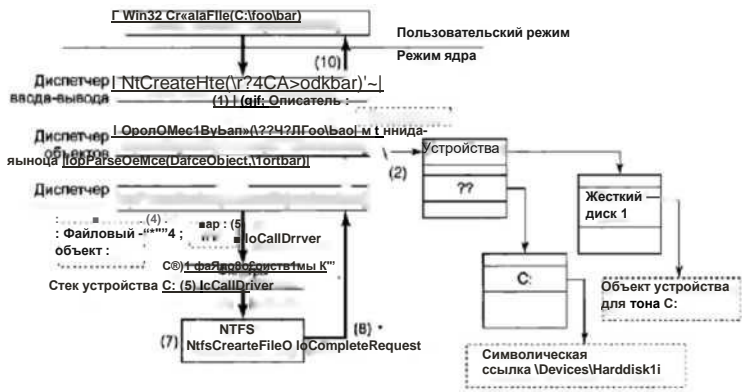


Рис 6.9

Процесс и поток являются очевидными. Есть один объект для каждого процесса и каждого потока, который хранит главные свойства (необходимые для управления процессом или потоком). Следующие три объекта (семафор, мьютекс и событие) относятся к межпроцессной синхронизации. Семафоры и мьютексы работают как обычно, но имеют разные дополнительные возможности (например, максимальные значения и тайм-ауты). События могут находиться в одном из двух состояний: сигнализированном или несигнализируемом. Если поток ждет события, которое находится в сигнализированном состоянии, то он немедленно освобождается. Если событие находится в несигнализируемом состоянии, то он блокируется до того момента,

когда какой-то другой поток просигнализирует это событие, после чего произойдет освобождение либо всех заблокированных потоков (для событий уведомления), либо только первого заблокированного потока (для событий синхронизации). Событие может быть также настроено таким образом, что после успешного получения сигнала оно автоматически перейдет в несигнализированное состояние (а не останется в сигнализированном состоянии). Объекты порта, таймера и очереди также относятся к обмену и синхронизации. Порты — это каналы между процессами (для обмена сообщениями LPC). Таймеры предоставляют способ блокирования на определенный интервал времени. Очереди используются для уведомления потоков о том, что ранее начатая операция асинхронного ввода-вывода завершилась, или о том, что в порту ждет сообщение. (Они созданы для управления уровнем параллелизма в приложении и используются в высокопроизводительных многопроцессорных приложениях, например, таких, как серверы систем управления базами данных.) Открытые файловые объекты создаются при открытии файла. Неоткрытые файлы не имеют объектов, которыми управляет диспетчер объектов. Маркеры доступа — это объекты безопасности. Они идентифицируют пользователя и рассказывают о том, какие специальные привилегии этот пользователь имеет (если они есть). Профили — это структуры, которые используются для хранения периодических отсчетов программного счетчика работающего потока (чтобы выяснить, где программа проводит свое время). Сегменты используются для представления объектов памяти, которые приложения могут попросить у диспетчера памяти отобразить на свое адресное пространство. Они хранят сведения о сегменте файла (или файла подкачки), который представляет страницы объекта памяти (когда они находятся на диске). Ключи представляют для пространства имен реестра точку монтирования в пространстве имен диспетчера объектов. Обычно имеется только один объект ключа (с названием `\REGISTRY`), который соединяет названия ключей реестра и их значения с пространством имен NT. Каталоги объектов и символические ссылки являются полностью локальными для той части пространства имен NT, которая управляется диспетчером объектов. Они

похожи на свои аналоги в файловой системе: каталоги позволяют собрать вместе связанные между собой объекты. Символические ссылки позволяют имени из одной части пространства имен объектов ссылаться на объект в другой части пространства имен объектов. Каждое известное операционной системе устройство имеет один (или несколько) объектов устройств, которые содержат информацию о нем и используются системой для ссылок на устройство. И наконец, каждый драйвер устройства (который был загружен) имеет объект драйвера в пространстве объектов. Объекты драйвера совместно используются всеми объектами устройств, которые представляют собой экземпляры устройств, управляемых этими драйверами. Прочие объекты (не указанные в таблице) служат более специализированным целям (взаимодействие с транзакциями ядра, пул рабочих потоков Win32). Интерфейсы прикладного программирования, DLL и службы пользовательского режима

## **Лекция 2.2 Процессы и потоки в Windows**

Для управления процессором и группировки ресурсов Windows имеет несколько концепций. Изучим и обсудим некоторые из соответствующих вызовов Win32 API, а также покажем, как они реализованы.

### **Фундаментальные концепции**

Windows процессы являются контейнерами для программ. Они содержат виртуальное адресное пространство, описатели объектов режима ядра, а также потоки. Как контейнеры для потоков они содержат также общие ресурсы (используемые для выполнения потоков), такие как указатель на структуру квоты, совместно используемый объект маркера, а также параметры по умолчанию (используемые для инициализации потоков), включая приоритет и класс планирования. Каждый процесс имеет системные данные пользовательского режима, называемые РЕВ (Process Environment Block — блок среды процесса). РЕВ включает список загруженных модулей (EXE и DLL), область памяти со строками окружения, текущий рабочий каталог, а также

данные для управления кучами процесса (и множество разнообразного хлама из Win32, который накопился со временем).

Потоки — это абстракции ядра для планирования процессора в Windows. Каждому потоку присваивается приоритет (в зависимости от значения приоритета его процесса). Потоки могут быть аффинизированными (affinitized), чтобы они выполнялись только на определенных процессорах. Это помогает параллельным программам (работающим на нескольких процессорах) явным образом распределять нагрузку. Каждый поток имеет два отдельных стека вызовов, один — для выполнения в пользовательском режиме, другой — для режима ядра. Есть также блок ТЕВ (Thread Environment Block — блок среды потока), который хранит специфичные для потока данные пользовательского режима, в том числе области хранения для потока (Thread Local Storage) и поля для Win32, локализации языка и культуры, а также прочие специальные поля, которые были добавлены различными средствами.

Помимо РЕВ и ТЕВ существует еще одна структура данных, которую режим ядра использует совместно со всеми процессами, — это «совместно используемые данные пользователя» (user shared data). Это страница, в которую ядро может вести запись, но процессы пользовательского режима могут из нее только читать. Она содержит некоторые поддерживаемые ядром значения, такие как различные формы времени, информация о версиях, количество физической памяти, а также большое количество флагов (совместно используемых различными компонентами пользовательского режима, такими как СОМ, службы терминалов, отладчики). Эта совместно используемая страница применяется исключительно для оптимизации производительности, поскольку все эти значения можно получить и при помощи системного вызова в режим ядра. Однако системные вызовы гораздо более дорогие, чем простое обращение к памяти, — поэтому для некоторых поддерживаемых системой полей (таких, как время) использовать эту страницу очень разумно. Другие поля (такие, как текущий часовой пояс) меняются редко, однако использующий их код должен часто запрашивать эти поля (чтобы определить, не изменились ли они).

## Процессы

Процессы создаются из объектов сегментов, каждый из которых описывает объект памяти (основанный на дисковом файле). При создании процесса создающий блестящей идеей. Для выполнения нового процесса в небольшом количестве памяти (при отсутствии аппаратных средств виртуальной памяти) приходилось процессы из памяти выгружать на диск. Первоначально fork в системе UNIX был реализован при помощи простой выгрузки родительского процесса и передачи его физической памяти дочернему процессу. Эта операция была почти «бесплатной». Отличие от тех времен, на момент написания командой Катлера системы NT обычной аппаратной средой были 32-битные многопроцессорные системы с аппаратной виртуальной памятью, которая использовала от 1 до 16 Мбайт физической памяти. Наличие нескольких процессоров позволяет одновременно выполнять части программ, поэтому NT применяла процессы как контейнеры для совместного использования памяти и ресурсов объектов, а потоки — как единицу параллельности (для планирования).

Конечно, те системы, которые появятся в течение последующих лет, не будут похожи ни на одну из этих двух целевых систем. У них будет 64-битное адресное пространство с десятками (или сотнями) процессорных ядер и много гигабайтов физической памяти, а также устройства флэш-памяти и другие энергонезависимые системы хранения, более широкая поддержка виртуализации, всеобъемлющая сетевая поддержка, а также поддержка инноваций в области синхронизации (типа «транзакционной памяти»). Windows и UNIX будут продолжать приспосабливаться к новым аппаратным средствам, но самое интересное — наблюдать за тем, какие новые операционные системы разрабатываются специально для использующих эти достижения систем.

## Задания и волокна

Windows может группировать процессы в задания, однако абстракция «задание» (job) имеет не очень общий характер. Она была специально создана для группирования процессов с целью применения

ограничений к содержащимся в них потокам, таких как ограничение использования ресурсов при помощи совместно используемой квоты или применение маркера ограниченного доступа (*restricted token*), который не позволяет потокам обращаться ко многим системным объектам. Самым важным свойством заданий (в плане управления ресурсами) является то, что с того момента, как процесс оказался в задании, все созданные (в этих процессах) потоками процессы будут также находиться в этом задании. Выхода нет. В полном соответствии со своим названием задания были предназначены для таких ситуаций, которые скорее напоминали пакетную обработку заданий, чем обычные интерактивные вычисления.

Процесс может находиться внутри только одного задания (максимум). Это разумно, поскольку трудно определить, как можно ограничить процесс несколькими квотами или маркерами ограниченного доступа. Однако это означает, что если несколько служб в системе попытаются использовать задания для управления процессами, то получатся конфликты (если они попытаются управлять одними и теми же процессами). Например, административный инструмент для ограничения использования ресурсов (посредством размещения процессов в заданиях) будет сбит с толку, если процесс сначала вставит себя в свое собственное задание (или если инструмент безопасности уже поместил процесс в задание с маркером ограниченного доступа — для ограничения его доступа к системным объектам). В результате задания в Windows применяются редко.

На рисунке 6.10 показана связь между заданиями (*jobs*), процессами (*processes*), потоками (*threads*), волокнами (*fibers*). Задания содержат процессы. Процессы содержат потоки. Но потоки не содержат волокна. Связь между потоками и волокнами обычно имеет тип «многие-ко-многим».



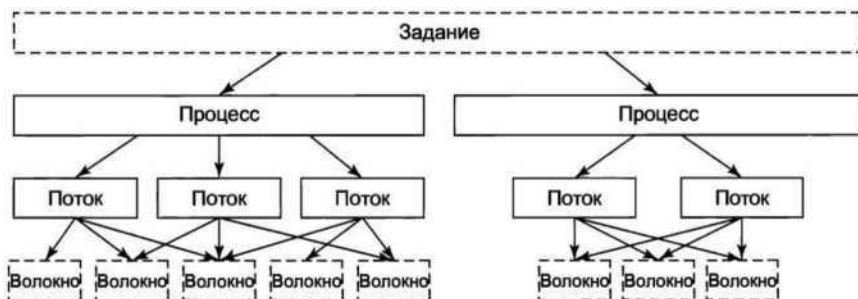


Рис. 6.10

Волокна создаются путем выделения места в стеке и структуры данных волокна в пользовательском режиме (для хранения регистров и данных, связанных с этим волокном). Потоки преобразуются в волокна, однако волокна могут создаваться и независимо от потоков. Такие волокна не будут выполняться до тех пор, пока уже выполняющееся в потоке волокно не вызовет явно `SwitchToFiber` (для запуска волокна). Потоки могут попытаться переключиться на то волокно, которое уже выполняется, так что программист должен предусмотреть синхронизацию (во избежание этого явления).

Любой процесс обычно начинается с одного потока, однако можно динамически создать дополнительные. Потоки являются основой планирования процессора, поскольку операционная система всегда выбирает для выполнения поток, а не процесс. Следовательно, каждый поток имеет состояние (готов, выполняется, блокирован и т. д.), а процессы не имеют состояний планирования. Потоки можно создавать динамически при помощи вызова `Win32`, в котором указывается адрес в адресном пространстве процесса (с которого он должен начинать работу).

Каждый поток имеет идентификатор потока, который выбирается из того же пространства, что и идентификатор процесса (так что процесс и поток никогда не могут иметь одинаковый идентификатор). Идентификаторы процессов и потоков кратны четырем, поскольку они выделяются исполнительным уровнем (с использованием специальной таблицы описателей, предназначенной для выделения идентификаторов). Таблица описателей не имеет ссылок на объекты, но использует поле указателя для указания на процесс или поток (так что

поиск процесса или потока по идентификатору очень эффективен). В современных версиях Windows используется порядок FIFO (очередь) для списка свободных описателей — так что повторное использование идентификаторов происходит не сразу.

Новые процессы создаются при помощи функции `CreateProcess` интерфейса Win32 API. Эта функция имеет много параметров и массу опций. Она принимает имя подлежащего выполнению файла, строки командной строки (без их разбора) и указатель на строки окружения. Есть также флаги и значения, которые управляют многими деталями, такими как настройка безопасности для процесса и первого потока, конфигурация отладчика, приоритеты планирования. При помощи флага указывается также, будут ли открытые описатели создателя передаваться новому процессу.

Функция принимает также текущий рабочий каталог для нового процесса и необязательную структуру данных с информацией о том окне графического интерфейса пользователя, которое должен использовать процесс. Вместо возврата идентификатора нового процесса Win32 возвращает и описатель, и идентификатор (как для нового процесса, так и для его исходного потока).

Большое количество параметров отражает те отличия, которые имеются по сравнению с созданием процессов в UNIX:

Реальный маршрут поиска подлежащей выполнению программы скрыт в библиотечном коде Win32, а в UNIX им можно управлять более явно.

Текущий рабочий каталог — это концепция режима ядра в UNIX, а в Windows — это строка пользовательского режима. Windows открывает описатель для текущего каталога для каждого процесса (с тем же назойливым эффектом, что и в UNIX — вы не сможете удалить каталог, если только это не сетевой каталог).

UNIX разбирает командную строку и передает массив параметров, а Win32 оставляет разбор аргументов программе. Вследствие этого некоторые программы могут обрабатывать групповые символы (например, \*.txt и прочие специальные символы) по-разному.

Способность наследования файловых дескрипторов в UNIX — это свойство описателя. В Windows это свойство и описателя, и параметра

создания процесса.

Win32 ориентирован на графический интерфейс пользователя, так что новым процессам напрямую передается информация об их первичном окне (в то время как в UNIX эта информация передается приложениям графического интерфейса пользователя как параметры).

Исполняемые файлы в Windows не имеют бита SETUID, но процесс может создать другой процесс, который выполняется как другой пользователь (если он может получить маркер с учетными данными этого пользователя).

Возвращенные из Windows описатели процесса и потока можно использовать для модификации нового процесса/потока многими способами, в том числе дублированием описателей настройкой переменных окружения нового процесса. UNIX просто модифицирует новый процесс между вызовами fork и exec.

### **Межпроцессный обмен**

Потоки могут вести обмен самыми разными способами, в том числе при помощи каналов, именованных каналов, почтовых слотов, сокетов, вызовов удаленных процедур, совместно используемых файлов.

Каналы имеют два режима: байтовый и режим сообщений (выбирается во время создания). В байтовом режиме каналы работают так же, как и в UNIX. Каналы в режиме сообщений немного похожи на них, но сохраняют границы сообщений — так что четыре записи по 128 байт будут прочитаны как четыре сообщения по 128 байт (а не как одно сообщение размером 512 байт, как это может случиться с каналом в байтовом режиме). Именованные каналы тоже имеют такие же два режима работы, как и обычные каналы. Именованные каналы могут также использоваться и по сети (а обычные — нет).

Почтовые слоты (mailslots) — это функция операционной системы OS/2, реализованная в Windows для совместимости. В некоторых отношениях они похожи на каналы, но не во всем. Например, они односторонние (а каналы — двусторонние). Их можно использовать по сети, но они не обеспечивают гарантированной доставки. И наконец, они позволяют посылающему процессу транслировать сообщение множеству получателей (а не только одному).

почтовые слоты, и именованные каналы реализованы в Windows как файловые системы, а не как функции исполнительного уровня ядра. Это позволяет осуществлять к ним доступ по сети при помощи существующих протоколов удаленных файловых систем.

Сокеты (sockets) подобны каналам, за исключением того, что они обычно соединяют процессы на разных машинах. Например, один процесс пишет в сокет, а другой (на удаленной машине) читает из него. Сокеты можно также использовать для соединения процессов на одной и той же машине, но поскольку их использование влечет за собой большие издержки, чем использование каналов, то их обычно используют только в сетевом контексте. Сокеты были изначально разработаны для Berkeley UNIX, и эта реализация стала широкодоступной. Некоторая часть кода и структур данных Berkeley и до нынешнего дня присутствуют в Windows (это признано в «Заметках версии», сопровождающих операционную систему).

RPC (удаленные вызовы процедур) — это способ для процесса А сделать так, чтобы процесс вызвал процедуру в адресном пространстве. От имени А и вернул результат в А. Параметры имеют различные ограничения. Например, нет смысла передавать указатель на другой процесс, поэтому структуры данных должны быть упакованы и переданы неспецифичным для процесса способом. RPC обычно реализованы как уровень абстракции над транспортным уровнем. В случае Windows транспортом могут быть сокеты TCP/IP, именованные каналы или ALPC. ALPC (Advanced Local Procedure Call — расширенный вызов локальных процедур) — это средство передачи сообщений в исполняющем уровне режима ядра. Оно оптимизировано для обмена между процессами локального компьютера и не работает по сети. Основная идея — посылать сообщения, которые генерируют ответы (реализуя таким образом облегченную версию вызовов удаленных процедур, на основе которой RPC может создать более богатый набор функциональных возможностей, чем те, которые имеются в ALPC). ALPC реализован при помощи сочетания копирования параметров и временного выделения совместно используемой памяти (в зависимости от размера сообщений).

Наконец, процессы могут совместно использовать объекты. Сюда

входят и объекты сегментов, соответствующие отображаемым на них файлам, которые можно отобразить на виртуальное адресное пространство разных процессов (в одно и то же время). Все операции записи, сделанные одним процессом в таком отображенном сегменте, появляются в адресных пространствах других процессов. При помощи этого механизма можно легко реализовать совместно используемый буфер, который применяется при решении проблем «поставщик—потребитель».

## **Синхронизация**

Процессы также могут использовать разные типы объектов синхронизации. Windows предоставляет множество механизмов синхронизации, - семафоры, мьютексы, критические области и события. Все эти механизмы работают с потоками (а не процессами), так что когда поток блокируется на семафоре, то другие потоки этого процесса (если они есть) могут продолжать выполнение.

Семафор создается при помощи функции `CreateSemaphore` интерфейса Win32 API, которая может инициализировать его заданным значением, а также задать максимальное значение. Семафоры — это объекты режима ядра и поэтому они имеют дескрипторы безопасности и описатели. Описатель для семафора может быть сдублирован при помощи `DuplicateHandle` и передан в другой процесс (чтобы по одному и тому же семафору могли синхронизироваться несколько процессов). Семафору можно дать имя в пространстве имен Win32, он может иметь ACL для своей защиты. Иногда совместное использование семафора по имени более удобно, чем дублирование описателя.

Имеются вызовы для операций `up` и `down`, хотя названия у них несколько странные: `ReleaseSemaphore` (это `up`) и `WaitForSingleObject` (это `down`). Можно также задать тайм-аут для `WaitForSingleObject`, чтобы вызывающий поток в итоге мог быть освобожден, даже если семафор останется на значении 0 (однако таймеры создают условия для возникновения эффекта гонок).

Мьютексы — это тоже объекты режима ядра, используемые для синхронизации, но они проще семафоров (поскольку не имеют счетчиков). По существу, это блокировки, имеющие функции API для

блокирования (`WaitForSingleObject`) и разблокирования (`ReleaseMutex`). Подобно описателям семафоров, описатели мьютексов могут дублироваться и передаваться между процессами, чтобы потоки в разных процессах могли получить доступ к одному и тому же мьютексу.

Третий механизм синхронизации называется «критическая секция» (`critical section`). Он реализует концепцию «критических областей». В Windows он похож на мьютекс, за исключением того, что он является локальным для адресного пространства создающего потока. Поскольку критические секции не являются объектами режима ядра, они не имеют явных описателей или дескрипторов безопасности и не могут передаваться между процессами. Блокирование и разблокирование выполняется соответственно вызовами `EnterCriticalSection` и `LeaveCriticalSection`. Поскольку эти функции API выполняются первоначально в пространстве пользователя и делают вызовы ядра только при необходимости в блокировке, то они гораздо быстрее мьютексов. Критические секции оптимизированы для комбинированного использования спин-блокировок (на многопроцессорных системах) и синхронизации ядра (при необходимости). Во многих приложениях большинство критических секций так редко становятся объектом соперничества или имеют такое краткое время удерживания, что необходимость в выделении объекта синхронизации ядра никогда не возникает. Это приводит к очень существенной экономии памяти ядра.

Обратите внимание, что некоторые из этих вызовов содержат значительное или не очень количество библиотечного кода, который отображает семантику Win32 на собственные вызовы интерфейса NT API. Другие же (относящиеся к интерфейсу волокон) являются исключительно функциями пользовательского режима, поскольку (как мы уже упоминали ранее) режим ядра в Windows NT 6 ничего не знает о волокнах. Они полностью реализованы средствами библиотек пользовательского режима.

## **Реализация процессов и потоков**

Рассмотрим подробно, как в Windows создается процесс (и начальный поток).

Процесс создается тогда, когда другой процесс делает вызов `CreateProcess` интерфейса `Win32`. Этот вызов запускает процедуру (пользовательского режима) из `kernel.dll`, которая создает процесс (выполняется несколько системных вызовов и другая работа):

Преобразуется имя исполняемого файла (заданное в виде параметра) из маршрута(пути) `Win32` в маршрут `NT`. Если исполняемый файл имеет только имя (без маршрута в виде каталогов), то поиск его ведется в тех каталогах, которые перечислены в качестве каталогов по умолчанию (они включают и те, которые содержатся в переменные окружения `PATH` — но не только их).

Собираются все параметры создания процесса и передаются (вместе с полным маршрутом к исполняемой программе) собственному интерфейсу `NtCreateUserProcess`. (Этот API был добавлен в `Windows Vista` для того, чтобы подробности создания процесса можно было реализовывать в режиме ядра, позволяя использовать процессы как защищенные области. Предыдущие собственные API (описанные выше) по-прежнему существуют, но вызовом `CreateProcess` они больше не используются.)

Работая в режиме ядра, `NtCreateUserProcess` обрабатывает параметры, а затем открывает образ программы и создает объект сегмента, который может использоваться для отображения программы на виртуальное адресное пространство нового процесса.

Диспетчер процессов выделяет и инициализирует объект процесса (структуру данных ядра, представляющую процесс как для ядра, так и для исполнительного уровня).

Диспетчер памяти создает адресное пространство для нового процесса, выделяя и инициализируя каталоги страниц и дескрипторы виртуальных адресов, описывающие режим ядра (и в том числе специфичные для процесса области, такие как элемент каталога страниц `self-map`,

который дает каждому процессу доступ в режиме ядра к физическим страницам всей таблицы страниц при помощи виртуальных адресов ядра). Мы опишем `self-map` более подробно в лекции «Управление памятью».

Для нового процесса создается таблица описателей, в которую

дублируются все те описатели вызвавшей стороны, для которых разрешается наследование.

Выполняется отображение совместно используемой страницы пользователя, а диспетчер памяти инициализирует структуры данных рабочего набора (используемые для того, чтобы принимать решение, какие страницы убирать из процесса при недостатке физической памяти).

Представленные объектом сегмента части образа исполняемого файла отображаются на адресное пространство пользовательского режима нового процесса.

Исполняющий уровень создает и инициализирует блок Process Environment Block (PEB) пользовательского режима, который используется как пользовательским режимом, так и ядром для поддержания информации о состоянии процесса (такой, как указатели кучи пользовательского режима и список загруженных библиотек (DLL)).

В новом процессе выделяется виртуальная память, которая используется для передачи параметров (в том числе строк окружения и командной строки).

Из специальной таблицы описателей (которую поддерживает ядро для эффективного выделения локально-уникальных идентификаторов для процессов и потоков) выделяется идентификатор процесса.

Выделяется и инициализируется объект потока. Выделяется стек пользовательского режима блок Thread Environment Block (TEB). Инициализируется запись CONTEXT, которая содержит начальные значения регистров процессора для потока (в том числе указатели команд и стека).

Объект процесса добавляется в глобальный список процессов. В таблице описателей вызвавшей стороны выделяется место под описатели для объектов процесса и потока. Для начального потока выделяется идентификатор (из таблицы идентификаторов).

NtCreateUserProcess возвращается в пользовательский режим с созданным новым процессом, содержащим единственный поток, который готов к работе, но находится в состоянии приостановки.

Если интерфейс NT API дает сбой, то код Win32 проверяет, не



принадлежит ли данный процесс к другой подсистеме (например, WOW64). Или, возможно, данная программа помечена для выполнения под управлением отладчика. Эти специальные случаи обрабатываются специальным кодом пользовательского режима в `CreateProcess`.

## Планирование

Ядро Windows не имеет центрального потока планирования. Вместо этого (когда поток не может больше выполняться) поток входит в режим ядра и вызывает планировщик, чтобы увидеть, на какой поток следует переключиться. К выполнению текущим потоком кода планировщика приводят такие условия:

Текущий выполняющийся поток блокируется на семафоре, мьютексе, событии, вводе-выводе и т. д.

Поток сигнализирует объекту синхронизации (то есть делает `up` на семафоре или приводит к сигнализированию события).

Интерфейс Win32 API предоставляет два API для работы с планированием потоков. Первый — вызов `SetPriorityClass`, который устанавливает класс приоритета для всех потоков вызывающего процесса. Допустимые значения: `real-time`, `high`, `above normal`, `normal` и `idle`. Класс приоритета определяет относительный приоритет процесса. (Начиная с Windows NT 6 класс приоритета процесса может также использоваться процессом для того, чтобы временно пометить самого себя как фоновый — это значит, что он не должен мешать никакой другой активности системы.) Обратите внимание, что класс приоритета устанавливается для процесса, но влияет на реальный приоритет каждого потока процесса (он устанавливает базовое значение приоритета, с которым стартует поток при создании).

Второй интерфейс Win32 API — это `SetThreadPriority`. Он устанавливает относительный приоритет потока (возможно, вызывающего потока — но это не обязательно) по отношению к классу приоритета своего процесса. Допустимые значения: `time critical`, `highest`, `above normal`, `normal`, `below normal`, `lowest` и `idle`. Потоки `time critical` получают самый высокий приоритет планирования, а потоки `idle` — самый низкий (независимо от класса приоритета). Планировщик работает следующим образом. В системе имеется 32 приоритета с

номерами от 0 до 31. Сочетание класса приоритета и относительного приоритета отображается на 32 абсолютных значения

## **Лекция 2.3 Управление памятью в Windows**

Принято считать, что каждый процесс, запущенный в Windows, получает в свое распоряжение виртуальное адресное пространство размером 4 Гб. Это число определяется разрядностью адресов в командах:  $2^{32}\text{байт} = 4 \text{ Гб}$ .

Конечно, трудно рассчитывать, что для каждого процесса найдется такое количество физической памяти, речь идет только о диапазоне возможных адресов.

Но даже и в этом смысле процессу доступно лишь около 2 Гб младших адресов виртуальной памяти. В частности, для WindowsNT старшие 2 Гб с адресами от 8000000016 до FFFFFFFF16 доступны только системе. Такое решение позволило уменьшить время, затрачиваемое при вызове системных функций, поскольку отпадает необходимость изменять при этом отображение страниц, нужно только разрешить их использование. Однако, чтобы сам вызов API-функций был возможен, системные библиотеки, которые содержат эти функции, размещаются в младшей, пользовательской половине виртуального пространства.

В Windows95 принято хулиганское решение: система и здесь располагается в старшей половине памяти, но эта половина доступна процессу пользователя и для чтения, и для записи. При этом вызов системы становится еще проще, но зато система становится беззащитной перед любой некорректной программой, лезущей куда не надо.

Кроме старших 2 Гб, процессу недоступны еще некоторые небольшие области в начале и в конце виртуального пространства. В WindowsNT недоступны адреса с 0000000016 по 0000FFFF16 и с 7P'PT000016 по 7FFFFFFF16, т.е. два кусочка по 64 Кб. Это сделано с целью выявления такой типичной ошибки программирования, как использование неинициализированных указателей, которые обычно попадают в запретные диапазоны адресов.

Для 64-разрядных процессоров размер виртуального адресного пространства возрастает до трудно представимых 264байт (17 миллиардов гигабайт, если угодно), однако Windows XP выделяет в распоряжение каждого процесса «всего лишь» 7152 гигабайта с адресами от 0 до 6FBFFFFFFF16, а остальное адресное пространство может использоваться только системой.

## Регионы

Рассмотрим теперь, каким образом программа процесса может использовать свое адресное пространство.

Попытка просто-напросто использовать в программе произвольно выбранный адрес в пределах адресного пространства процесса, скорее всего, приведет к выдаче сообщения об ошибке защиты памяти. На самом деле, использовать виртуальный адрес можно только после того, как ему поставлен в соответствие адрес физический. Такое сопоставление выполняется путем выделения регионов виртуальной памяти.

Регион памяти всегда имеет размеры, кратные 4 Кб (т.е. он содержит целое число страниц), а его начальный адрес кратен 64 Кб.

Для выделения региона используется функция VirtualAlloc. Она требует указания следующих параметров.

Начальный виртуальный адрес региона. Если указана константа NULL, то система сама выбирает адрес. Если указан адрес, не кратный 64 К, то система округляет его вниз.

Размер региона. При необходимости система округляет его до величины, кратной 4 Кб.

Тип выделения. Здесь указывается одна из констант MEM\_RESERVE (резервирование памяти) или MEM\_COMMIT (передача физической памяти), смысл которых будет подробно рассмотрен ниже, или комбинация обеих констант.

Тип доступа. Он определяет, какие операции могут выполняться со страницами выделенной памяти. Наиболее важны следующие типы доступа.

PAGE\_READONLY- доступ только для чтения, попытка записи в память приводит к ошибке.

PAGE\_READWRITE- доступ для чтения и записи.

PAGE\_GUARD- дополнительный флаг «охраны страниц», который должен комбинироваться с одним из предыдущих. При первой же попытке доступа к охраняемой странице генерируется прерывание, извещающее об этом систему. При этом флаг охраны автоматически снимается, так что дальнейшая работа со страницей выполняется без проблем.

Самое важное, что следует понять про выделение регионов, это смысл операций резервирования и передачи памяти.

Резервирование региона памяти (MEM\_RESERVE) означает всего лишь то, что диапазон виртуальных адресов, соответствующих данному региону, не будет использован ни под какие другие цели, система считает его занятым. Это как резервирование авиабилета: вы пока что не владеете билетом, но и никому другому его не продадут.

Попытка программы обратиться к адресу в зарезервированном, но не переданном регионе приведет к ошибке.

Передача физической памяти (MEM\_COMMIT) означает, что за каждой страницей виртуальной памяти региона система закрепляет... нет, вовсе не страницу физической памяти, как можно подумать. Закрепляется блок размером 4 Кб в страничном файле. В таблице страниц процесса переданные страницы помечаются как отсутствующие в памяти.

Теперь попытка обращения к адресу в регионе приведет уже к совсем другому результату. Поскольку страница отсутствует в памяти, произойдет прерывание. Однако это не будет рассматриваться как ошибка в программе. Система, обрабатывая прерывание, выполнит операцию чтения страницы с диска, из страничного файла, в основную память и занесет в таблицу страниц физический адрес, который теперь соответствует виртуальной странице. После этого команда, вызвавшая прерывание, будет повторена, но теперь уже с успехом, поскольку требуемая страница находится в памяти. Дальнейшие обращения к той же виртуальной странице будут выполняться без проблем, пока страница находится в памяти.

Резервирование и передача памяти могут выполняться одновременно, при одном обращении к функции VirtualAlloc, которой

для этого нужно передать комбинацию обеих констант: MEM\_RESERVE+MEM\_COMMIT. Есть и другой вариант: сначала зарезервировать регион памяти, а затем, по мере необходимости, передавать физическую память либо всему региону сразу, либо его отдельным частям (субрегионам). Для этого в первый раз функция VirtualAlloc вызывается с константой MEM\_RESERVE, как правило, без указания конкретного адреса. Затем VirtualAlloc->VirtualAlloc с константой MEM\_COMMIT и с указанием адреса ранее зарезервированного региона или соответствующего субрегиона.

Все описанное полностью соответствует понятию загрузки страниц по требованию, описанному в п. 4.5. В качестве особенностей реализации замещения страниц в Windows следует отметить следующее.

Для каждого процесса в системе определены максимальный и минимальный размер его рабочего множества. При выборе вытесняемой страницы система пытается добиться, чтобы за каждым процессом сохранялось не менее минимального, но не более максимального количества памяти. Это позволяет избежать ситуации, когда один процесс, расточительно использующий память, вытесняет из нее почти все страницы других процессов. Процесс может изменить размеры своего рабочего множества, но при этом суммарные требования всех процессов ограничиваются реальным размером имеющейся памяти.

Процесс может запереть в памяти некоторый диапазон адресов, чтобы воспрепятствовать вытеснению соответствующих страниц на диск. Суммарный размер памяти, запертой одним процессом, по умолчанию не должен превосходить 30 страниц. Длительное удержание одним процессом большого числа страниц запертыми в памяти привело бы к уменьшению объема памяти, доступного для других процессов (да и для незапертых страниц того же процесса).

## **Лекция 2.4 Ввод-вывод в Windows**

## Лекция 2.4 Ввод-вывод в Windows

Цель диспетчера ввода-вывода Windows — обеспечить обширную и гибкую основу для эффективной обработки очень широкого разнообразия устройств и служб ввода-вывода, поддержки автоматического распознавания устройств и инсталляции драйверов (Plug-and-Play), а также для управления электропитанием устройств и процессора — и все это с использованием в основном асинхронной структуры, которая позволяет выполнять вычисления одновременно с передачей данных при вводе-выводе. Существуют сотни тысяч устройств, которые работают с Windows NT. Для большого количества часто встречающихся устройств даже не нужно инсталлировать драйвер, поскольку в составе операционной системы уже есть такой драйвер. Даже с учетом этого (и если считать все версии) существует почти миллион различных двоичных драйверов, которые работают под управлением Windows NT6.

### Вызовы интерфейса прикладного программирования ввода-вывода

Интерфейсы системных вызовов, предоставляемые диспетчером ввода-вывода, не очень отличаются от предлагаемых большинством других операционных систем. Основные операции: *open*, *read*, *write*, *ioctl* и *close*, но есть также и другие операции: Plug-and-Play; управления энергопотреблением, установки параметров, сброса системных буферов и т.д. На уровне Win32 эти API заключаются в оболочку интерфейсов, которые предоставляют операции более высокого уровня (специфичные для конкретных устройств). На нижнем уровне эти оболочки открывают устройства и выполняют эти основные операции. Даже некоторые операции метаданных (такие, как переименование файла) реализованы без специфичных системных вызовов. Они просто используют специальную версию операции *ioctl*. Это станет более понятно, когда мы объясним реализацию стеков устройств ввода-вывода и использование пакетов запросов ввода-вывода (I/O request packets, IRP) диспетчером ввода-вывода.

Собственные системные вызовы ввода-вывода NT (в соответствии с общей философией Windows) имеют множество параметров и много вариантов. *NtCreateFile* используется для открытия существующих или

новых файлов. Он предоставляет дескрипторы безопасности для новых файлов, описание требуемых прав доступа, дает создателю новых файлов некоторые функции управления выделением блоков. *NtReadFile* и *NtWriteFile* принимают описатель файла, буфер и длину. Они также принимают явное смещение файла и позволяют указать ключ для доступа к заблокированным диапазонам байтов в файле. Большая часть параметров относится к указанию того, какие методы следует использовать для сообщения о завершении ввода/вывода (возможно, асинхронного), — они описаны выше.

*NtQueryDirectoryFile* — это пример стандартной парадигмы исполнительного уровня, где существуют различные API для обращения (или модификации) к информации об определенных типах объектов. В данном случае это объекты файлов, которые ссылаются на каталоги. Параметр указывает, какой тип информации запрашивается, например, список названий файлов в каталоге либо подробная информация о каждом файле (которая нужна для расширенного листинга каталога). Поскольку это фактически операция ввода-вывода, то поддерживаются все стандартные способы сообщений о завершении ввода-вывода. *NtQueryVolumeInformationFile* подобен операции запроса каталога, но ожидает получения описателя файла, представляющего открытый том, который может содержать (это необязательно) файловую систему. В отличие от каталогов, для томов есть параметры, которые можно модифицировать, поэтому существует отдельный вызов *NtSetVolumeInformationFile*.

*NtNotifyChangeDirectoryFile* — это пример интересной парадигмы NT. Поток может выполнять ввод-вывод, чтобы определить, происходят ли с объектами какие-либо изменения (в основном это каталоги файловых систем, как в данном случае, или ключи реестра). Поскольку ввод-вывод асинхронный, то поток возвращается и продолжает выполнение (а позже уведомляется, когда что-то модифицируется). Незавершенный запрос ставится в очередь в файловой системе как ожидающая выполнения операция ввода-вывода (с использованием пакета запросов IRP).

Если вы хотите удалить том с файловой системой из компьютера, то извещения становятся проблемой, поскольку есть незавершенные

операции ввода-вывода. Поэтому Windows поддерживает средства для отмены незавершенных операций ввода-вывода (и в том числе в файловых системах есть поддержка принудительного размонтирования тома с незавершенными операциями ввода-вывода).

*NtQueryInformationFile* — это специфичная (для файлов) версия системного вызова для каталогов. У нее есть напарник — системный вызов *NtSetInformationFile*. Эти интерфейсы обращаются и модифицируют все виды информации об именах файлов, файловых функциональных возможностях (типа шифрования, сжатия и разреженности), а также прочих атрибутах файлов (и их подробностях) — и в том числе они определяют внутренний идентификатор файла или присваивают файлу уникальное двоичное имя (идентификатор объекта).

Эти системные вызовы являются по существу специфичной для файлов формой *ioctl*. Для переименования или удаления файла можно использовать операцию *set*. Однако обратите внимание, что они принимают описатели (а не имена файлов), так что до переименования или удаления файл должен быть сначала открыт. Их можно также использовать для переименования альтернативных потоков данных в NTFS.

Отдельные вызовы (*NtLockFile* и *NtUnlockFile*) существуют для установки и удаления побайтовых блокировок для файлов. *NtCreateFile* позволяет ограничить доступ ко всему файлу (при помощи режима совместного использования). Альтернатива ему — эти вызовы блокировки, которые накладывают обязательные ограничения доступа на диапазон байтов в файле. Операции чтения и записи должны предоставлять ключ, совпадающий с заданным в *NtLockFile* (чтобы работать с заблокированными диапазонами).

Аналогичные средства имеются и в UNIX, но там приложения соблюдают блокировки диапазонов по своему усмотрению. *NtFsControlFile* во многом похожа на предыдущие операции *query* и *set*, но является операцией более общего типа, нацеленной на обработку специфичных для файлов операций (которые не выполняются другими вызовами). Например, некоторые операции специфичны для конкретной файловой системы.

И наконец, существуют разнообразные вызовы типа



*NtFlushBuffersFile*. Подобно вызову *sync* операционной системы UNIX, он заставляет записать данные файловой системы на диск. *NtCancelIoFile* отменяет незавершенные запросы ввода-вывода для конкретного файла. *NtDeviceIoControlFile* реализует операции *ioctl* для устройств. Список операций на самом деле гораздо длиннее. Существуют системные вызовы для удаления файлов по имени, для запроса атрибутов конкретного файла, но это просто оболочки для других операций диспетчера ввода-вывода (которые мы уже перечислили) и их не нужно реализовывать в виде отдельных системных вызовов. Есть также системные вызовы для работы с портами завершения ввода-вывода (I/O completion ports) — это средство формирования очередей в Windows, которое помогает многопоточным серверам эффективно использовать операции асинхронного ввода-вывода (посредством подготовки потоков по требованию и уменьшения количества переключений контекста, требуемых для обслуживания ввода-вывода специальными потоками).

### **Реализация ввода-вывода**

Система ввода-вывода в Windows состоит из служб Plug-and-Play, диспетчера электропитания, менеджера ввода-вывода, а также модели драйвера устройств. Plug-and-Play обнаруживает изменения в конфигурации аппаратного обеспечения и создает (или уничтожает) стеки устройств (для каждого устройства), а также загружает и выгружает драйверы устройств. Диспетчер электропитания настраивает состояние электропитания устройств ввода-вывода (чтобы уменьшить потребление энергии системой, когда устройства не используются). Диспетчер ввода-вывода предоставляет поддержку манипулирования объектами ядра для ввода-вывода, а также операций типа *IoCallDrivers* и *IoCompleteRequest*. Однако большая часть работы по поддержке ввода-вывода в Windows реализована в самих драйверах устройств.

### **Драйверы устройств**

Чтобы гарантировать, что драйверы устройств хорошо работают с Windows NT, компания *Microsoft* описала модель WDM (Windows Driver Model), которой должны соответствовать драйверы устройств. WDM была создана для работы как с Windows 98, так и с операционными

системами на базе Windows NT (начиная с Windows 2000), что позволяло корректно написанным драйверам работать в обеих системах. Существует набор разработчика (Windows Driver Kit, ранее известный как DDK), который предназначен для помощи в написании соответствующих данной модели драйверов. Большинство драйверов Windows начинается с копирования подходящего образцового драйвера и его модификации.

Компания *Microsoft* также предоставляет верификатор для драйверов (driver verifier), который проверяет многие действия драйвера — для уверенности в том, что он соответствует требованиям WDM (по структуре и протоколам запросов ввода-вывода, управлению памятью и т. д.). Верификатор поставляется вместе с системой, администраторы могут запустить его командой *verifier.exe*, которая позволяет им указать, какие драйверы будут проверяться и насколько всесторонними (то есть дорогими) должны быть эти проверки.

При всей этой поддержке разработки и верификации драйверов в Windows по-прежнему очень трудно написать даже простой драйвер, поэтому компания *Microsoft* создала систему оболочек под названием WDF (Windows Driver Foundation), которая работает поверх WDM и упрощает многие стандартные требования (в основном связанные с правильным взаимодействием с управлением электропитанием и операциями Plug-and-Play).

Чтобы еще больше упростить написание драйверов, а также повысить живучесть системы, WDF включает в себя инфраструктуру UMDF (User-Mode Driver Framework) для написания драйверов в виде выполняющихся в процессах служб. Существует также KMDF (Kernel-Mode Driver Framework) для написания драйверов как служб, выполняющихся в ядре, — при этом многие подробности WDM реализуются автоматически. Поскольку в основе лежит WDM (с ее моделью драйверов), то именно на ней мы и сосредоточимся в этом разделе.

Операции ввода-вывода инициируются диспетчером ввода-вывода, который вызывает интерфейс *IoCallDriver* исполнительного уровня с указателями на верхний объект устройства и на IRP (который представляет запрос ввода-вывода). Эта процедура находит объект

драйвера (связанный с объектом устройства). Указанные в IRP типы операций обычно соответствуют описанным выше системным вызовам диспетчера ввода-вывода (таким, как CREATE, READ и CLOSE).

На рисунке 6.11 показаны связи для одного уровня стека устройств. Для каждой из этих операций драйвер должен указать точку входа. *IoCallDriver* берет тип операции из IRP, использует объект устройства на текущем уровне стека устройств (для поиска объекта драйвера) и ищет (по типу операции) в таблице переходов для драйверов соответствующую точку входа в драйвер. Затем драйвер вызывается, и ему передаются объект устройства и IRP.

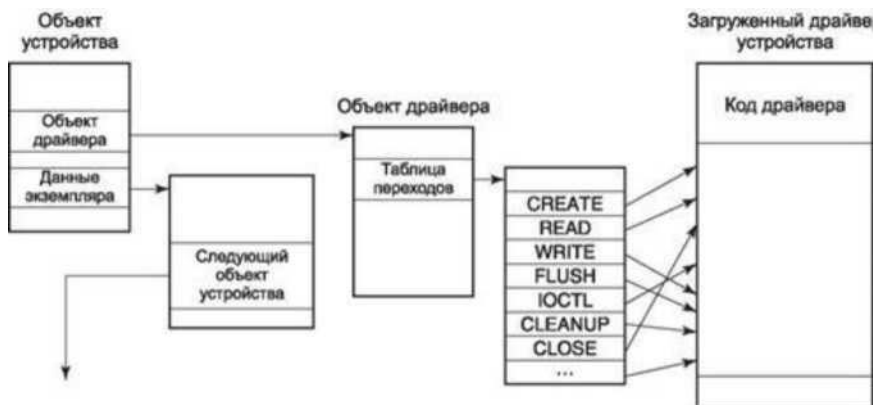


Рис. 6.11

### Пакеты запроса ввода-вывода

На рисунке 6.12 показаны основные поля IRP. По существу, IRP — это массив (с динамически изменяемым размером), содержащий поля, которые могут быть использованы любым драйвером (для обрабатывающего запрос стека устройств). Эти *стековые* поля позволяют драйверу указать процедуру, которую нужно вызвать при завершении запроса ввода-вывода. При завершении все уровни стека устройств проходятся в обратном порядке, при этом по очереди вызываются все процедуры завершения (для всех драйверов). На каждом уровне драйвер может завершить запрос или принять решение, что еще есть некая работа (которую нужно сделать), и оставить запрос незавершенным (отложив на данный момент завершение ввода-

вывода).



Рис. 6.12

При выделении IRP диспетчер ввода-вывода должен знать, насколько глубоок данный конкретный стек устройств, чтобы выделить достаточно большой IRP. Он отслеживает глубину стека в поле каждого объекта устройства (при формировании стека устройств). Обратите внимание, что не существует формального определения, какой следующий объект устройства в стеке. Эта информация содержится в частных структурах данных, принадлежащих предыдущему в стеке драйверу. Фактически стек может и не быть стеком вовсе. На каждом уровне драйвер может выделить новый IRP, продолжить использовать исходный IRP, послать операцию ввода-вывода в другой стек устройства либо даже переключиться на системный рабочий процесс для продолжения выполнения.

IRP содержит флаги, код операции для поиска по таблице переходов, указатели для пользовательского буфера и буфера ядра, а также список MDL (Memory Descriptor Lists), которые используются для описания представленных буферами физических страниц (то есть для операций DMA). Имеются также поля для операций отмены и завершения. Те поля в IRP, которые используются для постановки IRP в очередь к устройствам, используются повторно после завершения операции ввода-вывода (чтобы обеспечить память для управляющего объекта

APC, используемого для вызова процедуры завершения диспетчера ввода-вывода в контексте исходного потока). Есть также поле ссылки, используемое для связи всех незавершенных IRP с инициировавшим их потоком.

На рисунке 6.12 показаны также два формирующих стек драйвера.

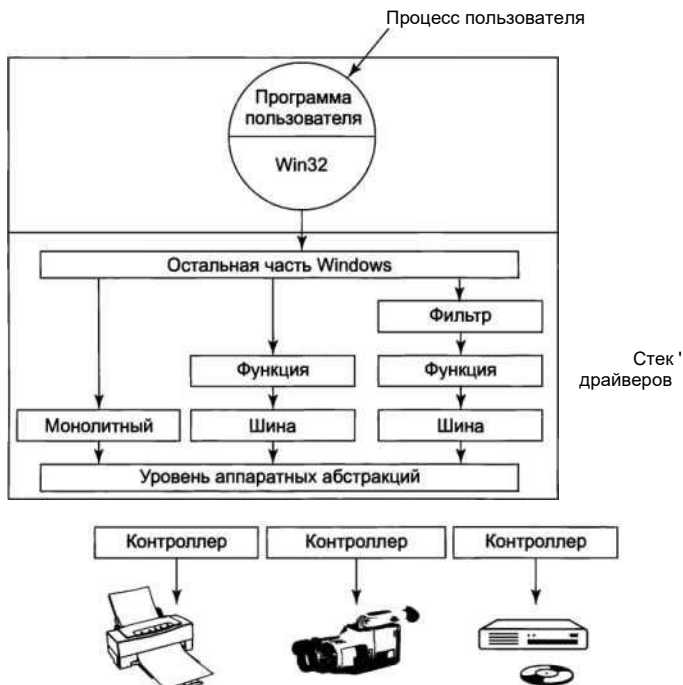


Рис. 6.12

Стеки драйверов часто используются для того, чтобы отделить управление шиной от работы по управлению устройством. Управление шиной PCI очень сложное (поскольку она имеет множество режимов и шинных транзакций). Отделив эту часть от специфичной для устройства части, авторы пдрайверов избавляются от необходимости изучать управление шиной. Они могут просто использовать в своем стеке драйверов стандартный драйвер для шины. Аналогичным же образом драйверы для SCSI и USB имеют специфичную для устройств часть и общую часть, причем часто используемые драйверы для общей части имеются в составе Windows.

Драйверы устройств режима ядра являются серьезной проблемой

для надежности и стабильности Windows. Большинство отказов ядра в Windows происходит из-за ошибок в драйверах устройств. Поскольку драйверы устройств режима ядра совместно используют одно и то же адресное пространство с уровнем ядра и исполнительным уровнем, то ошибки в драйверах могут повредить системные структуры данных (или сделать что-то худшее). Некоторые из этих ошибок возникают из-за потрепашающе большого количества существующих для Windows драйверов устройств либо из-за разработки драйверов неопытными системными программистами. Ошибки пвзникают и из-за большого количества подробностей, которые нужно знать для написания

Диспетчер электропитания (power manager) глаз не спускает с использования электроэнергии по всей системе. Исторически управление потреблением энергии состояло из отключения монитора и остановки вращения дисководов. Но эта проблема быстро становится все более сложной — из-за требований по увеличению продолжительности работы ноутбуков от батарей, а также соображений экономии энергии на настольных компьютерах (которые оставляют постоянно включенными) и из-за высокой стоимости потребляемой серверными центрами электроэнергии (компании вроде *Microsoft* и *Google* строят свои серверные центры около гидроэлектростанций — чтобы получить низкие тарифы).

Новые средства управления электропитанием включают уменьшение потребления энергии компонентами (когда система не используется)— для этого отдельные устройства переключаются в состояние резервирования или даже полностью отключаются (при помощи выключателя питания). Мультипроцессорные системы отключают отдельные процессоры (когда они не нужны) и даже могут уменьшать тактовую частоту процессоров (для уменьшения энергопотребления). Когда процессор бездействует, его потребление энергии также уменьшается — поскольку ему не нужно делать ничего, кроме ожидания возникновения прерbieaHUM. Windows поддерживает специальный режим выключения под названием гибернация (hibernation), при котором выполняется копирование всей физической памяти на диск, а затем потребление энергии снижается до минимального (в состоянии гибернации ноутбуки могут работать

неделями), при этом батарея разряжается минимально. Поскольку все состояние памяти записано на диск, то вы можете даже заменить батарею ноутбука (пока он находится в гибернации). Когда система загружается из гибернации, она восстанавливает сохраненное состояние памяти (и повторно инициализирует устройства). Это приводит компьютер в то же самое состояние, в котором он был перед гибернацией (без необходимости выполнять повторно регистрацию и запускать все приложения и службы (которые выполнялись). Несмотря на то что Windows старается оптимизировать этот процесс — в том числе она при этом игнорирует немодифицированные страницы (имеющие резервирование на диске) и сжимает остальные страницы памяти (для снижения требуемого объема ввода-вывода), — переход имеющего гигабайты памяти ноутбука (или настольного компьютера) в состояние гибернации может продолжаться много секунд.

## Лекция 2.5 Файловая система NTFS

Windows NT6 поддерживает несколько файловых систем, самыми важными из которых являются FAT-16, FAT-32 и NTFS (NT File System). FAT-16 - это старая файловая система операционной системы MS-DOS. Она использует 16-битные дисковые адреса, что ограничивает размер дисковых разделов двумя гигабайтами. В основном она применялась для доступа к флоппи-дискам (в настоящее время они практически не используются). FAT-32 использует 32-битные дисковые адреса и поддерживает дисковые разделы размером до двух терабайтов. FAT-32 не имеет никакой системы безопасности и на сегодняшний день она фактически используется только для переносных носителей (таких, как флэш-диски). Файловая система NTFS была разработана специально для версии Windows NT. Начиная с Windows XP ее по умолчанию устанавливает большинство производителей компьютеров, она существенно увеличивает безопасность и функциональность Windows. NTFS использует 64-битные дисковые адреса и теоретически может поддерживать дисковые разделы размером до 264 байт (однако некоторые соображения ограничивают этот размер до более низких значений).

### Структура файловой системы

Каждый том NTFS (например, дисковый раздел) содержит файлы, каталоги, битовые массивы и другие структуры данных. Каждый том организован как линейная последовательность блоков (которые в терминологии компании *Microsoft* называются кластерами), причем размер блоков для каждого тома фиксирован (в зависимости от размера тома он может изменяться от 512 байт до 64 Кбайт). Большинство дисков NTFS использует блоки размером 4 Кбайт — это компромисс между применением больших блоков (для эффективной передачи данных) и использованием маленьких блоков (для снижения внутренней фрагментации). Ссылки на блоки делаются с использованием смещения от начала тома (при помощи 64-битных чисел).

Главная структура данных каждого тома — это MFT (Master File Table — главная таблица файлов), которая является линейной



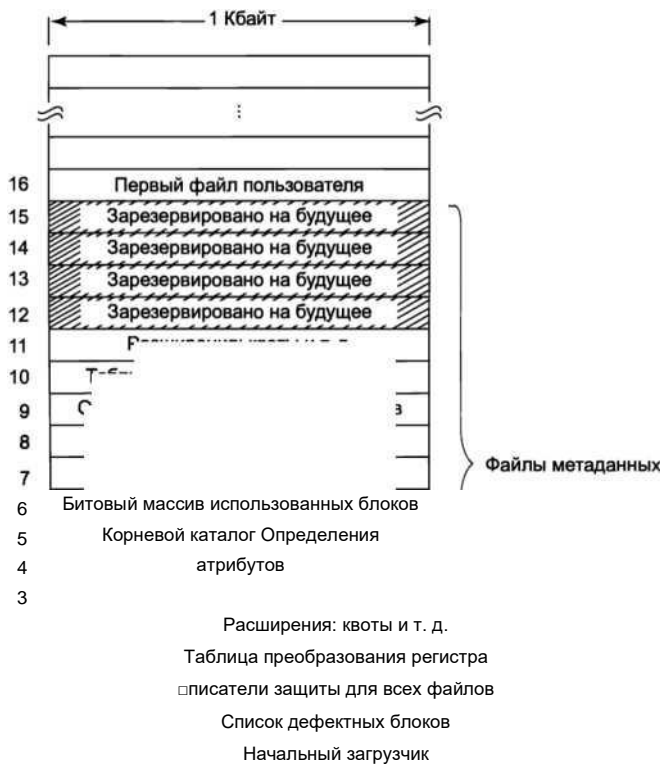
последовательностью записей фиксированного размера (1 Кбайт). Каждая запись MFT описывает один файл или один каталог. Она содержит атрибуты файла (такие, как его имя и временная метка), а также список дисковых адресов (где расположены его блоки). Если файл очень большой, то иногда приходится использовать две или более записей MFT (чтобы разместить в них список всех блоков) — в этом случае первая запись в MFT, называемая основной записью (base record), указывает на остальные записи в MFT. Такая схема переполнения ведет свое начало из CP/M, где каждый элемент каталога назывался экстендом. Битовый массив отслеживает свободные элементы MFT.

Сама MFT также является файлом и в качестве такового может быть размещена в любом месте тома (таким образом устраняется проблема наличия дефектных секторов на первой дорожке). Более того, при необходимости этот файл может расти (до максимального размера в 248 записей).

Первые 16 записей MFT резервируются для файлов метаданных NTFS (рисунок 6.13). Каждая из этих записей описывает нормальный файл, который имеет атрибуты и блоки данных (как и любой другой файл). Каждый из этих файлов имеет имя, которое начинается со знака доллара (чтобы обозначить его как файл метаданных). Первая запись (0) описывает сам файл MFT. В частности, в ней говорится, где находятся блоки файла MFT (чтобы система могла найти файл MFT). Очевидно, что Windows нужен способ нахождения первого блока файла MFT, чтобы найти остальную информацию по файловой системе. Windows смотрит в загрузочном блоке — именно туда записывается адрес первого блока файла MFT при форматировании тома.

Запись 1 является дубликатом начала файла MFT. Эта информация настолько ценная, что наличие второй копии может быть просто критическим (в том случае, если один из первых блоков MFT испортится). Вторая запись — файл журнала. Когда в файловой системе делаются структурные изменения (такие, как добавление

нового или удаление существующего каталога), то такое действие журналируется здесь до его выполнения (чтобы повысить вероятность корректного восстановления в случае сбоя во время операции — например такого, как отказ системы). Здесь также журналируются и изменения в файловых атрибутах. Фактически не журналируются здесь только изменения в пользовательских данных. Запись 3 содержит информацию о томе (такую, как его размер, метка и версия).



Файл тома
Журнал для восстановления
Зеркальная копия MFT
Главная файловая таблица

Рис. 6.13

Как уже упоминалось выше, каждая запись MFT содержит последовательность пар (заголовок атрибута — значение). Атрибуты определяются в файле *\$AttrDef*. Информация об этом файле содержится в MFT (в записи 4). Затем идет корневой каталог, который сам является файлом и может расти до произвольного размера. Он описывается записью номер 5 в MFT.

Свободное пространство тома отслеживается при помощи битового массива. Сам битовый массив — тоже файл, его атрибуты и дисковые адреса даны в записи 6 в MFT. Следующая запись MFT указывает на файл начального загрузчика. Запись 8 используется для того, чтобы связать вместе все плохие блоки (чтобы обеспечить невозможность их использования для файлов). Запись 9 содержит информацию безопасности. Запись 10 используется для установления соответствия регистра. Для латинских букв А—Z соответствие регистра очевидно (по крайней мере для тех, кто разговаривает на романских языках). Однако соответствие регистров для других языков (таких, как греческий, армянский или грузинский) для говорящих на романских языках не столь очевидно — поэтому данный файл рассказывает, как это сделать.

И наконец, запись 11 — это каталог, содержащий различные файлы для таких вещей, как дисковые квоты, идентификаторы объектов, точки повторной обработки и т. д. Последние четыре записи MFT зарезервированы для использования в будущем.

### **Выделение дискового пространства**

Модель отслеживания дисковых блоков состоит в том, что они выделяются последовательными участками, насколько это возможно (из соображений эффективности). Например, если первый логический блок потока помещен в блок 20 диска, то система будет очень стараться поместить второй логический блок в блок 21, третий логический блок — в блок 22 и т. д. Одним из способов достижения непрерывности этих участков является выделение дискового пространства по несколько блоков за один раз (по мере возможности). Блоки потока описываются последовательностью записей, каждая из которых описывает последовательность логически смежных блоков. Для потока без пропусков будет только одна такая запись. К этой категории

принадлежат такие потоки, которые записаны по порядку с начала и до конца. Для потока с одним пропуском будет две записи. Файлы с пропусками называются разреженными файлами (sparse files). Заголовок записи указывает количество блоков в файле.

За заголовком записи следует одна или несколько пар (в каждой дается дисковый адрес и длина участка). Дисковый адрес — это

смещение дискового блока от начала раздела; длина участка — это количество блоков в участке. В записи участка может быть столько пар, сколько необходимо.

Использование этой схемы для потока из трех участков и девяти блоков показано на рисунке 6.14. На этом рисунке у нас есть запись MFT для короткого потока из девяти блоков (заголовок 0-9).

Он состоит из трех участков последовательных блоков на диске. Первый участок — блоки 20-23, второй — блоки 64-65, третий — блоки 80-82. Каждый из этих участков заносится в запись MFT как пара (дисковый адрес, количество блоков). Количество участков зависит от того, насколько хорошо справился со своей работой модуль выделения блоков при создании потока. Для потока из  $n$  блоков количество участков может составлять от 1 до  $n$ .

Здесь нужно сделать несколько замечаний. Во-первых, для представленных таким способом потоков нет верхнего ограничения размера. Если не использовать сжатие адресов, то для каждой пары требуется два 64-битных числа (всего 16 байт). Однако пара может представлять 1 миллион (или более) смежных дисковых блоков. Фактически состоящий из 20 отдельных участков (каждый — по одному миллиону блоков размером в 1 Кбайт) поток размером в 20 Гбайт легко описывается одной записью MFT, а разбросанный по 60 изолированным блокам поток размером 60 Кбайт одной записью MFT не удастся описать.

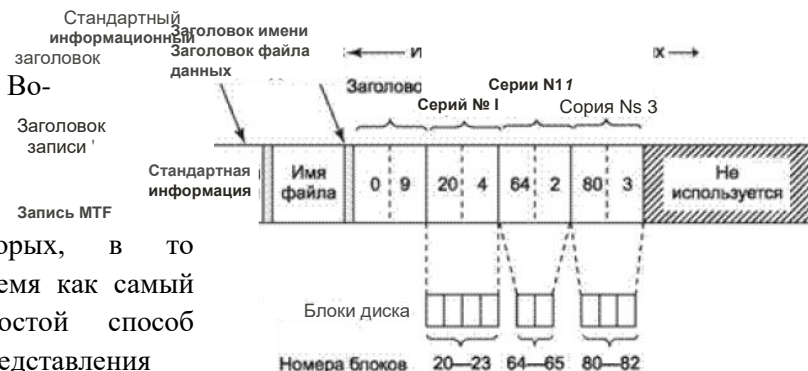


Рис. 6.14

вторых, в то время как самый простой способ представления каждой пары

требует 2 x 8 байт, имеется также и метод сжатия, который уменьшает размер пары меньше чем до 16 байт. Многие дисковые адреса имеют нулевые старшие байты. Их можно опустить. Заголовок данных сообщает о том, сколько их было опущено (то есть сколько байтов реально используется на один адрес). Используются также и другие виды сжатия. На практике пара часто занимает только 4 байта.

Наш первый пример был простым: вся информация файла уместилась в одной записи MFT. Что произойдет, если файл настолько большой или так сильно фрагментирован, что информация о блоках не помещается в одну запись MFT? Ответ простой: используются две или более записей MFT. На рисунке 6.15 мы видим файл, основная запись которого находится в записи 102 в MFT. Он имеет слишком много (для одной записи MFT) участков, поэтому вычисляется количество нужных записей расширения (например, две) и их индексы помещаются в основную запись. Остальная часть записи используется для первых  $k$  участков данных.

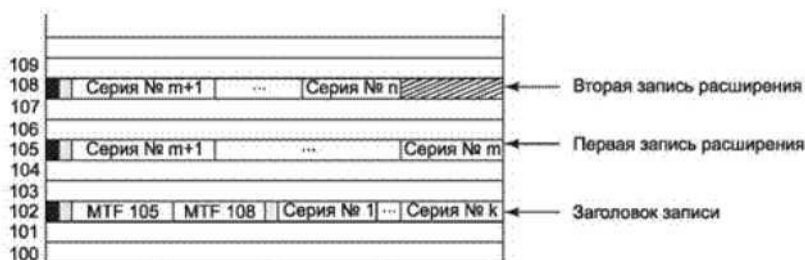


Рис. 6.15

Обратите внимание, что на рисунке 6.15 имеется некоторая избыточность. В теории не должно быть необходимости указывать конец последовательности участков, поскольку эту информацию можно вычислить по парам для участков. Цель избыточного указания этой информации в том, чтобы сделать более эффективным поиск: чтобы найти блок по заданному смещению в файле, нужно обследовать только заголовки записей (а не пары для участков).

NTFS поддерживает прозрачное сжатие файлов. Файл может

создаваться в сжатом режиме, а это означает, что NTFS пытается автоматически сжать блоки при их записи на диск и автоматически распаковывает их при чтении обратно. Те процессы, которые читают или пишут сжатые файлы, совершенно не в курсе того факта, что происходит сжатие или распаковка.

Сжатие работает следующим образом. Когда NTFS пишет файл (помеченный как сжатый) на диск, то она изучает первые 16 (логических) блоков файла— независимо от того, сколько участков они занимают. Затем она запускает по ним алгоритм сжатия. Если полученные данные можно записать в 15 или менее блоков, то сжатые данные записываются на диск (по возможности — одним участком). Если сжатые данные по-прежнему занимают 16 блоков, то эти 16 блоков записываются в несжатом виде. Затем исследуются блоки 1631, чтобы узнать, можно ли их сжать до размера 15 блоков (или менее), и т. д.

### **Журналирование**

NTFS поддерживает два механизма, при помощи которых программы могут обнаружить изменения в файлах и каталогах тома. Первый механизм — это операция ввода-вывода под названием *NtNotifyChangeDirectoryFile*, передающая системе буфер, который возвращается после обнаружения изменения в каталоге или подкаталоге. В результате ввода-вывода буфер заполняется списком записей об изменениях. Хорошо, если буфер достаточно большой — потому что в противном случае лишние записи теряются.

Второй механизм — это журнал изменений NTFS. NTFS содержит список всех записей об изменениях для каталогов и файлов тома в специальном файле, который программы могут читать при помощи специальных операций управления файловой системой (опция вызова *NtFsControlFile*). Файл журнала обычно очень большой и поэтому вероятность затирания записей до того, как они будут изучены, очень мала.

### **Шифрование файлов**

Сегодня компьютеры используются для хранения самых разнообразных конфиденциальных данных. Владельцы подобных данных, как правило, не желают, чтобы она попала в посторонние руки. Информация может оказаться потеряна, например, при потере или

краже переносного компьютера. Настольный компьютер можно загрузить с внешнего диска, чтобы обойти систему безопасности Windows NT. Наконец, жесткий диск можно просто вынуть из одного компьютера и установить на другой компьютер.

В операционной системе Windows на базе NT эти проблемы решаются при помощи возможности шифрования файлов. В результате применения шифрования, даже если компьютер будет украден или перезагружен, файлы останутся нечитаемыми. Чтобы использовать шифрование в операционной системе Windows NT, нужно пометить каталог как зашифрованный, в результате чего будут зашифрованы все файлы в этом каталоге, а все новые файлы, перемещены в этот каталог или созданные в нем, также будут зашифрованы. Само шифрование и дешифрование выполняется не файловой системой NTFS, а специальным драйвером EFS (Encrypting File System — шифрующая файловая система), размещающимся между NTFS и пользовательским процессом. Таким образом, прикладная программа не знает о шифровании, а сама файловая система NTFS только частично вовлечена в этот процесс.

Познакомимся теперь с тем, как шифруются файлы в операционной системе Windows на основе NT. Когда пользователь сообщает системе, что хочет зашифровать определенный файл, формируется случайный 128(192 или 256 в зависимости от версии ОС и используемого алгоритма)- разрядный ключ. Ключ используется для поблочного шифрования файла с помощью симметричного алгоритма, параметром в котором используется этот ключ. Каждый новый шифруемый файл получает новый случайный 128/192/256-разрядный ключ, так что никакие два файла не используют один и тот же ключ шифрования, что увеличивает защиту данных в случае, если какой-либо из ключей окажется скомпрометированным.

Чтобы файл мог быть впоследствии расшифрован, ключ файла должен где-то храниться. Если бы ключ хранился на диске в открытом виде, тогда злоумышленник, укравший файлы, мог бы легко найти его и воспользоваться им для расшифровки украденных файлов. В этом случае сама идея шифрования файлов оказалась бы бессмысленной. Поэтому ключи файлов сами должны храниться на диске в



зашифрованном виде. Для этого используется шифрование с открытым ключом.

После того как файл зашифрован, система с помощью информации в системном реестре ищет расположение открытого ключа пользователя. Открытый ключ можно без каких-либо опасений хранить прямо в реестре, так как, по открытому ключу невозможно определить закрытый ключ, необходимый для расшифровки файлов. Затем случайный 128/192/256-разрядный ключ файла шифруется открытым ключом, а результат сохраняется на диске вместе с файлом.

Чтобы расшифровать файл, с диска считывается зашифрованный случайный 128/192/256-разрядный ключ файла. Однако для его расшифровки необходим закрытый ключ. В идеале этот ключ должен храниться на смарт-карте, вне компьютера, и вставляться в считывающее устройство только тогда, когда требуется расшифровать файл. Хотя операционная система поддерживает смарт-карты, она не позволяет хранить на них закрытые ключи.

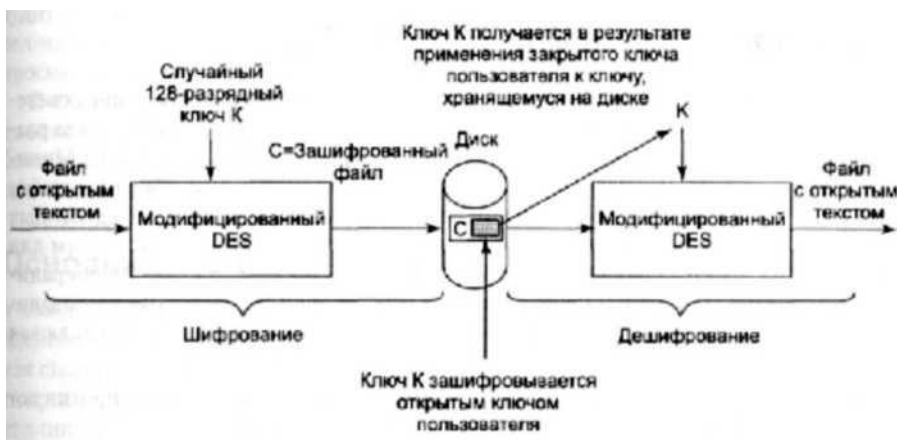


Рис. 6.16

Сложности возникают тогда, когда нескольким пользователям требуется доступ к одному и тому же зашифрованному файлу. Ключ шифрования файла будет зашифровываться несколько раз, отдельно для каждого авторизованного пользователя, его закрытым ключом. Все зашифрованные версии ключа могут добавляться к файлу.

Схема с использованием случайных ключей для шифрования файлов, но с шифрованием самих ключей при помощи симметричного алгоритма шифрования не будет работать. Проблема в том, что наличие симметричного ключа, хранящегося на диске в открытом виде, разрушит всю систему защиты — сформировать ключ дешифрации по ключу шифрования слишком легко. Таким образом, медленное шифрование с открытым ключом требуется для шифрования ключей файлов. Поскольку ключи шифрования все равно являются открытыми, хранение их в открытом виде не представляет опасности.

Вторая причина использования двухуровневой системы ключей заключается в производительности. Использование криптографии с открытым ключом для шифрования файлов было бы слишком медленным. Для повышения эффективности шифрование с открытым ключом применяется лишь для зашифровки коротких 128/192/256-разрядных ключей файлов, тогда как для шифрования самих файлов используется симметричный алгоритм.

## Лекция 2.6 Безопасность в Windows

Каждый пользователь (и группа) в Windows NT идентифицируется идентификатором безопасности SID (Security ID). SID — это двоичное число с коротким заголовком, за которым следует длинный случайный компонент. Каждый SID должен быть глобально-уникальным. Когда пользователь запускает процесс, то этот процесс и его потоки выполняются под пользовательским идентификатором SID. Большая часть системы безопасности спроектирована так, чтобы обеспечить доступ к любому объекту только потоков с авторизованными SID.

Каждый процесс имеет маркер доступа (access token), в котором указан SID и прочие свойства. Маркер обычно создается модулем winlogon (как описано ниже). Формат маркера показан на рисунке 6.17. Процессы могут вызвать GetTokenInformation (чтобы получить эту информацию). Заголовок содержит некоторую административную информацию. Поле «срок годности» может сказать, когда маркер утрачивает действительность (однако в настоящее время оно не используется). Поле Группы указывает группы, которым принадлежит процесс (это нужно для подсистемы POSIX). DACL (Discretionary ACL) — это список управления доступом, присваиваемый созданным процессом объектам (если не указан другой ACL). Пользовательский SID говорит о том, кто владеет процессом. Ограниченные идентификаторы SID позволяют ненадежным процессам принимать участие в заданиях вместе с надежными процессами (и при этом у них меньше возможностей что-то испортить).

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

Рис. 6.17

И наконец, привилегии (если они есть) дают процессу специальные возможности (которых нет у обычных пользователей) — такие, как право выключения компьютера или доступа к файлам. По существу, привилегии делят власть суперпользователя на несколько прав,

которые можно присвоить процессам. Таким образом, пользователь может получить некоторую власть суперпользователя (но не всю). Подводя итог — маркер доступа говорит о том, кто владеет процессом, а также какие значения по умолчанию и какие полномочия с ним связаны.

Когда пользователь регистрируется, то winlogon дает начальному процессу маркер доступа. Следующие процессы обычно наследуют этот маркер. Маркер доступа процесса первоначально применяется ко всем потокам процесса. Однако поток при выполнении может получить другой маркер доступа, в этом случае маркер доступа потока замещает маркер доступа процесса. В частности, клиентский поток может передать свои права доступа серверному потоку, чтобы сервер мог обратиться к защищенным файлам (и прочим объектам) клиента. Этот механизм называется олицетворением (impersonation). Он реализован в транспортных слоях (например, ALPC, именованные каналы, TCP/IP), используемых в RPC для обмена между клиентами и серверами. Транспортные слои используют внутренние интерфейсы монитора безопасности ядра для извлечения контекста безопасности для маркера доступа текущего потока и отправки его на сервер, где он используется для конструирования маркера, который сервер может использовать для олицетворения клиента.

Еще одна фундаментальная концепция — дескриптор безопасности (security descriptor).

Каждый объект имеет связанный с ним дескриптор безопасности, который говорит о том, кто и какие операции может выполнять с ним. Дескрипторы безопасности указываются при создании объектов. Файловая система NTFS и реестр поддерживают постоянную форму дескриптора безопасности, который используется для создания дескриптора безопасности для объектов File и Key (это объекты диспетчера объектов, представляющие открытые экземпляры файлов и ключей).

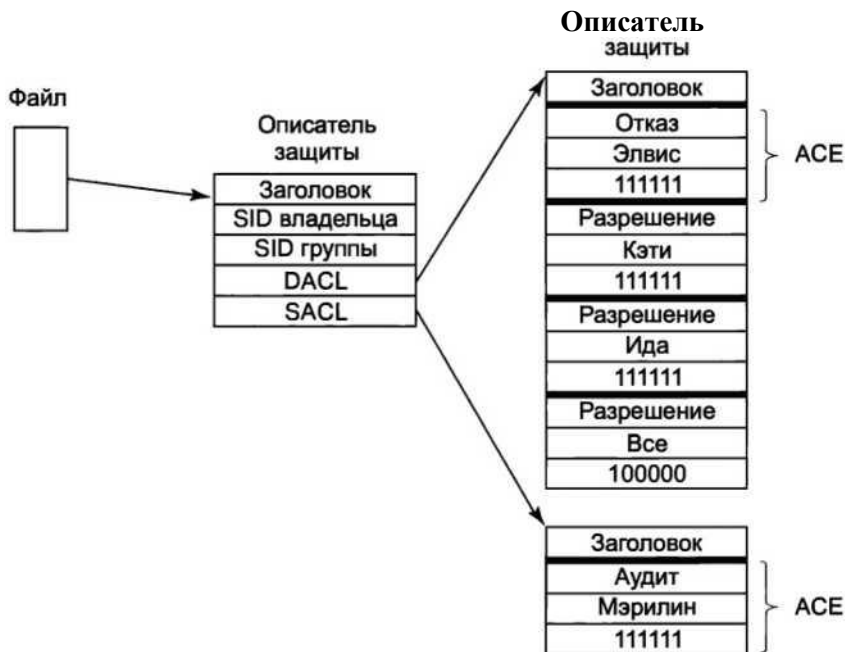


Рис. 6.18

Дескриптор безопасности состоит из заголовка, за которым следует список DACL с одним (или более) элементами управления доступом ACE (Access Control Entries). Два основных типа такого элемента — Allow и Deny. Элемент Allow указывает SID и битовый массив, который указывает, какие операции этот SID может выполнять над объектом. Элемент Deny работает аналогично (однако совпадение означает, что вызывающая сторона не может выполнять операцию). Например, Ида имеет файл, в дескрипторе безопасности которого указано, что доступ чтения имеют все, Элвис доступа не имеет, Кэти имеет доступ на чтение/запись, а сама Ида имеет полный доступ. Этот простой пример показан на рисунке 6.18. Идентификатор Все относится к множеству всех пользователей, но он перекрывается всеми (последующими) явными ACE.

В дополнение к списку DACL дескриптор безопасности имеет также и список SACL (System Access Control), который похож на DACL, за исключением того, что он указывает не тех, кто может использовать объект, а какие операции над объектом записываются в системный

журнал событий безопасности. На рис. 6.18 будет регистрироваться каждая операция, которую Мэрилин выполняет с файлом. SACL также содержит уровень целостности (который мы скоро обсудим).

### **Вызовы интерфейса прикладного программирования безопасности**

Большая часть механизма управления доступом в Windows на основе NT базируется на дескрипторах безопасности. Обычно схема такая: когда процесс создает объект, он предоставляет дескриптор безопасности в качестве одного из параметров CreateProcess или CreateFile (либо другого вызова создания объекта). Этот дескриптор безопасности затем становится прикрепленным к объекту дескриптором безопасности. Если при вызове создания объекта не предоставлен дескриптор безопасности, то вместо него используется безопасность по умолчанию из маркера доступа вызывающей стороны.

Многие вызовы системы безопасности в Win32 относятся к управлению дескрипторами безопасности, поэтому мы сосредоточимся здесь именно на них. При создании дескриптора безопасности сначала под него выделяется, а затем и инициализируется (при помощи InitializeSecurityDescriptor) место хранения. Этот вызов заполняет заголовок. Если SID владельца неизвестен, то его можно поискать по имени при помощи LookupAccountSid. Затем его можно вставить в дескриптор безопасности. То же самое можно сделать и с SID группы (если он есть). Обычно есть SID вызывающей стороны и одна из групп вызывающей стороны (однако системный администратор может вписать любые SID).

В этот момент можно инициализировать список DACL (или SACL) дескриптора безопасности (при помощи InitializeAcl). Элементы ACL можно добавлять при помощи AddAccessAllowedAce и AddAccessDeniedAce. Эти вызовы можно повторять много раз (для добавления необходимого количества элементов ACE). DeleteAce можно использовать для удаления элемента (при модификации существующего ACL). Когда ACL готов, можно использовать SetSecurityDescriptorDacl для прикрепления его к дескриптору безопасности. И наконец, когда объект создается, то свежесготовленный дескриптор безопасности можно передать как параметр (чтобы прикрепить его к объекту).

## Реализация безопасности

Безопасность отдельной системы под управлением Windows на основе NT реализована несколькими компонентами. Регистрацией управляет winlogon, а аутентификацией — lsass. Результатом успешной регистрации является новая оболочка с графическим интерфейсом пользователя (explorer.exe) со связанным с ней маркером доступа. Этот процесс использует ульи SECURITY и SAM реестра. Первый настраивает общую политику безопасности, а второй содержит информацию безопасности для отдельных пользователей.

После того как пользователь зарегистрировался, все операции системы безопасности происходят тогда, когда объект открывается для доступа. Для каждого вызова OpenXXX требуется имя открываемого объекта и набор требующихся прав. Во время обработки открытия монитор безопасности (см. рис. 11.4) проверяет, есть ли у вызывающей стороны все необходимые права. Он выполняет эту проверку путем изучения маркера доступа вызывающей стороны и списка DACL (связанного с объектом). Он просматривает по порядку список ACE внутри ACL. Как только он находит элемент, который совпадает с SID вызывающей стороны (или одной из групп вызывающей стороны), то найденный доступ принимается как окончательный. Если имеются все права, необходимые для вызывающей стороны, то операция открытия завершается успешно; в противном случае она заканчивается неудачей.

Списки DACL могут иметь как элементы Deny, так и элементы Allow (как мы уже видели). По этой причине обычно запрещающие доступ элементы размещают в ACL впереди разрешающих, чтобы тот пользователь, конкретно которому отказано в доступе, не мог его получить через «черный ход» (будучи членом группы, которая имеет законный доступ).

После открытия объекта вызывающей стороне возвращается его описатель. При последующих вызовах делается единственная проверка — была ли предпринимаемая сейчас операция в наборе запрошенных в момент открытия операций (чтобы не дать вызывающей стороне открыть файл для чтения, а затем записать в него). Кроме того, вызовы с использованием описателей могут приводить к появлению записей в журналах аудита (если это требуется списком SACL).

Начиная с Windows Vista добавлено еще одно средство для решения

возникающих при обеспечении безопасности системы (при помощи ACL) проблем. Это новый обязательный идентификатор уровня целостности (Integrity-level SID) в маркере процесса, причем объекты указывают список ACE уровня целостности в списке SACL. Уровень целостности предотвращает доступ на запись к объектам (вне зависимости от того, какие ACE есть в DACL). В частности, схема уровня целостности используется для защиты от скомпрометированного процесса Internet Explorer, который, возможно, был атакован посредством скачивания кода с незнакомого веб-сайта. Internet Explorer с низкими правами (он называется Low-rights IE) работает с установленным в значение low уровнем целостности. По умолчанию все файлы и ключи реестра имеют уровень целостности medium, так что работающий с уровнем low браузер Internet Explorer не может их модифицировать.

Если при наличии UAC делается попытка выполнить операцию, требующую административного доступа, то система показывает специальное уведомление и перехватывает управление, так что только ввод пользователя может разрешить этот доступ (аналогично тому, как работает CTRL-ALT-DEL по требованиям C2). Конечно, даже не став администратором, взломщик может уничтожить то, что ценно для пользователя (его персональные файлы). Однако UAC позволяет помешать атакам известных типов, причем всегда легче восстановить скомпрометированную систему в том случае, когда взломщик не смог модифицировать системные данные или файлы.

Можно создавать «защищенные процессы» (protected processes), которые обеспечивают периметр безопасности. Обычно периметр привилегий в системе определяет пользователь (представленный объектом маркера). Когда процесс создается, пользователь имеет доступ к процессу при помощи самых разных средств ядра (для создания процесса, его отладки, маршрутов и т. д.). Защищенные процессы изолированы от доступа пользователей. Эта функция в Windows NT6 используется только в программном обеспечении управления цифровыми правами Digital Rights Management для улучшения защиты контента.

Усилия компании Microsoft по повышению безопасности Windows в последние годы стали активизироваться, поскольку количество атак по



всему миру растет. Большая часть атак использует небольшие ошибки программирования, которые приводят к переполнению буфера, что позволяет взломщику вставить код (переписав адреса возврата, указатели на исключения и прочие данные, которые управляют выполнением программы). Многие из этих проблем можно избежать, если вместо языков С и С++ использовать безопасные в отношении типов языки. И даже при использовании этих небезопасных языков многих уязвимостей можно было бы избежать, если бы студентов лучше учили понимать важность проверки параметров и данных.

## ОСНОВНАЯ ЛИТЕРАТУРА

1. Вирт, Н. Разработка операционной системы и компилятора. Проект Оберон [Электронный ресурс] / Н. Вирт, Ю. Гуткнехт. — Электрон. дан. — Москва: ДМК Пресс, 2012. 560 с. Режим доступа: <https://e.lanbook.com/book/39992>.

## ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

2. Крищенко, В.А. Сервисы Windows [Электронный ресурс]: учебное пособие / В.А. Крищенко, Н.Ю. Рязанова. — Электрон. дан. — Москва: МГТУ им. Н.Э. Баумана, 2011. — 47 с. — Режим доступа: <https://e.lanbook.com/book/52416>

3. Войтов, Н.М. Администрирование ОС Red Hat Enterprise Linux. Учебный курс [Электронный ресурс]: учеб. пособие — Электрон. дан. — Москва: ДМК Пресс, 2011. 192 с. Режим доступа: <https://e.lanbook.com/book/1081>.

4. Стащук П.В. Администрирование и безопасность рабочих станций под управлением Mandriva Linux: лабораторный практикум. М: Флинта, 2015. <https://e.lanbook.com/book/70397>