**Управление памятью, вводвывод, файловая система и безопасность в ОС UNIX**

**Лекция**

# Содержание

История изменений 2

Содержание 3

Лекция. Управление памятью, ввод-вывод, файловые системы и безопасность в ОС

UNIX 4

Вопросы 4

Управление памятью 4

Основные понятия 4

Системные вызовы управления памятью в UNIX 6

Реализация управления памятью в UNIX 7

Алгоритм замещения страниц 9

Управление памятью в Linux 11

Ввод-вывод в системе UNIX 14

Системные вызовы ввода-вывода системы UNIX 16

Реализация ввода-вывода в системе UNIX 17

Потоки данных 19

Файловая система UNIX 21

Основные понятия 21

Вызовы файловой системы в UNIX 23

Реализация файловой системы UNIX 26

Файловая система Berkeley Fast 29

Файловая система Linux 31

Файловая система NFS 32

Безопасность в UNIX 37

Основные понятия 37

Системные вызовы безопасности в UNIX 38

Реализация безопасности в UNIX 39

Литература 39

# Лекция. Управление памятью, ввод-вывод, файловые системы и безопасность в ОС UNIX

## Вопросы

1. Управление памятью.
2. Организация ввода-вывода.
3. Файловые системы.
4. Система безопасности.

## Управление памятью

### Основные понятия

У каждого процесса в системе UNIX есть адресное пространство, состоящее из трех сегментов: текста (программы), данных и стека. Пример адресного пространства процесса изображен на рис. 1(а). **Текстовый (программный)** сегмент содержит машинные команды, образующие исполняемый код программы. Он создается компилятором и ассемблером при трансляции программы, написанной на языке высокого уровня (например, С 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.

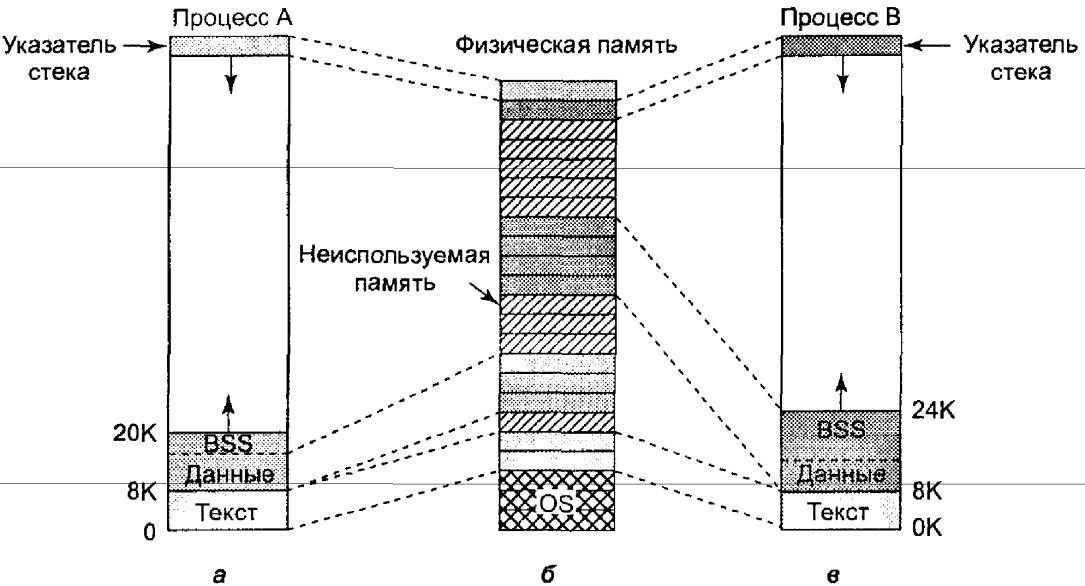


Рис. 1. Виртуальное адресное пространство процесса А (а); физическая память (б); виртуальное адресное пространство процесса В (в)

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

Неинициализированные данные необходимы лишь с точки зрения оптимизации. Когда начальное значение глобальной переменной явно не указано, то, согласно семантике языка С, ее значение устанавливается равным 0. На практике большинство глобальных переменных не инициализируются, и, таким образом, их начальное значение равно 0. Это можно реализовать следующим образом: создать целый сегмент исполняемого двоичного файла, точно равного по размеру числу байтов данных, и проинициализировать весь этот сегмент нулями.

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 ОС считывает это слово, выделяет нужное число байтов и обнуляет их.

Рассмотрим это еще раз на нашем примере, рис. 1(а). Здесь текст программы занимает 8 Кбайт, и инициализированные данные также занимают 8 Кбайт. Размер сегмента неинициализированных данных (BSS) равен 4 Кбайт. Исполняемый файл содержит только 16 Кбайт (текст + инициализированные данные), плюс короткий заголовок, в котором ОС указывается выделить программе дополнительно 4 Кбайт после неинициализированных данных и обнулить их перед выполнением программы. Этот трюк позволяет сэкономить 4 Кбайт нулей на диске в исполняемом файле.

В отличие от текстового сегмента, который не может изменяться, сегмент данных может модифицироваться. Программы изменяют свои переменные постоянно. Более того, многим программам требуется выделение дополнительной памяти динамически, во время выполнения. Чтобы реализовать это, ОС UNIX разрешает сегменту данных расти при динамическом выделении памяти программам и уменьшаться при освобождении памяти программами. Программа может установить размер своего сегмента данных с помощью системного вызова brk. Таким образом, чтобы получить больше памяти, программа может увеличить размер своего сегмента данных. Этим системным вызовом пользуется библиотечная процедура malloc, используемая для выделения памяти.

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

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 ср src dest

запускается программа ср со строкой «ср src dest» в стеке, что позволяет ей определить имена файлов, с которыми ей предстоит работать. Строка представляется в виде массива указателей па 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 рис. 1(б, в) видим два процесса, А и В, совместно использующие общий текстовый сегмент. Отображение выполняется аппаратным обеспечением виртуальной памяти.

Сегменты данных и стека никогда не бывают общими, кроме как после выполнения системного вызова fork, и то только те страницы, которые не модифицируются любым из процессов. Если размер любого из сегментов должен быть увеличен, то отсутствие свободного места в соседних страницах памяти не является проблемой, так как соседние виртуальные страницы памяти не обязаны отображаться на соседние физические страницы.

На некоторых компьютерах аппаратное обеспечение поддерживает раздельные 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 раздельных адресных пространств можно получить 232 бит адресного пространства для команд и еще 232 бит 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 рис. 2 показан файл, одновременно отображенный на адресные пространства двух процессов по различным виртуальным адресам.

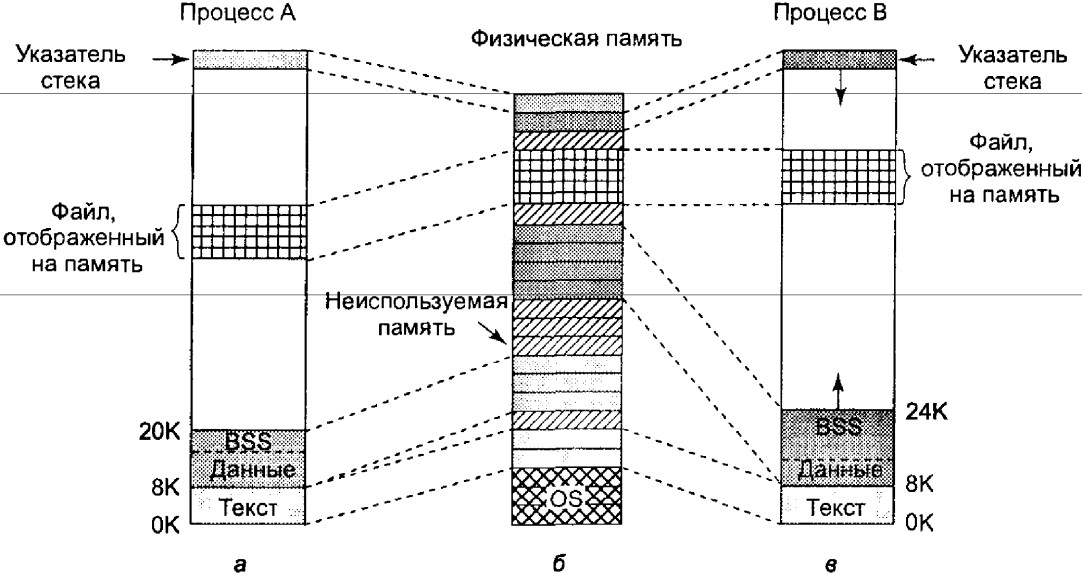


Рис. 2. Два процесса совместно используют один отображенный на память файл

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 функцию malloc (определенную стандартом ANSI С). 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.

На практике в большинстве систем UNIX есть системные вызовы для управления памятью. Наиболее распространенные системные вызовы перечислены в табл. 1. В случае ошибки возвращаемое значение s равно — 1; переменные а и addr представляют собой адреса в памяти, len — это длина, параметр prot управляет защитой, flags содержит различные биты, fd — это дескриптор файла, a offset — смещение. Системный вызов brk указывает размер сегмента данных, задавая адрес первого байта за его пределами. Если новое значение больше старого, сегмент данных увеличивается, в противном случае он уменьшается.

Таблица 1. Некоторые системные вызовы для управления памятью

|  |  |
| --- | --- |
| Системный вызов | Описание |
| s=brk(addr) | Изменить размер сегмента данных |
| a=mmap(addr, len, prot, flags, fd, offset) | Отобразить файл на память |
| s=unmap(addr, len) | Отменить отображения файла на память |

Системные вызовы mmap и unmap 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 ОС определяет этот адрес сама и возвращает его в а. 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, unmap, 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;

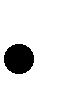
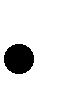
1. Системному вызову fork требовалась память для дочернего процесса.
2. Системный вызов brk собирался расширить сегмент данных.
3. Разросшемуся стеку требовалась дополнительная память.

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 блокированные (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. Таким образом, хорошими кандидатами на выгрузку были процессы, потребившие большое количество процессорного времени, либо находящиеся в памяти уже достаточно долгое время, даже если большую его часть они занимались вводомвыводом. Если блокированных процессов не было, тогда на основе тех же критериев выбирался готовый процесс.

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:

 на диске не оставалось процессов, готовых к работе,  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 перекачки учитывалось при помощи связного 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 ОС UNIX для компьютеров PDP-11 и Intcrdata, 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 опишем строение версии 4BSD, но System V 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, то 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. Если пользовательской структуры и таблицы страниц нет в памяти, то процесс не может быть запущен, пока свопер не загрузит их.

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 поддержать 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 — ink, как показано на рис. 3). 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.

Организация памяти в 4BSD показана на рис. 3. Память делится на три части. Первые две части, ядро ОС 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. При килобайтных страничных блоках и 16-байтовых записях карты памяти на карту памяти расходуется менее 2 % от общего объема памяти. Первые два поля записи карты памяти используются только тогда, когда соответствующий страничный блок находится в списке свободных страниц. В этом случае они сшивают свободные страницы в двусвязный список. Следующие три записи используются, когда страничный блок содержит информацию. 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.

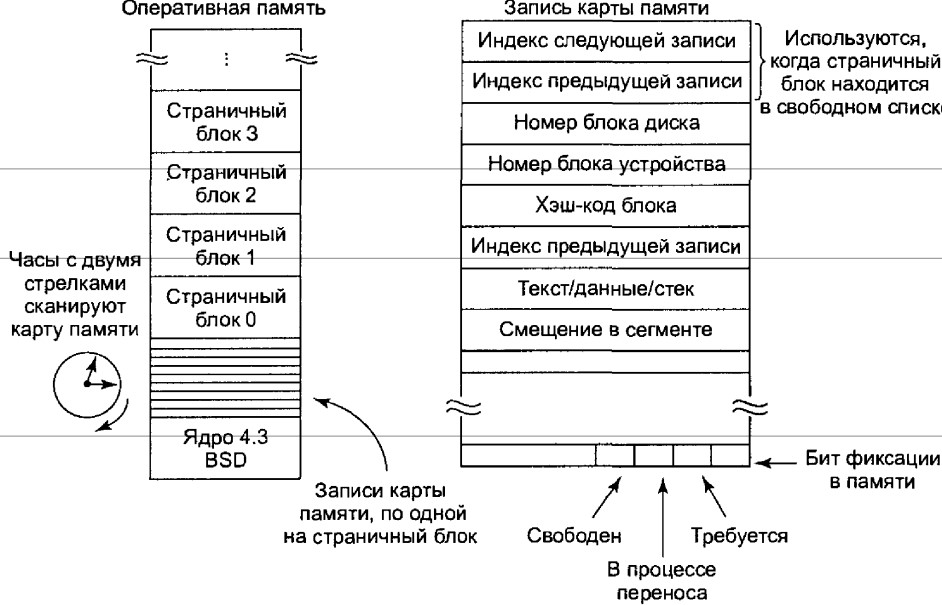


Рис. 3. Карта памяти в 4BSD

При запуске процесс может вызвать страничное прерывание, если одной или нескольких его страниц не окажется в памяти. При страничном прерывании ОС берет первый страничный блок из списка свободных 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 ОС 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, ПО расплачивалось за недостаточно развитую схему аппаратного обеспечения.

Изначально в Berkley UNIX использовался **основной алгоритм часов**, но затем было обнаружено, что при больших объемах ОЗУ проходы занимают слишком много времени. Тогда алгоритм был заменен **алгоритмом часов с двумя стрелками** (см. рис. 3). 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 ОС 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, изучаются 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 nice (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, поэтому 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. Свопер 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 ОС 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 течение 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 4BSD.

Рассмотрим ситуацию, в которой количество свободных страниц на единицу меньше, чем lotsfree, поэтому страничный демон освобождает одну страницу, чтобы привести количество свободных страниц в соответствие со значением lotsfree. 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 min 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. 4). 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 CPU Alpha, она также используется (в упрощенном виде) для всех архитектур. Каждый виртуальный адрес разбивается на четыре поля, как показано на рис. 4. Каталоговое 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 CPU 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.

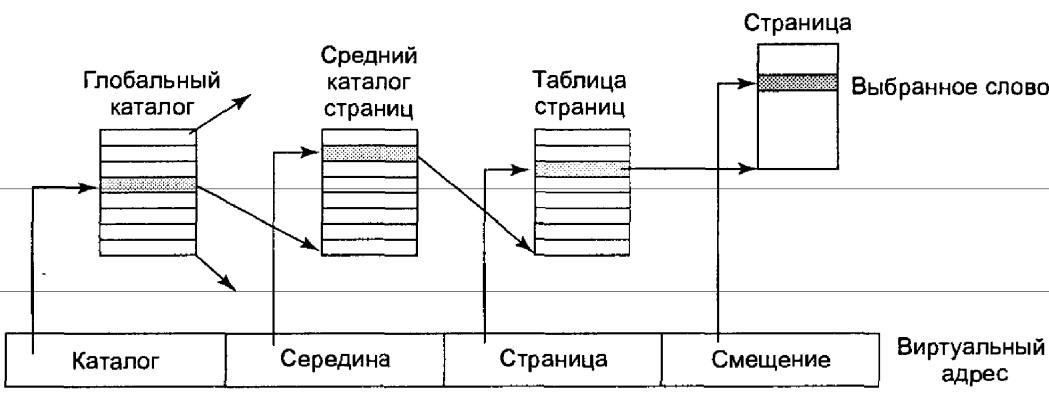


Рис. 4. В Linux используются трехуровневые таблицы страниц

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 *н* страничном кэше, потребуется снова, прежде чем она будет удалена из памяти, ее можно быстро объявить находящейся в памяти.

Кроме этого, ОС 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» алгоритм**. Он будет описан ниже.

Основная идея управления блоками памяти заключается в следующем. Изначально память состоит из единого непрерывного участка. В нашем примере на рис. 5(а) размер этого участка равен 64 страницам. Когда поступает запрос на выделение памяти, он сначала округляется до степени двух, например до 8 страниц. Затем весь блок памяти делится пополам, рис. 5(б). Так как получившиеся в результате этого деления надвое участки памяти все еще слишком велики, нижняя половина делится пополам еще, рис. 5(в) и еще рис. 5(г). Теперь получили участок памяти нужного размера. Этот участок предоставляется обратившемуся процессу, затененный на рис. 5(г).

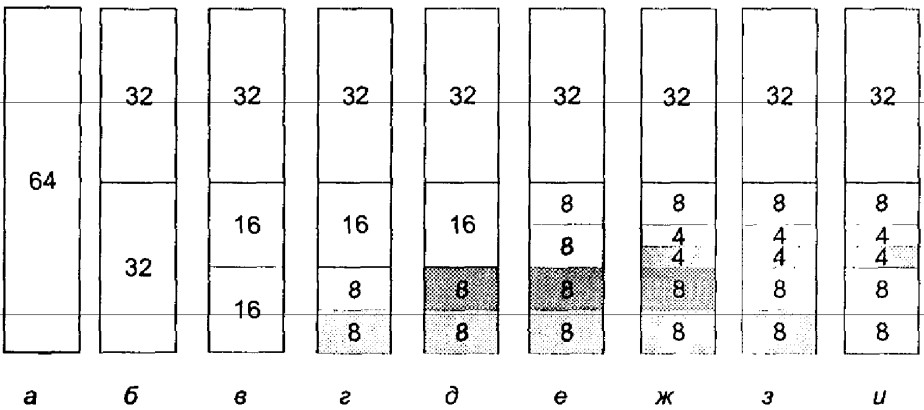


Рис. 5. Этапы работы дружественного алгоритма

Теперь предположим, что приходит второй запрос на 8 страниц. Он может быть удовлетворен немедленно, рис. 5(д). Следом поступает запрос на 4 страницы. При этом делится надвое наименьший участок, рис. 5(е) и выделяется половина от половины, рис. 5(ж). Затем освобождается второй 8-страничный участок, рис. 5(з). Наконец, освобождается оставшийся 8-страничный участок. Поскольку эти два участка были «приятелями», то есть они вышли из одного 16-страничного блока, они снова объединяются в 16-страничный блок, рис. 5(и).

ОС 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 «приятельского» 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.

ОС Linux является системой, предоставляющей страницы по требованию, без предварительной загрузки страниц и без концепции рабочего набора (хотя в ней есть системный вызов для указания пользователем страницы, которая ему может скоро понадобиться). Текстовые сегменты и отображаемые на адресное пространство памяти файлы подгружаются из соответствующих им файлов на диске. Все остальное выгружается либо в область подкачки, если она присутствует, либо в файлы подкачки фиксированной длины, которых может быть от одного до восьми. Файлы подкачки могут динамически добавляться и удаляться, и у каждого есть свой приоритет. Выгрузка страниц в отдельный раздел диска, доступ к которому осуществляется как к отдельному устройству, не содержащему файловой системы, более эффективна, чем выгрузка в файл, по нескольким причинам. **Во-первых**, не требуется преобразование блоков файла в блоки диска. **Во-вторых**, физическая запись может быть любого размера, а не только размера блока файла. **В-третьих**, страница всегда пишется прямо на устройство в виде единого непрерывного участка, а при записи в файл подкачки это может быть и не всегда так.

Страницы на устройстве подкачки или дисковом разделе подкачки не выделяются, пока они не потребуются. Каждое устройство или файл подкачки начинается с битового массива, в котором сообщается, какие страницы свободны. Когда страница, у которой нет места хранения на диске, должна быть удалена из памяти, из файлов (или разделов) подкачки, в которых еще есть свободное место, выбирается файл с наивысшим приоритетом, и в нем выделяется место для страницы. Как правило, у раздела подкачки (если таковой имеется) более высокий приоритет, чем у любого файла подкачки. Координата страницы на диске записывается в таблицу страниц.

Алгоритм замещения страниц работает следующим образом. Система Linux пытается поддерживать некоторые страницы свободными, чтобы их можно было предоставить при необходимости. Конечно, этот пул страниц должен постоянно пополняться, поэтому реальный алгоритм страничной подкачки заключается в том, как это происходит. Во время загрузки процесс ink запускает страничный демон kswapd, который работает раз в секунду. Он проверяет, есть ли достаточное количество свободных страниц. Если да, он отправляется спать еще секунду, хотя он может быть разбужен и раньше, если внезапно понадобятся дополнительные страницы. Страничный демон состоит из цикла, который выполняется до шести раз с возрастающей срочностью. Почему шесть? Вероятно, автор программы думал, что четырех будет недостаточно, а восемь будет слишком много. В отдельных местах ОС Linux реализована именно так.

Тело цикла выполняет обращения к трем процедурам, каждая из которых пытается получить различные типы страниц. Значение срочности передается в виде параметра, сообщающего процедуре, сколько усилий требуется предпринять, чтобы получить некоторые страницы. Как правило, это означает, сколько страниц нужно проверить, прежде чем опустить руки. В результате этот алгоритм сначала выбирает легко доступные страницы каждой категории, после чего переходит к труднодоступным. Когда получено достаточное количество страниц, страничный демон снова отправляется спать.

**Первая процедура** пытается получить те страницы из страничного кэша и буферного кэша файловой системы, к которым в последнее время не было обращений, для чего используется алгоритм часов. **Вторая процедура** ищет совместно используемые страницы, которыми никто из пользователей, похоже, не пользуется активно. **Третья процедура**, пытающаяся получить страницы, используемые одиночными пользователями, является наиболее интересной, поэтому рассмотрим ее подробнее.

Сначала выполняется цикл по всем процессам, в котором определяется, у какого процесса больше всего страниц на данный момент находится в памяти. Как только такой процесс найден, сканируются все его структуры vm\_area\_struct и изучаются все страницы в порядке виртуальных адресов, начиная с того места, на котором этот процесс был отложен в прошлый раз. Если страница недействительна, отсутствует в памяти, используется совместно, фиксирована в памяти или используется для DMA, то она пропускается. Если у страницы установлен бит обращения к ней, этот бит сбрасывается и страница отправляется в резерв. Если же бит сброшен, эта страница отнимается у процесса. В результате данный алгоритм подобен алгоритму часов (с той разницей, что страницы не сканируются в порядке FIFO).

Если страница, выбранная для удаления из памяти, чистая, она удаляется немедленно. Если страница «грязная» и у нее есть место резервного хранения на диске, она устанавливается в очередь записи на диск. Наконец, если у «грязной» страницы нет места резервного хранения на диске, она отправляется в страничный кэш, из которого она может быть снова получена позднее, если обращение к ней поступит прежде, чем она будет фактически выгружена на диск. В основе идеи сканирования страниц в порядке виртуальных адресов лежит надежда на то, что страницы, расположенные близко друг к другу в виртуальном адресном пространстве, скорее всего будут использоваться или не использоваться вместе, как единая группа, поэтому их следует записывать на диск как группу, а затем вместе считывать в память.

В управлении памятью принимает участие еще один демон, bdflush. Он периодически просыпается (а в некоторых случаях его явно будят), чтобы проверить, не превысило ли количество «грязных» страниц определенного предельного уровня. Если превысили, демон начинает сохранять их на диске.

## Ввод-вывод в системе UNIX

Система ввода-вывода в UNIX довольно проста. Как правило, все устройства вводавывода выглядят как файлы, и доступ к ним осуществляется с помощью тех же системных вызовов read и write, которые используются для доступа к обычным файлам. В некоторых случаях должны быть заданы параметры устройства, для чего служит специальный системный вызов.

### Основные понятия

Как и у всех компьютеров, у машин, работающих под управлением ОС 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 ОС 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 ср 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 ср 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.

Хотя к большинству символьных специальных файлов **невозможен произвольный доступ**, ими часто бывает нужно управлять таким способом, который не используется для блочных специальных файлов. Рассмотрим, например, строку, введенную с клавиатуры и отображенную на экране. Когда пользователь по ошибке нажимает не ту клавишу и хочет заменить последний символ, он нажимает специальную клавишу. Некоторые пользователи предпочитают использовать для этого клавишу **BACKSPACE**, а другие любят пользоваться клавишей **DEL** для удаления всей только что набранной строки тоже имеется большой выбор средств. Традиционно использовался символ @, но с распространением электронной почты (использующей символ @ в почтовом адресе) многие системы перешли на использование комбинации клавиш CTRL+U или других символов. Требуется и особая клавиша для прерывания работающей программы. Здесь также у разных пользователей различные вкусы.

В ОС 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 рис. 6.

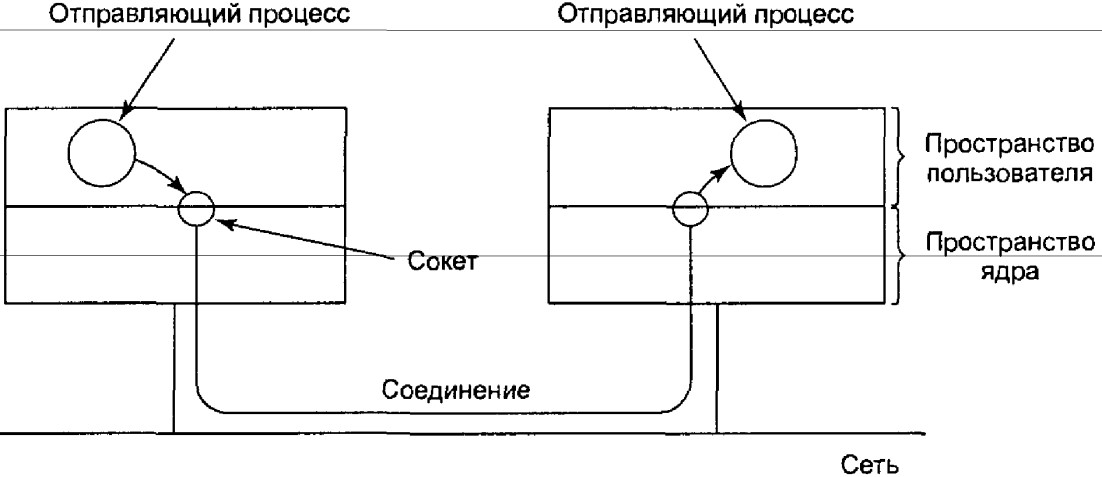


Рис. 6. Использование сокетов для соединения с сетью

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

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 прибудут на другой конец канала и прибудут 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 Data Protocol — пользовательский протокол данных). Все эти протоколы были разработаны для сети ARPANET Министерства обороны США и теперь составляют основу Интернета. Для надежного потока пакетов специального протокола нет.

Прежде чем сокет может быть использован для работы в сети, с ним должен быть связан адрес. Этот адрес может принадлежать к одному из нескольких пространств адресов. Наиболее распространенным пространством является пространство адресов Интернета, использующее 32-разрядные числа для идентификации конечных адресатов в протоколе IPv4 и 128-разрядные числа в протоколе IPv6.

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 прибудут данные. Другая сторона обращается к системному вызову connect, задавая в параметрах дескриптор файла для локального сокета и адрес удаленного сокета. Если удаленный компьютер принимает вызов, 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 ОС 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, перечисленные в табл. 2, используются для задания скорости терминала. Для управления вводом и выводом предоставлены различные вызовы, так как некоторые модемы работают в несимметричном режиме. Например, старые системы videotex 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 2. Основные вызовы стандарта POSIX для управления терминалом

|  |  |
| --- | --- |
| Системные вызовы | Описание |
| s=cfsetospeed(&termios, speed) | Задать исходящую скорость |
| s=cfsetispeed(&termios, speed) | Задать входящую скорость |
| s=cfgetospeed(&termios, speed) | Получить исходящую скорость |
| s=cfgetispeed(&termios, speed) | Получить входящую скорость |
| s=tcsetattr(fd, opt, &termios) | Задать атрибуты |
| s=tcgetattr(fd, &termios) | 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 ОС 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 ОС 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 — bdevsw для 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.

Наиболее важные поля массива cdevsw для типичной системы могут выглядеть, как показано в табл. 3. Каждый ряд таблицы относится к одному устройству ввода-вывода (то есть одному драйверу). Колонки соответствуют функциям, которые должны поддерживаться всеми драйверами. Существуют также некоторые другие функции. Когда выполняется операция с символьным специальным файлом, система обращается к массиву cdevsw, выбирая соответствующий ряд (структуру), затем обращается к функции в соответствующей записи структуры (колонка таблицы), чтобы выполнить требуемое действие. Таким образом, каждый ряд таблицы содержит указатели на функции, содержащиеся в одном драйвере.

**Таблица 3. Некоторые из полей типичной таблицы cdevsw**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Устройство | Open | Close | Read | Write | Ioctl | Other |
| Null | null | ull | null | null | null | … |
| Память | null | null | mem\_read | mem\_write | null | … |
| Клавиатура | k\_open | k\_close | k\_read | error | k\_ioctl | … |
| Tty | tty\_open | ttyclose | tty\_read | tty\_write | tty\_ioctl |  |
| Принтер | lp\_open | lp\_close | error | lp\_write | lp\_ioctl | … |

Каждый драйвер разделен на несколько частей. Верхняя часть драйвера работает в режиме вызывающего процесса и служит интерфейсом с остальной системой UNIX. Нижняя часть работает в контексте ядра и взаимодействует с устройством. Драйверам разрешается обращаться к процедурам ядра для выделения памяти, управления таймером, управления DMA и т. д. Набор функций ядра, которые они могут вызывать, определен в документе, называемом **Интерфейс Драйвер-Ядро**.

Система ввода-вывода разделена на два основных компонента: обработку блочных специальных файлов и обработку символьных специальных файлов. Рассмотрим по очереди 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** (рис. 7). 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 удается избежать. **Буферный кэш значительно улучшает производительность системы.**

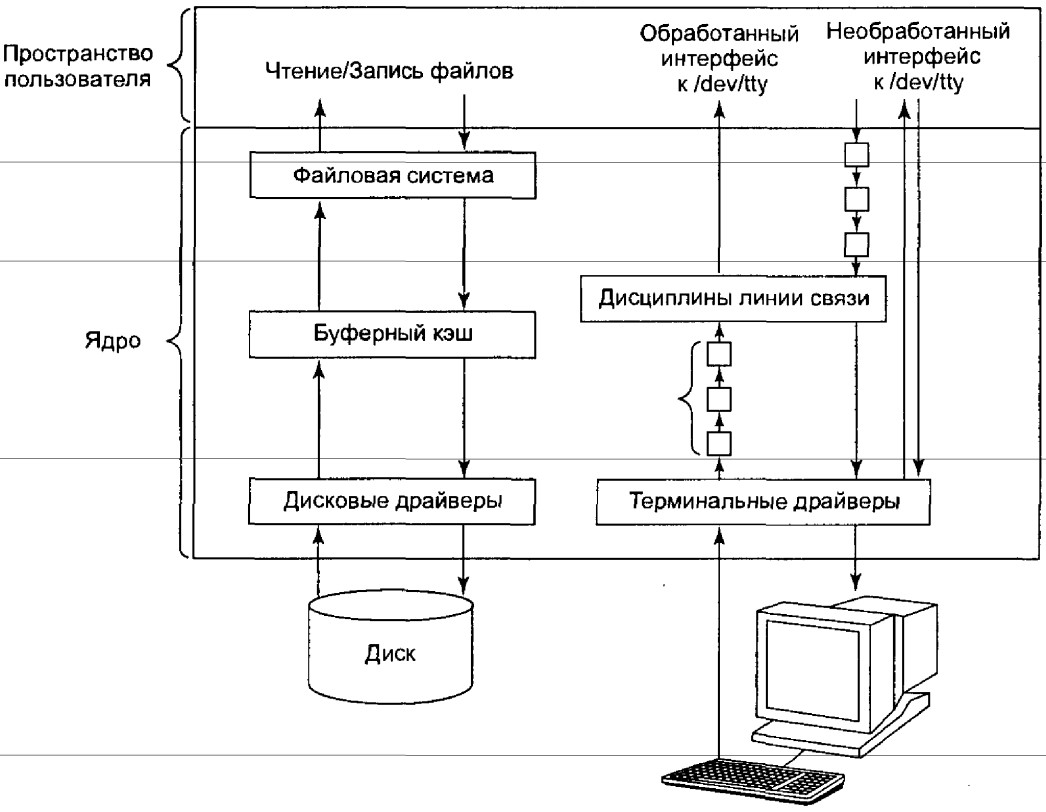


Рис. 7. Система ввода-вывода BSD UNIX

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.

Буферный кэш поддерживает не только операцию чтения с диска, но также и запись на диск. Когда программа пишет блок, этот блок не попадает напрямую на диск, а отправляется в кэш. Только когда кэш наполняется, модифицированные блоки кэша сохраняются на диске. Чтобы модифицированные блоки не хранились в кэше слишком долго, принудительная выгрузка на диск «грязных» блоков производится каждые 30 с.

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 станций, то есть в тех условиях, в которых росла и развивалась ОС UNIX. Как правило, компьютерный центр формировал ядро ОС, 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.

Все изменилось с появлением системы Linux, ориентированной в первую очередь на поддержку персональных компьютеров. Количество всевозможных устройств вводавывода на персональных компьютерах на порядок больше, чем у мини-компьютеров. Кроме того, хотя у пользователей системы Linux есть (или они легко могут его получить) полный набор исходных текстов ОС, подавляющее большинство пользователей будет испытывать существенные трудности с добавлением нового драйвера, обновлением файлов cdevsw или bdevsw, компоновкой ядра и установкой его как загружаемой системы (не говоря уже о том, что придется делать, если построенное новое ядро откажется загружаться).

В ОС Linux подобные проблемы были решены при помощи концепции подгружаемых модулей. Это куски кода, которые могут быть загружены в ядро во время работы ОС. 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.

При загрузке модуля должно выполняться несколько определенных действий. Вопервых, **модуль должен быть на лету перенастроен на новые адреса**. Во-вторых, **система должна проверить, доступны ли ресурсы, необходимые драйверу** (например, определенные уровни запроса прерывания), и если они доступны, то пометить их как используемые. В-третьих, **должны быть настроены все необходимые векторы прерываний**. В-четвертых, **для поддержки нового типа старшего устройства следует обновить таблицу переключения драйверов**. Наконец, **драйверу позволяется выполнить любую специфическую для данного устройства процедуру инициализации**. Когда все эти этапы выполнены, драйвер является полностью установленным, как и драйвер, установленный статически. Некоторые современные системы UNIX также поддерживают подгружаемые модули.

### 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.

Решение, реализованное в системе BSD, основано на структурах данных, присутствующих в классических системах UNIX, называемых **С-списками** (они показаны в виде квадратиков на рис. 7). 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 Ссписков. 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 называнием **потоков данных**, было разработано Деннисом Ритчи. Это общий подход (рис. 8). (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.

У потока данных всегда есть голова потока у вершины и соединение с драйвером у основания. В поток может быть вставлено столько модулей, сколько необходимо. Обработка может происходить в обоих направлениях, так что каждому модулю может понадобиться одна секция для чтения (из драйвера) и одна секция для записи (в драйвер). Когда процесс пользователя пишет данные в поток, программа в голове потока интерпретирует системный вызов и запаковывает данные в буферы потока, передаваемые от модуля к модулю вниз, при этом каждый модуль выполняет соответствующие преобразования. У каждого модуля есть очередь чтения и очередь записи, так что буферы обрабатываются в правильном порядке. У модулей есть строго определенные интерфейсы, определяемые инфраструктурой потока, что позволяет объединять вместе несвязанные модули.

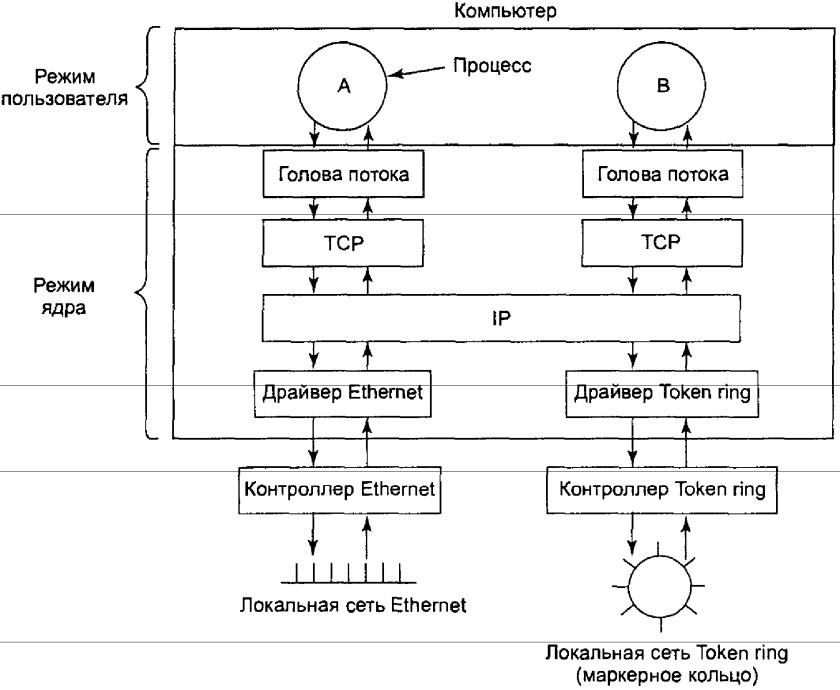


Рис. 8. Пример потоков в System V

В примере на рис. 8 показано, как применяются потоки при использовании Интернетпротокола TCP с двумя локальными сетями различного типа: Ethernet и Token ring. Данный пример иллюстрирует еще одно важное свойство потоков данных — мультиплексирование. Мультиплексный модуль может взять один поток и расщепить его на несколько потоков или, наоборот, объединить несколько потоков в единый поток. Модуль IP в данном примере выполняет обе функции.

## Файловая система UNIX

Прежде всего пользователь видит в любой ОС, включая систему UNIX, файловую систему. В следующих разделах рассмотрим основные идеи файловой системы UNIX, системные вызовы и детали реализации файловой системы. Некоторые принципы файловой системы UNIX были взяты у ОС MULTICS, в то время как многие другие позаимствованы в MS-DOS, Windows и других системах, хотя имеются и уникальные для UNIX идеи. Устройство файловой системы UNIX особенно интересно тем, что оно отчетливо иллюстрирует принцип разумной достаточности и красоту простых решений. При минимальном механизме и сильно ограниченном количестве системных вызовов ОС UNIX тем не менее предоставляет мощную и элегантную файловую систему.

### Основные понятия

**Файл в системе UNIX** — 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 NULL, поэтому допустимы даже имена файлов, состоящие, например, из трех символов возврата каретки (хотя такое имя и не слишком удобно в использовании).

По соглашению многие программы ожидают, что имена файлов должны состоять из основного имени и расширения, отделяемого от основного имени файла точкой (которая в системе UNIX также считается символом). Так, prog.c — это, как правило, программа на языке C, prog.f90 — обычно программа на языке FORTRAN 90, a prog.o — чаще всего объектный файл (выходные данные компилятора). Эти соглашения никак не регулируются ОС, но некоторые компиляторы и другие программы ожидают файлов именно с такими расширениями. Расширения могут иметь произвольную длину, кроме того, файлы могут иметь по нескольку расширений, как, например, prog.java.Z, что, скорее всего, представляет собой сжатую программу на языке Java.

Для удобства использования файлы могут группироваться в **каталоги**. Каталоги хранятся на диске в виде файлов, и до определенного предела с ними можно работать как с файлами. Каталоги могут содержать подкаталоги, что приводит к иерархической файловой системе. Корневой каталог называется / и, как правило, содержит несколько подкаталогов. Символ / также используется для разделения имен каталогов, поэтому имя /usr/ast/x означает файл х, расположенный в каталоге ast, который, в свою очередь, находится в каталоге usr.

Некоторые основные каталоги у вершины дерева показаны в табл. 4.

Таблица 4. Некоторые важные каталоги, существующие в большинстве систем UNIX

|  |  |
| --- | --- |
| Каталог | Содержимое |
| bin | Двоичные (исполняемые) программы |
| dev | Специальные файлы для устройств ввода-вывода |
| etc | Разные системные файлы |
| lib | Библиотеки |
| usr | Каталоги пользователей |

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 рис. 9(а). Здесь показан жесткий диск, который мы будем называть С:, а также гибкий диск, который мы будем называть А:. 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 х в каталог d (предполагая, что по умолчанию выбирается диск С:), следует ввести команду ср А:/х /a/d/x

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

Решение, применяемое в ОС UNIX, заключается в том, чтобы позволить монтировать один диск в дерево файлов другого диска. В примере можем смонтировать дискету в каталог /b, получая в результате файловую систему, показанную на рис. 9(б). 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/х /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.

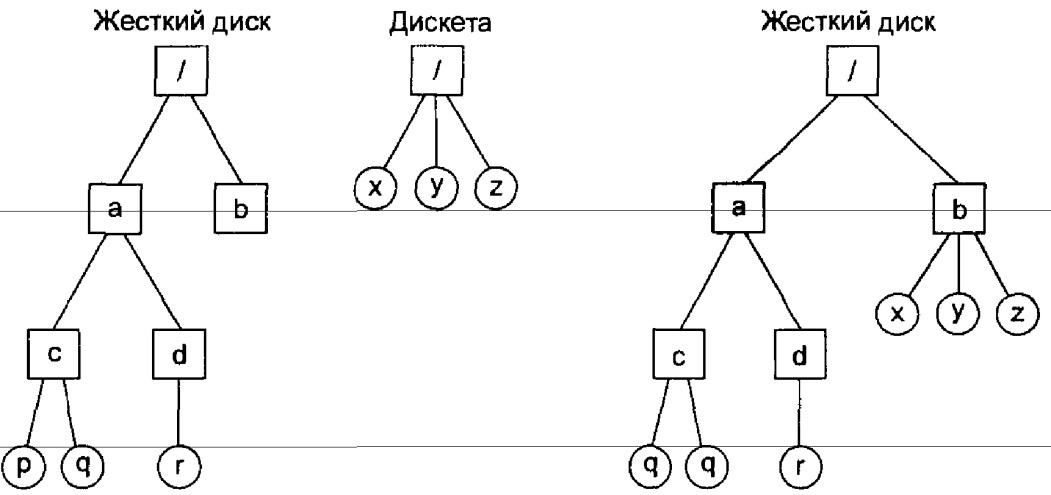


Рис. 9. Раздельные файловые системы (а); после монтирования (б)

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 другим, такой способ координации действии, 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.

По этой причине стандартом POSIX предоставляется гибкий и детальный механизм, позволяющий процессам за одну неделимую операцию блокировать даже единственный байт файла или целый файл, по желанию. Механизм блокировки требует от вызывающего его процесса указать блокируемый файл, начальный байт и количество байтов. Если операция завершается успешно, система создает запись в таблице, в которой указывается, что определенные байты файла заблокированы.

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 монопольная 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 рис. 10(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 б 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, рис. 10(в). Две предыдущие блокировки перекрываются с этой блокировкой. Поэтому обращающийся с новым запросом процесс будет заблокирован и останется заблокированным, пока оба процесса А и С не снимут свою блокировку.

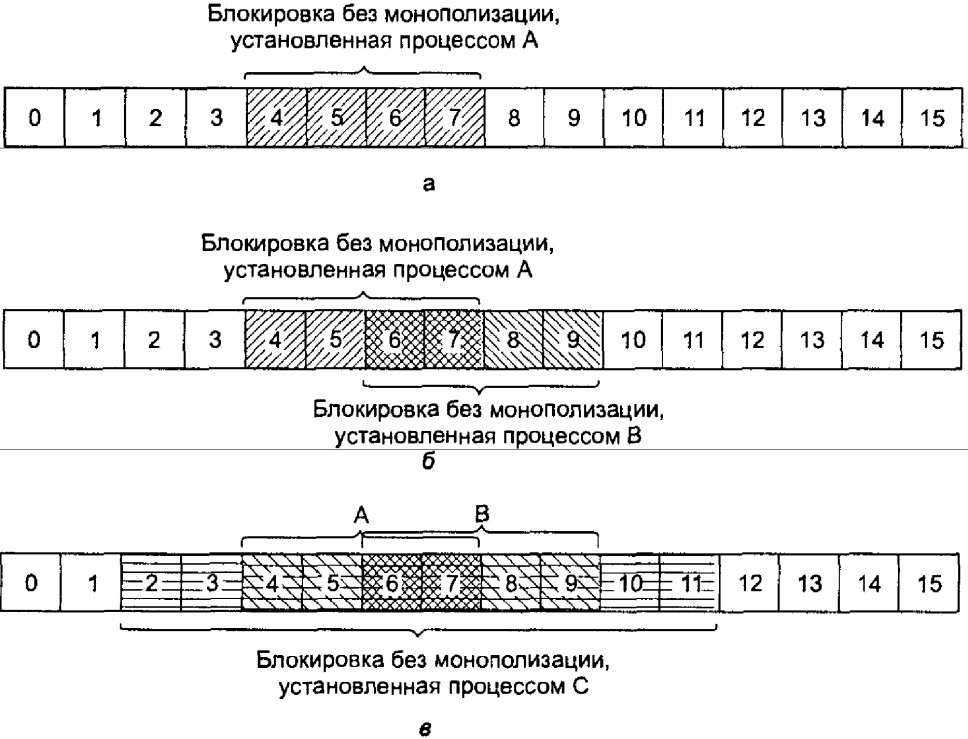


Рис. 10. Файл с одной блокировкой (а); добавление второй блокировки (б); третья блокировка (в)

### Системные вызовы файловой системы в UNIX

Многие системные вызовы относятся к файлам и файловой системе. Сначала рассмотрим системные вызовы, работающие с отдельными файлами. Затем изучим те системные вызовы, которые оперируют каталогами или всей файловой системой в целом. Для создания нового файла можно использовать системный вызов creat. 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. Вопросы защиты файлов и доступа к ним будут рассмотрены ниже.

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.

Теперь продолжим изучение основных файловых системных вызовов, перечисленных в табл. 5. (В случае ошибки возвращаемое значение*s* равно -1*,* fd *—* дескриптор файла, position — смещение в файле.) Чтобы прочитать данные из существующего файла или записать данные в существующий файл, файл сначала нужно открыть с помощью системного вызова open. Этому системному вызову следует указать имя файла, а также режим, в котором он должен быть открыт: для чтения, для записи или и для того и для другого. Также можно указать различные дополнительные параметры. Как и creat, системный вызов open возвращает дескриптор файла, который может быть использован для чтения или записи журнала. Затем файл может быть закрыт при помощи системного вызова close, после чего, чтобы писать в этот файл или читать из него, его снова нужно открыть. Системные вызовы creat и open возвращают наименьший неиспользуемый в данный момент дескриптор файла.

Таблица 5. Некоторые системные вызовы для работы с файлами

|  |  |
| --- | --- |
| Системные вызовы | Описание |
| fd=creat(name, mode) | Один из способов создания нового файла |
| fd=open(file, how, j) | Открыть файл для чтения, записи или и того и другого |
| s=close(fd) | Закрыть открытый файл |
| n=read(fd, buffer, nbytes) | Прочитать данные из файла в буфер |
| n=write(fd, buffer, nbytes) | Записать данные из буфера в файл |
| position=lseek(fd, offset, whence) | Переместить указатель в файле |
| s=stat(name, &buf) | Получить информацию о состоянии файла |
| s=fstat(fd, &buf) | Получить информацию о состоянии файла |
| s=pipe(&fd[0]) | Создать канал |
| s=fcntl(fd, cmd,...) | Блокировка файла и другие операции |

Когда программа начинает выполнение стандартным образом, файлы с дескрипторами 0, 1 и 2 уже открыты для стандартного ввода, стандартного вывода и стандартного потока сообщений об ошибках соответственно. Таким образом, фильтр, например программа *sort,* может просто читать свои входные данные из файла с дескриптором 0, а писать выходные данные в файл с дескриптором 1, не заботясь о том, где располагаются эти файлы. Работа этого механизма обеспечивается оболочкой, которая проверяет, чтобы эти дескрипторы соответствовали нужным файлам, прежде чем программа начнет свою работу.

Чаще всего программы используют системные вызовы read и write. У обоих вызовов по три параметра: дескриптор файла (указывающий, с каким из открытых файлов будет производиться операция чтения или записи), адрес буфера (сообщающий, куда положить данные или откуда их взять), а также счетчик байтов (указывающий, сколько байтов следует прочитать или записать). Вот и все. Очень простая схема. Пример типичного вызова:

#### n = read(fd, buffer, nbytes)

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.

Для каждого файла ОС UNIX хранит такие сведения, как тип (режим) файла (обычный, каталог, специальный файл), его размер, время последнего изменения и т. д. Программы могут получить эту информацию при помощи системного вызова stat. Первый параметр представляет собой имя файла. Второй является указателем на структуру, в которую следует поместить требуемую информацию. Системный вызов fstat представляет собой то же самое, что и системный вызов stat, с той разницей, что он работает с уже открытым файлом (имя которого может быть неизвестно), а не с путем. Системный вызов pipe используется для создания каналов оболочки. Он создает псевдофайл для буферирования данных, которыми обмениваются компоненты канала, и возвращает дескрипторы файлов для чтения и записи буфера. В команде sort <in 1 head -30

дескриптор файла 1 (стандартный вывод) в процессе, выполняющем программу sort*,* будет настроен оболочкой на запись в канал, а дескриптор файла 0 (стандартный ввод) в процессе, выполняющем программу head*,* будет настроен на чтение из канала. Программа sortпросто читает из файла с дескриптором 0 (установлен на файл in*)* и пишет в файл с дескриптором 1 (канал), даже не зная, что оба эти файла переопределены. Если бы ввод и вывод не были перенаправлены, программа sortчитала бы данные с клавиатуры и выводила бы их на экран (устройства по умолчанию). Подобным же образом, когда программа headсчитывает входные данные из файла с дескриптором 0, она получает данные, которые программа sortпоместила в буфер канала, даже не зная, что канал используется. Вот пример того, как простая концепция (перенаправление потока данных) плюс простая реализация (файлы с дескрипторами 0 и 1) вместе дают мощный инструмент (соединение программ произвольным образом без необходимости их модификации).

Последний системный вызов в табл. 5 — это fcntl. Он используется для блокировки и разблокирования файлов, а также некоторых других специфических для файлов операций.

Рассмотрим теперь некоторые системные вызовы, относящиеся скорее к каталогам или файловой системе в целом, нежели к какому-либо конкретному файлу. Наиболее часто употребляемые системные вызовы перечислены в табл. 6. (В случае ошибки возвращаемое значение s равно -1, dir идентифицирует каталог, a direct представляет собой запись каталога.) Каталоги создаются и удаляются при помощи системных вызовов mkdir и rmdir соответственно. Каталог может быть уничтожен, только если он пуст.

При создании связи с файлом создается новая запись в каталоге, указывающая на существующий файл. Связь создается при помощи системного вызова link. В параметрах этого системного вызова указываются исходное и новое имя. Записи в каталоге удаляются системным вызовом unlink. Когда удаляется последняя связь с файлом, файл также автоматически удаляется. Если для файла не было создано ни одной связи» то при первом же обращении к системному вызову unlink файл будет удален.

Таблица 6. Некоторые системные вызовы, имеющие отношение к работе с каталогом

|  |  |
| --- | --- |
| Системные вызовы | Описание |
| s=mkdir(path, mode) s=rmdir(path) s=link(oldpath, newpath) s=unlink(path) s=chdir(path) dir=opendir(path) s=closedir(dir) dirent=readdir(dir) rewinddir(dir) | Создать новый каталог  Удалить каталог  Создать связь с существующим файлом  Удалить связь  Изменить рабочий каталог  Открыть каталог для чтения  Закрыть каталог  Прочитать одну запись каталога  Установить указатель в каталоге на первую запись |

Рабочий каталог можно изменить при помощи системного вызова chdir. После выполнения этого системного вызова будут по-другому интерпретироваться относительные имена путей.

Последние четыре системных вызова в табл. 6 предназначены для чтения каталогов. Каталоги могут открываться, закрываться и читаться аналогично обычным файлам. Каждое обращение к системному вызову readdir возвращает ровно одну запись каталога фиксированного формата. Пользователям запрещено писать в каталоги (это делается, чтобы пользователи случайно не нарушили целостности системы). Файлы могут добавляться к каталогу при помощи системных вызовов creat и link, а удаляться с помощью системного вызова unlink. В ОС UNIX нет способа обращаться к файлу по расположению его описателя в каталоге, но есть системный вызов rewinddir, позволяющий начать читать открытый каталог с начала.

### Реализация файловой системы UNIX

В данном разделе будет описана реализация 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 рис. 11. 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.

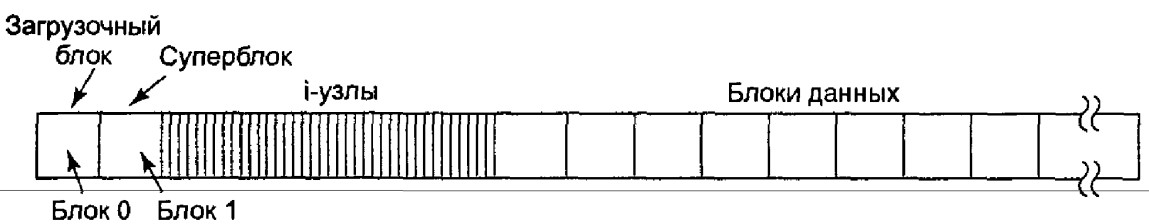


Рис. 11. Расположение классической файловой системы UNIX на диске

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-байт-ного имени файла и номера *i* -узла. 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. В табл. 7 показана структура *i* -узла, используемая в AT&T версиях системы UNIX от Version 7 до System V.

Таблица 7. Структура *i* -узла в System V

|  |  |  |
| --- | --- | --- |
| Поле | Байты | Описание |
| Mode | 2 | Тип файла, биты защиты, биты setuid и setgid |
| Minks | 2 | Количество каталоговых записей, указывающих на этот i-узел |
| Uid | 2 | UID владельца файла |
| Gid | 2 | GID владельца файла |
| Size | 4 | Размер файла в байтах |
| Addr | 39 | Адрес первых 10 дисковых блоков файла и 3 косвенных блоков |
| Gen | 1 | Номер «поколения» (увеличивается на единицу при каждом использовании *i* -узла) |
| Atime | 4 | Время последнего доступа к файлу |
| Mtime | 4 | Время последнего изменения файла |
| Ctime | 4 | Время последнего изменения i-узла (не считая других раз) |

Поиск файла по абсолютному пути, например /usr/ast/file*,* немного сложнее. Сначала система находит корневой каталог, как правило, использующий *i* -узел с номером 2 (*i* -узел 1 обычно резервируется для хранения дефектных блоков). Затем он ищет в корневом каталоге строку «usr», чтобы получить номер *i* -узла каталога /usr*.* Затем считывается этот *i* -узел, и из него извлекаются номера блоков, в которых располагается каталог /usr*.* После этого считывается каталог /usr*,* в котором ищется строка «ast». Когда нужная запись найдена, из нее извлекается номер *i* -узла для каталога /usr/astи т. д. Таким образом, использование относительного имени файла не только удобнее для пользователя, но также представляет существенно меньшее количество работы для файловой системы.

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>Х

то ожидается, что команда p1 будет писать свои выходные данные в файл х, а команда р2 будет также писать свои выходные данные в файл х, начиная с того места, на котором остановилась команда р1.

Когда оболочка запустит процесс p1*,* файл хбудет изначально пустым, поэтому команда p1просто начнет запись в файл в позиции 0. Однако когда она закончит свою работу, требуется некий механизм передачи указателя в файле х от процесса р1процессу р2*.* Если же позицию в файле хранить просто в таблице дескрипторов файла, процесс р2начнет запись в файл с позиции 0.

Способ передачи указателя от одного процесса другому показан на рис. 12. Трюк состоит в том, чтобы ввести в обращение новую таблицу, **таблицу открытых файлов,** между таблицей дескрипторов файлов и таблицей *i* -узлов, и хранить указатели в файле, а также бит чтения/записи в ней. На рисунке родительский процесс представляет собой оболочку, а дочерний процесс сначала является процессом р1, а затем процессом р2. Когда оболочка создает процесс р1, его пользовательская структура (включая таблицу дескрипторов файлов) представляет собой точную копию такой же структуры оболочки, поэтому обе они содержат указатели на одну и ту же таблицу открытых файлов. Когда процесс р1 завершает свою работу, таблица дескрипторов файлов оболочки продолжает указывать на таблицу открытых файлов, в которой содержится позиция в файле процесса p1. Когда теперь оболочка создает процесс р2, новый дочерний процесс автоматически наследует позицию в файле. При этом ни сам новый процесс, ни оболочка даже не должны знать текущее значение этой позиции.

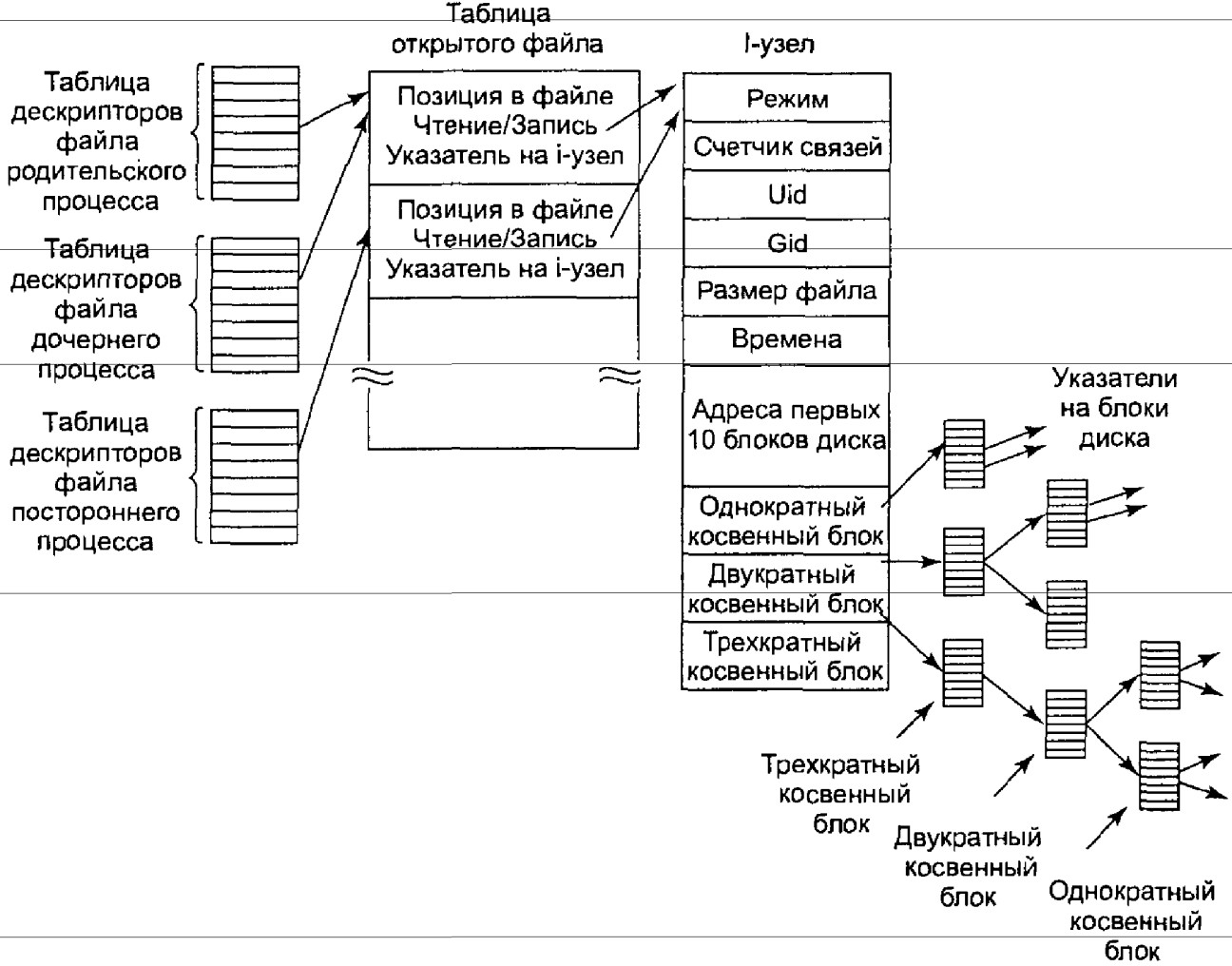


Рис. 12. Связь между таблицей дескрипторов файлов, таблицей открытых файлов и таблицей *i* -узлов

Однако если какой-нибудь посторонний процесс откроет файл, он получит свою собственную запись в таблице открытых файлов со своей позицией в файле, что как раз и нужно. Таким образом, **задача таблицы открытых файлов заключается в том, чтобы позволить родительскому и дочернему процессам совместно использовать один указатель в файле, но для посторонних процессов выделять отдельные указатели**.

Итак, мы показали, как работающие процессы получают доступ к позиции в файле и к *i* -узлу файла, *i* -узел содержит дисковые адреса первых 10 блоков файла. Если позиция в файле попадает в его первые 10 блоков, то считывается нужный блок файла, а данные копируются пользователю. Для поддержки файлов, длина которых превышает 10 блоков, в *i* -узле содержится дисковый адрес **одинарного косвенного блока** (см. рис. 12). Этот блок содержит дисковые адреса дополнительных блоков файла. Например, если размер блока составляет 1 Кбайт, а дисковый адрес занимает 4 байт, то одинарный косвенный блок может хранить до 256 дисковых адресов. Такая схема позволяет поддержать файлы размером до 256 Кбайт.

Для файлов, размер которых превосходит 256 Кбайт, используется **двойной косвенный блок**. Он содержит адреса 256 одинарных косвенных блоков, каждый из которых содержит адреса 256 блоков данных. Такая схема позволяет поддержать файлы размером до 10 216 блоков (67 119 104 байт). Если и этого оказывается недостаточно, в *i* -узле есть место для **тройного косвенного блока**. Его указатели показывают на 256 двойных косвенных блоков.

### 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, closedir, 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. 13. 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. 13 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-битовой границы. За ним могут следовать дополнительные байты-заполнители.

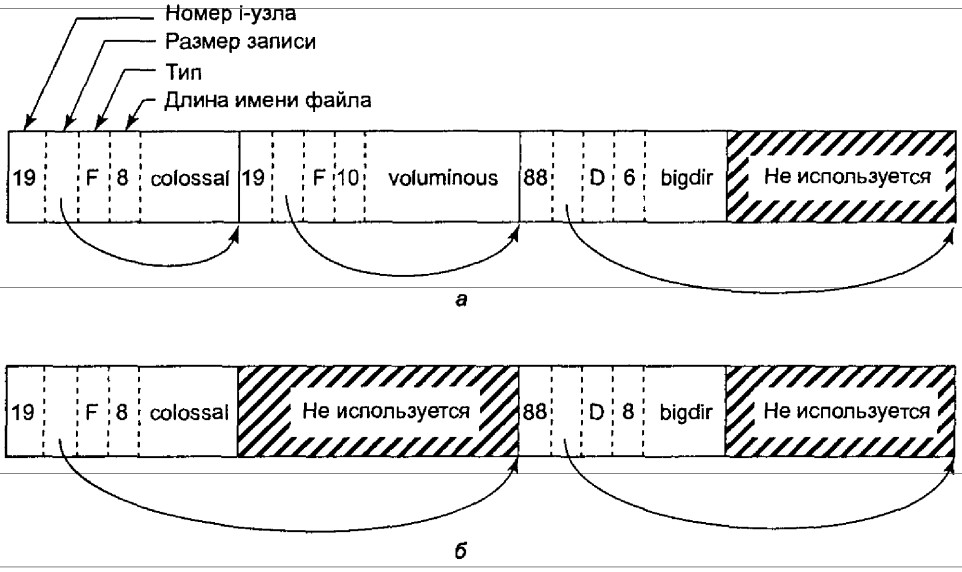


Рис. 13. Каталог BSD с тремя файлами (а); тот же каталог после удаления файла voluminous (б)

На рис. 13(б) 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 ОС Linux использовалась файловая система ОС 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 ОС 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 ОС Linux также поддерживает еще более десятка файловых систем, используя для этого файловую систему NFS (описанную далее). При компоновке ОС 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 рис. 14. 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. 14. Размещение файловой системы Ext2 на диске

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 8килобайтные 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.

Другой файловой системой Linux является файловая система **/ргос** (process — процесс). Идея этой файловой системы изначально была реализована в 8-й редакции ОС UNIX, созданной лабораторией Bell Labs, а позднее скопированной в 4.4BSD и System V. Однако в ОС Linux данная идея получила дальнейшее развитие. Основная концепция этой файловой системы заключается в том, что для каждого процесса системы создается подкаталог в каталоге /рrос*.* Имя подкаталога формируется из PID процесса в десятичном формате. Например, /рrос/619 *—* 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 ОС Linux, относятся к файлам и каталогам, расположенным в каталоге /рrос*.* Они содержат информацию о CPU, 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 /рrос*,* чтобы изменить параметры системы.

### Файловая система NFS

Сети играют главную роль в ОС UNIX с самого начала (первая сеть в системе UNIX была построена для переноса нового ядра с 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. Интересны три аспекта файловой системы NFS: архитектура, протокол и реализация.

### **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 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 рис. 15. (Каталоги показаны на рисунке в виде квадратов, а файлы — в виде кружков.)

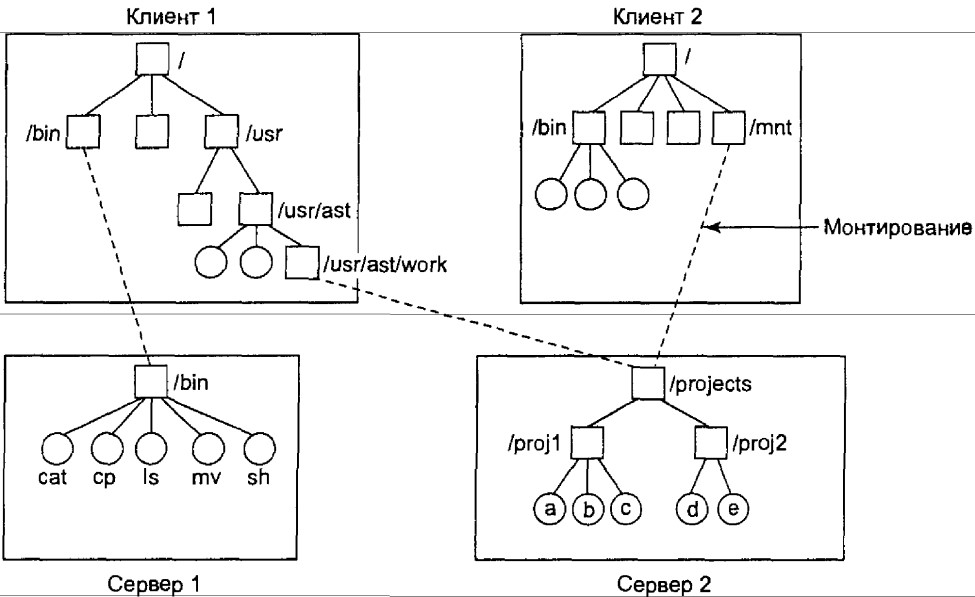


Рис. 15. Примеры монтирования удаленных файловых систем

В этом примере клиент 1 смонтировал каталог binсервера 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 projectsсервера 1 в своем каталоге usr/ast/work*,* поэтому он теперь может получать доступ к файлу акак 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 ОС 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 клиентсервер. **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 ОС UNIX, прежде чем перейти в многопользовательский режим, запускает сценарий оболочки /etc/rc*.* 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 ОС (не происходит даже контакта с сервером). Вместо этого при первом обращении к удаленному файлу (когда файл открывается) ОС 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/rc, 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).

С другой стороны, при таком подходе неявно подразумевается, что все указанные как альтернативные файловые системы идентичны для автомонтировки. Так как файловая система NFS не предоставляет поддержки репликации файлов или каталогов, то следить за идентичностью всех файловых систем должен сам пользователь. Поэтому автомонтировка, как правило, используется для файловых систем, в которых клиенту разрешено только чтение. Такие файловые системы обычно содержат системные двоичные файлы, а также другие редко изменяемые файлы.

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 ОС UNIX, за исключением, как ни странно, системных вызовов open и close.

Пропуск системных вызовов open и close не случаен. Это сделано намеренно. Нет необходимости открывать файл, прежде чем прочитать его. Также не нужно закрывать файл после того, как данные из него прочитаны. Вместо этого, чтобы прочитать файл, клиент посылает на сервер сообщение lookup, содержащее имя файла, с запросом найти этот файл и вернуть дескриптор файла, представляющий собой структуру, идентифицирующую файл (то есть содержащую идентификатор файловой системы и номер *i* -узла вместе с прочей информацией). В отличие от системного вызова open, операция lookup не копирует никакой информации во внутренние системные таблицы. Системному вызову read подается на входе дескриптор файла, который предстоит прочитать, смещение в файле, а также количество байтов, которые нужно прочитать. Таким образом, каждое сообщение является самодостаточным. Преимущество такой схемы заключается в том, что серверу не нужно помнить что-либо об открытых соединениях между обращениями к нему. Поэтому если на сервере произойдет сбой с последующей перезагрузкой, не будет потеряно никакой информации об открытых файлах, так как терять просто нечего. Такие серверы называются **серверами без** **состояния.**

К сожалению, метод файловой системы NFS усложняет достижение точной файловой семантики системы UNIX, Например, в ОС UNIX файл может быть открыт и заблокирован, так что никакой другой процесс не смог получить к этому файлу доступ. Когда файл закрывается, все его блокировки снимаются. В сервере без состояния, как в файловой системе NFS, с открытыми файлами нельзя связать блокировку, так как сервер не знает, какие файлы открыты. Следовательно, в файловой системе NFS требуется отдельный специальный механизм осуществления блокировки.

Файловая система NFS использует стандартный механизм защиты UNIX с битами rwx для владельца, группы и всех прочих пользователей (этот вопрос упоминался в главах 1 и 6, а также подробно обсуждается ниже). 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 рис. 17. 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).

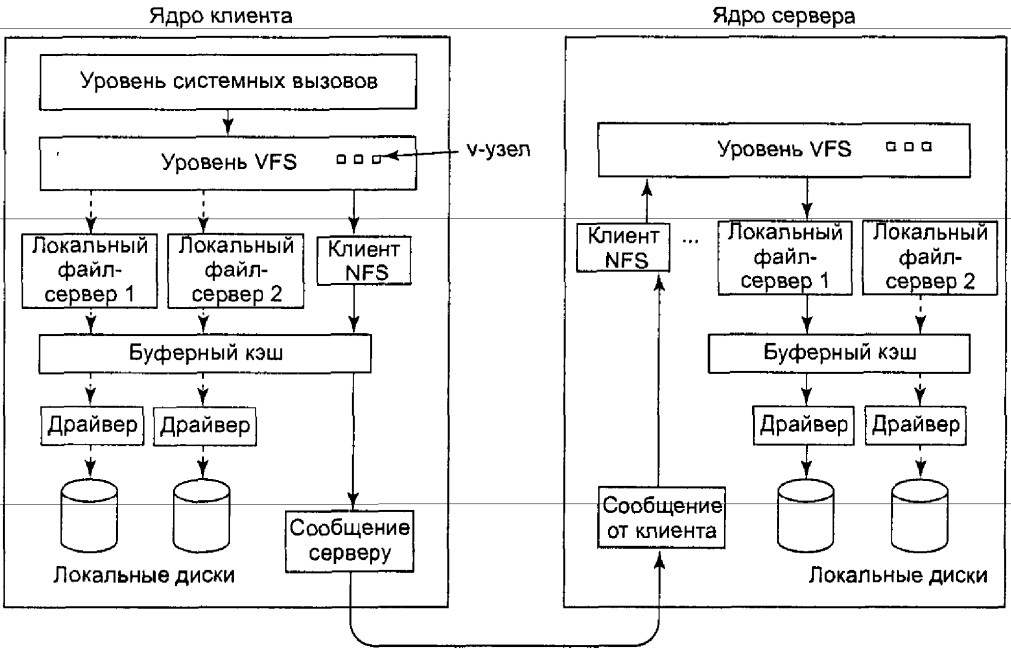


Рис. 16. Структура уровней файловой системы NFS

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 ОС, 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/rc*)* вызывает программу mount*,* указывая ей удаленный каталог, локальный каталог, в котором следует смонтировать удаленный каталог, и прочую информацию. Программа mountанализирует 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.

Затем ядро формирует для удаленного каталога *v* -узел и просит программу клиента NFS на рис. 16 создать в своих внутренних таблицах *r* -узел (удаленный *i* -узел) для хранения дескриптора файла. *v* -узел указывает на *r* -узел. Каждый *v* -узел на уровне

VFS будет в конечном итоге содержать либо указатель на *r* -узел в программе клиента NFS, либо указатель на *i* -узел в одной из локальных файловых систем (показанный на рис. 16 в виде штриховой линии). По содержимому *v* -узла можно понять, является ли файл или каталог локальным или удаленным. Если он локальный, то может быть найдена соответствующая файловая система и *i* -узел. Если файл удаленный, может быть найден удаленный хост и дескриптор файла.

Когда на клиенте открывается удаленный файл, при анализе пути файла ядро обнаруживает каталог, в котором смонтирована удаленная файловая система. Оно видит, что этот каталог удаленный, а в *v* -узле каталога находит указатель на *r* -узел. Затем она просит программу клиента NFS открыть файл. Программа клиента NFS просматривает оставшуюся часть пути на удаленном сервере, ассоциированном с монтированным каталогом, и получает обратно дескриптор файла для него. Он создает в своих таблицах *r* -узел для удаленного файла и докладывает об этом уровню VFS, который помещает в свои таблицы *v* -узел для файла, указывающий на *r* -узел. Мы видим, что и в этом случае у каждого открытого файла или каталога есть *v* -узел, указывающий на *r* -узел или *i* -узел.

Вызывающему процессу выдается дескриптор удаленного файла. Этот дескриптор файла отображается на *v* -узел при помощи таблиц уровня VFS. Обратите внимание, что на сервере не создается никаких записей в таблицах. Хотя сервер готов предоставить дескрипторы файлов по запросу, он не следит за состоянием дескрипторов файлов. Когда дескриптор файла присылается серверу для доступа к файлу, сервер проверяет дескриптор и использует его, если дескриптор действителен. При проверке может проверяться ключ аутентификации, содержащийся в заголовках вызова удаленной процедуры RPC.

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 проходится 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.

Учитывая серьезность данной проблемы, реализация NFS пытается смягчить ее остроту несколькими способами. Во-первых, **с каждым блоком кэша ассоциирован таймер**. **Когда время истекает, запись считается недействительной**. Как правило, для блоков с данными таймер устанавливается на 3 с, а для блоков каталога — 30 с. Таким образом, риск несколько снижается. Кроме того, **при каждом открытии кэшированного файла серверу посылается сообщение, чтобы определить, когда в последний раз был модифицирован этот файл**. Если последнее изменение произошло после того, как была сохранена в кэше локальная копия файла, эта копия из кэша удаляется, а с сервера получается новая копия. Наконец, **каждые 30 с истекает время таймера, и все «грязные» (модифицированные) блоки кэша посылаются на сервер**. Хотя такая схема и далека от совершенства, подобные «заплатки» позволяют пользоваться этой системой в большинстве практических случаев.

## Безопасность в UNIX

Несмотря на свое название, ОС UNIX была с самого начала многопользовательской системой. Это значит, что безопасность и контроль над информацией были встроены в систему с момента ее создания.

### Основные понятия

Сообщество пользователей ОС 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 16разрядными 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 ОС 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 rwx (read, write, execute). Возможность исполнять файл, конечно, имеет смысл только в том случае, если этот файл является исполняемой двоичной 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.

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 хинтерпретируется в отношении каталогов как разрешение не исполнения, а поиска в каталоге. Таким образом, каталог с режимом rwxr-xr-xпозволяет своему владельцу читать, изменять каталог, а также искать в нем файлы, а всем остальным пользователям разрешает 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. При этом режим доступа к файлу может быть установлен равным rw--------, чтобы все остальные пользователи не могли напрямую обращаться к принтеру. В противном случае при одновременной печати на принтере нескольких процессов получится полный хаос.

Конечно, тот факт, что файлом /dev/lp владеет демон и этот файл имеет режим доступа rw--------, означает, что более никто не может выводить данные на принтер.

Хотя такой способ и позволяет избежать множества неприятностей, однако время от времени пользователям бывает необходимо напечатать что-нибудь. В действительности существует более общая проблема регулируемого доступа ко всем устройствам вводавывода и другим системным ресурсам.

Эта проблема была решена с помощью добавления к перечисленным выше 9 бит нового бита защиты, бита **SETUID**. Когда выполняется программа с установленным битом SETUID, то запускаемому процессу присваивается не UID вызвавшего его пользователя или процесса, a UID владельца файла. Когда процесс пытается открыть файл, то проверяется его рабочий UID, а не UID запустившего его пользователя. Таким образом, если программой, обращающейся к принтеру, будет владеть демон с установленным битом SETUID, тогда любой пользователь сможет запустить ее и запущенный процесс будет обладать полномочиями демона (например, правами доступа к /dev/lp), но только для запуска этой программы (которая может устанавливать задания в очередь на принтер).

В ОС 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 вызовы перечислены в табл. 8. Чаще всего используется системный вызов chmod. С его помощью можно изменить режим защиты файла. Например, оператор s = chmod("/usr7ast/newgame", 0755);

устанавливает для файла newgameрежим 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 8. Некоторые системные вызовы, относящиеся к безопасности

|  |  |
| --- | --- |
| Системный вызов | Описание |
| s-chmod(path, mode) | Изменить режим защиты файла |
| s=access(path, mode) | Проверить разрешение доступа к файлу, используя действительные UID и GID |
| uid=getuid() | Получить действительный UID |
| uid=geteuid() | Получить рабочий UID |
| gid=getgid( ) | Получить действительный GID |
| gid=getegid() | Получить рабочий GID |
| s=chown(path, owner, group) | Изменить владельца и группу |
| s=setuid(uid) | Установить UID |
| s=setgid(gid) | 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). Хэширование применяется, чтобы избежать хранения пароля в незашифрованном виде где-либо в системе. Если пароль введен верно, программа регистрации считывает из файла /etc/passwd имя программы оболочки, которую любит пользователь. Ей может быть программа sh, но это также может быть и другая оболочка, например csh или ksh. Затем программа регистрации использует системные 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.

С этого момента начинает работу оболочка с установленными UID и GID, а также стандартными потоками ввода, вывода и ошибок, настроенными на устройства вводавывода по умолчанию. Все процессы, которые она запускает при помощи системного вызова fork (то есть команды, вводимые пользователем с клавиатуры), автоматически наследуют UID и GID оболочки, поэтому у них будет верное значение владельца и группы. Все файлы, создаваемые этими процессами, также будут иметь эти значения.

Когда любой процесс пытается открыть файл, система сначала проверяет биты защиты в *i* -узле файла для заданных значений рабочих UID и GID, чтобы определить, разрешен ли доступ для данного процесса. Если доступ разрешен, файл открывается и процессу возвращается дескриптор файла. В противном случае файл не открывается, а процессу возвращается значение -1. При последующих обращениях к системным вызовам read и write проверка не выполняется. В результате, если режим защиты файла изменяется уже после того, как файл открыт, новый режим не повлияет на процессы, которые уже успели открыть этот файл.

В ОС Linux защита файлов и ресурсов осуществляется так же, как и в UNIX. В системе Linux реализованы все функции защиты системы UNIX, и нет почти ничего такого в системе Linux в области защиты, чего нет в ОС UNIX.

# Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.
2. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.
3. Робачевский А.М. Операционная система UNIX. –СПб.: БХВ-Петербург, 2002. -528 с.