такое их количество, которое позволило бы хранить столько же информации, как на одной катушке ленты, очень быстро превратила их в весьма дорогое удовольствие. Но со временем цены снизились, и были разработаны примитивные файловые системы.

Типичным представителем этого нового витка развития был компьютер CDC 6600, представленный в 1964 году, который в течение многих лет оставался самым быстрым компьютером в мире. Пользователи могли создавать так называемые постоянные файлы, давая им имена в надежде, что никакой другой пользователь не решит, что, скажем, имя «data» является вполне подходящим для его файла. В них использовался одноуровневый каталог. Со временем для универсальных машин были разработаны сложные иерархические файловые системы, кульминацией которых, наверное, стала файловая система MULTICS.

Когда стали использоваться мини-компьютеры, то со временем они также обзавелись жесткими дисками. Стандартным для PDP-11 на момент его представления в 1970 году был диск RK05 емкостью 2,5 Мбайт, что составляло примерно половину емкости диска IBM RAMAC, но он был всего 40 см в диаметре и 5 см высотой. Поначалу он также имел всего один каталог. Когда пришло время микрокомпьютеров, то сначала доминирующей операционной системой была CP/M, и она тоже поддерживала только один каталог на гибком диске.

**Виртуальная память**

Виртуальная память (которая будет рассмотрена в главе 3) позволяет запускать про- граммы, размер которых превышает объем установленной на машине физической памяти, за счет быстрого перемещения фрагментов адресного пространства между оперативной памятью и диском. Она прошла похожий путь развития, появившись впервые на универсальных машинах, затем переместившись на мини- и микрокомпью- теры. Виртуальная память также позволяла программам во время работы динамически компоноваться с библиотеками, вместо того чтобы быть скомпилированными со всеми необходимыми библиотеками в единую программу. MULTICS была первой системой, позволявшей работать с такими программами. Со временем эта идея получила распро- странение и теперь широко используется на большинстве UNIX- и Windows-систем.

Во всех приведенных примерах развития мы видели идеи, которые изобретались в од- ном контексте, а позже в связи со сменой обстановки отметались (программирование на ассемблере, однозадачный режим, одноуровневые каталоги и т. д.), чтобы вновь появиться уже в другом контексте, зачастую спустя десятилетие. Поэтому в данной книге мы будем иногда рассматривать идеи и алгоритмы, которые могут показаться устаревшими для современных персональных компьютеров, оснащенных гигабайтной памятью, но имеют шанс на скорое возвращение во встроенных компьютерах и смарт- картах.

**1.6. Системные вызовы**

Мы выяснили, что операционные системы выполняют две основные функции: предо- ставляют абстракции пользовательским программам и управляют ресурсами компью- тера. В основном взаимодействие пользовательских программ и операционной системы касается первой функции — взять, к примеру, операции с файлами: создание, запись,

чтение и удаление. А управление ресурсами компьютера проходит большей частью незаметно для пользователей и осуществляется в автоматическом режиме. Так что интерфейс между пользовательскими программами и операционной системой строится в основном на абстракциях. Чтобы по-настоящему понять, что делает операционная система, мы должны более подробно рассмотреть этот интерфейс. Имеющиеся в интер- фейсе системные вызовы варьируются в зависимости от используемой операционной системы (хотя основные понятия практически ничем не различаются).

Теперь нужно выбрать между неопределенной обобщенностью («у операционных систем есть системные вызовы для чтения файлов») и какой-нибудь конкретной системой («в UNIX есть системный вызов чтения, имеющий три параметра: в одном из них определяется файл, в другом — куда поместить данные, а в третьем — сколько байтов следует считать»).

Мы выбрали второй подход. Пусть он сложнее, зато позволяет лучше понять, как на самом деле операционная система выполняет свою работу. Хотя все, что будет рассма- триваться, имеет непосредственное отношение к стандарту POSIX (Международный стандарт 9945-1), а следовательно к UNIX, System V, BSD, Linux, MINIX 3 и т. д., у большинства других современных операционных систем имеются системные вызовы, выполняющие аналогичные функции, при некоторых отличиях в деталях. Посколь- ку фактический механизм выполнения системного вызова существенно зависит от конкретной машины и зачастую должен быть реализован на ассемблере, разработаны библиотеки процедур, осуществляющие системные вызовы из программ, написанных, например, на языке C.

Очень полезно всегда помнить следующее. Любой однопроцессорный компьютер одно- моментно может выполнить только одну команду. Когда процесс выполняет пользова- тельскую программу в режиме пользователя и нуждается в какой-нибудь услуге опера- ционной системы, например в чтении данных из файла, он должен выполнить команду системного прерывания, чтобы передать управление операционной системе. Затем опера- ционная система по параметрам вызова определяет, что именно требуется вызывающему процессу. После этого она обрабатывает системный вызов и возвращает управление той команде, которая следует за системным вызовом. В некотором смысле выполнение си- стемного вызова похоже на выполнение особой разновидности вызова процедуры, с той лишь разницей, что системные вызовы входят в ядро, а процедурные — нет.

Для того чтобы прояснить механизм системных вызовов, рассмотрим системный вызов чтения — *read*. Как уже упоминалось, он имеет три параметра: первый служит для задания файла, второй указывает на буфер, а третий задает количество байтов, которое нужно прочитать. Как практически все системные вызовы, он осуществляется из программы на языке C с помощью вызова библиотечной процедуры, имя которой совпадает с именем системного вызова: *read*. Вызов из программы на C может иметь следующий вид:

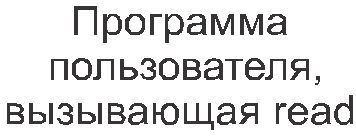
count = read(fd, buffer, nbytes);

Системный вызов (и библиотечная процедура) возвращает количество фактически считанных байтов, которое сохраняется в переменной *count*. Обычно это значение совпадает со значением параметра *nbytes*, но может быть и меньше, если, например, в процессе чтения будет достигнут конец файла.

Если системный вызов не может быть выполнен из-за неправильных параметров или ошибки диска, значение переменной *count* устанавливается в −1, а номер ошибки по-

мещается в глобальную переменную *errno*. Программам обязательно нужно проверять результаты системного вызова, чтобы отслеживать возникновение ошибки.

Выполнение системного вызова состоит из нескольких шагов. Для прояснения ситуа- ции вернемся к упоминавшемуся ранее примеру вызова *read*. Сначала, при подготовке вызова библиотечной процедуры *read*, которая фактически и осуществляет системный вызов *read*, вызывающая программа помещает параметры в стек (рис. 1.17, шаги *1*–*3*).



**Рис. 1.17.** 11 этапов выполнения системного вызова read(fd, buffer, nbytes)

Компиляторы C и C++ помещают параметры в стек в обратном порядке, следуя истори- чески сложившейся традиции (чтобы на вершине стека оказался первый параметр функ- ции *printf* — строка формата вывода данных). Первый и третий параметры передаются по значению, а второй параметр передается по ссылке, поскольку это адрес буфера (о чем свидетельствует знак &), а не его содержимое. Затем осуществляется фактический вызов библиотечной процедуры (шаг *4*). Эта команда представляет собой обычную команду вызова процедуры и используется для вызова любых процедур.

Библиотечная процедура, возможно, написанная на ассемблере, обычно помещает но- мер системного вызова туда, где его ожидает операционная система, например в регистр (шаг *5*). Затем она выполняет команду *TRAP* для переключения из пользовательского режима в режим ядра, и выполнение продолжается с фиксированного адреса, находя- щегося внутри ядра операционной системы (шаг *6*). Фактически команда *TRAP* очень похожа на команду вызова процедуры в том смысле, что следующая за ней команда берется из удаленного места, а адрес возврата сохраняется в стеке для последующего использования.

Тем не менее у команды *TRAP* и команды вызова процедуры есть два основных раз- личия. Во-первых, побочный эффект, заключающийся в переключении в режим ядра. Команда вызова процедуры не меняет используемый режим. А во-вторых, коман- да *TRAP* не может получить относительный или абсолютный адрес местонахождения процедуры, поскольку не может осуществить переход на произвольный адрес.

Начавшая работу после команды *TRAP* часть ядра (диспетчер на рис. 1. 17) проверяет номер системного вызова, а затем передает управление нужному обработчику. Обычно передача управления осуществляется посредством таблицы указателей на обработчики системных вызовов, которая индексирована по номерам этих вызовов (шаг *7*). После этого вступает в действие обработчик конкретного системного вызова (шаг *8*). Как только обработчик закончит работу, управление может быть возвращено библиотеч- ной процедуре, находящейся в пользовательской области памяти, той самой команде, которая следует за командой *TRAP* (шаг *9*). В свою очередь эта процедура вернет управ- ление пользовательской программе по обычной схеме возврата из процедуры (шаг *10*).

Чтобы завершить работу с процедурой *read*, пользовательская программа должна очи- стить стек, точно так же, как она это делает после любого вызова процедуры (шаг *11*). Если в нашем примере стек растет вниз (как это чаще всего и бывает), пользовательская программа в скомпилированном виде должна содержать команды увеличения указате- ля стека ровно настолько, чтобы были удалены параметры, помещенные в стек перед вызовом процедуры *read*. Теперь программа может продолжить свою работу.

При рассмотрении шага *9* было специально отмечено, что «управление может быть воз- вращено библиотечной процедуре, находящейся в пользовательской области памяти». Системный вызов может заблокировать вызывающую программу, препятствуя про- должению ее работы. Например, вызывающая программа должна быть заблокирована при попытке чтения с клавиатуры, когда на ней еще ничего не набрано. В этом случае операционная система ищет другой процесс, который может быть запущен. Позже, ког- да станут доступны требуемые входные данные, система вспомнит о заблокированном процессе и будут выполнены шаги с *9*-го по *11*-й.

В следующих разделах мы рассмотрим некоторые из наиболее востребованных си- стемных вызовов стандарта POSIX, или, точнее, библиотечных процедур, осущест- вляющих эти системные вызовы. В стандарте POSIX определено более 100 процедур, обеспечивающих обращение к системным вызовам. Некоторые наиболее важные из них и сгруппированные для удобства по категориям перечислены в табл. 1.1. Далее мы кратко опишем каждый вызов и его назначение.

Услуги, предоставляемые этими системными вызовами, в значительной степени опре- деляют большую часть возможностей операционной системы, поскольку управление ресурсами на персональных компьютерах осуществляется в минимальном объеме (во всяком случае, по сравнению с большими машинами, обслуживающими множество пользователей). Они включают в себя такие виды обслуживания, как создание и преры- вание процессов, создание, удаление, чтение и запись файлов, управление каталогами, ввод и вывод данных.

Отдельно стоит упомянуть, что отображение процедурных вызовов POSIX на систем- ные вызовы не является взаимно однозначным. В стандарте POSIX определены про- цедуры, которые должна предоставить совместимая с ним система, но он не указывает, чем именно они должны быть реализованы: системными, библиотечными вызовами или чем-нибудь еще. Если процедура может быть выполнена без системного вызова

(то есть без переключения в режим ядра), то она из соображений производительности обычно выполняется в пользовательском пространстве. Однако большинство процедур POSIX осуществляют системные вызовы, при этом обычно одна процедура непосред- ственно отображается на один системный вызов. В некоторых случаях, особенно когда несколько необходимых системе процедур мало чем отличаются друг от друга, один системный вызов обрабатывает более одного вызова библиотечных процедур.

**Таблица 1.1.** Некоторые важнейшие системные вызовы POSIX1

|  |  |
| --- | --- |
| **Вызов** | **Описание** |
| *Управление процессом* | |
| pid = fork() | Создает дочерний процесс, идентичный родительскому |
| pid = waitpid(pid, &statloc, options) | Ожидает завершения дочернего процесса |
| s = execve(name, argv, environp) | Заменяет образ памяти процесса |
| exit(status) | Завершает выполнение процесса и возвращает статус |
| *Управление файлами* | |
| fd = open(file, how...) | Открывает файл для чтения, записи или для того и дру- гого |
| s = close(fd) | Закрывает открытый файл |
| n = read(fd, buffer, nbytes) | Читает данные из файла в буфер |
| n = write(fd, buffer, nbytes) | Записывает данные из буфера в файл |
| position = lseek(fd, offset, whence) | Перемещает указатель файла |
| s = stat(name, &buf) | Получает информацию о состоянии файла |
| *Управление каталогами и файловой системой* | |
| s = mkdir(name, mode) | Создает новый каталог |
| s = rmdir(name) | Удаляет пустой каталог |
| s = link(name1, name2) | Создает новый элемент с именем name2, указывающий на name1 |
| s = unlink(name) | Удаляет элемент каталога |
| s = mount(special, name, flag) | Подключает файловую систему |
| s = umount(special) | Отключает файловую систему |
| *Разные* | |
| s = chdir(dirname) | Изменяет рабочий каталог |
| s = chmod(name, mode) | Изменяет биты защиты файла |
| s = kill(pid, signal) | Посылает сигнал процессу |
| seconds = time(&seconds) | Получает время, прошедшее с 1 января 1970 года |

1 При возникновении ошибки возвращаемое значение кода завершения *s* равно –1. Исполь- зуются следующие имена возвращаемых значений: *pid* — id процесса, *fd* — описатель файла, *n* — количество байтов, *position* — смещение внутри файла и *seconds* — прошедшее время. Описания параметров приведены в тексте.

**1.6.1. Системные вызовы для управления процессами**

Первая группа вызовов в табл. 1.1 предназначена для управления процессами. Начнем рассмотрение системного вызова *fork* (разветвление). Вызов *fork* является единствен- ным существующим в POSIX способом создания нового процесса. Он создает точную копию исходного процесса, включая все дескрипторы файлов, регистры и т. п. После выполнения вызова *fork* исходный процесс и его копия (родительский и дочерний процессы) выполняются независимо друг от друга. На момент разветвления все их со- ответствующие переменные имеют одинаковые значения, но поскольку родительские данные копируются в дочерний процесс, последующие изменения в одном из них не влияют на изменения в другом. (Текст программы, не подвергающийся изменениям, является общим для родительского и дочернего процессов.) Системный вызов *fork* возвращает нулевое значение для дочернего процесса и равное идентификатору до- чернего процесса или PID — для родительского. Используя возвращенное значение PID, два процесса могут определить, какой из них родительский, а какой — дочерний.

В большинстве случаев после вызова *fork* дочернему процессу необходимо выполнить программный код, отличный от родительского. Рассмотрим пример работы системной оболочки. Она считывает команду с терминала, создает дочерний процесс, ожидает, пока дочерний процесс выполнит команду, а затем считывает другую команду, если дочерний процесс завершается. Для ожидания завершения дочернего процесса роди- тельский процесс выполняет системный вызов *waitpid*, который просто ждет, пока до- черний процесс не закончит свою работу (причем здесь имеется в виду любой дочерний процесс, если их несколько). Системный вызов *waitpid* может ожидать завершения конкретного дочернего процесса или любого из запущенных дочерних процессов, если первый параметр имеет значение −1. Когда работа *waitpid* завершается, по адресу, ука- занному во втором параметре — *statloc*, заносится информация о статусе завершения дочернего процесса (нормальное или аварийное завершение и выходное значение). В третьем параметре определяются различные необязательные настройки.

Теперь рассмотрим, как вызов *fork* используется оболочкой. После набора команды оболочка создает дочерний процесс, который должен выполнить команду пользова- теля. Он делает это, используя системный вызов *execve*, который полностью заменяет образ памяти процесса файлом, указанным в первом параметре. (Фактически сам системный вызов называется *exec*, но некоторые библиотечные процедуры со слегка отличающимися именами вызывают его с различными параметрами. Здесь мы будем рассматривать все это как системные вызовы.) В листинге 1.1 показано использование *fork*, *waitpid* и *execve* предельно упрощенной оболочкой.

**Листинг 1.1.** Оболочка, усеченная до минимума

#define TRUE 1

while (TRUE) { /\* бесконечный цикл \*/ type\_prompt( ); /\* вывод приглашения на экран \*/ read\_command(command, parameters); /\* чтение ввода с терминала \*/

if (fork( ) != 0) { /\* ответвление дочернего процесса \*/

/\* код родительского процесса \*/

waitpid(-1, &status, 0); /\* ожидание завершения дочернего процесса \*/

} else {

/\* код дочернего процесса \*/

execve(command, parameters, 0); /\* выполнение command \*/

}

}

Далее в этой книге предполагается, что для *TRUE* определено значение 1.

В наиболее общем случае команда *execve* имеет три параметра: имя выполняемого фай- ла, указатель на массив аргументов и указатель на массив переменных окружения. Эти параметры мы рассмотрим в дальнейшем. Различные библиотечные подпрограммы, включая *execl*, *execv*, *execle* и *execve*, предусматривают возможность пропуска параме- тров или указания их различными способами. Далее в этой книге мы воспользуемся именем *exec* для представления системного вызова, который инициируется всеми этими подпрограммами.

Рассмотрим следующую команду:

cp file1 file2

Она используется для копирования одного файла в другой — file1 в file2. После созда- ния оболочкой дочернего процесса последний находит и выполняет файл cp и передает ему имена исходного и целевого файлов.

Основная программа cp (и большинство других основных программ на языке С) со- держит объявление

main(argc, argv, envp)

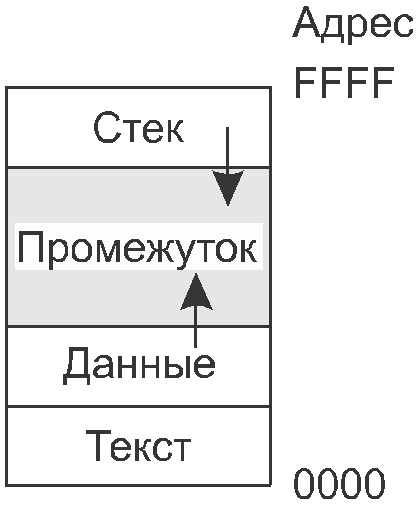
где *argc* — количество элементов командной строки, включая имя программы. Для предыдущего примера *argc* равен 3.

Второй параметр — *argv* — представляет собой указатель на массив. Элемент *i* этого массива является указателем на i-й строковый элемент командной строки. В нашем примере *argv[0]* будет указывать на строку *cp*, *argv[1]* — на строку *file1*, а *argv[2]* — на строку *file2*.

Третий параметр функции *main* — *envp* — является указателем на массив переменных окружения, то есть на массив строк вида *имя = значение*, используемый для передачи программе такой информации, как тип терминала и имя домашнего каталога программ. Существуют библиотечные процедуры, которые программа может вызвать для получе- ния переменных окружения. Они часто используются для настройки пользовательских предпочтений при выполнении определенных задач (например, для настройки прин- тера, используемого по умолчанию). В листинге 1.1 окружение дочернему процессу не передается, поэтому третий параметр *execve* имеет нулевое значение.

Если вызов *exec* кажется сложным, не стоит отчаиваться — это (с точки зрения семанти- ки) наиболее сложный из всех имеющихся в POSIX системных вызовов. Все остальные выглядят намного проще. В качестве более простого примера можно рассмотреть *exit*, который используется процессами, когда они заканчивают выполнение. У него всего один параметр — статус выхода (0–255), который возвращается родительской про- грамме через *statloc* в системном вызове *waitpid*.

В UNIX память каждого процесса делится на три сегмента: **текстовый сегмент** (то есть код программы), **сегмент данных** (переменные) и **сегмент стека**. Как показано на рис. 1.18, сегмент данных растет вверх, а стек растет вниз. Между ними существует часть неиспользованного адресного пространства. Стек заполняет пустое пространство автоматически по мере надобности. Расширение сегмента данных за счет пустого про-



**Рис. 1.18.** Память процессов состоит из трех сегментов:

текста, данных и стека

странства выполняется явным образом. Для этого предназначен системный вызов *brk*, указывающий новый адрес окончания сегмента данных. Однако этот вызов стандар- том POSIX не определен, так как программистам для динамического распределения памяти рекомендуется пользоваться библиотечной процедурой *malloc*. При этом низ- коуровневая реализация *malloc* не рассматривалась в качестве объекта, подходящего для стандартизации, поскольку мало кто из программистов использует ее в непо- средственном виде и весьма сомнительно, чтобы кто-нибудь даже заметил отсутствие в POSIX системного вызова *brk*.

**1.6.2. Системные вызовы для управления файлами**

Многие системные вызовы имеют отношение к файловой системе. В этом разделе будут рассмотрены вызовы, работающие с отдельными файлами, а в следующем разделе — те вызовы, которые оперируют каталогами или файловой системой в целом.

Чтобы прочитать данные из файла или записать их в файл, сначала его необходимо открыть. Для данного вызова необходимо указать имя открываемого файла (с указа- нием абсолютного пути либо пути относительно рабочего каталога) и код *O\_RDONLY*, *O\_WRONLY* или *O\_RDWR*, означающий, что файл открывается для чтения, записи или для чтения и записи. Для создания нового файла используется параметр *O\_CREAT*. Возвращаемый дескриптор файла впоследствии может быть использован для чтения или записи. После этого файл может быть закрыт с помощью системного вызова *close*, который делает дескриптор файла доступным для повторного использования при по- следующем системном вызове *open*.

Наиболее часто используемыми вызовами, несомненно, являются *read* и *write*. Вызов

*read* мы уже рассмотрели, вызов *write* имеет те же параметры.

Несмотря на то что большинство программ читают и записывают файлы последова- тельно, некоторым прикладным программам необходима возможность произвольного доступа к любой части файла. С каждым файлом связан указатель на текущую пози- цию. При последовательном чтении (записи) он обычно указывает на байт, который должен быть считан (записан) следующим. Системный вызов *lseek* может изменить значение указателя, при этом последующие вызовы *read* или *write* начнут свою работу с нового произвольно указанного места в файле.

Вызов *lseek* имеет три параметра: первый — это дескриптор файла, второй — позиция в файле, третий — указание, относительно чего задана позиция — начала файла, теку-

щей позиции или конца файла. Вызов *lseek* возвращает абсолютную позицию в файле

(в байтах) после изменения указателя.

Для каждого файла UNIX хранит следующие данные: код режима файла (обычный файл, специальный файл, каталог и т. д., а также права доступа к файлу), размер, вре- мя последнего изменения и другую информацию. Программы могут запрашивать эту информацию посредством системного вызова *stat*. Его первый параметр определяет файл, информацию о котором необходимо получить, второй является указателем на структуру, в которую она должна быть помещена. Для открытого файла то же самое делает системный вызов *fstat*.

**1.6.3. Системные вызовы для управления каталогами**

В этом разделе мы рассмотрим некоторые системные вызовы, относящиеся скорее к каталогам или к файловой системе в целом, чем к отдельным файлам, как в предыду- щем разделе. Первые два вызова — *mkdir* и *rmdir* — соответственно создают и удаляют пустые каталоги. Следующий вызов — *link*. Он позволяет одному и тому же файлу появляться под двумя или более именами, зачастую в разных каталогах. Этот вызов обычно используется, когда несколько программистов, работающих в одной команде, должны совместно использовать один и тот же файл. Тогда этот файл может появиться у каждого из них в собственном каталоге, возможно, под разными именами. Совместное использование файла отличается от предоставления каждому члену команды личной копии; наличие общего доступа к файлу означает, что изменения, которые вносит любой из представителей, тут же становятся видимыми другим, поскольку они ис- пользуют один и тот же файл. А при создании копии файла последующие изменения одной копии не влияют на другие.

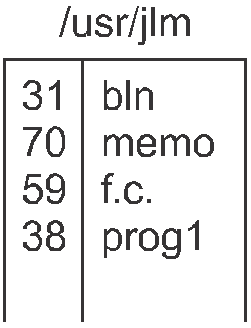
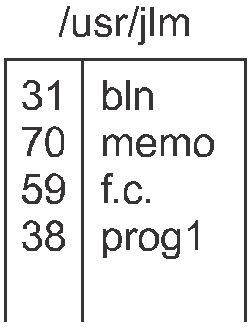
Чтобы увидеть, как работает вызов *link*, рассмотрим ситуацию, показанную на рис. 1.19, *a*. У каждого из двух пользователей с именами *ast* и *jim* есть собственные каталоги, в которых имеется ряд файлов. Если *ast* выполнит программу, содержащую системный вызов

link("/usr/jim/memo", "/usr/ast/note");

то файл memo в каталоге jim теперь будет входить в каталог ast под именем note. После этого /usr/jim/memo и /usr/ast/note будут ссылаться на один и тот же файл. Попутно следует заметить, что место, где хранятся каталоги пользователей — /usr, /user, /home или какое-нибудь другое, определяет местный системный администратор.

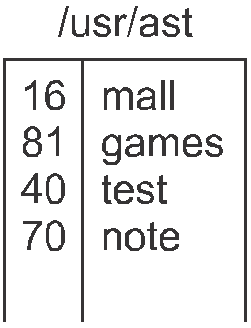
Возможно, станет понятнее, что именно делает *link*, если разобраться в том, как он работает. Каждый файл в UNIX имеет свой уникальный номер — идентификатор, или i-номер. Этот i-номер является единственным для каждого файла индексом в таблице **i-узлов** (i-nodes). Каждый i-узел (i-node или inode) хранит информацию о том, кто является владельцем файла, где расположены его блоки на диске и т. д. Каталог — это просто файл, содержащий набор пар (i-номер, ASCII-имя). В первой версии UNIX каждый элемент каталога занимал 16 байт: 2 байта для i-номера и 14 байт для имени. Сейчас для поддержки длинных имен файлов требуется более сложная структура, но концептуально каталог по-прежнему является набором пар (i-номер, ASCII-имя). На рис. 1.19 у файла mail имеется i-номер 16 и т. д. Системный вызов *link* просто создает новый элемент каталога, возможно, с новым именем, используя i-номер существующего файла. На рис. 1.19, *б* один и тот же i-номер (70) имеется у двух элементов, которые

таким образом ссылаются на один и тот же файл. Если позже с помощью системного вызова *unlink* одна из этих записей будет удалена, то вторая останется нетронутой. Если будут удалены обе записи, UNIX увидит, что записей для файла не существует (поле в i-узле отслеживает количество указывающих на данный файл элементов в каталогах), и удалит файл с диска.



**Рис. 1.19.** Два каталога: *а* — перед созданием ссылки на файл /usr/jim/memo в каталоге ast;

*б* — после создания ссылки

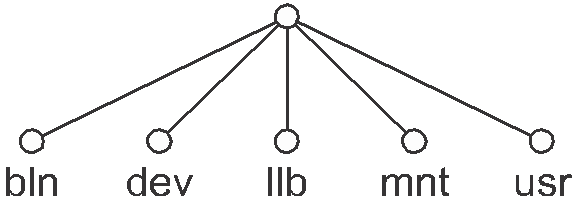
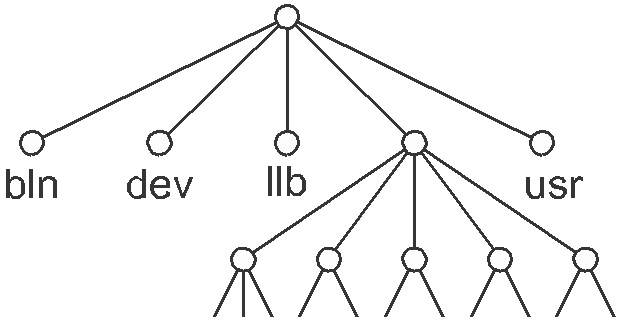


Как уже упоминалось, системный вызов *mount* позволяет объединять в одну две фай- ловые системы. Обычная ситуация такова: на разделе (или подразделе) жесткого диска находится корневая файловая система, содержащая двоичные (исполняемые) версии общих команд и другие интенсивно используемые файлы, а пользовательские файлы находятся на другом разделе (или подразделе). Затем пользователь может вставить USB-диск с файлами для чтения.

При помощи системного вызова *mount* файловая система USB-диска может быть под- ключена к корневой файловой системе (рис. 1.20). В языке C типичный оператор, вы- полняющий подключение («монтирование») файловой системы, выглядит так:

mount("/dev/sdb0", "/mnt", 0);

где первый параметр — это имя блочного специального файла для USB-устройства 0, второй параметр — место в дереве, куда будет происходить подключение, а третий па- раметр сообщает, будет ли файловая система подключена для использования в режиме чтения и записи или только чтения.



**Рис. 1.20.** Файловая система: *a* — до вызова mount; *б* —после вызова mount

После выполнения системного вызова *mount* для доступа к файлу на устройстве 0 можно использовать путь к нему из корневого или рабочего каталога, не обращая вни- мания на то, на каком устройстве он находится. Практически к любому месту дерева может быть подключено второе, третье и четвертое устройство. Вызов *mount* позволяет включать сменные носители в единую интегрированную файловую структуру, не об- ращая внимания на то, на каком устройстве находятся файлы. Хотя в этом примере

фигурирует компакт-диск, жесткие диски или их части (часто называемые **раздела- ми** — partition) также могут быть подключены этим способом. Аналогично могут быть подключены и внешние жесткие диски или флеш-накопители. Когда подключение файловой системы больше не требуется, она может быть отключена с помощью си- стемного вызова *umount*.

**1.6.4. Разные системные вызовы**

Помимо описанных ранее существуют и другие разновидности системных вызовов. Здесь будут рассмотрены только четыре из них. Системный вызов *chdir* изменяет те- кущий рабочий каталог. После вызова

chdir("/usr/ast/test");

при открытии файла xyz будет открыт файл /usr/ast/test/xyz. Использование понятия рабочего каталога избавляет от необходимости постоянно набирать длинные абсолют- ные пути файлов.

В UNIX каждый файл имеет код режима, хранящийся в i-узле и используемый для его защиты. Для управления правами доступа к файлу этот код включает биты чтения- записи-выполнения (read-write-execute) для владельца, для группы и для других поль- зователей. Системный вызов *chmod* позволяет изменять биты прав доступа к файлу. Например, чтобы сделать файл доступным для чтения для всех, а для владельца — до- ступным и для чтения и для записи, необходимо выполнить следующий системный вызов:

chmod("file", 0644);

Системный вызов *kill* позволяет пользователям и пользовательским процессам по- сылать сигналы. Если процесс готов принять определенный сигнал, то при его посту- плении запускается обработчик сигнала. Если процесс не готов к обработке сигнала, то его поступление уничтожает процесс (что соответствует имени системного вызова: kill — убивать, уничтожать).

Стандартом POSIX определен ряд процедур для работы со временем. Например, *time* просто возвращает текущее время в секундах, где 0 соответствует полуночи (началу, а не концу дня) 1 января 1970 года. На компьютерах, использующих 32-разрядные слова, максимальное значение, которое может быть возвращено процедурой *time*, рав- но 232 – 1 с (предполагается, что используется беззнаковое целое число — unsigned integer). Это значение соответствует периоду немногим более 136 лет. Таким образом, в 2106 году 32-разрядные системы UNIX «сойдут с ума», что сравнимо со знаменитой проблемой 2000 года (Y2K). Если сейчас вы работаете на 32-разрядной UNIX-системе, то ближе к 2106 году получите совет поменять ее на 64-битную.

**1.6.5. Windows Win32 API**

До сих пор мы ориентировались в основном на UNIX. Теперь настало время взглянуть и на Windows. Операционные системы Windows и UNIX фундаментально отличаются друг от друга в соответствующих моделях программирования. Программы UNIX со- стоят из кода, который выполняет те или иные действия, при необходимости обращаясь к системе с системными вызовами для получения конкретных услуг. В отличие от этого программой Windows управляют, как правило, события. Основная программа ждет,