**Лекция №2. Основные понятия операционных систем.**

**Введение.**

Данная лекция является второй в теме «Теоретические основы построения операционных систем». Мы рассмотрим основные понятия операционных систем такие как: процесс, адресные пространства, файловая система. Операционные системы выполняют две основные функции: предоставляют абстракции пользовательским программам и управляют ресурсами компьютера. Данные функции реализуются с помощью системных вызовов. Системный вызов является программным способом, в котором компьютерная программа запрашивает определенную операцию от ядра операционной системы. Иными словами, системный вызов возникает, когда пользовательский процесс требует некоторой службы реализуемой ядром и вызывает специальную функцию.

Сюда могут входить услуги, связанные с аппаратным обеспечением (например, доступ к жесткому диску), создание и выполнение новых процессов, связь с интегральными службами ядра, такими как планирование процессов. Системные вызовы обеспечивают необходимый интерфейс между процессом и операционной системой.

**Вопрос 1. Понятия операционных систем**

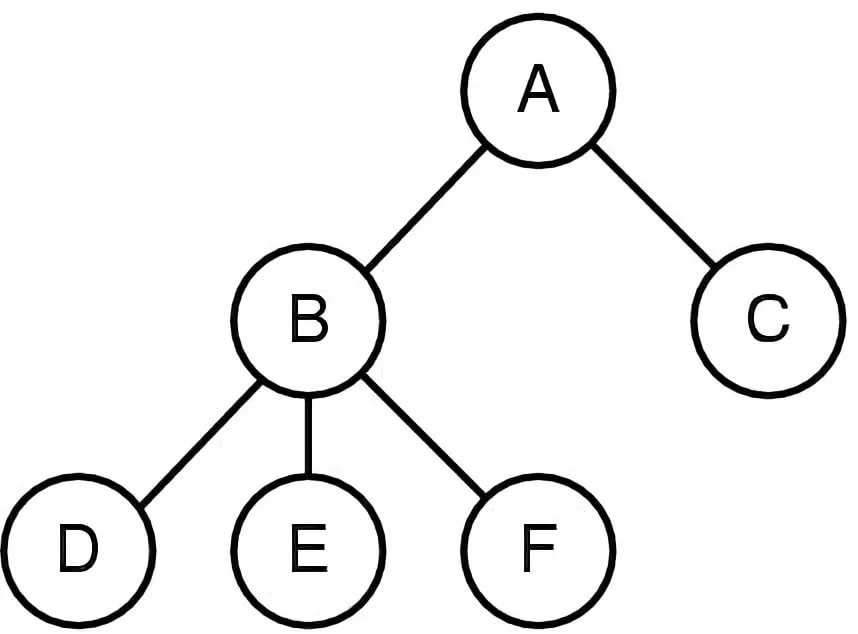
Ключевым понятием во всех операционных системах является процесс. Процессом, по существу, является программа вовремя ее выполнения. С каждым процессом связано его адресное пространство – список адресов ячеек памяти от нуля и до некоторого максимума, откуда процесс может считывать и куда записывает данные.

Адресное пространство содержит выполняемую программу, данные этой программы и ее стек. Кроме этого, с каждым процессом связан набор ресурсов, который обычно включает регистры, список открытых файлов, необработанные предупреждения, список связанных процессов и всю остальную информацию, необходимую в процессе работы программы.

Периодически операционная система будет принимать решения остановить работу одного процесса и запустить выполнение другого, например, потому, что первый исчерпал свою долю процессорного времени в предыдущую секунду или две. Если процесс приостанавливается, таким образом, позже он должен возобновиться именно в том состоянии, в котором был остановлен. Это означает, что на период приостановки вся информация о процессе должна быть явным образом где-то сохранена. Таким образом, процесс состоит из собственного адресного пространства, которое обычно называют образом памяти, и записи в таблице процессов, с содержимым его регистров, а также другой информацией, необходимой для последующего возобновления процесса.

Главными системными вызовами, используемыми при управлении процессами, являются вызовы, связанные с созданием и завершением процессов. Если процесс способен создавать несколько других процессов (называющихся дочерними процессами), а эти процессы в свою очередь могут создавать собственные дочерние процессы, то перед нами предстает дерево процессов (рис. 1.2). Процесс *А* создал два дочерних процесса, В и С, процесс В создал три дочерних процесса, *D*, *Е* и *F*. Связанные процессы, совместно работающие над выполнением какой-нибудь задачи, зачастую нуждаются в обмене данными друг с другом и в синхронизации своих действий, то есть в межпроцессном взаимодействии.

Другие системные вызовы, предназначенные для управления процессом, позволяют запросить выделение дополнительной памяти (или освобождение незадействованной), организовать ожидание завершения дочернего процесса или загрузку какой-нибудь другой программы поверх своей.

Рис. 1.2 Дерево процессов

Системный администратор присваивает каждому пользователю, которому разрешено работать с системой, идентификатор пользователя – *UID* (*User IDentification*). Пользователи могут входить в группу, каждая из которых имеет собственный идентификатор группы – *GID* (*Group IDentification*). Каждый запущенный процесс имеет *UID* того пользователя, который его запустил. Дочерние процессы имеют такой же *UID*, что и у родительского процесса.

Пользователь с особым значением *UID*, называемый в *Unix* «суперпользователем» (*superuser*), имеет особые полномочия, позволяющие пренебрегать многими правилами защиты. В крупных компьютерных системах только системный администратор знает пароль, необходимый для получения прав суперпользователя, но многие пользователи прикладывают немалые усилия, пытаясь отыскать бреши в системе, которые позволили бы им стать суперпользователем без пароля.

**Адресные пространства**. Основная проблема ОС компьютера – это управление адресным пространством процессов. Обычно каждому процессу отводится под использование некоторый непрерывный набор адресов, как правило, с нуля и до некоторого максимума. В простейшем случае максимальный объем адресного пространства, выделяемого процессу, меньше объема оперативной памяти. Таким образом, процесс может заполнить свое адресное пространство и для его размещения в оперативной памяти будет достаточно места.

При этом на многих компьютерах используется 32 или 64-разрядная адресация, позволяющая иметь адресное пространство размером в 232 или 264 байт соответственно. Если адресное пространство процесса превышает объем оперативной памяти, установленной на компьютере, а процессу требуется использовать все свое пространство целиком, то используется виртуальная память. Виртуальная память позволяет ОС хранить часть адресного пространства в оперативной памяти, а часть – на диске, по необходимости меняя их фрагменты местами. По сути, операционная система создает абстракцию адресного пространства в виде набора адресов, на которые может ссылаться процесс. Адресное пространство отделено от физической памяти машины и может быть, как больше, так и меньше ее по объему.

Другим ключевым понятием, поддерживаемым практически всеми ОС, является файловая система.

Многие операционные системы используют понятие каталога как способа объединения файлов в группы для хранения файлов. Для создания и удаления каталогов, для помещения в каталог существующего файла и удаления его оттуда используются системные вызовы. Элементами каталога могут быть либо файлы, либо другие каталоги. Эта модель стала прообразом иерархической структуры файловой системы, один из вариантов которой показан на рис. 1.3.

Иерархии файлов, как и иерархии процессов, организованы в виде деревьев, но на этом сходство заканчивается. Иерархии процессов не отличаются глубиной (обычно не более трех уровней), а иерархии файлов обычно имеют глубину в четыре, пять и более уровней.

Иерархии процессов обычно имеют короткий период существования, в большинстве своем не более нескольких минут, а иерархия каталогов может существовать годами. Определение принадлежности и меры защиты для процессов и файлов также имеют различия. Каждый файл, принадлежащий иерархии каталогов, может быть обозначен своим полным именем, с указанием пути к файлу, начиная с вершины иерархии – корневого каталога. Этот абсолютный путь состоит из списка каталогов, которые нужно пройти от корневого каталога, чтобы добраться до файла, где в качестве разделителей компонентов служат символы косой черты (слэша).

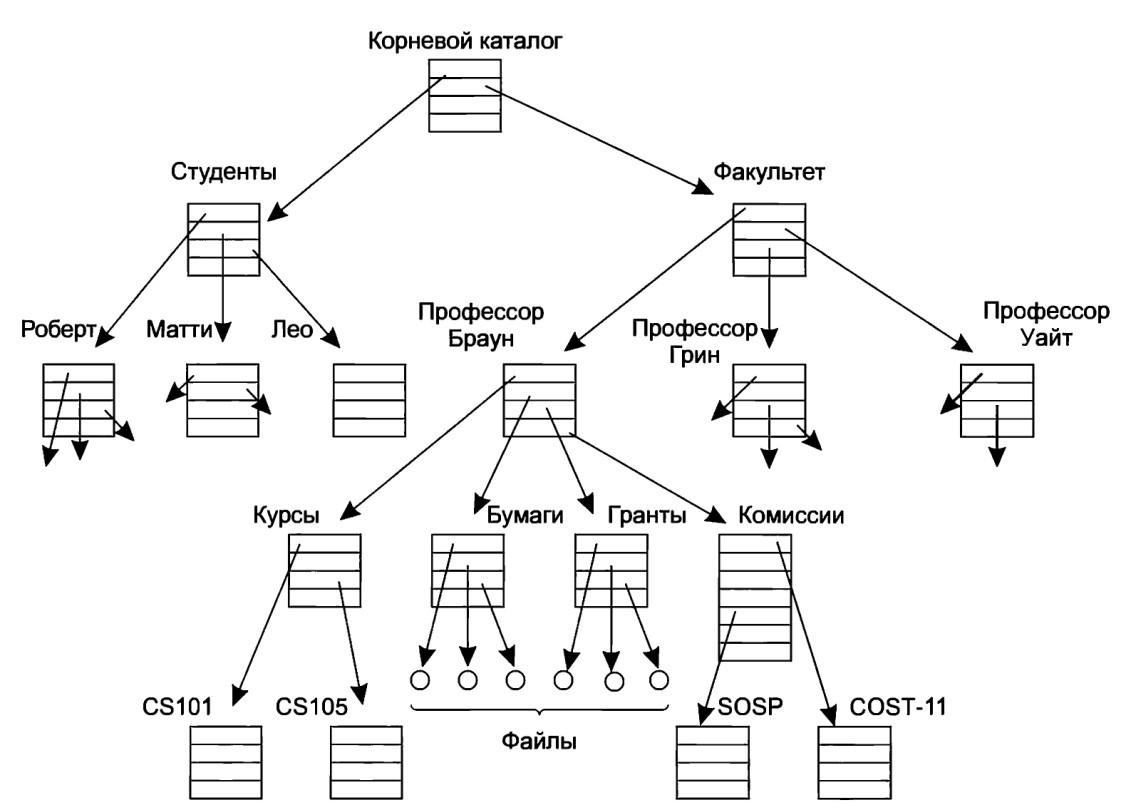


Рис. 1.3 Файловая система факультета университета

В любой момент времени у каждого процесса есть текущий рабочий каталог, относительно которого рассматриваются пути файлов, не начинающиеся с косой черты.

Например, на рис. 1.3, если */Faculty/Prof*.*Brown* будет рабочим каталогом, то при использовании пути *Courses*/*CS101* будет получен тот же самый файл, что и при указании рассмотренного выше абсолютного пути. Процесс может изменить свой рабочий каталог, воспользовавшись системным вызовом, определяющим новый рабочий каталог.

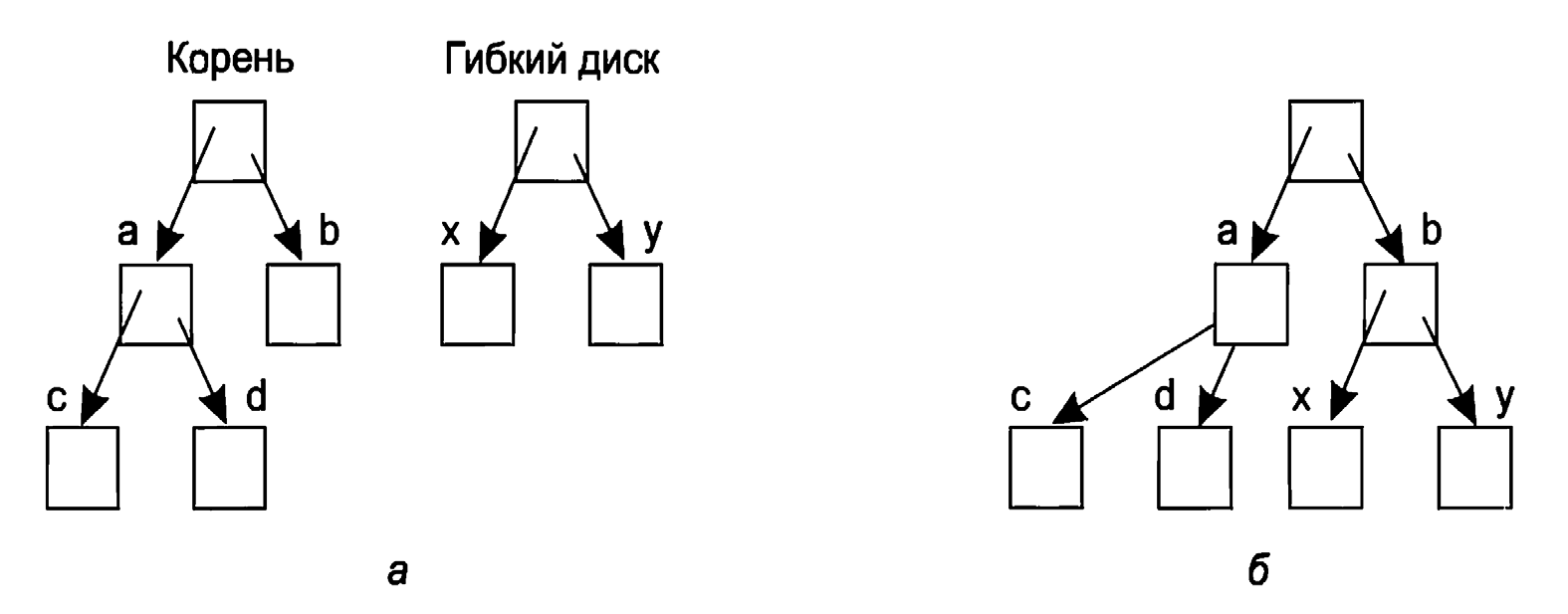
Другое важное понятие в *Unix* – смонтированная файловая система. Практически все персональные компьютеры оснащены одним и более приводами оптических дисков, в которые могут вставляться компакт-диски или диски *DVD*. У компьютеров есть *USB*-порты, к которым могут быть подключена *USB-*память или подключены внешние жесткие диски. Чтобы предоставить удобный способ работы с этими съемными носителями информации, *Unix* позволяет файловой системе на компакт-дисках или дисках *DVD* подключаться к основному дереву. Рассмотрим ситуацию, показанную на рис. 1.4а.

Перед вызовом команды mount корневая файловая система на жестком диске и вторая файловая система на компакт-диске существуют отдельно и не связаны друг с другом. Однако файлы на компакт-диске нельзя использовать, поскольку отсутствует способ определения для них полных имен. *Unix* не позволяет указывать в начале полного имени номер или имя устройства, поскольку это провело бы к жесткой зависимости от устройств, которой операционным системам лучше избегать. Вместо этого системный вызов mount позволяет подключить файловую систему на компакт-диске к корневой файловой системе в том месте, где этого потребует программа.

На рис. 1.4б файловая система на компакт-диске была подключена к каталогу *b*, открыв доступ к файлам /*b*/*х* и /*b*/*у*. Если в каталоге *b* содержались какие-нибудь файлы, то пока к нему подключена файловая система компакт-диска, эти файлы будут недоступны, поскольку путь /*b* стал ссылкой на корневой каталог компакт-диска. (Потеря доступа к этим файлам – во многом надуманная проблема: файловые системы практически всегда подключаются к пустым каталогам.) Если система оснащена несколькими жесткими дисками, то все они могут быть подключены к единому дереву аналогичным образом.

Еще одним важным понятием в *Unix* является специальный файл. Специальные файлы служат для того, чтобы устройства ввода-вывода были похожи на файлы. При этом с ними можно проводить операции чтения и записи, используя те же системные вызовы, которые применяются для чтения и записи файлов.

Существуют два вида специальных файлов: блочные специальные файлы и символьные специальные файлы. Блочные специальные файлы используются для моделирования устройств, содержащих набор блоков с произвольной адресацией, таких как диски. Аналогичным образом символьные специальные файлы используются для моделирования принтеров, модемов и других устройств, которые принимают или выдают поток символов. По принятому соглашению, специальные файлы хранятся в каталоге /*dev*. Например, путь /*dev*/*lp* может относиться к принтеру (который когда-то назывался строчным принтером – *line printer*).

Рис. 1.4 Перед подключением файлы на компакт-диске недоступны (а); после подключения они становятся частью корневой файловой системы (б)

Последним понятием в этом обзоре будут каналы, которые имеют отношение как к процессам, так и к файлам. Канал – это разновидность псевдофайла, которым можно воспользоваться для соединения двух процессов. Если процессам *А* и *В* необходимо обменяться данными с помощью канала, то они должны установить его заранее. Когда процессу *А* нужно отправить данные процессу *В*, он осуществляет запись в канал, как будто он имеет дело с выходным файлом.

Фактически реализация канала очень похожа на реализацию файла. Процесс *В* может прочитать данные, осуществляя операцию чтения из канала, как будто он имеет дело с входным файлом. Таким образом, обмен данными между процессами в *Unix* очень похож на обычные операции записи и чтения файла. Более того, только сделав специальный системный вызов, процесс может узнать, что запись выходных данных на самом деле производится не в файл, а в канал. Так как файловая система играет важную роль она будет более подробно рассмотрена в отдельном разделе.

**Ввод-вывод данных**. У всех компьютеров имеются физические устройства для получения входной и вывода выходной информации, в том числе: клавиатуры, мониторы, принтеры и т. д. Управление всеми этими устройствами возлагается на операционную систему, поэтому в ее составе существует своя подсистема ввода-вывода. Некоторые программы ввода-вывода не зависят от конкретного устройства, то есть в равной мере подходят для применения со многими или со всеми устройствами ввода-вывода. Другая часть программ, например, драйверы устройств, предназначена для определенных устройств ввода-вывода.

**Безопасность**. Компьютеры содержат большой объем информации, которую пользователям часто нужно защитить и сохранить ее конфиденциальность. Управление безопасностью системы также возлагается на операционную систему: например, она должна обеспечить доступ к файлам только пользователям, имеющим на это право.

Чтобы понять сам замысел возможной организации работы системы безопасности, обратимся в качестве простого примера к системе *Unix*. Файлам в *Unix* присваивается 9-разрядный двоичный код защиты. Этот код состоит из трех-битных полей. Одно поле – для владельца, второе – для представителей группы, в которую он входит, и третье – для всех остальных. В каждом поле есть бит, определяющий доступ для чтения, бит, определяющий доступ для записи, и бит, определяющий доступ для выполнения. Эти три бита называются *rwx*-битами (*read, write, execute*).

Например, код защиты *rwxr-x--x* означает, что владельцу доступны чтение, запись или выполнение файла, остальным представителям его группы разрешается чтение или выполнение файла (но не запись), а всем остальным разрешено выполнение файла (но не чтение или запись). Для каталога **х** означает разрешение на поиск. Дефис (минус) означает, что соответствующее разрешение отсутствует. Кроме защиты файлов, существует множество других аспектов безопасности. Один из них – это защита системы от нежелательных вторжений, как с участием, так и без участия людей (например, путем вирусных атак).

**Оболочка**. Операционная система представляет собой программу, выполняющую системные вызовы. Редакторы, компиляторы, ассемблеры, компоновщики и интерпретаторы команд по определению не являются частью операционной системы при всей своей важности и приносимой пользе. Рискуя внести некоторую путаницу, в этом разделе мы коротко рассмотрим и командный интерпретатор *Unix*, называемый оболочкой – *shell*. He являясь частью ОС, оболочка нашла широкое применение как средство доступа ко многим ее функциям и служит хорошим примером использования системных вызовов. Когда не применяется графический пользовательский интерфейс, она также является основным интерфейсом между пользователем, сидящим за своим терминалом, и операционной системой.

Существует множество оболочек, включая *sh, csh, ksh и bash*. Все они поддерживают рассматриваемые далее функции, происходящие из исходной оболочки (*sh*). Оболочка запускается после входа в систему любого пользователя. В качестве стандартного устройства ввода и вывода оболочка использует терминал. Свою работу она начинает с вывода приглашения – знака доллара, сообщающего пользователю, что оболочка ожидает приема команды.

**Вопрос 2. Системные вызовы.**

Операционные системы выполняют две основные функции: предоставляют абстракции пользовательским программам и управляют ресурсами компьютера.

В основном взаимодействие пользовательских программ и операционной системы касается первой функции: создание, запись, чтение и удаление, тогда как 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.

Рассматриваемый пример имеет непосредственное отношение к стандарту *POSIX* (Международный стандарт 9945-1), а, следовательно, к семейству ОС *Unix* (*System V, BSD, Linux, MINIX 3* и т. д.), но он является типовым для многих ОС. Поскольку фактический механизм осуществления системного вызова существенно зависит от конкретной машины и зачастую должен быть реализован на ассемблере, разработаны специальные библиотеки процедур (компиляторы), позволяющие осуществлять системные вызовы из программ, написанных, например, на языке С.

Очень полезно всегда помнить следующее. Любой однопроцессорный компьютер одновременно может выполнить только одну команду. Когда процесс выполняет пользовательскую программу в режиме пользователя и нуждается в какой-нибудь услуге операционной системы, например в чтении данных из файла, он должен выполнить команду системного прерывания, чтобы передать управление операционной системе. Затем операционная 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 системного вызова похоже на выполнение особой разновидности вызова процедуры, с той лишь разницей, что системные вызовы входят в ядро, а процедурные вызовы – нет. Для того чтобы прояснить механизм системных вызовов, рассмотрим системный вызов чтения – *read* (рис. 1.5).

Системный вызов чтения (*read*) 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 *С* с помощью вызова библиотечной процедуры, имя которой совпадает с именем системного вызова: read. Вызов из программы на *С* 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.

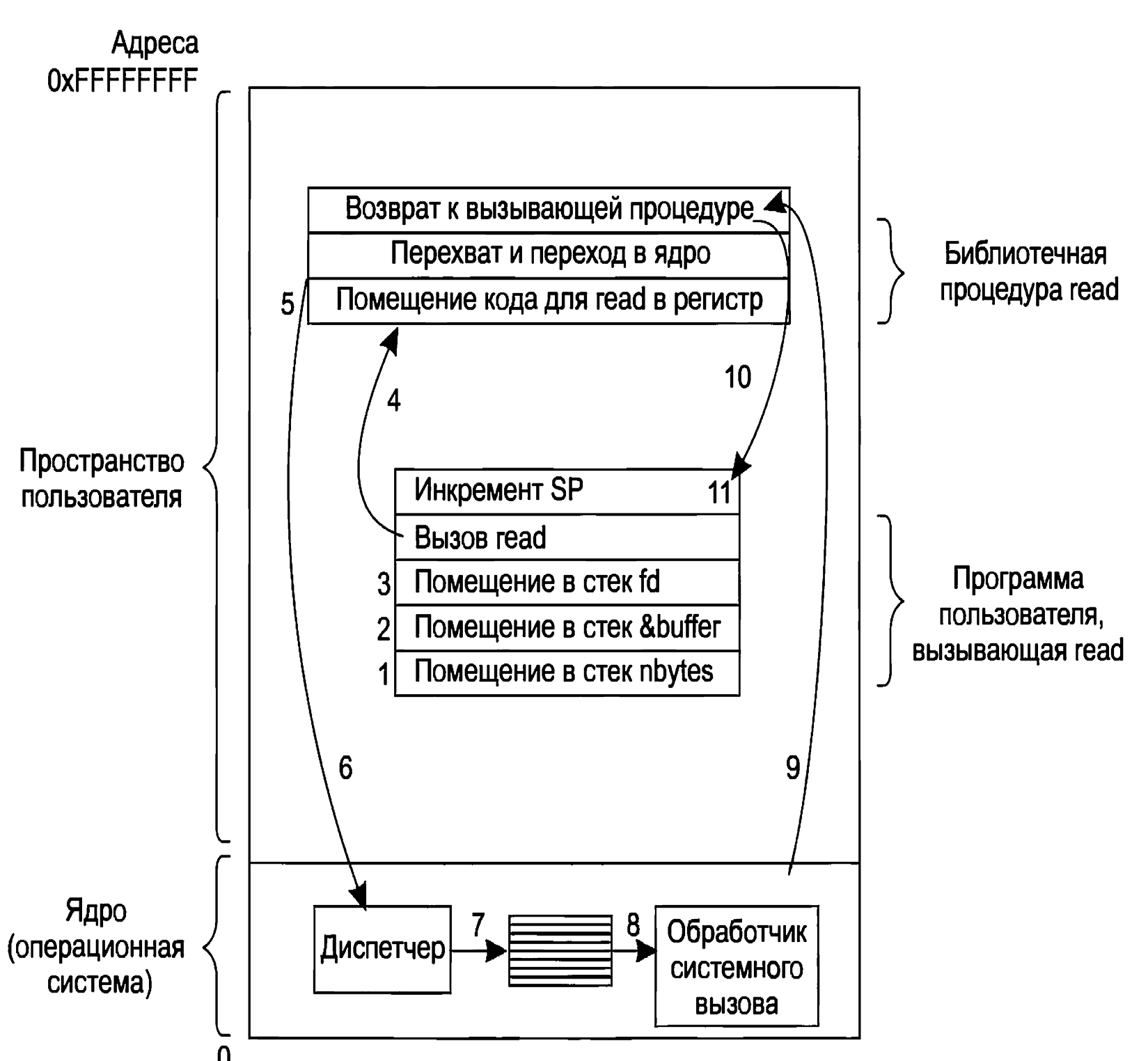


Рис. 1.5 Этапы выполнения системного вызова

*read* (*fd, buffer, nbytes*)

Если системный вызов не может быть выполнен из-за неправильных параметров или из-за ошибки диска, значение переменной *count* устанавливается в -1, а номер ошибки помещается в глобальную переменную errno. Программам обязательно нужно проверять результаты системного вызова, чтобы отслеживать возникновение ошибки.

Выполнение системного вызова состоит из нескольких шагов. Для прояснения ситуации вернемся к упоминавшемуся выше примеру вызова *read*. Сначала, при подготовке вызова библиотечной процедуры *read*, которая фактически и осуществляет системный вызов *read*, вызывающая программа помещает параметры в стек, как показано на рис. 1.5 шагами 1-3.

Компиляторы *С* и *C*++ помещают параметры в стек в обратном порядке, следуя исторически сложившейся традиции (чтобы на вершине стека оказался первый параметр функции 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 номер 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, находящегося 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 отличия. *Во-первых*, побочный эффект, заключающийся в переключении в режим ядра. Команда вызова процедуры не меняет используемый режим. А *во-вторых*, команда *TRAP* не может получить относительный или абсолютный адрес местонахождения процедуры, поскольку она не может осуществить переход на произвольный адрес.

Начавшая работу после команды *TRAP* часть ядра (диспетчер на рис. 1.5) 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 библиотечной процедуре, находящейся в пользовательской области памяти, той самой команде, которая следует за командой *TRAP* (шаг 9).

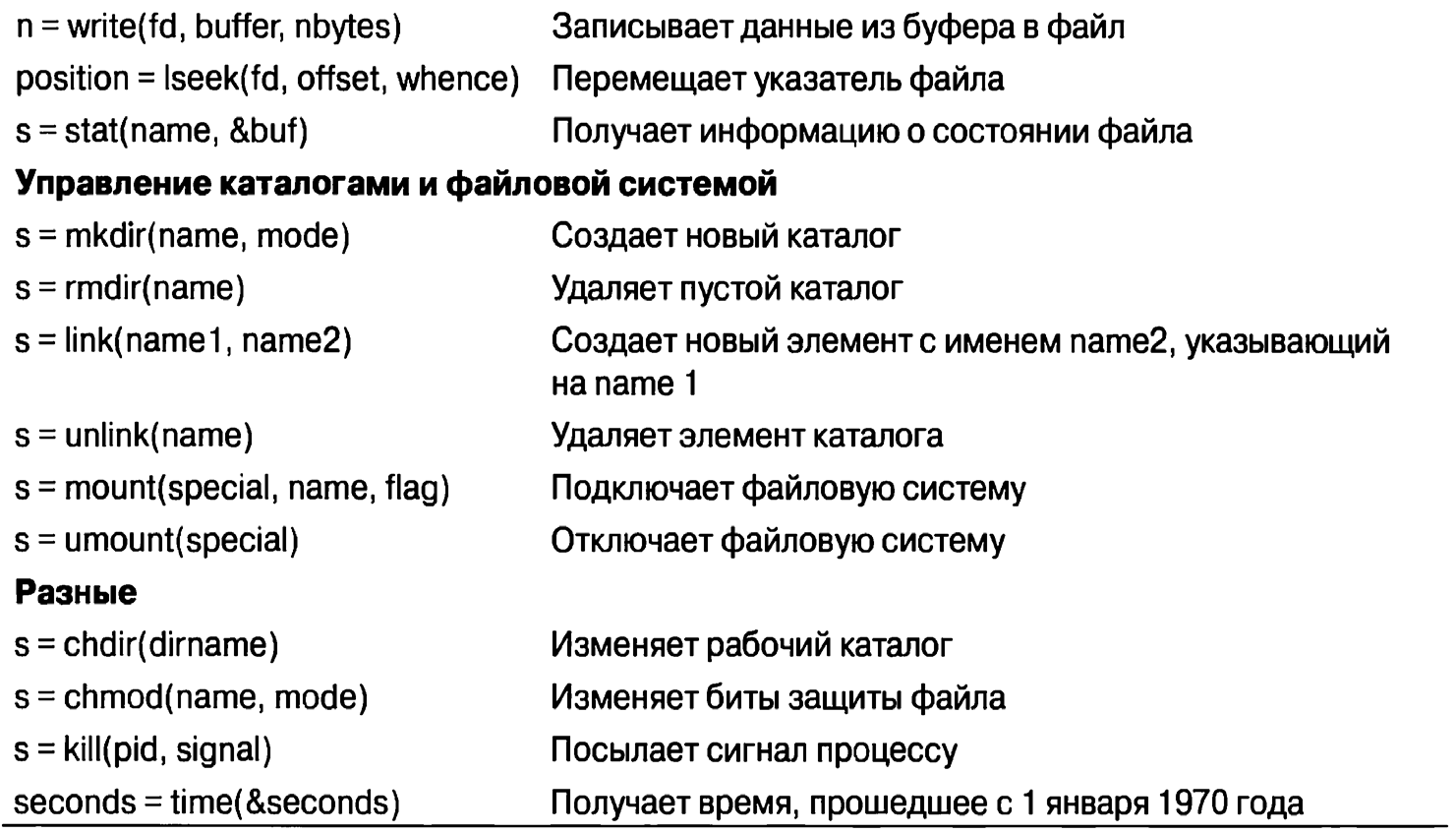
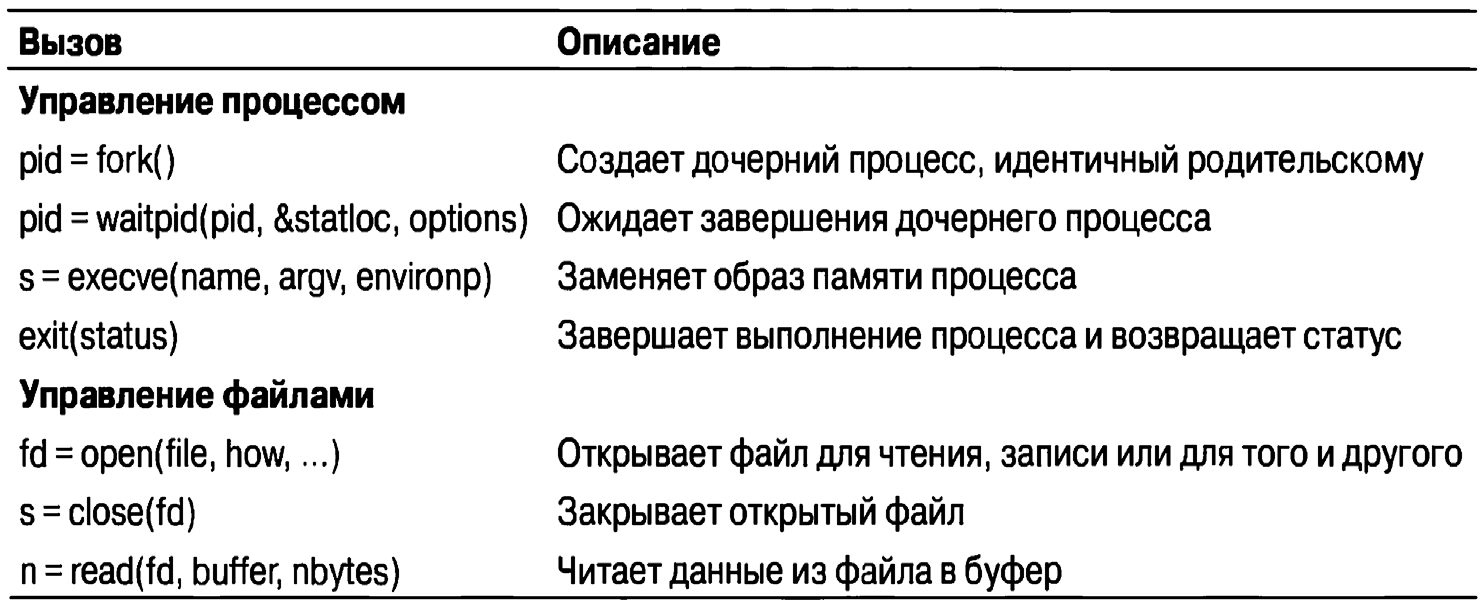
В свою очередь эта процедура вернет управление пользовательской программе по обычной схеме возврата из процедуры (шаг 10). Чтобы завершить работу с процедурой *read*, пользовательская программа должна очистить стек, точно так же, как она это делает после любого вызова процедуры (шаг 11). Полагая, что в нашем примере стек растет вниз (как это чаще всего и бывает), пользовательская программа в скомпилированном виде должна содержать команды увеличения указателя 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 возвращено библиотечной процедуре, находящейся в пользовательской области памяти». Системный вызов может заблокировать вызывающую программу, препятствуя продолжению 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, когда станут доступны требуемые входные данные, система вспомнит о заблокированном процессе и будут выполнены шаги с 9-го по 11-й.

В следующих темах мы рассмотрим некоторые из наиболее востребованных системных вызовов стандарта *POSIX*, или, точнее, библиотечных процедур, осуществляющих 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.

*Таблица 1.1*

Некоторые важнейшие системные вызовы *POSIX*



Первая группа вызовов в табл. 1.1 предназначена для управления процессами. Отдельно стоит упомянуть, что отображение процедурных вызовов *POSIX* на системные вызовы не является взаимно однозначным.

В стандарте *POSIX* определены процедуры, 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 (т. е. 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 непосредственно 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.

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. 1.6, память процессов состоит из трех сегментов: текста, данных и стека, причем сегмент данных растет вверх, а стек растет вниз. Между ними существует часть неиспользованного адресного пространства. Стек заполняет пустое пространство автоматически, по мере надобности.

Многие системные вызовы имеют отношение к файловой системе. Ниже рассмотрим вызовы, работающие с отдельными файлам, каталогами и всей файловой системой в целом.

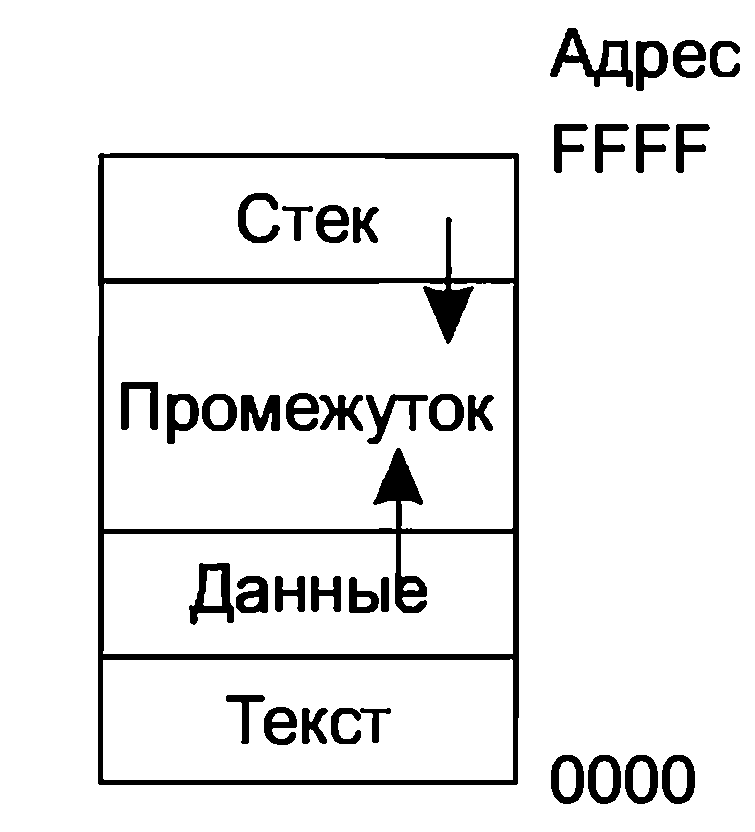
Чтобы прочитать данные из файла или записать их в файл, сначала его необходимо открыть с помощью системного вызова open. Для данного вызова необходимо указать имя открываемого файла и код O\_RDONLY, O\_WRONLY или O\_RDWR, означающий, что файл открывается для чтения, записи или для чтения и записи.

Рис. 1.6 Типовая структура памяти процессов

Для создания нового файла используется параметр *O\_CREAT*. Наиболее часто используемыми вызовами, несомненно, являются read и *write*. Вызов *read* мы уже рассмотрели, вызов *write* имеет те же параметры.

Вызов *lseek* имеет три параметра: *первый* – это дескриптор файла; *второй* – позиция в файле;

*третий* указывает, относительно чего задана позиция – относительно начала файла, текущей 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, время последнего изменения и другую информацию. Программы могут запрашивать эту информацию посредством системного вызова *stat*.

Первый параметр системного вызова stat определяет файл, информацию о котором необходимо получить, второй является указателем на структуру, куда она должна быть помещена. Для открытого файла то же самое делает системный вызов *fstat*.

Системные вызовы для управления каталогами *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.

Чтобы увидеть, как работает вызов *link*, рассмотрим ситуацию, показанную на рис. 1.7а.

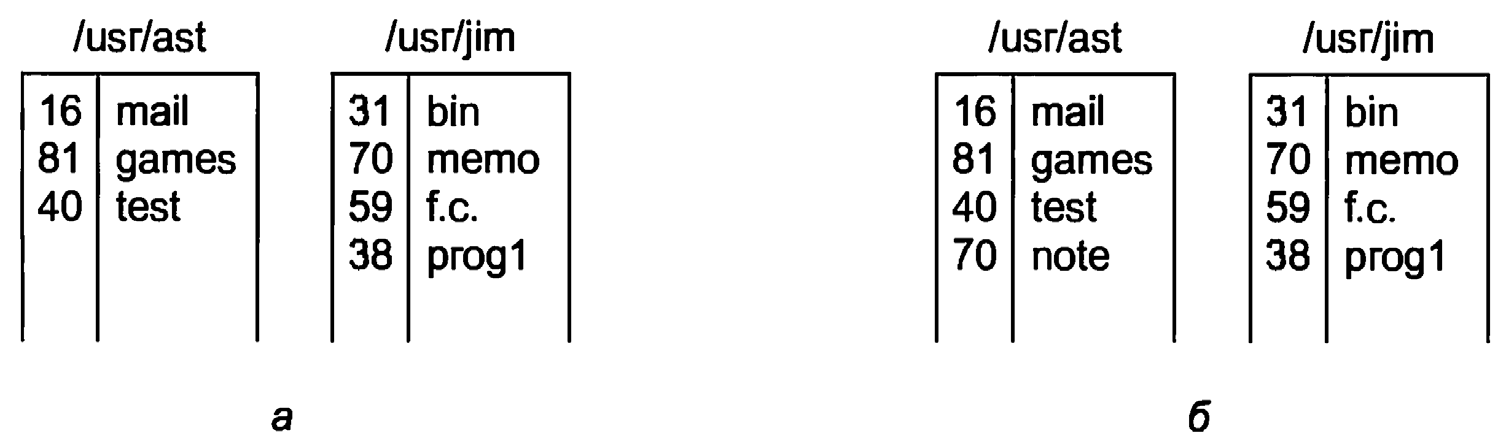


Рис. 1.7 Два каталога перед созданием ссылки на файл /*usr/jim/memo* в каталоге ast (a), и они же после создания ссылки (б)

У каждого из двух пользователей с именами *ast* и *jim* есть свои 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, определяется местным системным администратором.

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-номера, 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. 1.7 у файла *mail* имеется *i*-номер 16, и т. д. Системный вызов *link* просто создает новый элемент каталог, возможно, с новым именем, используя *i*номер существующего файла. На рис. 1.7б 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.

Помимо описанных выше, существуют также и другие разновидности системных вызовов. Здесь будут рассмотрены только четыре из них. Системный вызов *chdir* изменяет текущий рабочий каталог. После вызова *сhdiг* ("/*usr/ast/test*"), при открытии файла *xyz* будет открыт файл /*usr/ast/test/xyz*. Использование понятия рабочего каталога избавляет от необходимости постоянно набирать длинные абсолютные 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 пользователей. Системный вызов chmod позволяет изменять биты прав доступа к файлу. Например, чтобы сделать файл доступным по чтению для всех, а для владельца – доступным и для чтения и для записи, необходимо выполнить следующий системный вызов *chmod*.

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* состоят из кода, который выполняет те или иные действия, при необходимости обращаясь к системе с системными вызовами для получения конкретных услуг.

В отличие от этого программа *Windows* управляется, как правило, событиями. Основная программа ждет, пока возникнет какое-нибудь событие, а затем вызывает процедуру для его обработки. Типичные события – это нажатие клавиши, перемещение мыши, нажатие кнопки мыши или вставка в привод компакт-диска. Затем для обслуживания события, обновления экрана и обновления внутреннего состояния программы вызываются обработчики. В итоге все это приводит к несколько иному стилю программирования, чем в *Unix*, но поскольку эта дисциплина посвящена функциям и структурам операционных систем, различные модели программирования не будут вызывать у нас особого интереса.

**Заключение.**

Ключевым понятием во всех операционных системах является процесс. Процессом, по существу, является программа вовремя ее выполнения. С каждым процессом связано его адресное пространство – список адресов ячеек памяти от нуля и до некоторого максимума, откуда процесс может считывать и куда записывает данные.

Адресное пространство содержит выполняемую программу, данные этой программы и ее стек. Кроме этого, с каждым процессом связан набор ресурсов, который обычно включает регистры, список открытых файлов, необработанные предупреждения, список связанных процессов и всю остальную информацию, необходимую в процессе работы программы.

В любой операционной системе поддерживается механизм, который позволяет пользовательским программам обращаться к услугам ядра ОС. В операционных системах наиболее известной советской вычислительной машины БЭСМ-6 соответствующие средства «общения» с ядром назывались экстракодами, в операционных системах IBM они назывались системными макрокомандами и т. д. В ОС Unix такие средства называют системными вызовами.

Системные вызовы (system calls) — это интерфейс между операционной системой и пользовательской программой. Они создают, удаляют и> используют различные объекты, главные из которых — процессы и файлы. Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют прерывание процессора, после чего управление передается обработчику данного вызова, входящему в ядро операционной системы. Цель таких библиотек — сделать системный вызов похожим на обычный вызов подпрограммы.

Основное отличие состоит в том, что при системном вызове задача переходит в привилегированный режим или режим ядра (kernel mode). Поэтому системные вызовы иногда еще называют программными прерываниями, в отличие от аппаратных прерываний, которые чаще называют просто прерываниями.

В этом режиме работает код ядра операционной системы, причем исполняется он в адресном пространстве и в контексте вызвавшей его задачи. Таким образом, ядро операционной системы имеет полный доступ к памяти пользовательской программы, и при системном вызове достаточно передать адреса одной или нескольких областей памяти с параметрами вызова и адреса одной или нескольких областей памяти для результатов вызова.

В большинстве операционных систем системный вызов осуществляется командой программного прерывания (INT). Программное прерывание — это синхронное событие, которое может быть повторено при выполнении одного и того же программного кода.