**ПРОЦЕССЫ**

Ключевым понятием во всех операционных системах является **процесс**. Процессом, по существу, является программа во время ее выполнения. С каждым процессом связано его **адресное пространство** — список адресов ячеек памяти от нуля до некоторого максиму- ма, откуда процесс может считывать данные и куда может записывать их. Адресное про- странство содержит выполняемую программу, данные этой программы и ее стек. Кроме этого, с каждым процессом связан набор ресурсов, который обычно включает регистры (в том числе счетчик команд и указатель стека), список открытых файлов, необрабо- танные предупреждения, список связанных процессов и всю остальную информацию, необходимую в процессе работы программы. Таким образом, процесс — это контейнер, в котором содержится вся информация, необходимая для работы программы.

Более подробно понятие процесса будет рассмотрено в главе 2, а сейчас, для того чтобы выработать интуитивное представление о процессе, рассмотрим систему, работающую в мультипрограммном режиме. Пользователь может запустить программу редакти- рования видео и указать конвертирование одночасового видеофайла в какой-нибудь определенный формат (процесс займет несколько часов), а затем переключиться на блуждания по Интернету. При этом может заработать фоновый процесс, который пе- риодически «просыпается» для проверки входящей электронной почты. И у нас уже будет (как минимум) три активных процесса: видеоредактор, веб-браузер и программа получения (клиент) электронной почты. Периодически операционная система будет принимать решения остановить работу одного процесса и запустить выполнение дру- гого, возможно, из-за того, что первый исчерпал свою долю процессорного времени в предыдущую секунду или две.

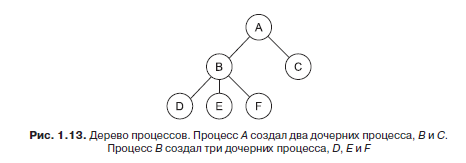
Если процесс приостанавливается таким образом, позже он должен возобновиться именно с того состояния, в котором был остановлен. Это означает, что на период приостановки вся информация о процессе должна быть явным образом где-то сохра- нена. Например, у процесса могут быть одновременно открыты для чтения несколь- ко файлов. С каждым из этих файлов связан указатель текущей позиции (то есть номер байта или записи, которая должна быть считана следующей). Когда процесс приостанавливается, все эти указатели должны быть сохранены, чтобы вызов *read*, выполняемый после возобновления процесса, приводил к чтению нужных данных. Во многих операционных системах вся информация о каждом процессе, за исклю- чением содержимого его собственного адресного пространства, хранится в таблице операционной системы, которая называется **таблицей процессов** и представляет со- бой массив (или связанный список) структур, по одной на каждый из существующих на данный момент процессов.

Таким образом, процесс (в том числе приостановленный) 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, явля- ются вызовы, связанные с созданием и завершением процессов. Рассмотрим простой пример. Процесс, называемый **интерпретатором команд**, или **оболочкой**, считывает команды с терминала. Пользователь только что набрал команду, требующую ком- пиляции программы. Теперь оболочка должна создать новый процесс, запускающий компилятор. Когда этот процесс завершит компиляцию, он произведет системный вызов для завершения собственного существования.

Если процесс способен создавать несколько других процессов (называющихся **дочер- ними процессами**), а эти процессы в свою очередь могут создавать собственные дочер- ние процессы, то перед нами предстает дерево процессов, подобное изображенному на рис. 1.13. Связанные процессы, совместно работающие над выполнением какой-нибудь задачи, зачастую нуждаются в обмене данными друг с другом и синхронизации своих действий. Такая связь называется **межпроцессным взаимодействием** и будет подробно рассмотрена в главе 2.

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.



Временами возникает потребность в передаче информации запущенному процессу, который не находится в состоянии ожидания этой информации. Можно привести в пример процесс, который обменивается информацией с другим процессом, за- пущенным на другом компьютере, и посылает удаленному процессу сообщение по сети. Чтобы застраховаться от возможной утраты сообщения или ответа на него, отправитель может запросить собственную операционную систему уведомить его по истечении определенного интервала времени, чтобы он мог повторно отправить сообщение, если не получит подтверждения его получения раньше. После установки такого таймера программа может продолжить выполнение другой работы.

Когда истечет заданный интервал времени, операционная система посылает процессу **сигнал тревоги**. Этот сигнал заставляет процесс приостановить выполняемую работу, сохранить в стеке состояние своих регистров и запустить специальную процедуру обработки сигнала тревоги, для того чтобы, к примеру, заново передать предполо- жительно утраченное сообщение. Когда обработчик сигнала завершит свою работу, запущенный процесс возобновится в том самом состоянии, которое было до поступле- ния сигнала. Сигналы являются программными аналогами аппаратных прерываний. Они могут генерироваться в различных ситуациях, а не только по истечении времени, установленного в таймере. Многие аппаратные прерывания (например, выполнение недопустимой команды или обращение по неверному адресу) также транслируются процессу, при выполнении которого произошла ошибка.

Каждому пользователю, которому разрешено работать с системой, системным админи- стратором присваивается **идентификатор пользователя** (User IDentification (**UID**)). Каждый запущенный процесс имеет UID того пользователя, который его запустил. Дочерние процессы имеют такой же UID, как и у родительского процесса. Пользо- ватели могут входить в какую-нибудь группу, каждая из которых имеет собственный **идентификатор группы** (Group IDentification (**GID**)).

Пользователь с особым значением UID, называемый в UNIX суперпользователем (superuser), а в Windows администратором (administrator), имеет особые полномочия, позволяющие пренебрегать многими правилами защиты. В крупных компьютерных системах только системный администратор знает пароль, необходимый для получения прав суперпользователя, но многие обычные пользователи (особенно студенты) при- кладывают немалые усилия, пытаясь отыскать бреши в системе, которые позволили бы им стать суперпользователем без пароля.

**ВЗАИМОБЛОКИРОВКА**

При работе многих приложений процессу нужен исключительный доступ не к одному, а сразу к нескольким ресурсам. Предположим, к примеру, что каждый из двух процес- сов захотел записать отсканированный документ на Blu-ray-диск. Процесс *A* запраши- вает разрешение на использование сканера и получает его. Процесс *B* запрограмми- рован по-другому: сначала он запрашивает разрешение на использование пишущего привода Blu-ray-дисков и также получает это разрешение. Теперь *A* запрашивает разре- шение на использование пишущего привода Blu-ray-дисков, но запрос отклоняется до тех пор, пока это устройство не будет освобождено процессом *B*. К сожалению, вместо того чтобы освободить привод, *B* запрашивает разрешение на использование сканера. И в этот момент оба процесса оказываются заблокированными навсегда. Такая ситуа- ция называется тупиковой ситуацией (тупиком), или **взаимоблокировкой** (deadlock).

Взаимоблокировки могут случаться и между машинами. К примеру, многие офисы оборудованы локальной сетью, к которой подключено множество компьютеров.

**ФАЙЛЫ**

Другим ключевым понятием, поддерживаемым практически всеми операционными системами, является файловая система. Как отмечалось ранее, основная функция операционной системы — скрыть специфику дисков и других устройств ввода-вывода и предоставить программисту удобную и понятную абстрактную модель, состоящую из независимых от устройств файлов. Вполне очевидно, что для создания, удаления, чтения и записи файлов понадобятся системные вызовы. Перед тем как файл будет готов к чтению, он должен быть найден на диске и открыт, а после считывания — за-крыт. Для проведения этих операций предусмотрены системные вызовы.

Чтобы предоставить место для хранения файлов, многие операционные системы персональных компьютеров используют **каталог** как способ объединения файлов в группы.

Например, у студента может быть по одному каталогу для каждого изучаемого курса (для программ, необходимых в рамках данного курса), каталог для электронной почты и еще один — для своей домашней веб-страницы. Для создания и удаления каталогов нужны системные вызовы. Они также нужны для помещения в каталог существующего файла и удаления его оттуда. Элементами каталога могут быть либо файлы, либо другие каталоги. Эта модель стала прообразом иерархической структуры файловой системы, один из вариантов которой показан на рис. 1.14.



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 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. Обычно только родительский процесс может управлять дочерним про- цессом или даже обращаться к нему, но практически всегда существуют механизмы, позволяющие читать файлы и каталоги не только их владельцу, но и более широкой группе пользователей.

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 (слеша). На рис. 1.14 путь к файлу CS101 будет иметь вид /Faculty/Prof.Brown/Courses/CS101. Первая косая черта является признаком использования абсолютного пути, который начинается в корневом каталоге. Следует заметить: в Windows в качестве разделителя вместо прямой косой черты (/) использует- ся обратная (\), поэтому показанный выше путь к файлу должен быть записан в следу- ющем виде: \Faculty\Prof.Brown\Courses\CS101. На страницах этой книги при указании путей к файлам будет в основном использоваться соглашение, действующее в UNIX.

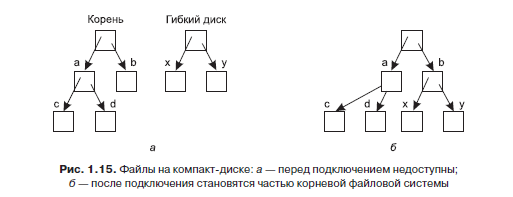
В любой момент времени у каждого процесса есть текущий **рабочий каталог**, отно- сительно которого рассматриваются пути файлов, не начинающиеся с косой черты. Например, на рис. 1.14, если /Faculty/Prof.Brown будет рабочим каталогом, то при ис- пользовании пути Courses/CS101 будет получен тот же самый файл, что и при указании рассмотренного ранее абсолютного пути. Процесс может изменить свой рабочий ката- лог, воспользовавшись системным вызовом, определяющим новый рабочий каталог.

Перед тем как с файлом можно будет работать в режиме записи или чтения, он должен быть открыт. На этом этапе происходит также проверка прав доступа. Если доступ раз- решен, система возвращает целое число, называемое **дескриптором файла**, который используется в последующих операциях. Если доступ запрещен, то возвращается код ошибки.

Другое важное понятие в UNIX — **смонтированная файловая система**. Большинство настольных компьютеров оснащено одним и более приводами оптических дисков, в которые могут вставляться компакт-диски, диски DVD и Blu-ray. У компьютеров, как правило, есть USB-порты, к которым может быть подключена USB-память (фактиче- ски это твердотельные устройства, заменяющие дисковые накопители), а некоторые компьютеры имеют приводы гибких дисков или подключенные к ним внешние жесткие диски. Чтобы предоставить удобный способ работы с этими съемными носителями информации, UNIX позволяет файловой системе на оптическом диске подключаться к основному дереву. Рассмотрим ситуацию, показанную на рис. 1.15, *а*. Перед вызовом команды *mount* **корневая файловая система** на жестком диске и вторая файловая си- стема 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 привело 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. 1.15, *б* файловая система на компакт-диске была подключена к каталогу b, открыв доступ к файлам /b/x и /b/y. 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. Открывая блочный специальный файл и считывая, скажем, блок 4, программа может напрямую получить доступ к четвертому блоку устройства независимо от структуры имеющейся у него файловой системы. Аналогичным образом символьные специальные файлы используются для моделирования принтеров, модемов и других устройств, которые принимают или выдают поток символов. В соответствии с принятым соглашением специальные файлы хранятся в каталоге /dev. Например, путь /dev/lp может отно- ситься к принтеру (который когда-то назывался строчным принтером — line printer).



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 (рис. 1.16). Если процессам *A* и *B* необ- ходимо обменяться данными с помощью канала, то они должны установить его заранее. Когда процессу *A* нужно отправить данные процессу *B*, он осуществляет запись в канал, как будто имеет дело с выходным файлом. Фактически реализация канала очень похо- жа на реализацию файла. Процесс *B* 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 чте- ния 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.



Файловая система играет очень важную роль.

**УПРАВЛЕНИЕ ПАМЯТЬЮ**

Память представляет собой очень важный ресурс, требующий четкого управления. Несмотря на то что в наши дни объем памяти среднего домашнего компьютера в де- сятки тысяч раз превышает ресурсы IBM 7094, бывшего в начале 1960-х годов самым большим компьютером в мире, размер компьютерных программ растет быстрее, чем объем памяти. Закон Паркинсона можно перефразировать следующим образом: «Программы увеличиваются в размерах, стремясь заполнить всю память, доступную для их размещения». В этой главе мы рассмотрим, как операционные системы созда- ют из памяти абстракции и как они этими абстракциями управляют.

В идеале каждому программисту хотелось бы иметь предоставленную только ему не- ограниченную по объему и скорости работы память, которая к тому же не теряет своего содержимого при отключении питания. Раз уж мы так размечтались, то почему бы не сделать память еще и совсем дешевой? К сожалению, существующие технологии пока не могут дать нам желаемого. Может быть, способ создания такой памяти удастся изо- брести именно вам.

Тогда чем же нам придется довольствоваться? Со временем была разработана кон- цепция **иерархии памяти**, согласно которой компьютеры обладают несколькими мегабайтами очень быстродействующей, дорогой и энергозависимой кэш-памяти, несколькими гигабайтами памяти, средней как по скорости, так и по цене, а так- же несколькими терабайтами памяти на довольно медленных, сравнительно де- шевых дисковых накопителях, не говоря уже о сменных накопителях, таких как DVD и флеш-устройства USB. Превратить эту иерархию в абстракцию, то есть в удобную модель, а затем управлять этой абстракцией — и есть задача операционной системы.

Та часть операционной системы, которая управляет иерархией памяти (или ее частью), называется **менеджером**, или **диспетчером**, **памяти**. Он предназначен для действен- ного управления памятью и должен следить за тем, какие части памяти используются, выделять память процессам, которые в ней нуждаются, и освобождать память, когда процессы завершат свою работу.

**ВВОД-ВЫВОД ДАННЫХ**

У всех компьютеров имеются физические устройства для получения входной и вы- вода выходной информации. Действительно, какой будет прок от компьютера, если пользователи не смогут поставить ему задачу и получить результаты по завершении заданной работы? Существует масса разнообразных устройств ввода-вывода: клавиатуры, мониторы, принтеры и т. д. Управление всеми этими устройствами возлагается на операционную систему.

Поэтому у каждой операционной системы для управления такими устройствами 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, напри- мер драйверы устройств, предназначена для определенных устройств ввода-вывода. Программное обеспечение подсистемы ввода-вывода будет рассмотрено в главе 5.

**БЕЗОПАСНОСТЬ**

Компьютеры содержат большой объем информации, и часто пользователям нужно защитить ее и сохранить ее конфиденциальность. Возможно, это электронная почта, бизнес-планы, налоговые декларации и многое другое. Управление безопасностью си- стемы 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 присваи- вается 9-разрядный двоичный код защиты. Этот код состоит из трехбитных полей. Одно поле — для владельца, второе — для представителей группы, в которую он входит (разделяет пользователей на группы системный администратор), третье — для всех остальных. В каждом поле есть бит, определяющий доступ для чтения, бит, опреде- ляющий доступ для записи, и бит, определяющий доступ для выполнения. Эти три бита называются **rwx-битами** (**r**ead, **w**rite, e**x**ecute). 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 ката- лога *x* 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). Различные вопросы, связанные с обес печением безопасности, будут рассматриваться в главе 9.

**ОБОЛОЧКА**

Операционная система представляет собой программу, выполняющую системные вызо- вы. Редакторы, компиляторы, ассемблеры, компоновщики, утилиты 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. Не являясь частью операционной системы, оболочка нашла широкое применение как средство доступа ко многим ее функциям и служит хорошим примером исполь- зования 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 стан- дартного устройства ввода и вывода оболочка использует терминал1. Свою работу она начинает с вывода **приглашения** — знака доллара, сообщающего пользователю, что оболочка ожидает приема команды. Например, если теперь пользователь наберет на клавиатуре

date

оболочка создаст дочерний процесс и запустит дочернюю программу date. Пока вы- полняется дочерний процесс, оболочка ожидает его завершения. После завершения дочернего процесса оболочка снова выведет приглашение и попытается прочитать следующую введенную строку.

Пользователь может указать, что стандартный вывод необходимо перенаправить в файл, например,

date >file

Точно так же может быть перенаправлен и стандартный ввод

sort <file1 >file2

Эта команда вызывает программу сортировки sort, входные данные для которой берутся из файла file1, а выходные данные отправляются в файл file2.

Выходные данные одной программы могут быть использованы в качестве входных для другой программы путем их соединения с помощью канала. Например, команда

cat file1 file2 file3 | sort >/dev/lp

вызывает программу cat для объединения трех файлов и отправки выходных данных программе sort, чтобы она расставила все строки в алфавитном порядке. Выходные данные программы sort перенаправляются в файл /dev/lp, которым обычно обознача- ется принтер.

Если пользователь после команды введет знак &, оболочка не станет ожидать ее завершения, а сразу же выведет приглашение. Следовательно, команда

cat file1 file2 file3 | sort >/dev/lp &

приступит к сортировке как к фоновому заданию, позволяя пользователю во время сортировки продолжить обычную работу. Оболочка имеет и ряд других интересных свойств, рассмотреть которые нам не позволяет объем книги. Но в большинстве книг по UNIX оболочка рассматривается довольно подробно (например, Kernighan and Pike, 1984; Kochan Quigley, 2004; Robbins, 2005).

В наши дни на большинстве персональных компьютеров используется графический пользовательский интерфейс. По сути, графический пользовательский интерфейс — это просто программа (или совокупность программ), работающая поверх операционной системы наподобие оболочки. В системах Linux этот факт проявляется явным образом, поскольку у пользователя есть выбор по крайней мере из двух сред, реализующих графи- ческий пользовательский интерфейс: Gnome и KDE. Или он может вообще не выбрать ни одну из них, воспользовавшись окном терминала из X11. В Windows также есть возможность заменить стандартный менеджер рабочего стола (Windows Explorer) какой-нибудь другой программой путем внесения изменений в некоторые значения реестра, хотя этой возможностью практически никто не пользуется.