Глава 4
Файловые системы

В хранении и извлечении информации нуждаются все компьютерные приложения. Работающий процесс в собственном адресном пространстве может хранить лишь ограниченное количество данных. Но емкость хранилища ограничена размером виртуального адресного пространства. Ряду приложений вполне достаточно и этого объема, но есть и такие приложения, например системы резервирования авиабилетов, системы банковского или корпоративного учета, для которых его явно недостаточно.

Вторая проблема, связанная с хранением информации в пределах адресного пространства процессов, заключается в том, что при завершении процесса эта информация теряется. Для многих приложений (например, баз данных) информация должна храниться неделями, месяцами или даже бесконечно. Ее исчезновение с завершением процесса абсолютно неприемлемо. Более того, она не должна утрачиваться и при аварийном завершении процесса при отказе компьютера.

Третья проблема заключается в том, что зачастую возникает необходимость в предоставлении одновременного доступа к какой-то информации (или ее части) нескольким процессам. Если интерактивный телефонный справочник будет храниться в пределах адресного пространства только одного процесса, то доступ к нему сможет получить только этот процесс. Эта проблема решается за счет придания информации как таковой независимости от любых процессов.

Таким образом, есть три основных требования к долговременному хранилищу информации:

1. Оно должно предоставлять возможность хранения огромного количества информации.
2. Информация должна пережить прекращение работы использующего ее процесса.
3. К информации должны иметь одновременный доступ несколько процессов.

В качестве такого долговременного хранилища долгие годы используются магнитные диски. В последние годы растет популярность твердотельных накопителей, поскольку у них нет склонных к поломке движущихся частей. К тому же они предлагают более быстрый произвольный доступ к данным. Также широко используются магнитные ленты и оптические диски, но их производительность значительно ниже, и они обычно используются в качестве резервных хранилищ. Более подробное изучение дисков предстоит в главе 5, но сейчас нам вполне достаточно представлять себе диск в виде устройства с линейной последовательностью блоков фиксированного размера, которое поддерживает две операции:

- чтение блока *k*;
- запись блока *k*.

На самом деле этих операций больше, но, в принципе, решить проблему долговременного хранения могут и эти две.

Тем не менее эти операции очень неудобны, особенно на больших системах, используемых многими приложениями и, возможно, несколькими пользователями (например, на сервере). Приведем навскидку лишь часть возникающих вопросов:

- Как ведется поиск информации?
- Как уберечь данные одного пользователя от чтения их другим пользователем?
- Как узнать, которые из блоков свободны?

А ведь таких вопросов значительно больше.

По аналогии с тем, что мы уже видели — как операционная система абстрагируется от понятия процессора, чтобы создать абстракцию процесса, и как она абстрагируется от понятия физической памяти, чтобы предложить процессам виртуальные адресные пространства, — мы можем решить эту проблему с помощью новой абстракции — файла. Взятые вместе абстракции процессов (и потоков), адресных пространств и файлов являются наиболее важными понятиями, относящимися к операционным системам. Если вы реально разбираетесь в этих понятиях от начала до конца, значит, вы на правильном пути становления в качестве специалиста по операционным системам.

Файлы являются логическими информационными блоками, создаваемыми процессами. На диске обычно содержатся тысячи или даже миллионы не зависящих друг от друга файлов. Фактически если рассматривать каждый файл как некую разновидность адресного пространства, то это будет довольно близко к истине, за исключением того, что файлы используются для моделирования диска, а не оперативной памяти.

Процессы могут считывать существующие файлы и, если требуется, создавать новые. Информация, хранящаяся в файлах, должна иметь **долговременный характер**, то есть на нее не должно оказывать влияния создание процесса и его завершение. Файл должен прекращать свое существование только в том случае, если его владелец удаляет его явным образом. Хотя операции чтения и записи файлов являются самыми распространенными, существует множество других операций, часть из которых будут рассмотрены далее.

Файлами управляет операционная система. Структура файлов, их имена, доступ к ним, их использование, защита, реализация и управление ими являются основными вопросами разработки операционных систем. В общем и целом, та часть операционной системы, которая работает с файлами, и будет темой этой главы.

С позиции пользователя наиболее важным аспектом файловой системы является ее представление, то есть что собой представляет файл, как файлы именуются, какой защитой обладают, какие операции разрешено проводить с файлами и т. д. А подробности о том, что именно используется для отслеживания свободного пространства хранилища — связанные списки или битовая матрица, и о том, сколько секторов входит в логический дисковый блок, ему неинтересны, хотя они очень важны для разработчиков файловой системы. Поэтому мы разбили главу на несколько разделов. Первые два раздела посвящены пользовательскому интерфейсу для работы с файлами и каталогами соответственно. Затем следует подробное рассмотрение порядка реализации файловой системы и управления ею. И наконец, будут приведены несколько примеров реально существующих файловых систем.

4.1. Файлы

На следующих страницах мы взглянем на файлы с пользовательской точки зрения, то есть рассмотрим, как они используются и какими свойствами обладают.

4.1.1. Имена файлов

Файл является механизмом абстрагирования. Он предоставляет способ сохранения информации на диске и последующего ее считывания, который должен оградить пользователя от подробностей о способе и месте хранения информации и деталей фактической работы дисковых устройств.

Наверное, наиболее важной характеристикой любого механизма абстрагирования является способ управления объектами и их именования, поэтому исследование файловой системы начнется с вопроса, касающегося имен файлов. Когда процесс создает файл, он присваивает ему имя. Когда процесс завершается, файл продолжает существовать, и к нему по этому имени могут обращаться другие процессы.

Конкретные правила составления имен файлов варьируются от системы к системе, но все ныне существующие операционные системы в качестве допустимых имен файлов позволяют использовать от одной до восьми букв. Поэтому для имен файлов можно использовать слова andrea, bruce и cathy. Зачастую допускается также применение цифр и специальных символов, поэтому допустимы также такие имена, как 2, urgent! и Fig.2-14. Многие файловые системы поддерживают имена длиной до 255 символов.

Некоторые файловые системы различают буквы верхнего и нижнего регистров, а некоторые не делают таких различий. Система UNIX подпадает под первую категорию, а старая MS-DOS — под вторую. (Кстати, при всей своей древности MS-DOS до сих пор довольно широко используется во встроенных системах, так что она отнюдь не устарела.) Поэтому система UNIX может рассматривать сочетания символов maria, Maria и MARIA как имена трех разных файлов. В MS-DOS все эти имена относятся к одному и тому же файлу.

Наверное, будет кстати следующее отступление, касающееся файловых систем. Обе операционные системы, Windows 95 и Windows 98, использовали файловую систему MS-DOS под названием **FAT-16**, и поэтому они унаследовали множество ее свойств, касающихся, например, построения имен файлов. В Windows 98 было представлено расширение FAT-16, которое привело к системе **FAT-32**, но обе эти системы очень похожи друг на друга. Вдобавок к этому Windows NT, Windows 2000, Windows XP, Windows Vista, Windows 7 и Windows 8 по-прежнему поддерживают обе файловые системы FAT, которые к настоящему времени фактически уже устарели. Но новые операционные системы имеют собственную намного более совершенную файловую систему NTFS, которая обладает несколько иными свойствами (к примеру, допускает имена файлов в кодировке Unicode). На самом деле для Windows 8 имеется вторая файловая система, известная как ReFS (или Resilient File System — восстанавливаемая файловая система), но она предназначена для серверной версии. В этой главе все ссылки на MS-DOS или файловую систему FAT будут, если не указано иное, подразумевать системы FAT-16 и FAT-32, используемые в Windows. Далее в этой главе мы рассмотрим файловую систему FAT, а систему NTFS — в главе 12, когда будем подробно изучать операционную систему Windows 8. Кстати, есть также новая FAT-подобная файловая система, известная как **exFAT**. Это созданное компанией Microsoft расширение к FAT-32, опти-

мизированное для флеш-накопителей и больших файловых систем. ExFAT является единственной современной файловой системой компании Microsoft, в отношении которой в OS X допускаются чтение и запись.

Многие операционные системы поддерживают имена файлов, состоящие из двух частей, разделенных точкой, как, например, prog.c. Та часть имени, которая следует за точкой, называется **расширением имени файла** и, как правило, несет в себе некоторую информацию о файле. К примеру, в MS-DOS имена файлов состоят из 1–8 символов и имеют (необязательно) расширение, состоящее из 1–3 символов. В UNIX количество расширений выбирает сам пользователь, так что имя файла может иметь два и более расширений, например homepage.html.zip, где .html указывает на наличие веб-страницы в коде HTML, а .zip — на то, что этот файл (homepage.html) был сжат архиватором. Некоторые широко распространенные расширения и их значения показаны в табл. 4.1.

Таблица 4.1. Некоторые типичные расширения имен файлов

Расширение	Значение
.bak	Резервная копия файла
.c	Исходный текст программы на языке C
.gif	Изображение формата GIF
.hlp	Файл справки
.html	Документ в формате HTML
.jpg	Статическое растровое изображение в формате JPEG
.mp3	Музыка в аудиоформате MPEG layer 3
.mpg	Фильм в формате MPEG
.o	Объектный файл (полученный на выходе компилятора, но еще не прошедший компоновку)
.pdf	Документ формата PDF
.ps	Документ формата PostScript
.tex	Входной файл для программы форматирования TEX
.txt	Обычный текстовый файл
.zip	Архив, сжатый программой zip

В некоторых системах (например, во всех разновидностях UNIX) расширения имен файлов используются в соответствии с соглашениями и не навязываются операционной системой. Файл file.txt может быть текстовым файлом, но это скорее напоминание его владельцу, чем передача некой значимой информации компьютеру. В то же время компилятор языка C может выдвигать требование, чтобы компилируемые им файлы имели расширение .c, и отказываться выполнять компиляцию, если они не имеют такого расширения. Но операционную систему это не волнует.

Подобные соглашения особенно полезны, когда одна и та же программа должна управлять различными типами файлов. Например, компилятору языка C может быть предоставлен список файлов, которые он должен откомпилировать и скомпоновать, причем некоторые из этих файлов могут содержать программы на языке C, а другие — являться ассемблерными файлами. В таком случае компилятор сможет отличить одни файлы от других именно по их расширениям.

Система Windows, напротив, знает о расширениях имен файлов и присваивает каждому расширению вполне определенное значение. Пользователи (или процессы) могут регистрировать расширения в операционной системе, указывая программу, которая станет их «владельцем». При двойном щелчке мыши на имени файла запускается программа, назначенная этому расширению, с именем файла в качестве параметра. Например, двойной щелчок мыши на имени file.docx запускает Microsoft Word, который открывает файл file.docx в качестве исходного файла для редактирования.

4.1.2. Структура файла

Файлы могут быть структурированы несколькими различными способами. Три наиболее вероятные структуры показаны на рис. 4.1. Файл на рис. 4.1, *а* представляет собой бессистемную последовательность байтов. В сущности, операционной системе все равно, что содержится в этом файле, — она видит только байты. Какое-либо значение этим байтам придают программы на уровне пользователя. Такой подход используется как в UNIX, так и в Windows.

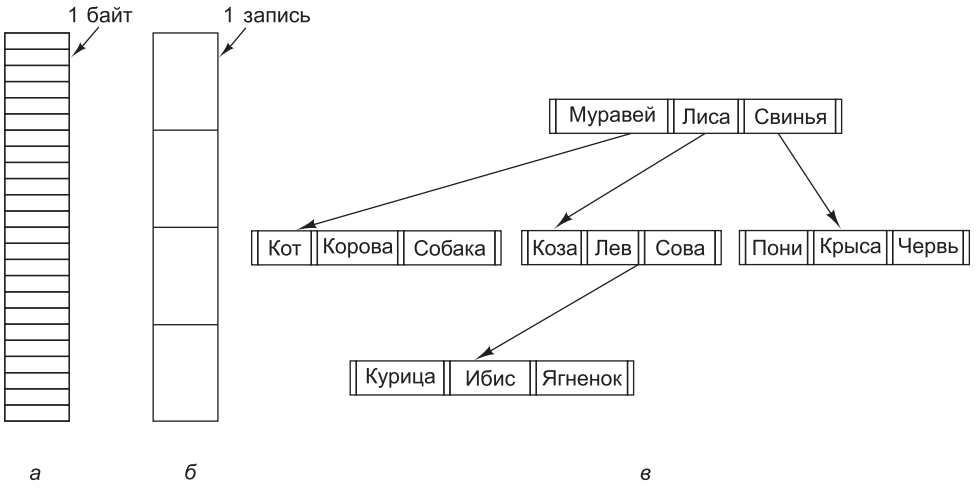

Рис. 4.1. Три типа файлов: *а* — последовательность байтов; *б* — последовательность записей; *в* — дерево

Когда операционная система считает, что файлы — это не более чем последовательность байтов, она предоставляет максимум гибкости. Программы пользователя могут помещать в свои файлы все, что им заблагорассудится, и называть их, как им удобно. Операционная система ничем при этом не помогает, но и ничем не мешает. Последнее обстоятельство может иметь особое значение для тех пользователей, которые хотят сделать что-либо необычное. Эта файловая модель используется всеми версиями UNIX (включая Linux и OS X) и Windows.

Первый шаг навстречу некой структуре показан на рис. 4.1, *б*. В данной модели файл представляет собой последовательность записей фиксированной длины, каждая из которых имеет собственную внутреннюю структуру. Основная идея файла как последовательности записей состоит в том, что операция чтения возвращает одну из записей, а операция записи перезаписывает или дополняет одну из записей. В каче-

стве исторического отступления заметим, что несколько десятилетий назад, когда в компьютерном мире властвовали перфокарты на 80 столбцов, многие операционные системы универсальных машин в основе своей файловой системы использовали файлы, состоящие из 80-символьных записей, — в сущности, образы перфокарт. Эти операционные системы поддерживали также файлы, состоящие из 132-символьных записей, предназначавшиеся для строковых принтеров (которые в то время представляли собой большие печатающие устройства, имеющие 132 столбца). Программы на входе читали блоки по 80 символов, а на выходе записывали блоки по 132 символа, даже если заключительные 52 символа были пробелами. Ни одна современная универсальная система больше не использует эту модель в качестве своей первичной файловой системы, но, возвращаясь к временам 80-столбцовых перфокарт и 132-символьной принтерной бумаги, следует отметить, что это была весьма распространенная модель для универсальных компьютеров.

Третья разновидность структуры файла показана на рис. 4.2, *в*. При такой организации файл состоит из дерева записей, необязательно одинаковой длины, каждая из которых в конкретной позиции содержит **ключевое** поле. Дерево сортируется по ключевому полю, позволяя выполнять ускоренный поиск по конкретному ключу.

Здесь основной операцией является не получение «следующей» записи, хотя возможно проведение и этой операции, а получение записи с указанным ключом. Для файла зоопарк (см. рис. 4.1, *в*) можно, к примеру, запросить систему выдать запись с ключом *пони*, нисколько не заботясь о ее конкретной позиции в файле. Более того, к файлу могут быть добавлены новые записи, и решение о том, куда их поместить, будет принимать не пользователь, а операционная система. Совершенно ясно, что этот тип файла отличается от бессистемных битовых потоков, используемых в UNIX и Windows, и он используется в некоторых больших универсальных компьютерах, применяемых при обработке коммерческих данных.

4.1.3. Типы файлов

Многие операционные системы поддерживают несколько типов файлов. К примеру, в системах UNIX (опять же включая OS X) и Windows имеются обычные файлы и каталоги. В системе UNIX имеются также символьные и блочные специальные файлы. **Обычными** считаются файлы, содержащие информацию пользователя. Все файлы на рис. 4.1 являются обычными. **Каталоги** — это системные файлы, предназначенные для поддержки структуры файловой системы. Мы рассмотрим их чуть позже. **Символьные специальные файлы** имеют отношение к вводу-выводу и используются для моделирования последовательных устройств ввода-вывода, к которым относятся терминалы, принтеры и сети. **Блочные специальные файлы** используются для моделирования дисков. В данной главе нас в первую очередь будут интересовать обычные файлы.

Как правило, к обычным файлам относятся либо файлы ASCII, либо двоичные файлы. ASCII-файлы состоят из текстовых строк. В некоторых системах каждая строка завершается символом возврата каретки. В других системах используется символ перевода строки. Некоторые системы (например, Windows) используют оба символа. Строки не обязательно должны иметь одинаковую длину.

Большим преимуществом ASCII-файлов является возможность их отображения и распечатки в исходном виде, также они могут быть отредактированы в любом текстовом редакторе. Более того, если большое количество программ используют ASCII-файлы для

ввода и вывода информации, это упрощает подключение выхода одной программы ко входу другой, как это делается в конвейерах оболочки. (При этом обмен данными между процессами ничуть не упрощается, но интерпретация информации, несомненно, становится проще, если для ее выражения используется стандартное соглашение вроде ASCII.)

Все остальные файлы относятся к двоичным — это означает, что они не являются ASCII-файлами. Их распечатка будет непонятным и бесполезным набором символов. Обычно у них есть некая внутренняя структура, известная использующей их программе.

Например, на рис. 4.2, *а* показан простой исполняемый двоичный файл, взятый из одной из ранних версий UNIX. Хотя с технической точки зрения этот файл представляет собой всего лишь последовательность байтов, операционная система исполнит его только в том случае, если он будет иметь допустимый формат. Файл состоит из пяти разделов: заголовка, текста, данных, битов перемещения и таблицы символов. Заголовок начинается с так называемого **магического числа**, идентифицирующего файл в качестве исполняемого (чтобы предотвратить случайное исполнение файла, не соответствующего данному формату). Затем следуют размеры различных частей файла, адрес, с которого начинается его выполнение, и ряд битов-флагов. За заголовком следуют текст программы и данные. Они загружаются в оперативную память и перемещаются с использованием битов перемещения. Таблица символов используется для отладки.

В качестве второго примера двоичного файла служит архив, также взятый из UNIX (см. рис. 4.2, *б*). Он состоит из набора откомпилированных, но не скомпонованных библиотечных процедур (модулей). Каждому модулю предшествует заголовок, сообщающий о его имени, дате создания, владельце, коде защиты и размере. Как и в исполняемом файле, заголовки модулей заполнены двоичными числами. При их распечатке на принтере будет получаться тарабарщина.

Каждая операционная система должна распознавать по крайней мере один тип файла — собственный исполняемый файл, но некоторые операционные системы распознают и другие типы файлов. Старая система TOPS-20 (для компьютера DECsystem 20) дошла даже до проверки времени создания каждого предназначенного для выполнения файла. Затем она находила исходный файл и проверяла, не был ли он изменен со времени создания исполняемого файла. Если он был изменен, она автоматически перекомпилировала исходный файл. В терминах UNIX это означает, что программа make была встроена в оболочку. Использование расширений имен файлов было обязательным, чтобы операционная система могла определить, какая двоичная программа от какого исходного файла произошла.

Столь строгая типизация файлов создает проблемы, как только пользователь делает что-нибудь неожиданное для разработчиков системы. Представьте, к примеру, систему, в которой выходные файлы программы имеют расширение .dat (файлы данных). Если пользователь пишет программу форматирования, которая считывает файл с расширением .c (программа на языке C), преобразует его (например, конвертируя в вид со стандартными отступами), а затем записывает преобразованный файл в качестве выходного, то выходной файл приобретает тип .dat. Если пользователь попытается предложить этот файл компилятору C, чтобы тот его откомпилировал, система откажет ему в этом, поскольку у имени файла неверное расширение. Попытки скопировать file.dat в file.c будут отвергнуты системой как недопустимые (чтобы уберечь пользователя от ошибок).

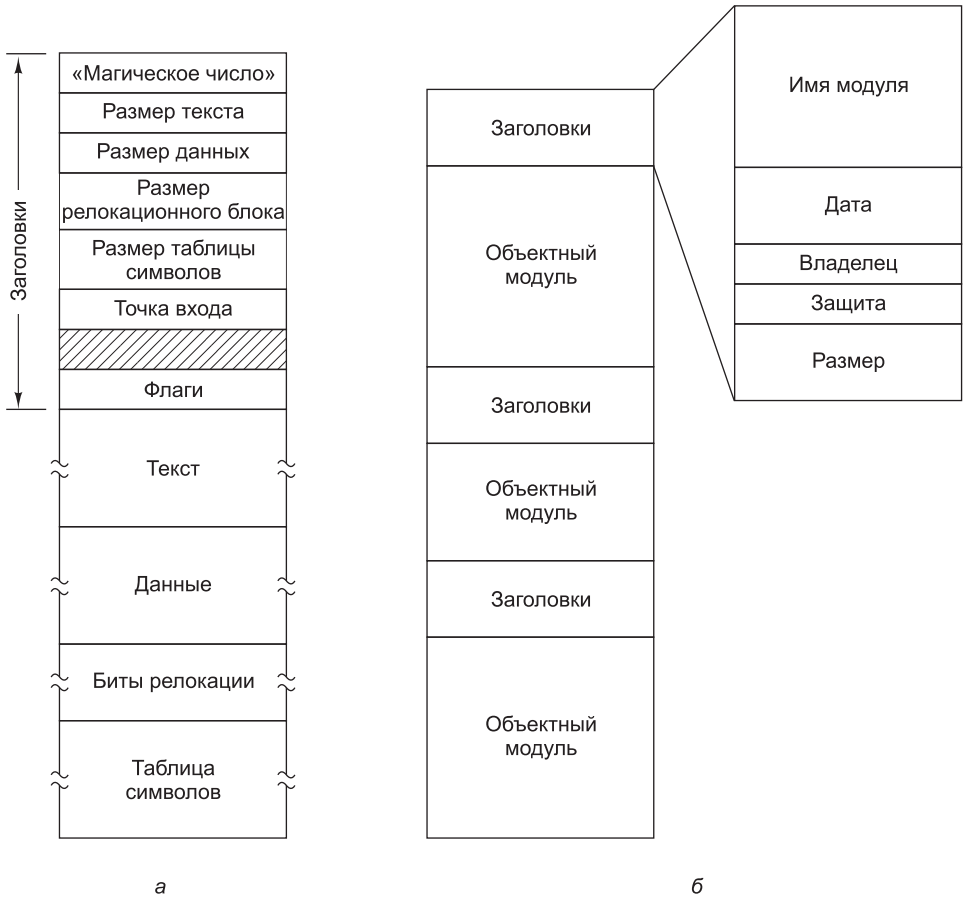

Рис. 4.2. Примеры структур двоичных файлов: *а* — исполняемый файл; *б* — архив

Хотя подобное «дружелюбие» по отношению к пользователю и может помочь новичкам, оно ставит в тупик опытных пользователей, поскольку им приходится прикладывать значительные усилия, чтобы обойти представления операционной системы о том, что приемлемо, а что нет.

4.1.4. Доступ к файлам

В самых первых операционных системах предоставлялся только один тип доступа к файлам — **последовательный**. В этих системах процесс мог читать все байты или записи файла только по порядку, с самого начала, но не мог перепрыгнуть и считать их вне порядка их следования. Но последовательные файлы можно было перемотать назад, чтобы считать их по мере надобности. Эти файлы были удобны в те времена, когда носителем в хранилищах данных служила магнитная лента, а не диск.

Когда для хранения файлов стали использоваться диски, появилась возможность считывать байты или записи файла вне порядка их размещения или получать доступ к записям по ключу, а не по позиции. Файлы, в которых байты или записи могли быть

считаны в любом порядке, стали называть **файлами произвольного доступа**. Они востребованы многими приложениями.

Файлы произвольного доступа являются неотъемлемой частью многих приложений, например систем управления базами данных. Если авиапассажир заказывает себе место на конкретный рейс, программа бронирования должна иметь возможность доступа к записи, относящейся к этому рейсу, не обременяя себя необходимостью предварительного считывания записей, относящихся к нескольким тысячам других рейсов.

Для определения места начала считывания могут быть применены два метода. При первом методе позиция в файле, с которой начинается чтение, задается при каждой операции чтения *read*. При втором методе для установки на текущую позицию предоставляется специальная операция поиска нужного места *seek*. После этой операции файл может быть считан последовательно с только что установленной позиции. Последний метод используется в UNIX и Windows.

4.1.5. Атрибуты файлов

У каждого файла есть свои имя и данные. Вдобавок к этому все операционные системы связывают с каждым файлом и другую информацию, к примеру дату и время последней модификации файла и его размер. Мы будем называть эти дополнительные сведения **атрибутами файла**. Также их называют **метаданными**. Список атрибутов существенно варьируется от системы к системе. В табл. 4.2 показаны некоторые из возможных атрибутов, но кроме них существуют и другие атрибуты. Ни одна из существующих систем не имеет всех этих атрибутов, но каждый из них присутствует в какой-либо системе.

Первые четыре атрибута относятся к защите файла и сообщают о том, кто может иметь к нему доступ, а кто нет. Возможно применение разнообразных схем, часть из них мы рассмотрим чуть позже. В некоторых системах для доступа к файлу пользователь должен ввести пароль, в этом случае пароль может быть одним из атрибутов файла.

Флаги представляют собой биты или небольшие поля, с помощью которых происходит управление некоторыми конкретными свойствами или разрешение их применения. Например, скрытые файлы не появляются в листинге файлов. Флаг архивации представляет собой бит, с помощью которого отслеживается, была ли недавно сделана резервная копия файла. Этот флаг сбрасывается программой архивирования и устанавливается операционной системой при внесении в файл изменений. Таким образом программа архивирования может определить, какие файлы следует архивировать. Флаг «временный» позволяет автоматически удалять помеченный им файл по окончании работы создавшего его процесса.

Поля длины записи, позиции ключа и длины ключа имеются только у тех файлов, записи которых можно искать по ключу. Они предоставляют информацию, необходимую для поиска ключей.

Различные показатели времени позволяют отслеживать время создания файла, последнего доступа к этому файлу, его последнего изменения. Эти сведения могут оказаться полезными для достижения различных целей. К примеру, если исходный файл был изменен уже после создания соответствующего объектного файла, то он нуждается в перекомпиляции. Необходимую для этого информацию предоставляют поля времени.

Таблица 4.2. Некоторые из возможных атрибутов

Атрибут	Значение
Защита	Кто и каким образом может получить доступ к файлу
Пароль	Пароль для получения доступа к файлу
Создатель	Идентификатор создателя файла
Владелец	Текущий владелец
Флаг «только для чтения»	0 — для чтения и записи; 1 — только для чтения
Флаг «скрытый»	0 — обычный; 1 — не предназначенный для отображения в перечне файлов
Флаг «системный»	0 — обычный; 1 — системный
Флаг «архивный»	0 — прошедший резервное копирование; 1 — нуждающийся в резервном копировании
Флаг «ASCII/двоичный»	0 — ASCII; 1 — двоичный
Флаг произвольного доступа	0 — только последовательный доступ; 1 — произвольный доступ
Флаг «временный»	0 — обычный; 1 — удаляемый по окончании работы процесса
Флаги блокировки	0 — незаблокированный; ненулевое значение — заблокированный
Длина записи	Количество байтов в записи
Позиция ключа	Смещение ключа внутри каждой записи
Длина ключа	Количество байтов в поле ключа
Время создания	Дата и время создания файла
Время последнего доступа	Дата и время последнего доступа к файлу
Время внесения последних изменений	Дата и время внесения в файл последних изменений
Текущий размер	Количество байтов в файле
Максимальный размер	Количество байтов, до которого файл может увеличиваться

Текущий размер показывает, насколько большим является файл в настоящее время. Некоторые старые операционные системы универсальных машин требуют при создании файла указывать его максимальный размер, чтобы позволить операционной системе заранее выделить максимальное место для его хранения. Операционные системы рабочих станций и персональных компьютеров достаточно разумны, чтобы обойтись без этой особенности.

4.1.6. Операции с файлами

Файлы предназначены для хранения информации с возможностью ее последующего извлечения. Разные системы предоставляют различные операции, позволяющие со-

хранять и извлекать информацию. Далее рассматриваются наиболее распространенные системные вызовы, относящиеся к работе с файлами.

- *Create* (Создать). Создает файл без данных. Цель вызова состоит в объявлении о появлении нового файла и установке ряда атрибутов.
- *Delete* (Удалить). Когда файл больше не нужен, его нужно удалить, чтобы освободить дисковое пространство. Именно для этого и предназначен этот системный вызов.
- *Open* (Открыть). Перед использованием файла процесс должен его открыть. Цель системного вызова *open* — дать возможность системе извлечь и поместить в оперативную память атрибуты и перечень адресов на диске, чтобы ускорить доступ к ним при последующих вызовах.
- *Close* (Закрыть). После завершения всех обращений к файлу потребность в его атрибутах и адресах на диске уже отпадает, поэтому файл должен быть закрыт, чтобы освободить место во внутренней таблице. Многие системы устанавливают максимальное количество открытых процессами файлов, определяя смысл существования этого вызова. Информация на диск пишется блоками, и закрытие файла вынуждает к записи последнего блока файла, даже если этот блок и не заполнен.
- *Read* (Произвести чтение). Считывание данных из файла. Как правило, байты поступают с текущей позиции. Вызывающий процесс должен указать объем необходимых данных и предоставить буфер для их размещения.
- *Write* (Произвести запись). Запись данных в файл, как правило, с текущей позиции. Если эта позиция находится в конце файла, то его размер увеличивается. Если текущая позиция находится где-то в середине файла, то новые данные пишутся поверх существующих, которые утрачиваются навсегда.
- *Append* (Добавить). Этот вызов является усеченной формой системного вызова *write*. Он может лишь добавить данные в конец файла. Как правило, у систем, предоставляющих минимальный набор системных вызовов, вызов *append* отсутствует, но многие системы предоставляют множество способов получения того же результата, и иногда в этих системах присутствует вызов *append*.
- *Seek* (Найти). При работе с файлами произвольного доступа нужен способ указания места, с которого берутся данные. Одним из общепринятых подходов является применение системного вызова *seek*, который перемещает указатель файла к определенной позиции в файле. После завершения этого вызова данные могут считываться или записываться с этой позиции.
- *Get attributes* (Получить атрибуты). Процессу для работы зачастую необходимо считать атрибуты файла. К примеру, имеющаяся в UNIX программа make обычно используется для управления проектами разработки программного обеспечения, состоящими из множества сходных файлов. При вызове программа make проверяет время внесения последних изменений всех исходных и объектных файлов и для обновления проекта обходится компиляцией лишь минимально необходимого количества файлов. Для этого ей необходимо просмотреть атрибуты файлов, а именно время внесения последних изменений.
- *Set attributes* (Установить атрибуты). Значения некоторых атрибутов могут устанавливаться пользователем и изменяться после того, как файл был создан. Такую возможность дает именно этот системный вызов. Характерным примером может

послужить информация о режиме защиты. Под эту же категорию подпадает большинство флагов.

- *Rename* (Переименовать). Нередко пользователю требуется изменить имя существующего файла. Этот системный вызов помогает решить эту задачу. Необходимость в нем возникает не всегда, поскольку файл может быть просто скопирован в новый файл с новым именем, а старый файл затем может быть удален.

4.1.7. Пример программы, использующей файловые системные вызовы

В этом разделе будет рассмотрена простая UNIX-программа, копирующая один файл из файла-источника в файл-приемник. Текст программы показан в листинге 4.1. У этой программы минимальные функциональные возможности и очень скромные возможности сообщения об ошибках, но она дает довольно четкое предоставление о некоторых системных вызовах, относящихся к работе с файлами.

Листинг 4.1. Простая программа копирования файла

```
/* Программа копирования файла. Контроль ошибок и сообщения об их возникновении
сведены к минимуму. */

#include <sys/types.h>             /* включение необходимых заголовочных файлов
*/
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);  /* ANSI-прототип */

#define BUF_SIZE 4096              /* используется буфер размером
                                   4096 байт */
#define OUTPUT_MODE 0700           /* биты защиты для выходного файла */
int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);        /* если argc не равен 3, возникает
                                   синтаксическая ошибка */

    /* Открытие входного и создание выходного файла */
    in_fd = open(argv[1], O_RDONLY);        /* открытие исходного файла */
    if (in_fd < 0) exit(2);                 /* если он не открывается, выйти */
    out_fd = creat(argv[2], OUTPUT_MODE);   /* создание файла-приемника */
    if (out_fd < 0) exit(3);                /* если он не создается, выйти */

    /* Цикл копирования */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* чтение блока данных */
    if (rd_count <= 0) break;      /* в конце файла или при ошибке – выйти из
                                   цикла */
        wt_count = write(out_fd, buffer, rd_count); /* запись данных */
```

```
        if (wt_count <= 0) exit(4); /* при wt_count <= 0 возникает ошибка */
    }

    /* Закрытие файлов */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)            /* при последнем чтении ошибки не возникло */
        exit(0);
    else
        exit(5);                  /* ошибка при последнем чтении */
}
```

Эта программа с именем copyfile может быть вызвана, к примеру, из командной строки

```
copyfile abc xyz
```

чтобы скопировать файл abc в файл xyz. Если файл xyz уже существует, то он будет переписан. Если этого файла не существует, то он будет создан. Программа должна быть вызвана с обязательным указанием двух аргументов, являющихся допустимыми именами файлов. Первый аргумент должен быть именем файла-источника, а второй — именем выходного файла.

Благодаря четырем операторам *#include* в самом начале программы в нее включается большое количество определений и прототипов функций. Это нужно для совместимости программы с соответствующими международными стандартами, но больше это нас интересовать не будет. Следующая строка, согласно требованию стандарта ANSI C, содержит прототип функции *main*, но сейчас это нас тоже не интересует.

Первый оператор *#define* является макроопределением строки *BUF_SIZE* как макроса, который при компиляции заменяется в тексте программы числом 4096. Программа будет считывать и записывать данные блоками по 4096 байт. Создание таких констант и использование их вместо непосредственного указания чисел в программе считается хорошим стилем программирования. При этом программу не только удобнее читать, но и проще изменять в случае необходимости. Второй оператор *#define* определяет круг пользователей, которые могут получить доступ к выходному файлу.

У основной программы, которая называется main, имеется два аргумента — *argc* и *argv*. Значения этим аргументам присваиваются операционной системой при вызове программы. В первом аргументе указывается количество строковых значений, присутствующих в командной строке, вызывающей программу, включая имя самой программы. Его значение должно быть равно 3. Второй аргумент представляет собой массив указателей на аргументы командной строки. В примере, приведенном ранее, элементы этого массива будут содержать указатели на следующие значения:

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

Через этот массив программа получает доступ к своим аргументам.

В программе объявляются пять переменных. В первых двух переменных, *in_ fd* и *out_ fd*, будут храниться **дескрипторы файлов** — небольшие целые числа, возвращаемые при открытии файла. Следующие две переменные, *rd_count* и *wt_count*, являются байтовыми счетчиками, возвращаемыми процедурами *read* и *write* соответственно. Последняя переменная, *buffer*, используется для хранения считанных и предоставления записываемых данных.

Первый исполняемый оператор проверяет, не равно ли значение счетчика аргументов *argc* 3. Если счетчик *argc* не равен 3, программа завершается с кодом 1. Любой код завершения программы отличный от 0 означает, что произошла ошибка. Единственный применяемый в этой программе способ сообщения об ошибках — это код завершения программы. Окончательный вариант этой программы выводил бы сообщения об ошибках.

Затем программа пытается открыть входной файл и создать выходной файл. Если открытие файла проходит успешно, операционная система присваивает переменной *in_fd* небольшое целочисленное значение, чтобы идентифицировать файл. Это целое число может включаться в последующие вызовы, чтобы система знала, какой файл им нужен. Аналогично этому, если успешно создается выходной файл, переменной *out_fd* также присваивается идентифицирующее его значение. Второй аргумент процедуры *creat* устанавливает код защиты создаваемого файла. Если не удается открыть файл или создать его, то значение соответствующего дескриптора файла устанавливается в –1 и происходит выход из программы с соответствующим кодом ошибки.

Затем наступает черед цикла копирования, который начинается с попытки считать в буфер *buffer* 4 Кбайт данных. Это делается путем вызова библиотечной процедуры *read*, которая на самом деле осуществляет системный вызов *read*. Первый параметр идентифицирует файл, второй указывает буфер, а третий сообщает, сколько байтов нужно считать. Значение, присвоенное *rd_count*, дает количество реально считанных байтов. Обычно это значение равно 4096, за исключением того случая, когда в файле останется меньше байтов. При достижении конца файла это значение будет равно 0. Как только значение *rd_count* станет нулевым или отрицательным, копирование не сможет продолжаться, поэтому для прекращения цикла (который в противном случае был бы бесконечным) выполняется оператор *break*.

Обращение к процедуре *write* приводит к выводу содержимого буфера в выходной файл. Первый параметр идентифицирует файл, второй указывает буфер, а третий, аналогично параметрам процедуры *read*, сообщает, сколько байтов нужно записать. Следует заметить, что счетчик байтов содержит количество реально считанных байтов, а не значение *BUF_SIZE*. Это важно, поскольку при последнем считывании не будет возвращено число 4096, если только длина файла не окажется кратной 4 Кбайт.

После обработки всего файла при первом же вызове, выходящем за пределы файла, в *rd_count* будет возвращено значение 0, которое и заставит программу выйти из цикла. После этого оба файла закрываются и происходит выход из программы со статусом, свидетельствующем о ее нормальном завершении.

Несмотря на то что системные вызовы Windows отличаются от системных вызовов UNIX, общая структура программы Windows, предназначенной для копирования файлов и запускаемой из командной строки, примерно такая же, как у программы, показанной в листинге 4.1. Системные вызовы Windows 8 будут рассмотрены в главе 11.

4.2. Каталоги

Обычно в файловой системе для упорядочения файлов имеются **каталоги** или **папки**, которые сами по себе являются файлами. В этом разделе будут рассмотрены каталоги, их организация, их свойства и операции, которые к ним могут применяться.

4.2.1. Системы с одноуровневыми каталогами

Самая простая форма системы каталогов состоит из одного каталога, содержащего все файлы. Иногда он называется **корневым каталогом**, но поскольку он один-единственный, то имя особого значения не имеет. Эта система была широко распространена на первых персональных компьютерах отчасти из-за того, что у них был всего один пользователь. Как ни странно, первый в мире суперкомпьютер, CDC 6600, также имел один каталог для всех файлов, даже притом, что на нем одновременно работало много пользователей. Несомненно, это решение было принято с целью упростить разработку программного обеспечения.

Пример системы, имеющей всего один каталог, показан на рис. 4.3. На нем изображен каталог, в котором содержатся четыре файла. Преимущества такой схемы заключаются в ее простоте и возможности быстрого нахождения файлов, поскольку поиск ведется всего в одном месте. Такая система иногда все еще используется в простых встроенных устройствах — цифровых камерах и некоторых переносных музыкальных плеерах.

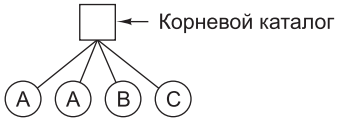

Рис. 4.3. Система с одноуровневым каталогом, содержащим четыре файла

4.2.2. Иерархические системы каталогов

Одноуровневая система больше подходит для очень простых специализированных приложений (и применялась даже на первых персональных компьютерах), но современным пользователям, работающим с тысячами файлов, найти что-нибудь, если все файлы находятся в одном каталоге, будет практически невозможно.

Поэтому нужен способ, позволяющий сгруппировать родственные файлы. К примеру, у профессора может быть коллекция файлов, составляющая книгу, которую он написал для одного учебного курса, вторая коллекция файлов, в которой содержатся студенческие программы, сданные при изучении другого курса, третья группа файлов, в которых содержится создаваемая им инновационная система компиляции, четвертая группа файлов, в которых содержатся предложения по грантам, а также другие файлы, используемые для электронной почты, протоколов совещаний, неоконченных статей, игр и т. д.

Для организации всего этого нужна иерархия (то есть дерево каталогов). Такой подход позволяет иметь столько каталогов, сколько необходимо для группировки файлов естественным образом. Кроме того, если общий файловый сервер совместно используется несколькими пользователями, как это бывает во многих сетях организаций, каждый пользователь может иметь корневой каталог для собственной иерархии. Именно этот подход показан на рис. 4.4. На нем изображены каталоги *A*, *B* и *C*, которые содержатся в корневом каталоге и принадлежат разным пользователям, двое из которых создали подкаталоги для проектов, над которыми работают.

Возможность создания произвольного количества подкаталогов предоставляет пользователям мощный инструмент структуризации, позволяющий организовать их работу. Поэтому таким образом устроены практически все современные файловые системы.

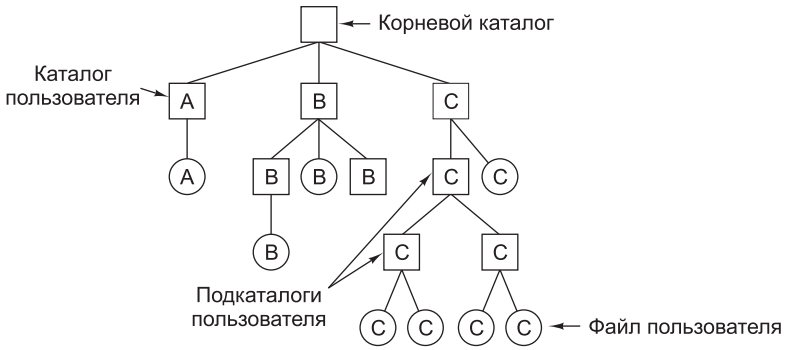

Рис. 4.4. Иерархическая система каталогов

4.2.3. Имена файлов

Когда файловая система организована в виде дерева каталогов, нужен какой-нибудь способ указания имен файлов. Чаще всего для этого используются два метода. В первом методе каждому файлу дается **абсолютное имя (полное имя)**, состоящее из пути от корневого каталога к файлу. Например, имя /usr/ast/mailbox означает, что корневой каталог содержит подкаталог usr, который, в свою очередь, содержит подкаталог ast, в котором содержится файл mailbox. Абсолютные имена файлов всегда начинаются с названия корневого каталога и являются уникальными именами. В системе UNIX компоненты пути разделяются символом «слеш» — /. В системе Windows разделителем служит символ «обратный слеш» — \. В системе MULTICS этим разделителем служила угловая скобка — >. В этих трех системах одно и то же имя будет выглядеть следующим образом:

```
Windows \usr\ast\mailbox
UNIX /usr/ast/mailbox
MULTICS >usr>ast>mailbox
```

Если в качестве первого символа в имени файла используется разделитель, то независимо от символа, используемого в этом качестве, путь будет абсолютным.

Другой разновидностью имени является **относительное имя**. Оно используется совместно с понятием **рабочего каталога** (называемого также **текущим каталогом**). Пользователь может определить один каталог в качестве текущего, и тогда все имена файлов станут рассматриваться относительно рабочего каталога и не будут начинаться с корневого каталога. К примеру, если текущим рабочим каталогом будет /usr/ast, то к файлу, имеющему абсолютное имя /usr/ast/mailbox, можно будет обращаться, просто указывая mailbox. Иначе говоря, команда UNIX

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

и команда

```
cp mailbox mailbox.bak
```

делают одно и то же, если рабочим каталогом является /usr/ast. Относительная форма указания имен зачастую более удобна, но при этом делает то же самое, что и абсолютная форма.

Некоторым программам нужен доступ к конкретному файлу безотносительно того, какой каталог является рабочим. В таком случае им всегда нужно использовать абсолютные имена. К примеру, программе проверки правописания в процессе работы может понадобиться чтение файла /usr/lib/dictionary. В таком случае ей следует использовать полное, абсолютное имя, поскольку она не знает, какой каталог будет при ее вызове рабочим. Абсолютное имя файла будет работать всегда, независимо от того, какой именно каталог будет рабочим.

Разумеется, если программа проверки правописания нуждается в большом количестве файлов из каталога /usr/lib, то альтернативным подходом будет следующий: использовать системный вызов для смены рабочего каталога на /usr/lib, а затем в качестве первого параметра системного вызова *open* можно будет использовать лишь имя dictionary. За счет явного изменения рабочего каталога программа точно знает, в каком месте дерева каталогов она работает, поэтому она может использовать относительные пути к файлам.

У каждого процесса есть свой рабочий каталог, поэтому, когда процесс меняет свой рабочий каталог и потом завершает работу, это не влияет на работу других процессов и в файловой системе от подобных изменений не остается никаких следов. Таким образом, процесс может когда угодно изменить свой рабочий каталог, абсолютно не беспокоясь о последствиях. В то же время, если *библиотечная процедура* поменяет свой рабочий каталог и при возврате управления не восстановит прежний рабочий каталог, то вызвавшая ее программа может оказаться не в состоянии продолжить работу, так как ее предположения о текущем каталоге окажутся неверными. Из-за этого библиотечные процедуры редко меняют свои рабочие каталоги, а когда им все-таки приходится это делать, они обязательно восстанавливают прежний рабочий каталог перед возвратом управления.

Большинство операционных систем, поддерживающих иерархическую систему каталогов, имеют в каждом каталоге специальные элементы «.» и «..», которые обычно произносятся как «точка» и «точка-точка». Точка является ссылкой на текущий каталог, а двойная точка — на родительский каталог (за исключением корневого каталога, где этот элемент является ссылкой на сам корневой каталог). Чтобы увидеть, как они используются, обратимся к дереву каталогов системы UNIX (рис. 4.5). Пусть у нас есть некий процесс, для которого каталог /usr/ast является рабочим. Чтобы переместиться вверх по дереву, он может использовать обозначение «...». К примеру, он может копировать файл /usr/lib/dictionary в собственный каталог при помощи команды

```
cp ../lib/dictionary .
```

Первый указанный путь предписывает системе подняться вверх по дереву (к каталогу usr), затем опуститься вниз к каталогу lib и найти в нем файл dictionary.

Второй аргумент (точка) заменяет имя текущего каталога. Когда в качестве последнего аргумента команда *cp* получает имя каталога (включая точку), она копирует все файлы в этот каталог. Разумеется, куда более привычным способом копирования будет использование полного абсолютного имени пути к файлу-источнику:

```
cp /usr/lib/dictionary .
```

Здесь использование точки избавляет пользователя от необходимости второй раз набирать имя dictionary. Тем не менее, если набрать

```
cp /usr/lib/dictionary dictionary
```

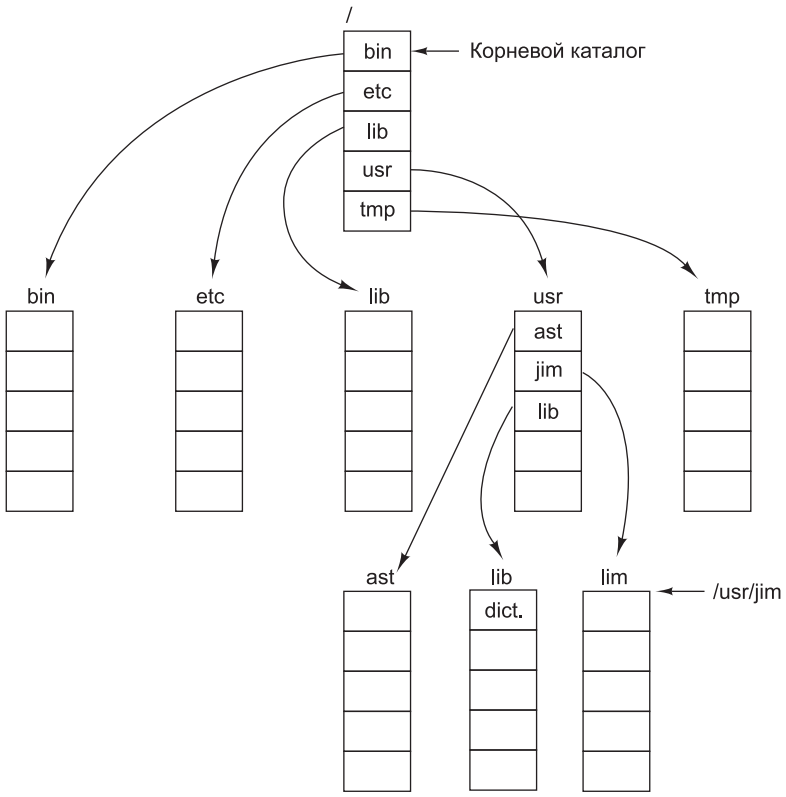

Рис. 4.5. Дерево каталогов UNIX

команда будет работать так же, как и при наборе

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

Все эти команды приводят к одному и тому же результату.

4.2.4. Операции с каталогами

Допустимые системные вызовы для управления каталогами имеют большее количество вариантов от системы к системе, чем системные вызовы, управляющие файлами. Рассмотрим примеры, дающие представление об этих системных вызовах и характере их работы (взяты из системы UNIX).

- *Create* (Создать каталог). Каталог создается пустым, за исключением точки и двойной точки, которые система помещает в него автоматически (или в некоторых случаях при помощи программы mkdir).
- *Delete* (Удалить каталог). Удалить можно только пустой каталог. Каталог, содержащий только точку и двойную точку, рассматривается как пустой, поскольку они не могут быть удалены.
- *Opendir* (Открыть каталог). Каталоги могут быть прочитаны. К примеру, для вывода имен всех файлов, содержащихся в каталоге, программа ls открывает каталог

для чтения имен всех содержащихся в нем файлов. Перед тем как каталог может быть прочитан, он должен быть открыт по аналогии с открытием и чтением файла.

- *Closedir* (Закрыть каталог). Когда каталог прочитан, он должен быть закрыт, чтобы освободить пространство во внутренних таблицах системы.
- *Readdir* (Прочитать каталог). Этот вызов возвращает следующую запись из открытого каталога. Раньше каталоги можно было читать с помощью обычного системного вызова *read*, но недостаток такого подхода заключался в том, что программист вынужден был работать с внутренней структурой каталогов, о которой он должен был знать заранее. В отличие от этого, *readdir* всегда возвращает одну запись в стандартном формате независимо от того, какая из возможных структур каталогов используется.
- *Rename* (Переименовать каталог). Во многих отношениях каталоги подобны файлам и могут быть переименованы точно так же, как и файлы.
- *Link* (Привязать). Привязка представляет собой технологию, позволяющую файлу появляться более чем в одном каталоге. В этом системном вызове указываются существующий файл и новое имя файла в некотором существующем каталоге и создается привязка существующего файла к указанному каталогу с указанным новым именем. Таким образом, один и тот же файл может появиться в нескольких каталогах, возможно, под разными именами. Подобная привязка, увеличивающая показания файлового счетчика i-узла (предназначенного для отслеживания количества записей каталогов, в которых фигурирует файл), иногда называется **жесткой связью**, или **жесткой ссылкой** (hard link).
- *Unlink* (Отвязать). Удалить запись каталога. Если отвязываемый файл присутствует только в одном каталоге (что чаще всего и бывает), то этот вызов удалит его из файловой системы. Если он фигурирует в нескольких каталогах, то он будет удален из каталога, который указан в имени файла. Все остальные записи останутся. Фактически системным вызовом для удаления файлов в UNIX (как ранее уже было рассмотрено) является *unlink*.

В приведенном списке перечислены наиболее важные вызовы, но существуют и другие вызовы, к примеру для управления защитой информации, связанной с каталогами.

Еще одним вариантом идеи привязки файлов является **символическая ссылка** (symbolic link). Вместо двух имен, указывающих на одну и ту же внутреннюю структуру данных, представляющую файл, может быть создано имя, указывающее на очень маленький файл, в котором содержится имя другого файла. Когда используется первый файл, например он открывается, файловая система идет по указанному пути и в итоге находит имя. Затем она начинает процесс поиска всех мест, где используется это новое имя. Преимуществом символических ссылок является то, что они могут пересекать границы дисков и даже указывать на имена файлов, находящихся на удаленных компьютерах. И тем не менее их реализация несколько уступает в эффективности жестким связям.

4.3. Реализация файловой системы

Настала пора перейти от пользовательского взгляда на файловую систему к взгляду специалистов на ее реализацию. Пользователей волнует, как можно назвать файлы, какие операции над ними допустимы, как выглядит дерево каталогов и другие подобные

вопросы, касающиеся взаимодействия с файловой системой. Разработчиков интересует, как хранятся файлы и каталоги, как осуществляется управление дисковым пространством и как добиться от всего этого эффективной и надежной работы. В следующих разделах будет рассмотрен ряд перечисленных вопросов, чтобы можно было понять, какие проблемы и компромиссы встречаются на этом пути.

4.3.1. Структура файловой системы

Файловые системы хранятся на дисках. Большинство дисков может быть разбито на один или несколько разделов, на каждом из которых будет независимая файловая система. Сектор 0 на диске называется **главной загрузочной записью** (Master Boot Record (**MBR**)) и используется для загрузки компьютера. В конце MBR содержится таблица разделов. Из этой таблицы берутся начальные и конечные адреса каждого раздела. Один из разделов в этой таблице помечается как активный. При загрузке компьютера BIOS (базовая система ввода-вывода) считывает и выполняет MBR. Первое, что делает программа MBR, — находит расположение активного раздела, считывает его первый блок, который называется **загрузочным**, и выполняет его. Программа в загрузочном блоке загружает операционную систему, содержащуюся в этом разделе. Для достижения единообразия каждый раздел начинается с загрузочного блока, даже если он не содержит загружаемой операционной системы. Кроме того, в будущем он может содержать какую-нибудь операционную систему.

Во всем остальном, кроме того, что раздел начинается с загрузочного блока, строение дискового раздела значительно различается от системы к системе. Зачастую файловая система будет содержать некоторые элементы, показанные на рис. 4.6. Первым элементом является **суперблок**. В нем содержатся все ключевые параметры файловой системы, которые считываются в память при загрузке компьютера или при первом обращении к файловой системе. Обычно в информацию суперблока включаются «магическое» число, позволяющее идентифицировать тип файловой системы, количество блоков в файловой системе, а также другая важная административная информация.

Далее может находиться информация о свободных блоках файловой системы, к примеру в виде битового массива или списка указателей. За ней могут следовать i-узлы, массив структур данных — на каждый файл по одной структуре, в которой содержится

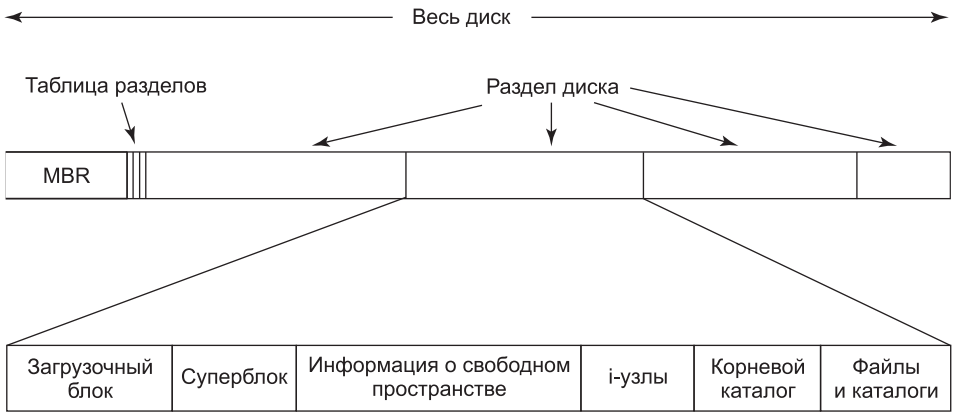

Рис. 4.6. Возможная структура файловой системы

вся информация о файле. Затем может размещаться корневой каталог, содержащий вершину дерева файловой системы. И наконец, оставшаяся часть диска содержит все остальные каталоги и файлы.

4.3.2. Реализация файлов

Возможно, самым важным вопросом при реализации файлового хранилища является отслеживание соответствия файлам блоков на диске. В различных операционных системах используются разные методы. Некоторые из них будут рассмотрены в этом разделе.

Непрерывное размещение

Простейшая схема размещения заключается в хранении каждого файла на диске в виде непрерывной последовательности блоков. Таким образом, на диске с блоками, имеющими размер 1 Кбайт, файл размером 50 Кбайт займет 50 последовательных блоков. При блоках, имеющих размер 2 Кбайт, под него будет выделено 25 последовательных блоков.

Пример хранилища с непрерывным размещением приведен на рис. 4.7, *а*. На нем показаны 40 первых блоков, начинающихся с блока 0 слева. Изначально диск был пустым. Затем на него начиная с блока 0 был записан файл *A* длиной четыре блока. Затем правее окончания файла *A* записан файл *B*, занимающий шесть блоков.

Следует заметить, что каждый файл начинается от границы нового блока, поэтому, если файл *A* фактически имел длину 3,5 блока, то в конце последнего блока часть пространства будет потеряна впустую. Всего на рисунке показаны семь файлов, каждый

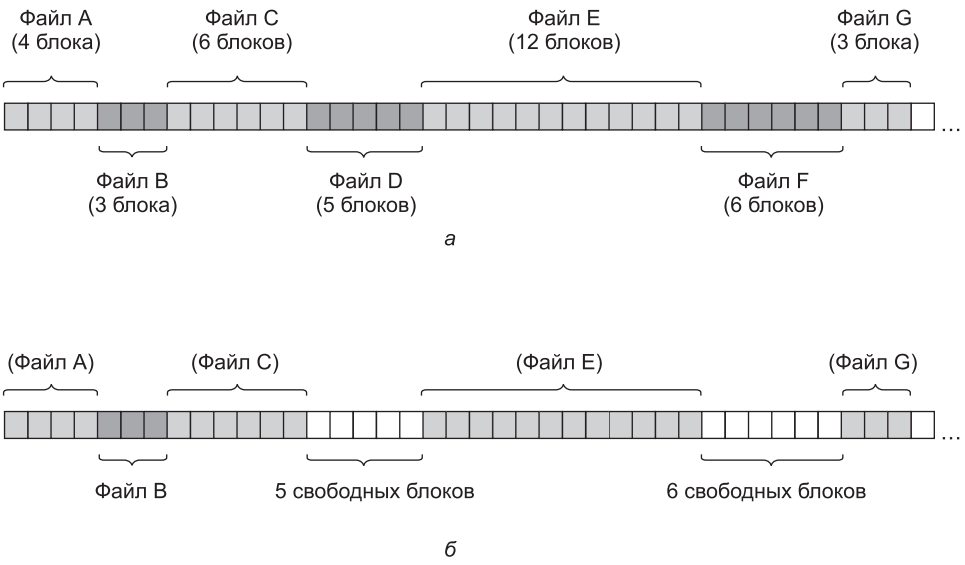

Рис. 4.7. Дисковое пространство: *а* — непрерывное размещение семи файлов; *б* — состояние диска после удаления файлов *D* и *F*

из которых начинается с блока, который следует за последним блоком предыдущего файла. Затенение использовано только для того, чтобы упростить показ деления пространства на файлы. В отношении самого хранилища оно не имеет никакого практического значения.

У непрерывного распределения дискового пространства есть два существенных преимущества. Во-первых, его просто реализовать, поскольку отслеживание местонахождения принадлежащих файлу блоков сводится всего лишь к запоминанию двух чисел: дискового адреса первого блока и количества блоков в файле. При наличии номера первого блока номер любого другого блока может быть вычислен путем простого сложения.

Во-вторых, у него превосходная производительность считывания, поскольку весь файл может быть считан с диска за одну операцию. Для нее потребуется только одна операция позиционирования (на первый блок). После этого никаких позиционирований или ожиданий подхода нужного сектора диска уже не потребуется, поэтому данные поступают на скорости, равной максимальной пропускной способности диска. Таким образом, непрерывное размещение характеризуется простотой реализации и высокой производительностью.

К сожалению, у непрерывного размещения есть также очень серьезный недостаток: со временем диск становится фрагментированным. Как это происходит, показано на рис. 4.7, *б*. Были удалены два файла — *D* и *F*. Естественно, при удалении файла его блоки освобождаются и на диске остается последовательность свободных блоков. Немедленное уплотнение файлов на диске для устранения такой последовательности свободных блоков («дыры») не осуществляется, поскольку для этого потребуется скопировать все блоки, — а их могут быть миллионы, — следующие за ней, что при использовании больших дисков займет несколько часов или даже дней. В результате, как показано на рис. 4.7, *б*, диск содержит вперемешку файлы и последовательности свободных блоков.

Сначала фрагментация не составляет проблемы, поскольку каждый новый файл может быть записан в конец диска, следуя за предыдущим файлом. Но со временем диск заполнится и понадобится либо его уплотнить, что является слишком затратной операцией, либо повторно использовать последовательности свободных блоков между файлами, для чего потребуется вести список таких свободных участков, что вполне возможно осуществить. Но при создании нового файла необходимо знать его окончательный размер, чтобы выбрать подходящий для размещения участок.

Представьте себе последствия использования такого подхода. Пользователь запускает текстовый процессор, чтобы создать документ. В первую очередь программа спрашивает, сколько байтов в конечном итоге будет занимать документ. Без ответа на этот вопрос она не сможет продолжить работу. Если в конце выяснится, что указан слишком маленький размер, программа будет вынуждена преждевременно прекратить свою работу, поскольку выбранная область на диске будет заполнена и остаток файла туда просто не поместится. Если пользователь попытается обойти эту проблему, задавая заведомо большой конечный объем, скажем, 1 Гбайт, редактор может и не найти столь большой свободной области и объявит, что файл создать нельзя. Разумеется, ничто не помешает пользователю запустить программу повторно и задать на сей раз 500 Мбайт и т. д., пока не будет найдена подходящая свободная область. Но такая система вряд ли осчастливит пользователей.

Тем не менее есть одна сфера применения, в которой непрерывное размещение вполне приемлемо и все еще используется на практике, — это компакт-диски. Здесь все размеры файлов известны заранее и никогда не изменяются в процессе дальнейшего использования файловой системы компакт-диска.

С DVD ситуация складывается несколько сложнее. В принципе, 90-минутный фильм может быть закодирован в виде одного-единственного файла длиной около 4,5 Гбайт, но в используемой файловой системе **UDF** (Universal Disk Format — универсальный формат диска) для представления длины файла применяется 30-разрядное число, которое ограничивает длину файлов одним гигабайтом. Вследствие этого DVD-фильмы, как правило, хранятся в виде трех-четырех файлов размером 1 Гбайт, каждый из которых является непрерывным. Такие физические части одного логического файла (фильма) называются **экстентами**.

Как говорилось в главе 1, в вычислительной технике при появлении технологии нового поколения история часто повторяется. Непрерывное размещение благодаря своей простоте и высокой производительности использовалось в файловых системах магнитных дисков много лет назад (удобство для пользователей в то время еще не было в цене). Затем из-за необходимости задания конечного размера файла при его создании эта идея была отброшена. Но неожиданно с появлением компакт-дисков, DVD, Blu-ray-дисков и других однократно записываемых оптических носителей непрерывные файлы снова оказались весьма кстати. Поэтому столь важное значение имеет изучение старых систем и идей, обладающих концептуальной ясностью и простотой, поскольку они могут пригодиться для будущих систем совершенно неожиданным образом.

Размещение с использованием связанного списка

Второй метод хранения файлов заключается в представлении каждого файла в виде связанного списка дисковых блоков (рис. 4.8). Первое слово каждого блока используется в качестве указателя на следующий блок, а вся остальная часть блока предназначается для хранения данных.

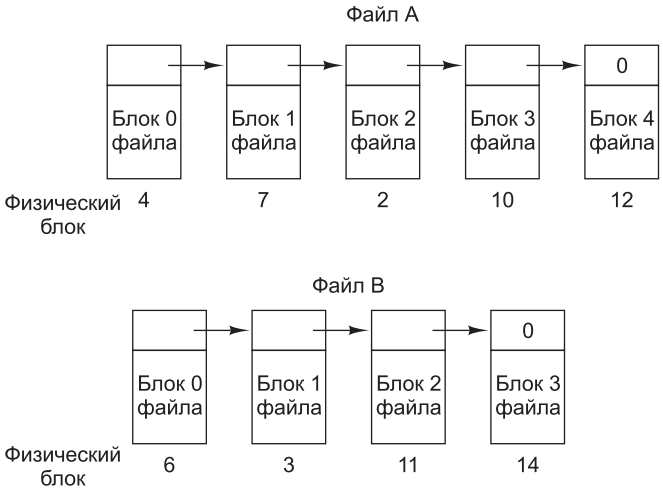

Рис. 4.8. Хранение файла в виде связанного списка дисковых блоков

В отличие от непрерывного размещения, в этом методе может быть использован каждый дисковый блок. При этом потери дискового пространства на фрагментацию отсутствуют (за исключением внутренней фрагментации в последнем блоке). Кроме того, достаточно, чтобы в записи каталога хранился только дисковый адрес первого блока. Всю остальную информацию можно найти начиная с этого блока.

В то же время по сравнению с простотой последовательного чтения файла произвольный доступ является слишком медленным. Чтобы добраться до блока *n*, операционной системе нужно начать со стартовой позиции и прочитать поочередно $n - 1$ предшествующих блоков. Понятно, что осуществление стольких операций чтения окажется мучительно медленным.

К тому же объем хранилища данных в блоках уже не кратен степени числа 2, поскольку несколько байтов отнимает указатель. Хотя это и не смертельно, но необычный размер менее эффективен, поскольку многие программы ведут чтение и запись блоками, размер которых кратен степени числа 2. Когда первые несколько байтов каждого блока заняты указателем на следующий блок, чтение полноценного блока требует получения и соединения информации из двух дисковых блоков, из-за чего возникают дополнительные издержки при копировании.

Размещение с помощью связанного списка, использующего таблицу в памяти

Оба недостатка размещения с помощью связанных списков могут быть устранены за счет изъятия слова указателя из каждого дискового блока и помещения его в таблицу в памяти. На рис. 4.9 показано, как выглядит таблица для примера, приведенного на рис. 4.8. На обоих рисунках показаны два файла. Файл *A* использует в указанном порядке дисковые блоки 4, 7, 2, 10 и 12, а файл *B* — блоки 6, 3, 11 и 14. Используя таблицу, показанную на рис. 4.9, можно пройти всю цепочку от начального блока 4 до самого конца. То же самое можно проделать начиная с блока 6. Обе цепочки заканчиваются специальным маркером (например, –1), который не является допустимым номером блока. Такая таблица, находящаяся в оперативной памяти, называется **FAT** (File Allocation Table — таблица размещения файлов).

При использовании такой организации для данных доступен весь блок. Кроме того, намного упрощается произвольный доступ. Хотя для поиска заданного смещения в файле по-прежнему нужно идти по цепочке, эта цепочка целиком находится в памяти, поэтому проход по ней может осуществляться без обращений к диску. Как и в предыдущем методе, в записи каталога достаточно хранить одно целое число (номер начального блока) и по-прежнему получать возможность определения местоположения всех блоков независимо от того, насколько большим будет размер файла.

Основным недостатком этого метода является то, что для его работы вся таблица должна постоянно находиться в памяти. Для 1-терабайтного диска, имеющего блоки размером 1 Кбайт, потребовалась бы таблица из 1 млрд записей, по одной для каждого из 1 млрд дисковых блоков. Каждая запись должна состоять как минимум из 3 байт. Для ускорения поиска размер записей должен быть увеличен до 4 байт. Таким образом, таблица будет постоянно занимать 3 Гбайт или 2,4 Гбайт оперативной памяти в зависимости от того, как оптимизирована система, под экономию пространства или под экономию времени, что с практической точки зрения выглядит не слишком привлекательно. Становится очевидным, что идея FAT плохо масштабируется на диски

Рис. 4.9. Размещение с помощью связанного списка, использующего таблицу размещения файлов в оперативной памяти

больших размеров. Изначально это была файловая система MS-DOS, но она до сих пор полностью поддерживается всеми версиями Windows.

I-узлы

Последним из рассматриваемых методов отслеживания принадлежности конкретного блока конкретному файлу является связь с каждым файлом структуры данных, называемой **i-узлом (index-node** — индекс-узел), содержащей атрибуты файла и дисковые адреса его блоков. Простой пример приведен на рис. 4.10. При использовании i-узла появляется возможность найти все блоки файла. Большим преимуществом этой схемы перед связанными списками, использующими таблицу в памяти, является то, что i-узел должен быть в памяти только в том случае, когда открыт соответствующий файл. Если каждый i-узел занимает n байт, а одновременно может быть открыто максимум k файлов, общий объем памяти, занимаемой массивом, хранящим i-узлы открытых файлов, составляет всего лишь kn байт. Заранее нужно будет зарезервировать только этот объем памяти.

Обычно этот массив значительно меньше того пространства, которое занимает таблица расположения файлов, рассмотренная в предыдущем разделе. Причина проста. Таблица, предназначенная для хранения списка всех дисковых блоков, пропорциональна размеру самого диска. Если диск имеет n блоков, то таблице нужно n записей. Она растет пропорционально росту размера диска. В отличие от этого, для схемы, использующей i-узлы, нужен массив в памяти, чей размер пропорционален максимальному количеству одновременно открытых файлов. При этом неважно, будет ли размер диска 100, 1000 или 10 000 Гбайт.

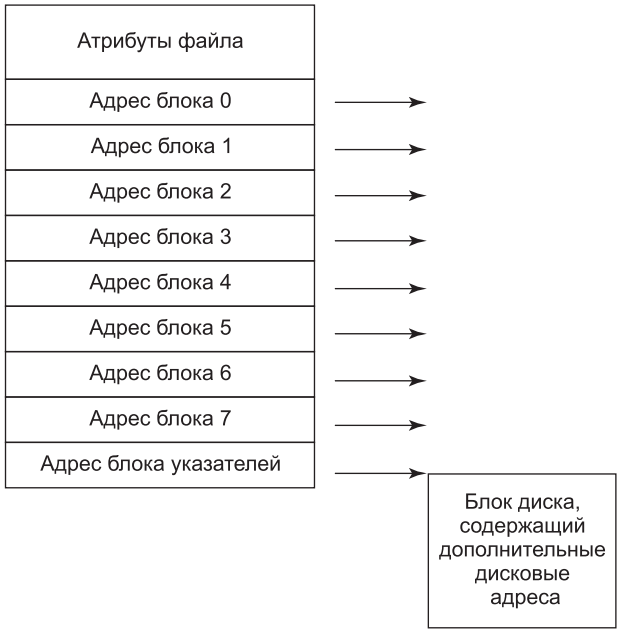

Рис. 4.10. Пример i-узла

С i-узлами связана одна проблема: если каждый узел имеет пространство для фиксированного количества дисковых адресов, то что произойдет, когда файл перерастет этот лимит? Одно из решений заключается в резервировании последнего дискового адреса не для блока данных, а для блока, содержащего дополнительные адреса блоков (см. рис. 4.10). Более того, можно создавать целые цепочки или даже деревья адресных блоков, поскольку их может понадобиться два или более. Может потребоваться даже дисковый блок, указывающий на другие, полные адресов дисковые блоки. Мы еще вернемся к i-узлам при изучении системы UNIX в главе 10. По аналогии с этим в файловой системе Windows NTFS используется такая же идея, но только с более крупными i-узлами, в которых также могут содержаться небольшие файлы.

4.3.3. Реализация каталогов

Перед тем как прочитать файл, его нужно открыть. При открытии файла операционная система использует предоставленное пользователем имя файла для определения местоположения соответствующей ему записи каталога на диске. Эта запись предоставляет информацию, необходимую для поиска на диске блоков, занятых данным файлом. В зависимости от применяемой системы эта информация может быть дисковым адресом всего файла (с непрерывным размещением), номером первого блока (для обеих схем, использующих связанные списки) или номером i-узла. Во всех случаях основной функцией системы каталогов является преобразование ASCII-имени файла в информацию, необходимую для определения местоположения данных.

Со всем этим тесно связан вопрос: где следует хранить атрибуты? Каждая файловая система работает с различными атрибутами файлов, такими как имя владельца файла

и время создания, и их нужно где-то хранить. Одна из очевидных возможностей заключается в хранении их непосредственно в записи каталога. Именно так некоторые системы и делают. Этот вариант показан на рис. 4.11, *а*. В этой простой конструкции каталог состоит из списка записей фиксированного размера, по одной записи на каждый файл, в которой содержатся имя файла (фиксированной длины), структура атрибутов файла, а также один или несколько дисковых адресов (вплоть до некоторого максимума), сообщающих, где находятся соответствующие файлу блоки на диске.

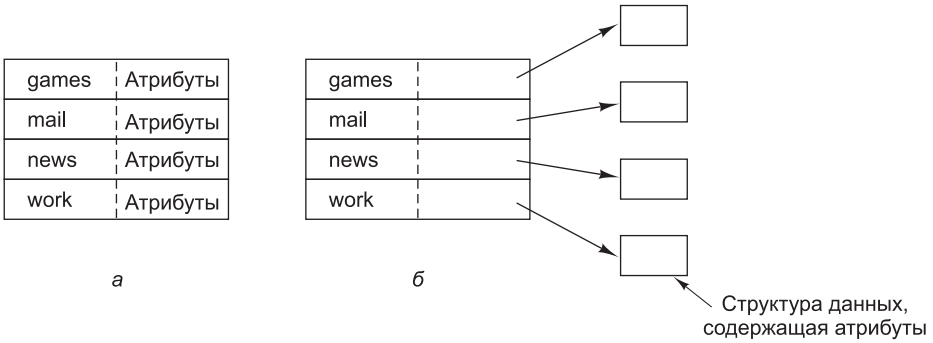

Рис. 4.11. Каталог: *а* — содержит записи фиксированного размера с дисковыми адресами и атрибутами; *б* — каждая запись всего лишь ссылается на i-узел

Для систем, использующих i-узлы, имеется возможность хранить атрибуты в самих i-узлах. При этом запись каталога может быть укорочена до имени файла и номера i-узла. Этот вариант изображен на рис. 4.11, *б*. Позже мы увидим, что этот метод имеет некоторые преимущества перед методом размещения атрибутов в записи каталога.

До сих пор мы предполагали, что файлы имеют короткие имена фиксированной длины. В MS-DOS у файлов имелось основное имя, состоящее из 1–8 символов, и необязательное расширение имени, состоящее из 1–3 символов. В UNIX версии 7 имена файлов состояли из 1–14 символов, включая любые расширения. Но практически все современные операционные системы поддерживают длинные имена переменной длины. Как это может быть реализовано?

Проще всего установить предел длины имени файла — как правило, он составляет 255 символов, — а затем воспользоваться одной из конструкций, показанных на рис. 4.11, отводя по 255 символов под каждое имя. Этот подход при всей своей простоте ведет к пустой трате пространства, занимаемого каталогом, поскольку такие длинные имена бывают далеко не у всех файлов. Из соображений эффективности нужно использовать какую-то другую структуру.

Один из альтернативных подходов состоит в отказе от предположения о том, что все записи в каталоге должны иметь один и тот же размер. При таком подходе каждая запись в каталоге начинается с порции фиксированного размера, обычно начинающейся с длины записи, за которой следуют данные в фиксированном формате, чаще всего включающие идентификатор владельца, дату создания, информацию о защите и прочие атрибуты. Следом за заголовком фиксированной длины идет часть записи переменной длины, содержащая имя файла, каким бы длинным оно ни было (рис. 4.12, *а*), с определенным для данной системы порядком следования байтов в словах (например, для SPARC — начиная со старшего). В приведенном примере

показаны три файла: project-budget, personnel и foo. Имя каждого файла завершается специальным символом (обычно 0), обозначенным на рисунке перечеркнутыми квадратиками. Чтобы каждая запись в каталоге могла начинаться с границы слова, имя каждого файла дополняется до целого числа слов байтами, показанными на рисунке закрашенными прямоугольниками.

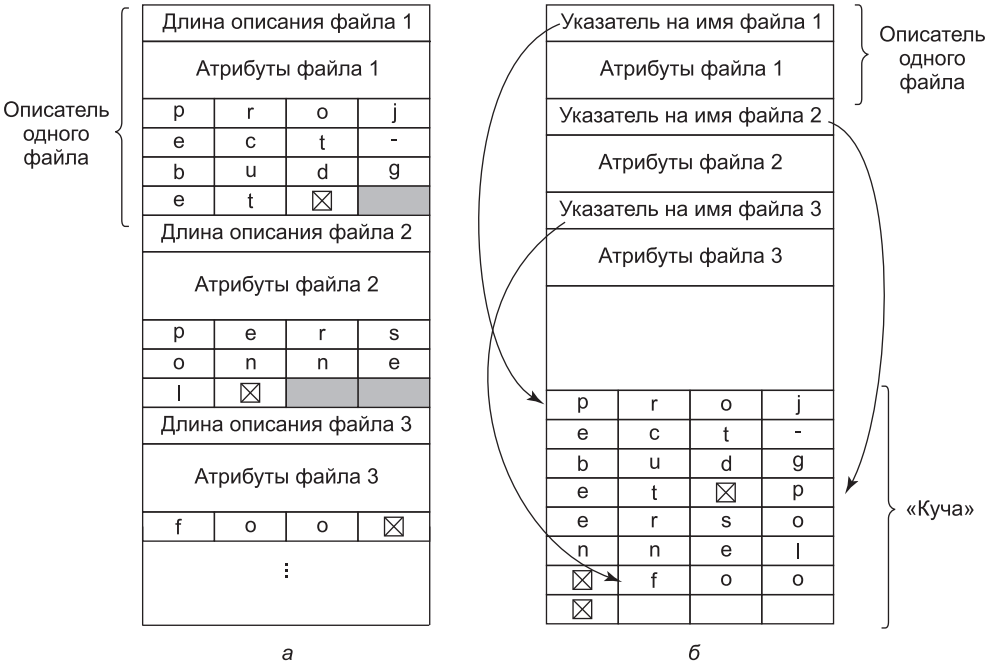

Рис. 4.12. Два способа реализации длинных имен в каталоге: *а* — непосредственно в записи; *б* — в общем хранилище имен (куче)

Недостаток этого метода состоит в том, что при удалении файла в каталоге остается промежуток произвольной длины, в который описатель следующего файла может и не поместиться. Эта проблема по сути аналогична проблеме хранения на диске непрерывных файлов, только здесь уплотнение каталога вполне осуществимо, поскольку он полностью находится в памяти. Другая проблема состоит в том, что какая-нибудь запись каталога может разместиться на нескольких страницах памяти и при чтении имени файла может произойти ошибка отсутствия страницы.

Другой метод реализации имен файлов переменной длины заключается в том, чтобы сделать сами записи каталога фиксированной длины, а имена файлов хранить отдельно в общем хранилище (куче) в конце каталога (рис. 4.12, *б*). Преимущество этого метода состоит в том, что при удалении записи в каталоге (при удалении файла) на ее место всегда сможет поместиться запись другого файла. Но общим хранилищем имен по-прежнему нужно будет управлять, и при обработке имен файлов все так же могут происходить ошибки отсутствия страниц. Небольшой выигрыш заключается в том, что уже не нужно, чтобы имена файлов начинались на границе слов, поэтому отпадает надобность в символах-заполнителях после имен файлов, показанных на рис. 4.12, *б*, в отличие от тех имен, которые показаны на рис. 4.12, *а*.

При всех рассмотренных до сих пор подходах к организации каталогов, когда нужно найти имя файла, поиск в каталогах ведется линейно, от начала до конца. Линейный поиск в очень длинных каталогах может выполняться довольно медленно. Ускорить поиск поможет присутствие в каждом каталоге хэш-таблицы. Пусть размер такой таблицы будет равен *n*. При добавлении в каталог нового имени файла оно должно хэшироваться в число от 0 до $n - 1$, к примеру, путем деления его на *n* и взятия остатка. В качестве альтернативы можно сложить слова, составляющие имя файла, и получившуюся сумму разделить на *n* или сделать еще что-либо подобное[1].

В любом случае просматривается элемент таблицы, соответствующий полученному хэш-коду. Если элемент не используется, туда помещается указатель на запись о файле. Эти записи следуют сразу за хэш-таблицей. Если же элемент таблицы уже занят, то создается связанный список, объединяющий все записи о файлах, имеющих одинаковые хэш-коды, и заголовок этого списка помещается в элемент таблицы.

При поиске файла производится такая же процедура. Для выбора записи в хэш-таблице имя файла хэшируется. Затем на присутствие имени файла проверяются все записи в цепочке, чей заголовок помещен в элементе таблицы. Если искомое имя файла в этой цепочке отсутствует, значит, в каталоге файла с таким именем нет.

Преимущество использования хэш-таблицы состоит в существенном увеличении скорости поиска, а недостаток заключается в усложнении процесса администрирования. Рассматривать применение хэш-таблицы стоит только в тех системах, где ожидается применение каталогов, содержащих сотни или тысячи файлов.

Другим способом ускорения поиска в больших каталогах является кэширование результатов поиска. Перед началом поиска проверяется присутствие имени файла в кэше. Если оно там есть, то местонахождение файла может быть определено немедленно. Разумеется, кэширование поможет, только если результаты поиска затрагивают относительно небольшое количество файлов.

4.3.4. Совместно используемые файлы

Когда над проектом вместе работают несколько пользователей, зачастую возникает потребность в совместном использовании файлов. Поэтому нередко представляется удобным, чтобы совместно используемые файлы одновременно появлялись в различных каталогах, принадлежащих разным пользователям. На рис. 4.13 еще раз показана файловая система, изображенная на рис. 4.4, только теперь один из файлов, принадлежащих пользователю *C*, представлен также в одном из каталогов, принадлежащих пользователю *B*. Установленное при этом отношение между каталогом, принадлежащим *B*, и совместно используемыми файлами называется **связью**. Теперь сама файловая система представляет собой не дерево, а **ориентированный ациклический граф** (Directed Acyclic Graph (**DAG**)). Потребность в том, чтобы файловая система была представлена как DAG, усложняет ее обслуживание, но такова жизнь.

---

[1] Иными словами, производится отображение символьного имени файла в целое число из требуемого диапазона по некоторому алгоритму, рассматривающему имя как некоторое число (битовую строку) или последовательность чисел (например, кодов символов и т. п.). При этом такое преобразование не является взаимно-однозначным. — *Примеч. ред.*

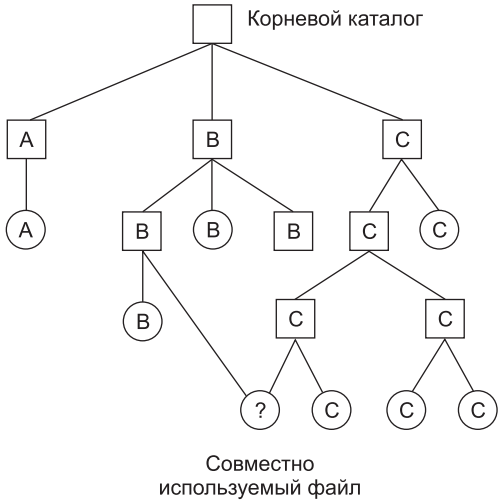

Рис. 4.13. Файловая система, содержащая совместно используемый файл

При всем удобстве совместное использование файлов вызывает ряд проблем. Для начала следует заметить: если собственно каталоги содержат адреса всех дисковых блоков файла, то при установке связи с файлом они должны быть скопированы в каталог, принадлежащий пользователю *B*. Если кто-либо из пользователей, *B* или *C*, чуть позже добавит к файлу какие-то новые данные, то новые блоки попадут в список каталога, принадлежащего только тому пользователю, который производил дополнение. Другому пользователю изменения будут не видны, и совместное использование потеряет всякий смысл.

Эта проблема может быть решена двумя способами. Первое решение заключается в том, что дисковые блоки не указываются в каталогах. Вместо этого с самим файлом связывается некоторая небольшая структура данных. В этом случае каталоги должны лишь указывать на эту структуру данных. Такой подход используется в UNIX (где в качестве такой структуры данных выступает i-узел).

Второе решение заключается в том, что каталог пользователя *B* привязывается к одному из файлов пользователя *C*, заставляя систему создать новый файл типа LINK и включить этот файл в каталог пользователя *B*. Новый файл содержит только имя того файла, с которым он связан. Когда пользователь *B* читает данные из связанного файла, операционная система видит, что файл, из которого они считываются, относится к типу LINK, находит в нем имя файла и читает данные из этого файла. Этот подход в отличие от традиционной (жесткой) связи называется **символической ссылкой**.

У каждого из этих методов имеются недостатки. В первом методе, когда пользователь *B* устанавливает связь с совместно используемым файлом, в i-узле владельцем файла числится пользователь *C*. Создание связи не приводит к изменению владельца (рис. 4.14), а увеличивает показания счетчика связей в i-узле, благодаря чему система знает, сколько записей в каталогах указывает на файл в данный момент.

Если впоследствии пользователь *C* попытается удалить файл, система сталкивается с проблемой. Если она удаляет файл и очищает i-узел, то в каталоге у пользователя *B* будет запись, указывающая на неверный i-узел. Если i-узел чуть позже будет назначен

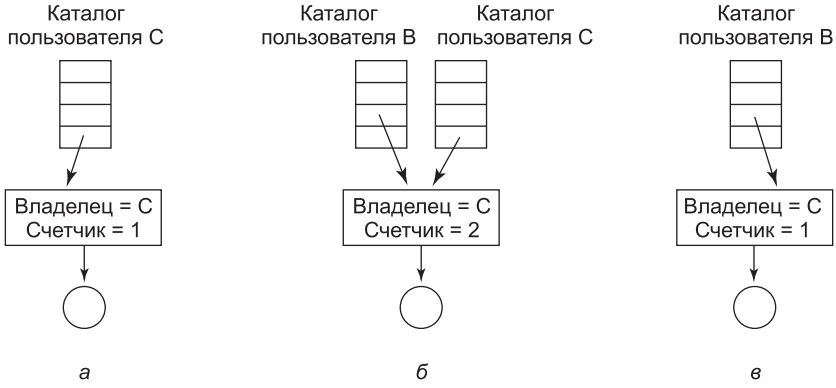

Рис. 4.14. Ситуация, сложившаяся: *а* — перед созданием связи; *б* — после создания связи; *в* — после того как владелец (исходный пользователь) удаляет файл

другому файлу, связь, принадлежащая пользователю *B*, будет указывать на неверный файл (новый, а не тот, для которого она создавалась). По счетчику в i-узле система сможет увидеть, что файл по-прежнему используется, но простого способа найти в каталогах все записи, относящиеся к этому файлу, чтобы их удалить, у нее не будет. Указатели на каталоги не могут быть сохранены в i-узле, поскольку их количество может быть неограниченным.

Единственное, что можно сделать, — это удалить запись в каталоге пользователя *C*, но оставить нетронутым i-узел, установив его счетчик связей в 1 (см. рис. 4.14, *в*). В результате возникнет ситуация, когда *B* является единственным пользователем, имеющим в своем каталоге ссылку на файл, которым владеет пользователь *C*. Если в системе ведется учет использования ресурсов или выделяются какие-то квоты, то пользователь *C* будет по-прежнему получать счета за этот файл, до тех пор пока пользователь *B* не примет решение его удалить. Тогда счетчик будет сброшен до нуля, а файл — удален.

При использовании символической ссылки такой проблемы не возникает, поскольку указатель на i-узел есть только у владельца файла. А у пользователей, создавших ссылку на файл, есть только пути к файлам и нет указателей на i-узлы. Файл при удалении его *владельцем* уничтожается. Последующие попытки использовать этот файл при помощи символической ссылки не будут иметь успеха, так как система не сможет найти файл. Удаление символической ссылки никак на файл не повлияет.

Проблема использования символических ссылок заключается в дополнительных издержках. Файл, в котором содержится путь, должен быть прочитан, затем путь должен быть разобран и покомпонентно пройден, пока не будет достигнут i-узел. Все эти действия могут потребовать многократных дополнительных обращений к диску. Более того, для каждой символической ссылки нужен дополнительный i-узел, равно как и дополнительный дисковый блок для хранения пути, хотя, если имя пути не слишком длинное, система в порядке оптимизации может хранить его в самом i-узле. Преимущество символических ссылок заключается в том, что они могут использоваться для связи с файлами, которые находятся на удаленных компьютерах в любой точке земного шара. Для этого нужно лишь в дополнение к пути к файлу указать сетевой адрес машины, на которой он находится.

Символические или иные ссылки вызывают и еще одну проблему. Когда разрешается использование ссылок, могут появиться два и более путей к файлу. Программы, начинающие свою работу с заданного каталога и ведущие поиск всех файлов в этом каталоге и его подкаталогах, могут обнаруживать файл, на который имеются ссылки, по нескольку раз. Например, программа, архивирующая все файлы в каталоге и всех его подкаталогах на магнитную ленту, может сделать множество копий файла, на который имеются ссылки. Более того, если потом эта лента будет прочитана на другой машине, то вместо создания ссылок на файл он может быть повторно скопирован, если только программа архивации не окажется достаточно «умной».

4.3.5. Файловые системы с журнальной структурой

На современные файловые системы оказывают влияние и технологические изменения, в частности, постоянно растущая скорость центральных процессоров, увеличение емкости и удешевление дисковых накопителей (при не столь впечатляющем увеличении скорости их работы), рост в геометрической прогрессии объема оперативной памяти. Единственным параметром, не демонстрирующим столь стремительного роста, остается время позиционирования блока головок на нужный цилиндр диска (за исключением твердотельных дисков, у которых данный параметр отсутствует).

Сочетание всех этих факторов свидетельствует о том, что у многих файловых систем возникает узкое место в росте производительности. Во время исследований, проведенных в университете Беркли, была предпринята попытка смягчить остроту этой проблемы за счет создания совершенно нового типа файловой системы — **LFS** (Log-structured File System — файловая система с журнальной структурой). Этот раздел будет посвящен краткому описанию работы LFS. Более полное изложение вопроса можно найти в исходной статье по LFS Розенблюма и Остераута (Rosenblum and Ousterhout, 1991).

В основу LFS заложена идея о том, что по мере повышения скорости работы центральных процессоров и увеличения объема оперативной памяти существенно повышается и уровень кэширования дисков. Следовательно, появляется возможность удовлетворения весьма существенной части всех дисковых запросов на чтение прямо из кэша файловой системы без обращения к диску. Из этого наблюдения следует, что в будущем основную массу обращений к диску будут составлять операции записи, поэтому механизм опережающего чтения, применявшийся в некоторых файловых системах для извлечения блоков еще до того, как в них возникнет потребность, уже не дает значительного прироста производительности.

Усложняет ситуацию то, что в большинстве файловых систем запись производится очень малыми блоками данных. Запись малыми порциями слишком неэффективна, поскольку записи на диск, занимающей 50 мкс, зачастую предшествуют позиционирование на нужный цилиндр, на которое затрачивается 10 мс, и ожидание подхода под головку нужного сектора, на что уходит 4 мс. При таких параметрах эффективность работы с диском падает до долей процента.

Чтобы понять, откуда берутся все эти мелкие записи, рассмотрим создание нового файла в системе UNIX. Для записи этого файла должны быть записаны i-узел для каталога, блок каталога, i-узел для файла и сам файл. Эти записи могут быть отложены, но если произойдет сбой до того, как будут выполнены все записи, файловая система столкнется с серьезными проблемами согласованности данных. Поэтому, как правило, записи i-узлов производятся немедленно.

Исходя из этих соображений разработчики LFS решили переделать файловую систему UNIX таким образом, чтобы добиться работы диска с полной пропускной способностью, даже если объем работы состоит из существенного количества небольших произвольных записей. В основу была положена идея структурировать весь диск в виде очень большого журнала.

Периодически, когда в этом возникает особая надобность, все ожидающие осуществления записи, находящиеся в буфере памяти, собираются в один непрерывного сегмент и в таком виде записываются на диск в конец журнала. Таким образом, отдельный сегмент может вперемешку содержать i-узлы, блоки каталога и блоки данных. В начале каждого сегмента находится сводная информация, в которой сообщается, что может быть найдено в этом сегменте. Если средний размер сегмента сможет быть доведен примерно до 1 Мбайт, то будет использоваться практически вся пропускная способность диска.

В этой конструкции по-прежнему используются i-узлы той же структуры, что и в UNIX, но теперь они не размещаются в фиксированной позиции на диске, а разбросаны по всему журналу. Тем не менее, когда определено местоположение i-узла, местоположение блоков определяется обычным образом. Разумеется, теперь нахождение i-узла значительно усложняется, поскольку его адрес не может быть просто вычислен из его i-номера, как в UNIX. Для поиска i-узлов ведется массив i-узлов, проиндексированный по i-номерам. Элемент *i* в таком массиве указывает на i-узел на диске. Массив хранится на диске, но также подвергается кэшированию, поэтому наиболее интенсивно использующиеся фрагменты большую часть времени будут находиться в памяти.

Подытоживая все ранее сказанное: все записи сначала буферизуются в памяти, и периодически все, что попало в буфер, записывается на диск в единый сегмент в конец журнала. Открытие файла теперь состоит из использования массива для определения местоположения i-узла для этого файла. После определения местоположения i-узла из него могут быть извлечены адреса блоков. А все блоки будут находиться в сегментах, расположенных в различных местах журнала.

Если бы диски были безразмерными, то представленное описание на этом и закончилось бы. Но существующие диски не безграничны, поэтому со временем журнал займет весь диск и новые сегменты не смогут быть записаны в него. К счастью, многие существующие сегменты могут иметь уже ненужные блоки. К примеру, если файл перезаписан, его i-узел теперь будет указывать на новые блоки, но старые блоки все еще будут занимать пространство в ранее записанных сегментах.

Чтобы справиться с этой проблемой, LFS использует **очищающий поток**, который занимается тем, что осуществляет круговое сканирование журнала с целью уменьшения его размера. Сначала он считывает краткое содержание первого сегмента журнала, чтобы увидеть, какие i-узлы и файлы в нем находятся. Затем проверяет текущий массив i-узлов, чтобы определить, актуальны ли еще i-узлы и используются ли еще файловые блоки. Если они уже не нужны, то информация выбрасывается. Те i-узлы и блоки, которые еще используются, перемещаются в память для записи в следующий сегмент. Затем исходный сегмент помечается как свободный, и журнал может использовать его для новых данных. Таким же образом очищающий поток перемещается по журналу, удаляя позади устаревшие сегменты и помещая все актуальные данные в память для их последующей повторной записи в следующий сегмент. В результате диск становится большим кольцевым буфером с пишущим потоком, добавляющим впереди новые сегменты, и очищающим потоком, удаляющим позади устаревшие сегменты.

Управление использованием блоков на диске в этой системе имеет необычный характер, поскольку, когда файловый блок опять записывается на диск в новый сегмент, должен быть найден i-узел файла (который находится где-то в журнале), после чего он должен быть обновлен и помещен в память для записи в следующий сегмент. Затем должен быть обновлен массив i-узлов, чтобы в нем присутствовал указатель на новую копию. Тем не менее такое администрирование вполне осуществимо, и результаты измерения производительности показывают, что все эти сложности вполне оправданны. Результаты замеров, приведенные в упомянутой ранее статье, свидетельствуют о том, что при малых записях LFS превосходит UNIX на целый порядок, обладая при этом производительностью чтения и записи больших объемов данных, которая по крайней мере не хуже, чем у UNIX.

4.3.6. Журналируемые файловые системы

При всей привлекательности идеи файловых систем с журнальной структурой они не нашли широкого применения отчасти из-за их крайней несовместимости с существующими файловыми системами. Тем не менее одна из позаимствованных у них идей — устойчивость к отказам — может быть внедрена и в более привычные файловые системы. Основной принцип заключается в журналировании всех намерений файловой системы перед их осуществлением. Поэтому, если система терпит аварию еще до того, как у нее появляется возможность выполнить запланированные действия, после перезагрузки она может посмотреть в журнал, определить, что она собиралась сделать на момент аварии, и завершить свою работу. Такие файловые системы, которые называются **журналируемыми файловыми системами**, нашли свое применение. Журналируемыми являются файловая система NTFS, разработанная Microsoft, а также файловые системы Linux ext3 и ReiserFS. В OS X журналируемая файловая система предлагается в качестве дополнительной. Далее будет дано краткое введение в эту тему.

Чтобы вникнуть в суть проблемы, рассмотрим заурядную, часто встречающуюся операцию удаления файла. Для этой операции в UNIX нужно выполнить три действия:

1. Удалить файл из его каталога.
2. Освободить i-узел, поместив его в пул свободных i-узлов.
3. Вернуть все дисковые блоки файла в пул свободных дисковых блоков.

В Windows требуются аналогичные действия. В отсутствие отказов системы порядок выполнения этих трех действий не играет роли, чего нельзя сказать о случае возникновения отказа. Представьте, что первое действие завершено, а затем в системе возник отказ. Не станет файла, из которого возможен доступ к i-узлу и к блокам, занятым данными файла, но они не будут доступны и для переназначения — они превратятся в ничто, сокращая объем доступных ресурсов. А если отказ произойдет после второго действия, то будут потеряны только блоки.

Если последовательность действий изменится и сначала будет освобожден i-узел, то после перезагрузки системы его можно будет переназначить, но на него будет по-прежнему указывать старый элемент каталога, приводя к неверному файлу. Если первыми будут освобождены блоки, то отказ до освобождения i-узла будет означать, что действующий элемент каталога указывает на i-узел, в котором перечислены блоки, которые теперь находятся в пуле освободившихся блоков и которые в ближайшее

время, скорее всего, будут использованы повторно, что приведет к произвольному совместному использованию одних и тех же блоков двумя и более файлами. Ни один из этих результатов нас не устраивает.

В журналируемой файловой системе сначала делается запись в журнале, в которой перечисляются три намеченных к выполнению действия. Затем журнальная запись сбрасывается на диск (дополнительно, возможно, эта же запись считывается с диска, чтобы убедиться в том, что она было записана правильно). И только после сброса журнальной записи на диск выполняются различные операции. После успешного завершения операций журнальная запись удаляется. Теперь после восстановления системы при ее отказе файловая система может проверить журнал, чтобы определить наличие какой-либо незавершенной операции. Если таковая найдется, то все операции могут быть запущены заново (причем по нескольку раз в случае повторных отказов), и так может продолжаться до тех пор, пока файл не будет удален по всем правилам.

При журналировании все операции должны быть **идемпонентными**, что означает возможность их повторения необходимое число раз без нанесения какого-либо вреда. Такие операции, как «обновить битовый массив, пометив i-узел *k* или блок *n* свободными», могут повторяться до тех пор, пока все не завершится должным и вполне безопасным образом. По аналогии с этим поиск в каталоге и удаление любой записи с именем *foobar* также является идемпонентным действием. В то же время добавление только что освободившихся блоков из i-узла *k* к концу перечня свободных блоков не является идемпонентным действием, поскольку они уже могут присутствовать в этом перечне. Более ресурсоемкая операция «просмотреть перечень свободных блоков и добавить к нему блок *n*, если он в нем отсутствовал» является идемпонентной. Журналируемые файловые системы должны выстраивать свои структуры данных и журналируемые операции таким образом, чтобы все они были идемпонентными. При таких условиях восстановление после отказа может проводиться быстро и безопасно.

Для придания дополнительной надежности в файловой системе может быть реализована концепция **атомарной транзакции**, присущая базам данных. При ее использовании группа действий может быть заключена между операциями начала транзакции — *begin transaction* и завершения транзакции — *end transaction*. Распознающая эти операции файловая система должна либо полностью выполнить все заключенные в эту пару операции, либо не выполнить ни одной из них, не допуская никаких других комбинаций.

NTFS обладает исчерпывающей системой журналирования, и ее структура довольно редко повреждается в результате системных отказов. Ее разработка продолжалась и после первого выпуска в Windows NT в 1993 году. Первой журналируемой файловой системой Linux была ReiserFS, но росту ее популярности помешала несовместимость с применяемой в ту пору стандартной файловой системой ext2. В отличие от нее ext3, представляющая собой менее амбициозный проект, чем ReiserFS, также журналирует операции, но при этом обеспечивает совместимость с предыдущей системой ext2[1].

---

[1] Для Linux существуют и другие журналируемые файловые системы, обладающие своими достоинствами и недостатками. Как минимум необходимо упомянуть ext4, ныне являющуюся файловой системой по умолчанию в ряде распространенных дистрибутивов Linux. — *Примеч. ред.*

4.3.7. Виртуальные файловые системы

Люди пользуются множеством файловых систем, зачастую на одном и том же компьютере и даже для одной и той же операционной системы. Система Windows может иметь не только основную файловую систему NTFS, но и устаревшие приводы или разделы с файловой системой FAT-32 или FAT-16, на которых содержатся старые, но все еще нужные данные, а время от времени могут понадобиться также флеш-накопитель, старый компакт-диск или DVD (каждый со своей уникальной файловой системой). Windows работает с этими совершенно разными файловыми системами, идентифицируя каждую из них по разным именам дисководов, таким как C:, D: и т. д. Когда процесс открывает файл, имя дисковода фигурирует в явном или неявном виде, поэтому Windows знает, какой именно файловой системе передать запрос. Интегрировать разнородные файловые системы в одну унифицированную никто даже не пытается.

В отличие от этого для всех современных систем UNIX предпринимаются весьма серьезные попытки интегрировать ряд файловых систем в единую структуру. У систем Linux в качестве корневой файловой системы может выступать ext2, и она может иметь ext3-раздел, подключенный к каталогу /usr, и второй жесткий диск, имеющий файловую систему ReiserFS, подключенный к каталогу /home, а также компакт-диск, отвечающий стандарту ISO 9660, временно подключенный к каталогу /mnt. С пользовательской точки зрения это будет единая иерархическая файловая система, поскольку объединение нескольких несовместимых файловых систем невидимо для пользователей или процессов.

Существование нескольких файловых систем становится необходимостью, и начиная с передовой разработки Sun Microsystems (Kleiman, 1986) большинство UNIX-систем, пытаясь интегрировать несколько файловых систем в упорядоченную структуру, использовали концепцию **виртуальной файловой системы** (virtual file system (**VFS**)). Ключевая идея состоит в том, чтобы выделить какую-то часть файловой системы, являющуюся общей для всех файловых систем, и поместить ее код на отдельный уровень, из которого вызываются расположенные ниже конкретные файловые системы с целью фактического управления данными. Вся структура показана на рис. 4.15. Рассматриваемый далее материал не имеет конкретного отношения к Linux, или FreeBSD, или любой другой версии UNIX, но дает общее представление о том, как в UNIX-системах работают виртуальные файловые системы.

Все относящиеся к файлам системные вызовы направляются для первичной обработки в адрес виртуальной файловой системы. Эти вызовы, поступающие от пользователь-

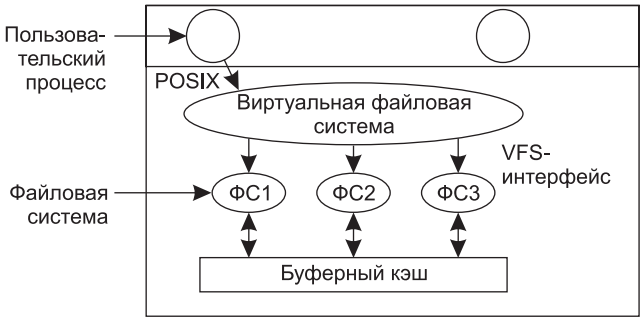

Рис. 4.15. Расположение виртуальной файловой системы

ских процессов, являются стандартными POSIX-вызовами, такими как *open*, *read*, *write*, *lseek* и т. д. Таким образом, VFS обладает «верхним» интерфейсом к пользовательским процессам, и это хорошо известный интерфейс POSIX.

У VFS есть также «нижний» интерфейс к конкретной файловой системе, который на рис. 4.15 обозначен как VFS-интерфейс. Этот интерфейс состоит из нескольких десятков вызовов функций, которые VFS способна направлять к каждой файловой системе для достижения конечного результата. Таким образом, чтобы создать новую файловую систему, работающую с VFS, ее разработчики должны предоставить вызовы функций, необходимых VFS. Вполне очевидным примером такой функции является функция, считывающая с диска конкретный блок, помещающая его в буферный кэш файловой системы и возвращающая указатель на него. Таким образом, у VFS имеются два интерфейса: «верхний» — к пользовательским процессам и «нижний» — к конкретным файловым системам.

Хотя большинство файловых систем, находящихся под VFS, представляют разделы локального диска, так бывает не всегда. На самом деле исходной мотивацией для компании Sun при создании VFS служила поддержка удаленных файловых систем, использующих протокол **сетевой файловой системы** (Network File System (**NFS**)). Конструктивная особенность VFS состоит в том, что пока конкретная файловая система предоставляет требуемые VFS функции, VFS не знает или не заботится о том, где данные хранятся или что собой представляет находящаяся под ней файловая система.

По внутреннему устройству большинство реализаций VFS являются объектно-ориентированными, даже если они написаны на C, а не на C++. Как правило, в них поддерживается ряд ключевых типов объектов. Среди них суперблок (superblock), описывающий файловую систему, v-узел (v-node), описывающий файл, и каталог (directory), описывающий каталог файловой системы. Каждый из них имеет связанные операции (методы), которые должны поддерживаться конкретной файловой системой. Вдобавок к этому в VFS имеется ряд внутренних структур данных для собственного использования, включая таблицу монтирования и массив описателей файлов, позволяющий отслеживать все файлы, открытые в пользовательских процессах.

Чтобы понять, как работает VFS, разберем пример в хронологической последовательности. При загрузке системы VFS регистрирует корневую файловую систему. Вдобавок к этому при подключении (монтировании) других файловых систем, либо во время загрузки, либо в процессе работы они также должны быть зарегистрированы в VFS. При регистрации файловой системы главное, что она делает, — предоставляет список адресов функций, необходимых VFS, в виде либо длинного вектора вызова (таблицы), либо нескольких таких векторов, по одному на каждый VFS-объект, как того требует VFS. Таким образом, как только файловая система зарегистрируется в VFS, виртуальная файловая система будет знать, каким образом, скажем, она может считать блок из зарегистрировавшейся файловой системы, — она просто вызывает четвертую (или какую-то другую по счету) функцию в векторе, предоставленном файловой системой. Также после этого VFS знает, как можно выполнить любую другую функцию, которую должна поддерживать конкретная файловая система: она просто вызывает функцию, чей адрес был предоставлен при регистрации файловой системы.

После установки файловую систему можно использовать. Например, если файловая система была подключена к каталогу /usr и процесс осуществил вызов

```
open("/usr/include/unistd.h",ORDONLY)
```

то при анализе пути VFS увидит, что к /usr была подключена новая файловая система, определит местоположение ее суперблока, просканировав список суперблоков установленных файловых систем. После этого она может найти корневой каталог установленной файловой системы, а в нем — путь include/unistd.h. Затем VFS создает v-узел и направляет вызов конкретной файловой системе, чтобы вернулась вся информация, имеющаяся в i-узле файла. Эта информация копируется в v-узел (в оперативной памяти) наряду с другой информацией, наиболее важная из которой — указатель на таблицу функций, вызываемых для операций над v-узлами, таких как чтение — *read*, запись — *write*, закрытие — *close* и т. д.

После создания v-узла VFS создает запись в таблице описателей файлов вызывающего процесса и настраивает ее так, чтобы она указывала на новый v-узел. (Для особо дотошных — описатель файла на самом деле указывает на другую структуру данных, в которой содержатся текущая позиция в файле и указатель на v-узел, но эти подробности для наших текущих задач не очень важны.) И наконец, VFS возвращает описатель файла вызывавшему процессу, чтобы тот мог использовать его при чтении, записи и закрытии файла.

В дальнейшем, когда процесс осуществляет чтение, используя описатель файла, VFS находит v-узел из таблиц процесса и описателей файлов и следует по указателю к таблице функций, каждая из которых имеет адрес внутри конкретной файловой системы, где и расположен нужный файл. Теперь вызывается функция, управляющая чтением, и внутри конкретной файловой системы запускается код, извлекающий требуемый блок. VFS не знает, откуда приходят данные, с локального диска или по сети с удаленной файловой системы, с флеш-накопителя USB или из другого источника. Задействованные структуры данных показаны на рис. 4.16. Они начинаются с номера вызывающего процесса и описателя файла, затем задействуется v-узел, указатель на функцию *read* и отыскивается доступ к функции внутри конкретной файловой системы.

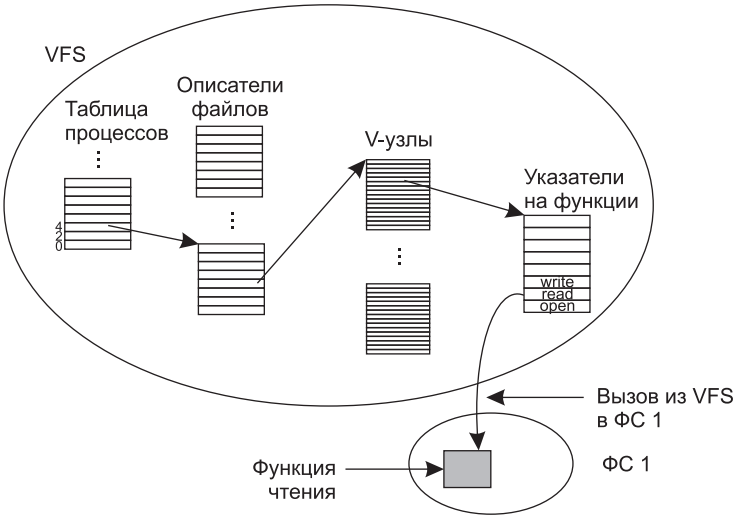

Рис. 4.16. Упрощенный взгляд на структуру данных и код, используемые VFS и конкретной файловой системой для операции чтения

Таким образом, добавление файловых систем становится относительно простой задачей. Чтобы добавить какую-нибудь новую систему, разработчики берут перечень вызовов функций, ожидаемых VFS, а затем пишут свою файловую систему таким образом, чтобы она предоставляла все эти функции. В качестве альтернативы, если файловая система уже существует, им нужно предоставить функции-оболочки, которые делают то, что требуется VFS, зачастую за счет осуществления одного или нескольких вызовов, присущих конкретной файловой системе.

4.4. Управление файловой системой и ее оптимизация

Заставить файловую систему работать — это одно, а вот добиться от нее эффективной и надежной работы — совсем другое. В следующих разделах будет рассмотрен ряд вопросов, относящихся к управлению дисками.

4.4.1. Управление дисковым пространством

Обычно файлы хранятся на диске, поэтому управление дисковым пространством является основной заботой разработчиков файловой системы. Для хранения файла размером n байт возможно использование двух стратегий: выделение на диске n последовательных байтов или разбиение файла на несколько непрерывных блоков. Такая же дилемма между чистой сегментацией и страничной организацией присутствует и в системах управления памятью.

Как уже было показано, при хранении файла в виде непрерывной последовательности байтов возникает очевидная проблема: вполне вероятно, что по мере увеличения его размера потребуется его перемещение на новое место на диске. Такая же проблема существует и для сегментов в памяти, с той лишь разницей, что перемещение сегмента в памяти является относительно более быстрой операцией по сравнению с перемещением файла с одной дисковой позиции на другую. По этой причине почти все файловые системы разбивают файлы на блоки фиксированного размера, которые не нуждаются в смежном расположении.

Размер блока

Как только принято решение хранить файлы в блоках фиксированного размера, возникает вопрос: каким должен быть размер блока? Кандидатами на единицу размещения, исходя из способа организации дисков, являются сектор, дорожка и цилиндр (хотя все эти параметры зависят от конкретного устройства, что является большим минусом). В системах со страничной организацией памяти проблема размера страницы также относится к разряду основных.

Если выбрать большой размер блока (один цилиндр), то каждый файл, даже однобайтовый, занимает целый цилиндр. Это также означает, что существенный объем дискового пространства будет потрачен впустую на небольшие файлы. В то же время при небольшом размере блока (один физический сектор) большинство файлов будет разбито на множество блоков, для чтения которых потребуется множество операций позиционирования головки и ожиданий подхода под головку нужного сектора, сни-

жающих производительность системы. Таким образом, если единица размещения слишком большая, мы тратим впустую пространство, а если она слишком маленькая — тратим впустую время.

Чтобы сделать правильный выбор, нужно обладать информацией о распределении размеров файлов. Вопрос распределения размеров файлов был изучен автором (Tanenbaum et al., 2006) на кафедре информатики крупного исследовательского университета (VU) в 1984 году, а затем повторно изучен в 2005 году, исследовался также коммерческий веб-сервер, предоставляющий хостинг политическому веб-сайту (www.electoral-vote.com). Результаты показаны в табл. 4.3, где для каждого из трех наборов данных перечислен процент файлов, меньших или равных каждому размеру файла, кратному степени числа 2. К примеру, в 2005 году 59,13 % файлов в VU имели размер 4 Кбайт или меньше, а 90,84 % — 64 Кбайт или меньше. Средний размер файла составлял 2475 байт. Кому-то такой небольшой размер может показаться неожиданным.

Какой же вывод можно сделать исходя из этих данных? Прежде всего, при размере блока 1 Кбайт только около 30–50 % всех файлов помещается в единичный блок, тогда как при размере блока 4 Кбайт количество файлов, помещающихся в блок, возрастает до 60–70 %. Судя по остальным данным, при размере блока 4 Кбайт 93 % дисковых блоков используется 10 % самых больших файлов. Это означает, что потеря некоторого пространства в конце каждого небольшого файла вряд ли имеет какое-либо значение, поскольку диск заполняется небольшим количеством больших файлов (видеоматериалов), а то, что основной объем дискового пространства занят небольшими файлами, едва ли вообще имеет какое-то значение. Достойным внимания станет лишь удвоение пространства 90 % файлов.

Таблица 4.3. Процент файлов меньше заданного размера

Длина	VU 1984	VU 2005	Веб-сайт	Длина	VU 1984	VU 2005	Веб-сайт
1 байт	1,79	1,38	6,67	16 Кбайт	92,53	78,92	86,79
2 байта	1,88	1,53	7,67	32 Кбайт	97,27	85,87	91,65
4 байта	2,01	1,65	8,33	64 Кбайт	99,18	90,84	94,80
8 байтов	2,31	1,80	11,30	128 Кбайт	99,84	93,73	96,93
16 байтов	3,32	2,15	11,46	256 Кбайт	99,96	96,12	98,48
32 байта	5,13	3,15	12,33	512 Кбайт	100,00	97,73	98,99
64 байта	8,71	4,98	26,10	1 Мбайт	100,00	98,87	99,62
128 байтов	14,73	8,03	28,49	2 Мбайт	100,00	99,44	99,80
256 байтов	23,09	13,29	32,10	4 Мбайт	100,00	99,71	99,87
512 байта	34,44	20,62	39,94	8 Мбайт	100,00	99,86	99,94
1 Кбайт	48,05	30,91	47,82	16 Мбайт	100,00	99,94	99,97
2 Кбайта	60,87	46,09	59,44	32 Мбайт	100,00	99,97	99,99
4 Кбайта	75,31	59,13	70,64	64 Мбайт	100,00	99,99	99,99
8 Кбайтов	84,97	69,96	79,69	128 Мбайт	100,00	99,99	100,00

В то же время использование небольших блоков означает, что каждый файл будет состоять из множества блоков. Для чтения каждого блока обычно требуется потратить время на позиционирование блока головок и ожидание подхода под головку нужного

сектора (за исключением твердотельного диска), поэтому чтение файла, состоящего из множества небольших блоков, будет медленным.

В качестве примера рассмотрим диск, у которого на каждой дорожке размещается по 1 Мбайт данных. На ожидание подхода нужного сектора затрачивается 8,33 мс, а среднее время позиционирования блока головок составляет 5 мс. Время в миллисекундах, затрачиваемое на чтение блока из k байт, складывается из суммы затрат времени на позиционирование блока головок, ожидание подхода нужного сектора и перенос данных:

$$5 + 4{,}165 + (k/1\,000\,000) \cdot 8{,}33.$$

Пунктирная кривая на рис. 4.17 показывает зависимость скорости передачи данных такого диска от размера блока. Для вычисления эффективности использования дискового пространства нужно сделать предположение о среднем размере файла. В целях упрощения предположим, что все файлы имеют размер 4 Кбайт. Хотя это число несколько превышает объем данных, определенный в VU, у студентов, возможно, больше файлов небольшого размера, чем в корпоративном центре хранения и обработки данных, так что в целом это может быть наилучшим предположением. Сплошная кривая на рис. 4.17 показывает зависимость эффективности использования дискового пространства от размера блока.

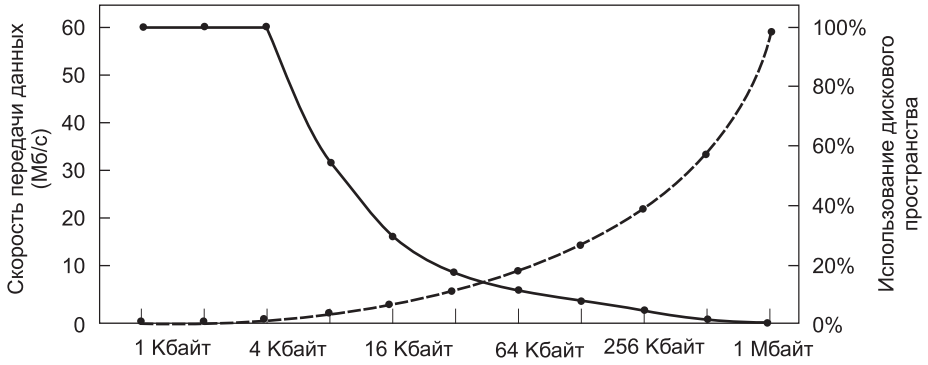

Рис. 4.17. Пунктирная кривая (по шкале слева) показывает скорость передачи данных с диска, сплошная кривая (по правой шкале) показывает эффективность использования дискового пространства. Все файлы имеют размер 4 Кбайт

Эти две кривые можно понимать следующим образом. Время доступа к блоку полностью зависит от времени позиционирования блока головок и ожидания подхода под головки нужного сектора. Таким образом, если затраты на доступ к блоку задаются на уровне 9 мс, то чем больше данных извлекается, тем лучше. Поэтому с ростом размера блока скорость передачи данных возрастает практически линейно (до тех пор, пока перенос данных не займет столько времени, что его уже нужно будет брать в расчет).

Теперь рассмотрим эффективность использования дискового пространства. Потери при использовании файлов размером 4 Кбайт и блоков размером 1, 2 или 4 Кбайт практически отсутствуют. При блоках по 8 Кбайт и файлах по 4 Кбайт эффективность использования дискового пространства падает до 50 %, а при блоках по 16 Кбайт — до 25 %. В реальности точно попадают в кратность размера дисковых блоков всего несколько файлов, поэтому потери пространства в последнем блоке файла бывают всегда.

Кривые показывают, что производительность и эффективность использования дискового пространства по своей сути конфликтуют. Небольшие размеры блоков вредят производительности, но благоприятствуют эффективности использования дискового пространства. В представленных данных найти какой-либо разумный компромисс невозможно. Размер, находящийся поблизости от пересечения двух кривых, составляет 64 Кбайт, но скорость передачи данных в этой точке составляет всего лишь 6,6 Мбайт/с, а эффективность использования дискового пространства находится на отметке, близкой к 7 %. Ни то ни другое нельзя считать приемлемым результатом. Исторически сложилось так, что в файловых системах выбор падал на диапазон размеров от 1 до 4 Кбайт, но при наличии дисков, чья емкость сегодня превышает 1 Тбайт, может быть лучше увеличить размер блоков до 64 Кбайт и смириться с потерями дискового пространства. Вряд ли дисковое пространство когда-либо будет в дефиците.

В рамках эксперимента по поиску существенных различий между использованием файлов в Windows NT и в UNIX Вогельс провел измерения, используя файлы, с которыми работают в Корнелльском университете (Vogels, 1999). Он заметил, что в NT файлы используются более сложным образом, чем в UNIX. Он написал следующее: «Набор в Блокноте нескольких символов с последующим сохранением в файле приводит к 26 системным вызовам, включая 3 неудачные попытки открытия файла, 1 переписывание файла и 4 дополнительные последовательности его открытия и закрытия».

При этом Вогельс проводил исследования с файлами усредненного размера (определенного на практике). Для чтения брались файлы размером 1 Кбайт, для записи — 2,3 Кбайт, для чтения и записи — 4,2 Кбайт. Если принять в расчет различные технологии измерения набора данных и то, что заканчивается 2014 год, эти результаты вполне совместимы с результатами, полученными в VU.

Отслеживание свободных блоков

После выбора размера блока возникает следующий вопрос: как отслеживать свободные блоки? На рис. 4.18 показаны два метода, нашедшие широкое применение. Первый метод состоит в использовании связанного списка дисковых блоков, при этом в каждом блоке списка содержится столько номеров свободных дисковых блоков, сколько в него может поместиться. При блоках размером 1 Кбайт и 32-разрядном номере дискового блока каждый блок может хранить в списке свободных блоков номера 255 блоков. (Одно слово необходимо под указатель на следующий блок.) Рассмотрим диск емкостью 1 Тбайт, имеющий около 1 млрд дисковых блоков. Для хранения всех этих адресов по 255 на блок необходимо около 4 млн блоков. Как правило, для хранения списка свободных блоков используются сами свободные блоки, поэтому его хранение обходится практически бесплатно.

Другая технология управления свободным дисковым пространством использует битовый массив. Для диска, имеющего n блоков, требуется битовый массив, состоящий из n битов. Свободные блоки представлены в массиве в виде единиц, а распределенные — в виде нулей (или наоборот). В нашем примере с диском размером 1 Тбайт массиву необходимо иметь 1 млрд битов, для хранения которых требуется около 130 000 блоков размером 1 Кбайт каждый. Неудивительно, что битовый массив требует меньше пространства на диске, поскольку в нем используется по одному биту на блок, а не по 32 бита, как в модели, использующей связанный список. Только если диск почти заполнен (то есть имеет всего несколько свободных блоков), для системы со связанными списками требуется меньше блоков, чем для битового массива.

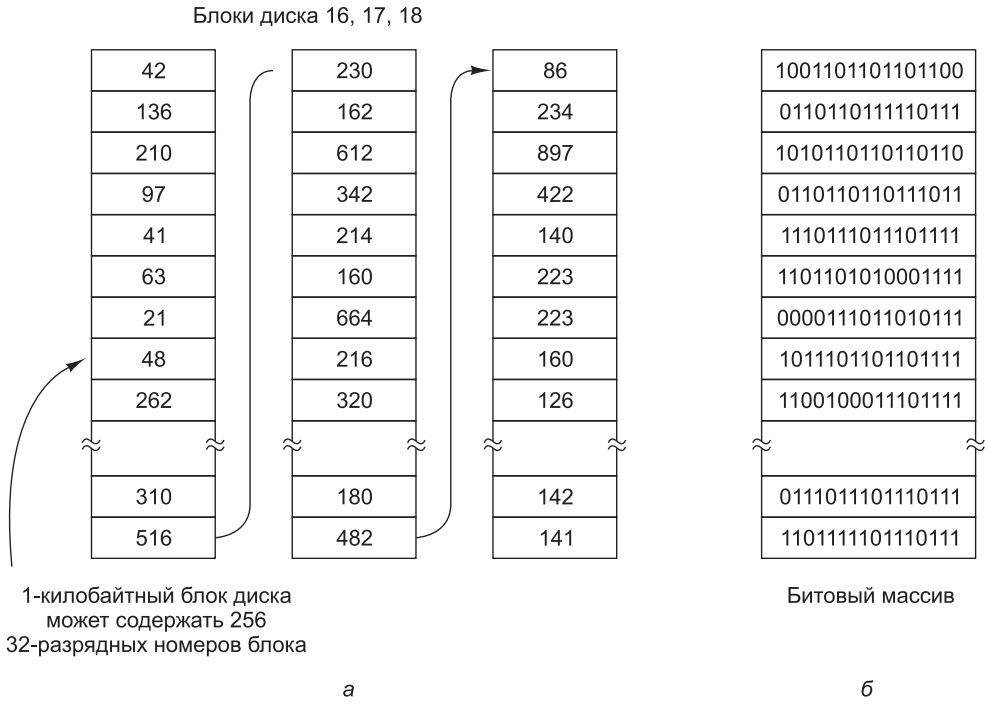

Рис. 4.18. Хранение сведений о свободных блоках: *а* — в связанном списке; *б* — в битовом массиве

Если свободные блоки выстраиваются в длинные непрерывные последовательности блоков, система, использующая список свободных блоков, может быть модифицирована на отслеживание последовательности блоков, а не отдельных блоков. С каждым блоком, дающим номер последовательных свободных блоков, может быть связан 8-, 16- или 32-разрядный счетчик. В идеале в основном пустой диск может быть представлен двумя числами: адресом первого свободного блока, за которым следует счетчик свободных блоков. Но если диск становится слишком фрагментированным, отслеживание последовательностей менее эффективно, чем отслеживание отдельных блоков, поскольку при этом должен храниться не только адрес, но и счетчик.

Это иллюстрирует проблему, с которой довольно часто сталкиваются разработчики операционных систем. Для решения проблемы можно применить несколько структур данных и алгоритмов, но для выбора наилучших из них требуются сведения, которых разработчики не имеют и не будут иметь до тех пор, пока система не будет развернута и испытана временем. И даже тогда сведения могут быть недоступными. К примеру, наши собственные замеры размеров файлов в VU, данные веб-сайта и данные Корнелльского университета — это всего лишь четыре выборки. Так как это лучше, чем ничего, мы склонны считать, что они характерны и для домашних компьютеров, корпоративных машин, компьютеров госучреждений и других вычислительных систем. Затратив определенные усилия, мы могли бы получить несколько выборок для других категорий компьютеров, но даже тогда их было бы глупо экстраполировать на все компьютеры исследованной категории.

Ненадолго возвращаясь к методу, использующему список свободных блоков, следует заметить, что в оперативной памяти нужно хранить только один блок указателей. При создании файла необходимые для него блоки берутся из блока указателей. Когда он будет исчерпан, с диска считывается новый блок указателей. Точно так же при удалении файла его блоки освобождаются и добавляются к блоку указателей, который находится в оперативной памяти. Когда этот блок заполняется, он записывается на диск.

При определенных обстоятельствах этот метод приводит к выполнению излишних дисковых операций ввода-вывода. Рассмотрим ситуацию, показанную на рис. 4.19, *а*, где находящийся в оперативной памяти блок указателей имеет свободное место только для двух записей. Если освобождается файл, состоящий из трех блоков, блок указателей переполняется и должен быть записан на диск, что приводит к ситуации, показанной на рис. 4.19, *б*. Если теперь записывается файл из трех блоков, опять должен быть считан полный блок указателей, возвращающий нас к ситуации, изображенной на рис. 4.19, *а*. Если только что записанный файл из трех блоков был временным файлом, то при его освобождении требуется еще одна запись на диск, чтобы сбросить на него обратно полный блок указателей. Короче говоря, когда блок указателей почти пуст, ряд временных файлов с кратким циклом использования может стать причиной выполнения множества дисковых операций ввода-вывода.

Альтернативный подход, позволяющий избежать большинства операций ввода-вывода, состоит в разделении полного блока указателей на две части. Тогда при освобождении трех блоков вместо перехода от ситуации, изображенной на рис. 4.19, *а*, к ситуации, проиллюстрированной на рис. 4.19, *б*, мы переходим от ситуации, показанной на рис. 4.19, *а*, к ситуации, которую видим на рис. 4.19, *в*. Теперь система может справиться с серией временных файлов без каких-либо операций дискового ввода-вывода. Если блок в памяти заполняется, он записывается на диск, а с диска считывается полузаполненный блок. Идея здесь в том, чтобы хранить большинство блоков указателей на диске полными (и тем самым свести к минимуму использование диска), а в памяти хранить полупустой блок, чтобы он мог обслуживать как создание файла, так и его удаление без дисковых операций ввода-вывода для обращения к списку свободных блоков.

При использовании битового массива можно также содержать в памяти только один блок, обращаясь к диску за другим блоком только при полном заполнении или опусто-

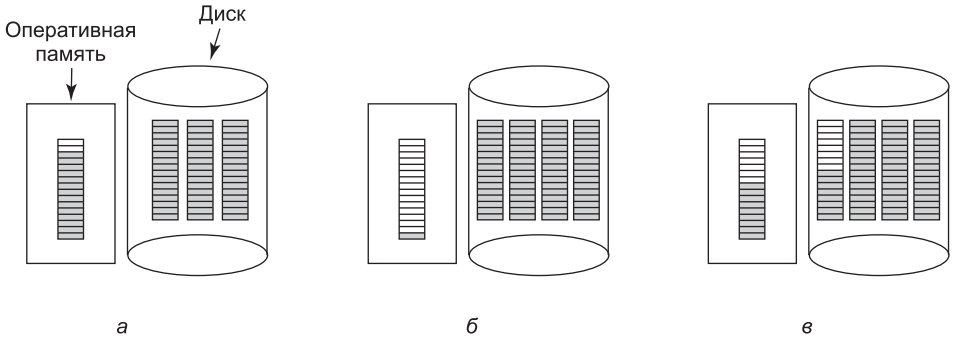

Рис. 4.19. Три ситуации: *а* — почти заполненный блок указателей на свободные дисковые блоки, находящийся в памяти, и три блока указателей на диске; *б* — результат освобождения файла из трех блоков; *в* — альтернативная стратегия обработки трех свободных блоков. Закрашены указатели на свободные дисковые блоки

шении хранящегося в памяти блока. Дополнительное преимущество такого подхода состоит в том, что при осуществлении всех распределений из одного блока битового массива дисковые блоки будут находиться близко друг к другу, сводя к минимуму перемещения блока головок. Поскольку битовый массив относится к структурам данных с фиксированным размером, если ядро частично разбито на страницы, битовая карта может размещаться в виртуальной памяти и иметь страницы, загружаемые по мере надобности.

Дисковые квоты

Чтобы не дать пользователям возможности захватывать слишком большие области дискового пространства, многопользовательские операционные системы часто предоставляют механизм навязывания дисковых квот. Замысел заключается в том, чтобы системный администратор назначал каждому пользователю максимальную долю файлов и блоков, а операционная система гарантировала невозможность превышения этих квот. Далее будет описан типичный механизм реализации этого замысла.

Когда пользователь открывает файл, определяется местоположение атрибутов и дисковых адресов и они помещаются в таблицу открытых файлов, находящуюся в оперативной памяти. В числе атрибутов имеется запись, сообщающая о том, кто является владельцем файла. Любое увеличение размера файла будет засчитано в квоту владельца.

Во второй таблице содержится запись квоты для каждого пользователя, являющегося владельцем какого-либо из открытых в данный момент файлов, даже если этот файл был открыт кем-нибудь другим. Эта таблица показана на рис. 4.20. Она представляет собой извлечение из имеющегося на диске файла квот, касающееся тех пользователей, чьи файлы открыты в настоящее время. Когда все файлы закрыты, записи сбрасывается обратно в файл квот.

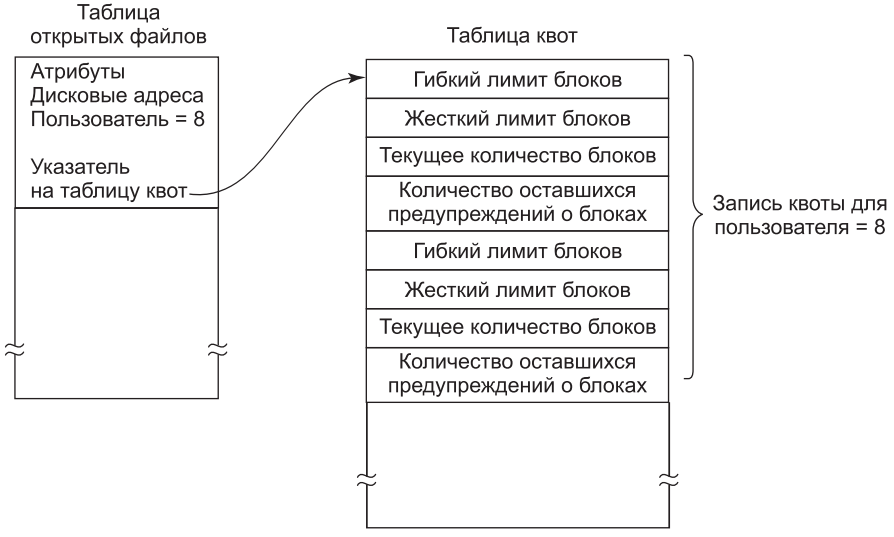

Рис. 4.20. Квоты отслеживаются для каждого пользователя в таблице квот

Когда в таблице открытых файлов делается новая запись, в нее включается указатель на запись квоты владельца, облегчающий поиск различных лимитов. При каждом добавлении блока к файлу увеличивается общее число блоков, числящихся за владельцем, и оно сравнивается как с назначаемым, так и с жестким лимитом. Назначаемый лимит может быть превышен, а вот жесткий лимит — нет. Попытка добавить что-нибудь к файлу, когда достигнут жесткий лимит, приведет к ошибке. Аналогичные сравнения проводятся и для количества файлов, чтобы запретить пользователю дробление i-узлов.

Когда пользователь пытается зарегистрироваться, система проверяет квоту файлов на предмет превышения назначенного лимита как по количеству файлов, так и по количеству дисковых блоков. Когда какой-либо из лимитов превышен, выводится предупреждение и счетчик оставшихся предупреждений уменьшается на единицу. Если когда-нибудь счетчик дойдет до нуля, это значит, что пользователь сразу проигнорировал слишком много предупреждений и ему не будет предоставлена возможность зарегистрироваться. Для повторного разрешения на регистрацию пользователю нужно будет обращаться к системному администратору.

У этого метода есть особенность, которая позволяет пользователям превысить назначенные им лимиты в течение сеанса работы при условии, что они ликвидируют превышение перед выходом из системы. Жесткий лимит не может быть превышен ни при каких условиях.

4.4.2. Резервное копирование файловой системы

Выход из строя файловой системы зачастую оказывается куда более серьезной неприятностью, чем поломка компьютера. Если компьютер ломается из-за пожара, удара молнии или чашки кофе, пролитой на клавиатуру, это, конечно же, неприятно и ведет к непредвиденным расходам, но обычно дело обходится покупкой новых комплектующих и не причиняет слишком много хлопот[1]. Если же на компьютере по аппаратной или программной причине будет безвозвратно утрачена его файловая система, восстановить всю информацию будет делом нелегким, небыстрым и во многих случаях просто невозможным. Для людей, чьи программы, документы, налоговые записи, файлы клиентов, базы данных, планы маркетинга или другие данные утрачены навсегда, последствия могут быть катастрофическими. Хотя файловая система не может предоставить какую-либо защиту против физического разрушения оборудования или носителя, она может помочь защитить информацию. Решение простое: нужно делать резервные копии. Но проще сказать, чем сделать. Давайте ознакомимся с этим вопросом.

Многие даже не задумываются о том, что резервное копирование их файлов стоит потраченного на это времени и усилий, пока в один прекрасный день их диск внезапно не выйдет из строя, — тогда они клянут себя на чем свет стоит. Но компании (обычно) прекрасно осознают ценность своих данных и в большинстве случаев делают резервное копирование не реже одного раза в сутки, чаще всего на ленту. Современные ленты содержат сотни гигабайт, и один гигабайт обходится в несколько центов. Несмотря на это создание резервной копии — не такое уж простое дело, поэтому далее мы рассмотрим ряд вопросов, связанных с этим процессом.

---

[1] Следует заметить, что в случае пожара или удара молнии возможна и обратная ситуация с очень большими хлопотами (из-за утраты данных или физической утраты самого компьютера), причем даже на значительном расстоянии от места происшествия. — *Примеч. ред.*

Резервное копирование на ленту производится обычно для того, чтобы справиться с двумя потенциальными проблемами:

- восстановлением после аварии;
- восстановлением после необдуманных действий (ошибок пользователей).

Первая из этих проблем заключается в возвращении компьютера в строй после поломки диска, пожара, наводнения или другого стихийного бедствия. На практике такое случается нечасто, поэтому многие люди даже не задумываются о резервном копировании. Эти люди по тем же причинам также не склонны страховать свои дома от пожара.

Вторая причина восстановления вызвана тем, что пользователи часто случайно удаляют файлы, потребность в которых в скором времени возникает опять. Эта происходит так часто, что когда файл «удаляется» в Windows, он не удаляется навсегда, а просто перемещается в специальный каталог — Корзину, где позже его можно отловить и легко восстановить. Резервное копирование делает этот принцип более совершенным и дает возможность файлам, удаленным несколько дней, а то и недель назад, восстанавливаться со старых лент резервного копирования.

Резервное копирование занимает много времени и пространства, поэтому эффективность и удобство играют в нем большую роль. В связи с этим возникают следующие вопросы. Во-первых, нужно ли проводить резервное копирование всей файловой системы или только какой-нибудь ее части? Во многих эксплуатирующихся компьютерных системах исполняемые (двоичные) программы содержатся в ограниченной части дерева файловой системы. Если все они могут быть переустановлены с веб-сайта производителя или установочного DVD-диска, то создавать их резервные копии нет необходимости. Также у большинства систем есть каталоги для хранения временных файлов. Обычно включать их в резервную копию также нет смысла. В UNIX все специальные файлы (устройства ввода-вывода) содержатся в каталоге /dev. Проводить резервное копирование этого каталога не только бессмысленно, но и очень опасно, поскольку программа резервного копирования окончательно зависнет, если попытается полностью считать его содержимое. Короче говоря, желательно проводить резервное копирование только указанных каталогов со всем их содержимым, а не копировать всю файловую систему.

Во-вторых, бессмысленно делать резервные копии файлов, которые не изменились со времени предыдущего резервного копирования, что наталкивает на мысль об **инкрементном резервном копировании**. Простейшей формой данного метода будет периодическое создание полной резервной копии, скажем, еженедельное или ежемесячное, и ежедневное резервное копирование только тех файлов, которые были изменены со времени последнего полного резервного копирования. Еще лучше создавать резервные копии только тех файлов, которые изменились со времени их последнего резервного копирования. Хотя такая схема сводит время резервного копирования к минимуму, она усложняет восстановление данных, поскольку сначала должна быть восстановлена самая последняя полная резервная копия, а затем в обратном порядке все сеансы инкрементного резервного копирования. Чтобы упростить восстановление данных, зачастую используются более изощренные схемы инкрементного резервного копирования.

В-третьих, поскольку обычно резервному копированию подвергается огромный объем данных, может появиться желание сжать их перед записью на ленту. Но у многих алгоритмов сжатия одна сбойная область на ленте может нарушить работу алгоритма распаковки и сделать нечитаемым весь файл или даже всю ленту. Поэтому, прежде

чем принимать решение о сжатии потока резервного копирования, нужно хорошенько подумать.

В-четвертых, резервное копирование активной файловой системы существенно затруднено. Если в процессе резервного копирования происходит добавление, удаление и изменение файлов и каталогов, можно получить весьма противоречивый результат. Но поскольку архивация данных может занять несколько часов, то для выполнения резервного копирования может потребоваться перевести систему в автономный режим на большую часть ночного времени, что не всегда приемлемо. Поэтому были разработаны алгоритмы для создания быстрых копий текущего состояния файловой системы за счет копирования критических структур данных, которые при последующем изменении файлов и каталогов требуют копирования блоков вместо обновления их на месте (Hutchinson et al., 1999). При этом файловая система эффективно «замораживается» на момент создания быстрой копии текущего состояния, и ее резервная копия может быть сделана чуть позже в любое удобное время.

И наконец, в-пятых, резервное копирование создает для организации множество нетехнических проблем. Самая лучшая в мире постоянно действующая система безопасности может оказаться бесполезной, если системный администратор хранит все диски или ленты с резервными копиями в своем кабинете и оставляет его открытым и без охраны, когда спускается в зал за распечаткой. Шпиону достаточно лишь войти на несколько секунд, положить в карман одну небольшую кассету с лентой или диск и спокойно уйти прочь. Тогда с безопасностью можно будет распрощаться. Ежедневное резервное копирование вряд ли пригодится, если огонь, охвативший компьютер, испепелит и ленты резервного копирования. Поэтому диски резервного копирования должны храниться где-нибудь в другом месте, но это повышает степень риска (поскольку теперь нужно позаботиться о безопасности уже двух мест). Более подробно этот и другие проблемы администрирования рассмотрены в работе Немета (Nemeth et al., 2000). Далее обсуждение коснется только технических вопросов, относящихся к резервному копированию файловой системы.

Для резервного копирования диска можно воспользоваться одной из двух стратегий: физической или логической архивацией. **Физическая архивация** ведется с нулевого блока диска, при этом все дисковые блоки записываются на лету в порядке их следования и, когда скопирован последний блок, запись останавливается. Эта программа настолько проста, что, возможно, она может быть избавлена от ошибок на все 100 %, чего, наверное, нельзя сказать о любых других полезных программах.

Тем не менее следует сделать несколько замечаний о физической архивации. Прежде всего, создавать резервные копии неиспользуемых дисковых блоков не имеет никакого смысла. Если программа архивирования может получить доступ к структуре данных, регистрирующей свободные блоки, она может избежать копирования неиспользуемых блоков. Но пропуск неиспользуемых блоков требует записывать номер блока перед каждым из них (или делать что-нибудь подобное), потому что теперь уже не факт, что блок k на резервной копии был блоком k на диске.

Вторая неприятность связана с архивированием дефектных блоков. Создать диски больших объемов без каких-либо дефектов практически невозможно. На них всегда найдется несколько дефектных блоков. Время от времени при низкоуровневом форматировании дефектные блоки выявляются, помечаются как плохие и подменяются запасными блоками, находящимися на всякий случай в резерве в конце каждой дорожки.

Во многих случаях контроллер диска справляется с плохими блоками самостоятельно, и операционная система даже не знает об их существовании.

Но иногда блоки портятся уже после форматирования, что когда-нибудь будет обнаружено операционной системой. Обычно эта проблема решается операционной системой путем создания «файла», в котором содержатся все плохие блоки. Это делается хотя бы для того, чтобы они никогда не попали в пул свободных блоков и никогда не попали под распределение. Наверное, излишне говорить, что эти файлы абсолютно нечитаемы.

Если, как говорилось ранее, все плохие блоки перераспределены контроллером диска и скрыты от операционной системы, то физическое архивирование проходит без проблем. Однако если такие блоки находятся в поле зрения операционной системы и собраны в один или несколько файлов или битовых массивов, то очень важно, чтобы программа, осуществляющая физическое архивирование, имела доступ к этой информации и избегала архивирования этих блоков, чтобы предотвратить бесконечные ошибки чтения диска при попытках сделать резервную копию файла, состоящего из плохих блоков.

У систем Windows имеются файлы подкачки и гибернации, которые не нуждаются в восстановлении и не должны подвергаться резервному копированию в первую очередь. У конкретных систем могут иметься и другие внутренние файлы, которые не должны подвергаться резервному копированию, поэтому программы архивации должны о них знать.

Главным преимуществом физического архивирования являются простота и высокая скорость работы (в принципе, архивирование может вестись со скоростью работы диска). А главным недостатком является невозможность пропуска выбранных каталогов, осуществления инкрементного архивирования и восстановления по запросу отдельных файлов. Исходя из этого в большинстве эксплуатирующихся компьютерных систем проводится логическое архивирование.

Логическая архивация начинается в одном или нескольких указанных каталогах и рекурсивно архивирует все найденные там файлы и каталоги, в которых произошли изменения со времени какой-нибудь заданной исходной даты (например, даты создания резервной копии при инкрементном архивировании или даты установки системы для полного архивирования). Таким образом, при логической архивации на магнитную ленту записываются последовательности четко идентифицируемых каталогов и файлов, что упрощает восстановление по запросу указанного файла или каталога.

Поскольку наибольшее распространение получила логическая архивация, рассмотрим подробности ее общего алгоритма на основе примера (рис. 4.21). Этот алгоритм используется в большинстве UNIX-систем. На рисунке показана файловая система с каталогами (квадраты) и файлами (окружности). Закрашены те элементы, которые подверглись изменениям со времени исходной даты и поэтому подлежат архивированию. Светлые элементы в архивации не нуждаются.

Согласно этому алгоритму архивируются также все каталоги (даже не подвергшиеся изменениям), попадающиеся на пути к измененному файлу или каталогу, на что было две причины. Первая — создать возможность восстановления файлов и каталогов из архивной копии в только что созданной файловой системе на другом компьютере. Таким образом, архивирование и восстановление программ может быть использовано для переноса всей файловой системы между компьютерами.

Второй причиной архивирования неизмененных каталогов, лежащих на пути к измененным файлам, является возможность инкрементного восстановления отдельного файла (например, чтобы восстановить файл, удаленный по ошибке). Предположим, что полное архивирование файловой системы сделано в воскресенье вечером, а инкрементное архивирование сделано в понедельник вечером. Во вторник каталог /usr/jhs/proj/nr3 был удален вместе со всеми находящимися в нем каталогами и файлами. В среду спозаранку пользователю захотелось восстановить файл /usr/jhs/proj/nr3/plans/summary. Но восстановить файл summary не представляется возможным, поскольку его некуда поместить. Сначала должны быть восстановлены каталоги nr3 и plans. Чтобы получить верные сведения об их владельцах, режимах использования, метках времени и других атрибутах, эти каталоги должны присутствовать на архивном диске, даже если сами они не подвергались изменениям со времени последней процедуры полного архивирования.

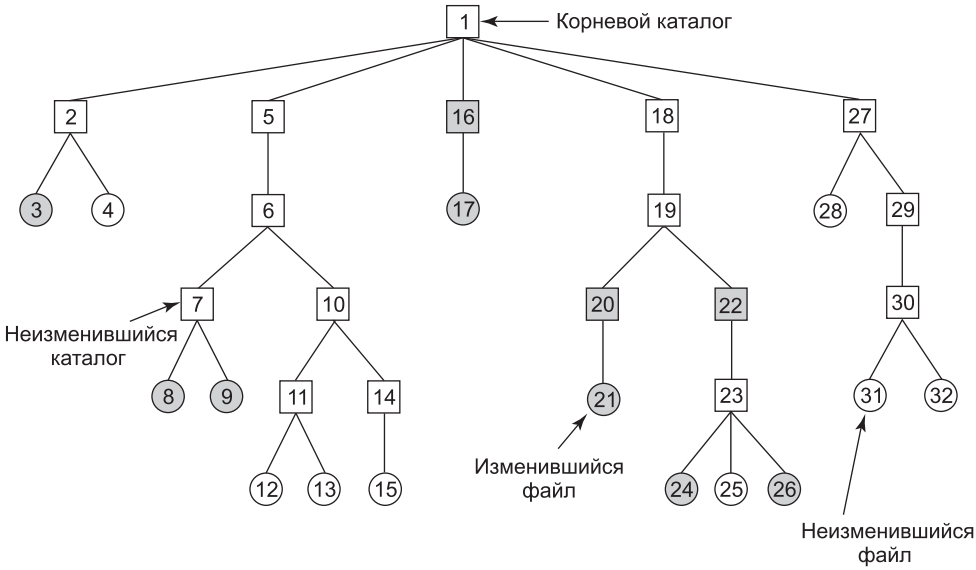

Рис. 4.21. Архивируемая файловая система. Квадратами обозначены каталоги, а окружностями — файлы. Закрашены те элементы, которые были изменены со времени последнего архивирования. Каждый каталог и файл помечен номером своего i-узла

Согласно алгоритму архивации создается битовый массив, проиндексированный по номеру i-узла, в котором на каждый i-узел отводится несколько битов. При реализации алгоритма эти биты будут устанавливаться и сбрасываться. Реализация проходит в четыре этапа. Первый этап начинается с исследования всех элементов начального каталога (например, корневого). В битовом массиве помечается i-узел каждого измененного файла. Также помечается каждый каталог (независимо от того, был он изменен или нет), а затем рекурсивно проводится аналогичное исследование всех помеченных каталогов.

В конце первого этапа помеченными оказываются все измененные файлы и все каталоги, что и показано закрашиванием на рис. 4.22, *а*. Согласно концепции на втором этапе происходит повторный рекурсивный обход всего дерева, при котором снимаются пометки со всех каталогов, которые не содержат измененных файлов или каталогов

как непосредственно, так и в нижележащих поддеревьях каталогов. После этого этапа битовый массив приобретает вид, показанный на рис. 4.22, *б*. Следует заметить, что каталоги 10, 11, 14, 27, 29 и 30 теперь уже не помечены, поскольку ниже этих каталогов не содержится никаких измененных элементов. Они не будут сохраняться в резервной копии. В отличие от них каталоги 5 и 6, несмотря на то что сами они не подверглись изменениям, будут заархивированы, поскольку понадобятся для восстановления сегодняшних изменений на новой машине. Для повышения эффективности первый и второй этапы могут быть объединены в одном проходе дерева.

К этому моменту уже известно, какие каталоги и файлы нужно архивировать. Все они отмечены на рис. 4.22, *б*. Третий этап состоит из сканирования i-узлов в порядке их нумерации и архивирования всех каталогов, помеченных для архивации. Эти каталоги показаны на рис. 4.22, *в*. В префиксе каждого каталога содержатся его атрибуты (владелец, метки времени и т. д.), поэтому они могут быть восстановлены. И наконец, на четвертом этапе также архивируются файлы, помеченные на рис. 4.22, *г*, в префиксах которых также содержатся их атрибуты. На этом архивация завершается.

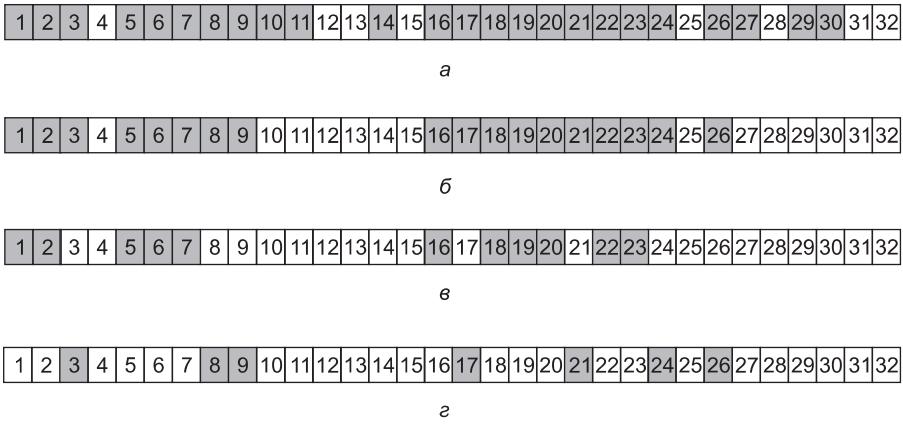

Рис. 4.22. Битовые массивы, используемые алгоритмом логической архивации

Восстановление файловой системы с архивного диска не представляет особого труда. Для начала на диске создается пустая файловая система. Затем восстанавливается самая последняя полная резервная копия. Поскольку первыми на резервный диск попадают каталоги, то все они восстанавливаются в первую очередь, задавая основу файловой системы, после чего восстанавливаются сами файлы. Затем этот процесс повторяется с первым инкрементным архивированием, сделанным после полного архивирования, потом со следующим и т. д.

Хотя логическая архивация довольно проста, в ней имеется ряд хитростей. К примеру, поскольку список свободных блоков не является файлом, он не архивируется, значит, после восстановления всех резервных копий он должен быть реконструирован заново. Этому ничего не препятствует, поскольку набор свободных блоков является всего лишь дополнением того набора блоков, который содержится в сочетании всех файлов.

Другая проблема касается связей. Если файл связан с двумя или несколькими каталогами, нужно, чтобы файл был восстановлен только один раз, а во всех каталогах, которые должны указывать на него, появились соответствующие ссылки.

Еще одна проблема связана с тем обстоятельством, что файлы UNIX могут содержать дыры. Вполне допустимо открыть файл, записать в него всего несколько байтов, переместить указатель на значительное расстояние, а затем записать еще несколько байтов. Блоки, находящиеся в промежутке, не являются частью файла и не должны архивироваться и восстанавливаться. Файлы, содержащие образы памяти, зачастую имеют дыру в сотни мегабайт между сегментом данных и стеком. При неверном подходе у каждого восстановленного файла с образом памяти эта область будет заполнена нулями и, соответственно, будет иметь такой же размер, как и виртуальное адресное пространство (например, 2^{32} байт или, хуже того, 2^{64} байт).

И наконец, специальные файлы, поименованные каналы (все, что не является настоящим файлом) и им подобные никогда не должны архивироваться, независимо от того, в каком каталоге они могут оказаться (они не обязаны находиться только в каталоге /dev). Дополнительную информацию о резервном копировании файловой системы можно найти в трудах Червенака (Chervenak et al., 1998), а также Звиски (Zwicky, 1991).

4.4.3. Непротиворечивость файловой системы

Еще одной проблемой, относящейся к надежности файловой системы, является обеспечение ее непротиворечивости. Многие файловые системы считывают блоки, вносят в них изменения, а потом записывают их обратно на носитель. Если сбой системы произойдет до того, как записаны все модифицированные блоки, файловая система может остаться в противоречивом состоянии. Особую остроту эта проблема приобретает, когда среди незаписанных блоков попадаются блоки i-узлов, блоки каталогов или блоки, содержащие список свободных блоков.

Для решения проблемы противоречивости файловых систем на многих компьютерах имеются служебные программы, проверяющие их непротиворечивость. К примеру, в системе UNIX есть fsck, а в системе Windows — sfc (и другие программы). Эта утилита может быть запущена при каждой загрузке системы, особенно после сбоя. Далее будет дано описание работы утилиты fsck. Утилита sfc имеет ряд отличий, поскольку работает на другой файловой системе, но в ней также действует общий принцип использования избыточной информации для восстановления файловой системы. Все средства проверки файловых систем работают над каждой файловой системой (разделом диска) независимо от всех остальных файловых систем.

Могут применяться два типа проверки непротиворечивости: блочный и файловый. Для проверки блочной непротиворечивости программа создает две таблицы, каждая из которых состоит из счетчика для каждого блока, изначально установленного в нуль. Счетчики в первой таблице отслеживают количество присутствия каждого блока в файле, а счетчики во второй таблице регистрируют количество присутствий каждого блока в списке свободных блоков (или в битовом массиве свободных блоков).

Затем программа считывает все i-узлы, используя непосредственно само устройство, при этом файловая структура игнорируется и возвращаются все дисковые блоки начиная с нулевого. Начиная с i-узла можно построить список всех номеров блоков, используемых в соответствующем файле. Как только будет считан номер каждого блока, увеличивается значение его счетчика в первой таблице. Затем программа проверяет список или битовый массив свободных блоков, чтобы найти все неиспользуемые блоки. Каждое появление блока в списке свободных блоков приводит к увеличению значения его счетчика во второй таблице.

Если у файловой системы нет противоречий, у каждого блока будет 1 либо в первой, либо во второй таблице (рис. 4.23, *а*). Но в результате отказа таблицы могут принять вид, показанный на рис. 4.23, *б*, где блок 2 отсутствует в обеих таблицах. Он будет фигурировать как **пропавший блок**. Хотя пропавшие блоки не причиняют существенного вреда, они занимают пространство, сокращая емкость диска. В отношении пропавших блоков принимается довольно простое решение: программа проверки файловой системы включает их в список свободных блоков.

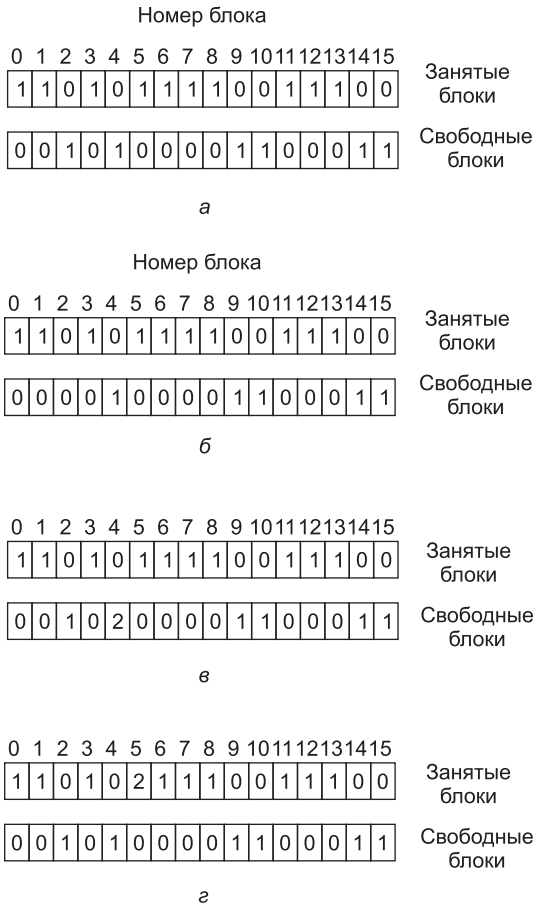

Рис. 4.23. Состояния файловой системы: *а* — непротиворечивое; *б* — с пропавшим блоком; *в* — с блоком, дважды фигурирующим в списке свободных блоков; *г* — с блоком, дважды фигурирующим в данных

Другая возможная ситуация показана на рис. 4.23, *в*. Здесь блок 4 фигурирует в списке свободных блоков дважды. (Дубликаты могут появиться, только если используется именно список свободных блоков, а не битовый массив, с которым этого просто не может произойти.) Решение в таком случае тоже простое: перестроить список свободных блоков.

Хуже всего, если один и тот же блок данных присутствует в двух или нескольких файлах, как случилось с блоком 5 (рис. 4.23, *г*). Если какой-нибудь из этих файлов

удаляется, блок 5 помещается в список свободных блоков, что приведет к тому, что один и тот же блок будет используемым и свободным одновременно. Если будут удалены оба файла, то блок попадет в список свободных блоков дважды.

Приемлемым действием программы проверки файловой системы станет выделение свободного блока, копирование в него содержимого блока 5 и вставка копии в один из файлов. При этом информационное содержимое файлов не изменится (хотя у одного из них оно почти всегда будет искаженным), но структура файловой системы во всяком случае приобретет непротиворечивость. При этом будет выведен отчет об ошибке, позволяющий пользователю изучить дефект.

Вдобавок к проверке правильности учета каждого блока программа проверки файловой системы проверяет и систему каталогов. Она также использует таблицу счетчиков, но теперь уже для каждого файла, а не для каждого блока. Начиная с корневого каталога она рекурсивно спускается по дереву, проверяя каждый каталог файловой системы. Для каждого i-узла в каждом каталоге она увеличивает значение счетчика, чтобы оно соответствовало количеству использований файла.

Вспомним, что благодаря жестким связям файл может фигурировать в двух и более каталогах. Символические ссылки в расчет не берутся и не вызывают увеличения значения счетчика целевого файла.

После того как программа проверки все это сделает, у нее будет список, проиндексированный по номерам i-узлов, сообщающий, сколько каталогов содержит каждый файл. Затем она сравнивает эти количества со счетчиками связей, хранящимися в самих i-узлах. Эти счетчики получают значение 1 при создании файла и увеличивают свое значение при создании каждой жесткой связи с файлом. В непротиворечивой файловой системе оба счетчика должны иметь одинаковые значения. Но могут возникать два типа ошибок: счетчик связей в i-узле может иметь слишком большое или слишком маленькое значение.

Если счетчик связей имеет более высокое значение, чем количество элементов каталогов, то даже если все файлы будут удалены из каталогов, счетчик по-прежнему будет иметь ненулевое значение и i-узел не будет удален. Эта ошибка не представляет особой важности, но она отнимает пространство диска для тех файлов, которых уже нет ни в одном каталоге. Ее следует исправить, установив правильное значение счетчика связей в i-узле.

Другая ошибка может привести к катастрофическим последствиям. Если два элемента каталогов имеют связь с файлом, а i-узел свидетельствует только об одной связи, то при удалении любого из элементов каталога счетчик i-узла обнулится. Когда это произойдет, файловая система помечает его как неиспользуемый и делает свободными все его блоки. Это приведет к тому, что один из каталогов будет указывать на неиспользуемый i-узел, чьи блоки в скором времени могут быть распределены другим файлам. Решение опять-таки состоит в принудительном приведении значения счетчика связей i-узла в соответствие с реально существующим количеством записей каталогов.

Эти две операции, проверка блоков и проверка каталогов, часто совмещаются в целях повышения эффективности (то есть для этого требуется только один проход по всем i-узлам). Возможно проведение и других проверок. К примеру, каталоги должны иметь определенный формат, равно как и номера i-узлов и ASCII-имена. Если номер i-узла превышает количество i-узлов на диске, значит, каталог был поврежден.

Более того, у каждого i-узла есть режим использования (определяющий также права доступа к нему), и некоторые из них могут быть допустимыми, но весьма странными, к примеру 0007, при котором владельцу и его группе вообще не предоставляется ника-

кого доступа, но всем посторонним разрешаются чтение, запись и исполнение файла. Эти сведения могут пригодиться для сообщения о том, что у посторонних больше прав, чем у владельцев. Каталоги, состоящие из более чем, скажем, 1000 элементов, также вызывают подозрения. Если файлы расположены в пользовательских каталогах, но в качестве их владельцев указан привилегированный пользователь и они имеют установленный бит SETUID, то они являются потенциальной угрозой безопасности, поскольку приобретают полномочия привилегированного пользователя, когда выполняются любым другим пользователем. Приложив небольшие усилия, можно составить довольно длинный список технически допустимых, но довольно необычных ситуаций, о которых стоит вывести сообщения.

В предыдущем разделе рассматривалась проблема защиты пользователя от аварийных ситуаций. Некоторые файловые системы также заботятся о защите пользователя от него самого. Если пользователь намеревается набрать команду

```
rm *.o
```

чтобы удалить все файлы, оканчивающиеся на .o (сгенерированные компилятором объектные файлы), но случайно набирает

```
rm * .o
```

(с пробелом после звездочки), то команда *rm* удалит все файлы в текущем каталоге, а затем пожалуется, что не может найти файл .o. В MS-DOS и некоторых других системах при удалении файла устанавливается лишь бит в каталоге или i-узле, помечая файл удаленным. Дисковые блоки не возвращаются в список свободных блоков до тех пор, пока в них не возникнет насущная потребность[1]. Поэтому, если пользователь тут же обнаружит ошибку, можно будет запустить специальную служебную программу, которая вернет назад (то есть восстановит) удаленные файлы. В Windows удаленные файлы попадают в Корзину (специальный каталог), из которой впоследствии при необходимости их можно будет извлечь[2]. Разумеется, пока они не будут реально удалены из этого каталога, пространство запоминающего устройства возвращено не будет.

4.4.4. Производительность файловой системы

Доступ к диску осуществляется намного медленнее, чем доступ к оперативной памяти. Считывание 32-разрядного слова из памяти может занять 10 нс. Чтение с жесткого диска может осуществляться на скорости 100 Мбит/с, что для каждого 32-разрядного

---

[1] В случае файловых систем FAT-12 и FAT-16 для MS-DOS это не совсем верно: блоки (кластеры), занимавшиеся файлом, отмечаются в FAT как свободные, но запись файла в каталоге с потерей первого имени файла сохраняется до момента ее использования новым файлом в данном каталоге. По сохранившимся в записи файла указателю на первый блок и размеру файла его можно полностью восстановить, если он не был фрагментирован и освободившиеся блоки еще не заняты другими файлами. Иначе возможно лишь частичное восстановление или же оно вообще невозможно. — *Примеч. ред.*

[2] Это верно по умолчанию при удалении файлов средствами Проводника Windows. Однако его поведение может быть изменено средствами настройки операционной системы или сторонними программами. Для других программ поведение также может различаться: например, оболочка командной строки cmd.exe удаляет файлы напрямую, не сохраняя их в Корзине. Некоторые программы могут предоставлять выбор способа удаления файлов.

слова будет в четыре раза медленнее, но к этому следует добавить 5–10 мс на установку головок на дорожку и ожидание, когда под головку подойдет нужный сектор. Если нужно считать только одно слово, то доступ к памяти примерно в миллион раз быстрее, чем доступ к диску. В результате такой разницы во времени доступа во многих файловых системах применяются различные усовершенствования, предназначенные для повышения производительности. В этом разделе будут рассмотрены три из них.

Кэширование

Наиболее распространенным методом сокращения количества обращений к диску является **блочное кэширование** или **буферное кэширование**. (Термин «кэш» происходит от французского cacher — скрывать.) В данном контексте кэш представляет собой коллекцию блоков, логически принадлежащих диску, но хранящихся в памяти с целью повышения производительности.

Для управления кэшем могут применяться различные алгоритмы, но наиболее распространенный из них предусматривает проверку всех запросов для определения того, имеются ли нужные блоки в кэше. Если эти блоки в кэше имеются, то запрос на чтение может быть удовлетворен без обращения к диску. Если блок в кэше отсутствует, то он сначала считывается в кэш, а затем копируется туда, где он нужен. Последующий запрос к тому же самому блоку может быть удовлетворен непосредственно из кэша.

Работа кэша показана на рис. 4.24. Поскольку в кэше хранится большое количество (обычно несколько тысяч) блоков, нужен какой-нибудь способ быстрого определения, присутствует ли заданный блок в кэше или нет. Обычно хэшируются адрес устройства и диска, а результаты ищутся в хэш-таблице. Все блоки с таким же значением хэша собираются в цепочку (цепочку коллизий) в связанном списке.

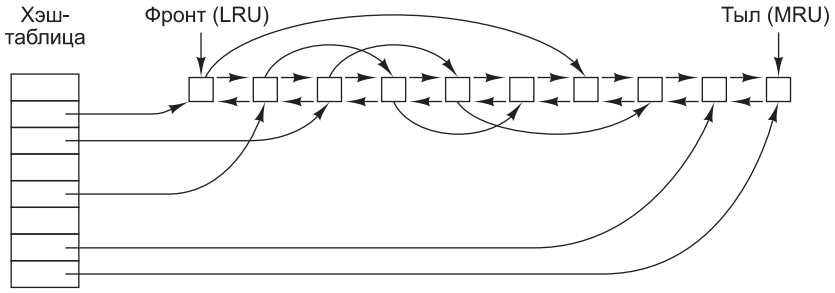

Рис. 4.24. Структура данных буферного кэша

Когда блок необходимо загрузить в заполненный кэш, какой-то блок нужно удалить (и переписать на диск, если он был изменен со времени его помещения в кэш). Эта ситуация очень похожа на ситуацию со страничной организацией памяти, и все общепринятые алгоритмы замещения страниц, рассмотренные в главе 3, включая FIFO, «второй шанс» и LRU, применимы и в данной ситуации. Одно приятное отличие кэширования от страничной организации состоит в том, что обращения к кэшу происходят относительно нечасто, поэтому вполне допустимо хранить все блоки в строгом порядке LRU (замещения наименее востребованного блока), используя связанные списки.

На рис. 4.24 показано, что в дополнение к цепочкам коллизий, начинающихся в хэш-таблице, используется также двунаправленный список, в котором содержатся номера всех блоков в порядке их использования, с наименее востребованным блоком (LRU) в начале этого списка и с наиболее востребованным блоком (MRU) в его конце. Когда происходит обращение к блоку, он может удаляться со своей позиции в двунаправленном списке и помещаться в его конец. Таким образом может поддерживаться точный LRU-порядок.

К сожалению, здесь имеется один подвох. Теперь, когда сложилась ситуация, позволяющая в точности применить алгоритм LRU, оказывается, что это как раз и нежелательно. Проблема возникает из-за сбоев и требований непротиворечивости файловой системы, рассмотренных в предыдущем разделе. Если какой-нибудь очень важный блок, например блок i-узла, считывается в кэш и изменяется, но не переписывается на диск, то сбой оставляет файловую систему в противоречивом состоянии. Если блок i-узла помещается в конец LRU-цепочки, может пройти довольно много времени, перед тем как он достигнет ее начала и будет записан на диск.

Более того, к некоторым блокам, например блокам i-узлов, редко обращаются дважды за короткий промежуток времени. Исходя из этих соображений можно прийти к модифицированной LRU-схеме, в которой берутся в расчет два фактора:

1. Велика ли вероятность того, что данный блок вскоре снова понадобится?
2. Важен ли данный блок с точки зрения непротиворечивости файловой системы?

При ответе на оба вопроса блоки могут быть разделены на такие категории, как блоки i-узлов, косвенные блоки, блоки каталогов, заполненные блоки данных и частично заполненные блоки данных. Блоки, которые, возможно, в ближайшее время не понадобятся, помещаются в начало, а не в конец списка LRU, поэтому вскоре занимаемые ими буферы будут использованы повторно. Блоки, которые вскоре могут снова понадобиться, например частично заполненные блоки, в которые производится запись, помещаются в конец списка, поэтому они останутся в кэше надолго.

Второй вопрос не связан с первым. Если блок важен для непротиворечивости файловой системы (в основном таковы все блоки, за исключением блоков данных) и он был изменен, его тут же нужно записать на диск независимо от того, в каком конце LRU-списка он находится. Быстрая запись важных блоков существенно снижает вероятность аварии файловой системы. Конечно, пользователь может расстроиться, если один из его файлов будет поврежден в случае аварии, но он расстроится еще больше, если будет потеряна вся файловая система.

Даже при таких мерах сохранения целостности файловой системы, слишком долгое хранение в кэше блоков данных без их записи на диск также нежелательно. Войдем в положение писателя, который работает на персональном компьютере. Даже если он периодически заставляет редактор текста сбрасывать редактируемый файл на диск, есть вполне реальная вероятность того, что весь его труд все еще находится в кэше, а на диске ничего нет. Если произойдет сбой, структура файловой системы повреждена не будет, но вся работа за день окажется утрачена.

Подобная ситуация складывается нечасто, если только не попадется слишком невезучий пользователь. Чтобы справиться с этой проблемой, система может применять два подхода. В UNIX используется системный вызов *sync*, вынуждающий немедленно записать все измененные блоки на диск. При запуске системы в фоновом режиме начинает действовать программа, которая обычно называется update. Она работает

в бесконечном цикле и осуществляет вызовы *sync*, делая паузу в 30 с между вызовами. В результате при аварии теряется работа не более чем за 30 с.

Хотя в Windows теперь есть системный вызов, эквивалентный *sync*, который называется *FlushFileBuffers*, в прошлом такого вызова не было. Вместо этого использовалась другая стратегия, которая в чем-то была лучше, чем подход, используемый в UNIX (но в чем-то хуже). При ее применении каждый измененный блок записывался на диск сразу же, как только попадал в кэш. Кэш, в котором все модифицированные блоки немедленно записываются обратно на диск, называется **кэшем со сквозной записью**. Он требует большего объема операций дискового ввода-вывода по сравнению с кэшем без сквозной записи.

Различия между этими двумя подходами можно заметить, когда программа посимвольно записывает полный блок размером 1 Кбайт. Система UNIX будет собирать все символы в кэше и записывать блок на диск или каждые 30 с, или в случае, когда блок будет удаляться из кэша. При использовании кэша со сквозной записью обращение к диску осуществляется при записи каждого символа. Разумеется, большинство программ выполняют внутреннюю буферизацию, поэтому, как правило, они записывают при каждом системном вызове *write* не символ, а строку или еще более крупный фрагмент.

Последствия этого различия в стратегии кэширования состоят в том, что удаление диска из системы UNIX без осуществления системного вызова *sync* практически никогда не обходится без потери данных, а часто приводит еще и к повреждению файловой системы. При кэшировании со сквозной записью проблем не возникает. Столь разные стратегии были выбраны из-за того, что система UNIX разрабатывалась в среде, где все используемые диски были жесткими и не удалялись из системы, а первая файловая система Windows унаследовала свои черты у MS-DOS, которая вышла из мира гибких дисков. Когда в обиход вошли жесткие диски, стал нормой подход, реализованный в UNIX, обладающий более высокой эффективностью (но меньшей надежностью), и теперь он также используется для жестких дисков в системе Windows. Но в файловой системе NTFS, как рассматривалось ранее, для повышения надежности предприняты другие меры (например, журналирование).

В некоторых операционных системах буферное кэширование объединено со страничным. Такое объединение особенно привлекательно при поддержке файлов, отображаемых на память. Если файл отображен на память, то некоторые из его страниц могут находиться в памяти, поскольку они были востребованы. Такие страницы вряд ли отличаются от файловых блоков в буферном кэше. В таком случае они могут рассматриваться одинаковым образом в едином кэше, используемом как для файловых блоков, так и для страниц.

Опережающее чтение блока

Второй метод, улучшающий воспринимаемую производительность файловой системы, заключается в попытке получить блоки в кэш еще до того, как они понадобятся, чтобы повысить соотношение удачных обращений к кэшу. В частности, многие файлы читаются последовательно. Когда у файловой системы запрашивается блок k какого-нибудь файла, она выполняет запрос, но, завершив его выполнение, проверяет присутствие в кэше блока $k + 1$. Если этот блок в нем отсутствует, она планирует чтение блока $k + 1$ в надежде, что, когда он понадобится, он уже будет в кэше. В крайнем случае уже будет в пути.

Разумеется, стратегия опережающего чтения работает только для тех файлов, которые считываются последовательно. При произвольном обращении к файлу опережающее чтение не поможет. Будет досадно, если скорость передачи данных с диска снизится из-за чтения бесполезных блоков и ради них придется удалять полезные блоки из кэша (и, возможно, при этом еще больше снижая скорость передачи данных с диска из-за возвращения на него измененных блоков). Чтобы определить, стоит ли использовать опережающее чтение блоков, файловая система может отслеживать режим доступа к каждому открытому файлу. К примеру, в связанном с каждым файлом бите можно вести учет режима доступа к файлу, устанавливая его в режим последовательного доступа или в режим произвольного доступа. Изначально каждому открываемому файлу выдается кредит доверия, и бит устанавливается в режим последовательного доступа. Но если для этого файла проводится операция позиционирования указателя текущей позиции в файле, бит сбрасывается. При возобновлении последовательного чтения бит будет снова установлен. Благодаря этому файловая система может выстраивать вполне обоснованные догадки о необходимости опережающего чтения. И ничего страшного, если время от времени эти догадки не будут оправдываться, поскольку это приведет всего лишь к незначительному снижению потока данных с диска.

Сокращение количества перемещений блока головок диска

Кэширование и опережающее чтение — далеко не единственные способы повышения производительности файловой системы. Еще одним важным методом является сокращение количества перемещений головок диска за счет размещения блоков с высокой степенью вероятности обращений последовательно, рядом друг с другом, предпочтительно на одном и том же цилиндре. При записи выходного файла файловой системе нужно распределить блоки по одному в соответствии с этим требованием. Если запись о свободных блоках ведется в битовом массиве и весь этот массив находится в памяти, несложно выбрать свободный блок как можно ближе к предыдущему блоку. Если ведется список свободных блоков, часть из которых находится на диске, то задача размещения блоков как можно ближе друг к другу значительно усложняется.

Но даже при использовании списка свободных блоков несколько блоков можно объединить в кластеры. Секрет в том, чтобы отслеживать пространство носителя не в блоках, а в группах последовательных блоков. Если все сектора содержат 512 байт, в системе могут использоваться блоки размером 1 Кбайт (2 сектора), но дисковое пространство может распределяться единицами по 2 блока (4 сектора). Это не похоже на использование дисковых блоков по 2 Кбайт, поскольку в кэше по-прежнему будут использоваться блоки размером 1 Кбайт и передача данных с диска будет вестись блоками размером 1 Кбайт, но последовательное чтение файла на ничем другим не занятой системе сократит количество поисковых операций вдвое, существенно повысив производительность. Вариацией на ту же тему является сокращение количества позиционирований блока головок на нужную дорожку. При распределении блоков система стремится поместить последовательные блоки файла на один и тот же цилиндр.

Еще одно узкое место файловых систем, использующих i-узлы или что-то им подобное, заключается в том, что даже короткие файлы требуют двух обращений к диску: одного для i-узла и второго для блока данных. Обычное размещение i-узлов показано на рис. 4.25, *а*. Здесь все i-узлы размещены ближе к началу диска, поэтому среднее расстояние между i-узлом и его блоками будет составлять половину всего количества цилиндров, требуя больших перемещений блока головок.

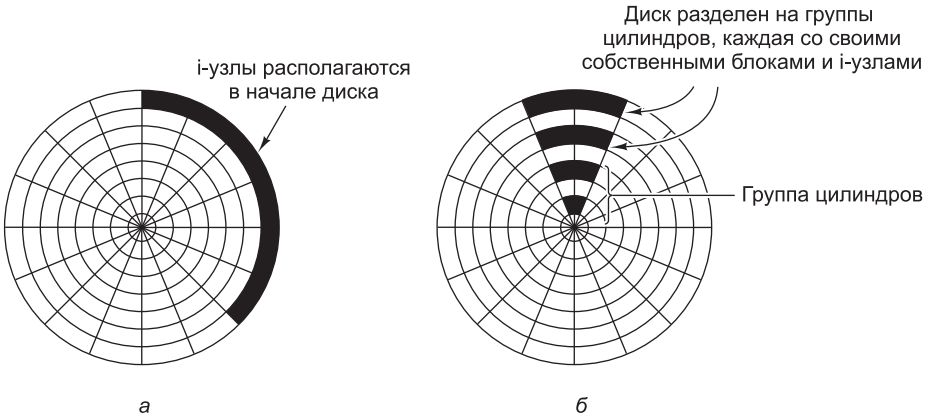

Рис. 4.25. Диск: *а* — i-узлы расположены в его начале; *б* — разделен на группы цилиндров, каждая со своими собственными блоками и i-узлами

Слегка улучшить производительность можно за счет помещения i-узлов в середину диска, а не в его начало, сократив таким образом вдвое среднее число перемещений головок между i-узлом и первым блоком. Еще одна идея, показанная на рис. 4.25, *б*, заключается в разделении диска на группы цилиндров, каждая из которых будет иметь свои i-узлы, блоки и список свободных узлов (McKusick et al., 1984). При создании нового файла может быть выбран любой i-узел, но система стремится найти блок в той же группе цилиндров, в которой находится i-узел. Если это невозможно, используется блок в ближайшей группе цилиндров. Разумеется, перемещение блока головок и время подхода нужного сектора уместны, только если они есть у диска. Все больше компьютеров оснащается **твердотельными дисками** (solid-state disk, SSD), у которых вообще нет подвижных частей. У этих дисков, созданных по такой же технологии, что и флеш-накопители, произвольный доступ осуществляется почти так же быстро, как и последовательный, и многие проблемы традиционных дисков уходят в прошлое. К сожалению, возникают новые. Например, когда дело доходит до чтения, записи и удаления, у SSD-накопителей проявляются особые свойства. В частности, в каждый блок запись может производиться ограниченное количество раз, поэтому большое внимание уделяется равномерному распределению износа по диску.

4.4.5. Дефрагментация дисков

При начальной установке операционной системы все нужные ей программы и файлы устанавливаются последовательно с самого начала диска, каждый новый каталог следует за предыдущим. За установленными файлами единым непрерывным участком следует свободное пространство. Но со временем по мере создания и удаления файлов диск обычно приобретает нежелательную фрагментацию, где повсеместно встречаются файлы и области свободного пространства. Вследствие этого при создании нового файла используемые им блоки могут быть разбросаны по всему диску, что ухудшает производительность.

Производительность можно восстановить за счет перемещения файлов с места на место, чтобы они размещались непрерывно, и объединения всего (или, по крайней мере, основной части) свободного дискового пространства в один или в несколько непрерывных участков на диске. В системе Windows имеется программа defrag, которая

именно этим и занимается. Пользователи Windows должны регулярно запускать эту программу, делая исключение для SSD-накопителей.

Дефрагментация проводится успешнее на тех файловых системах, которые располагают большим количеством свободного пространства в непрерывном участке в конце раздела. Это пространство позволяет программе дефрагментации выбрать фрагментированные файлы ближе к началу раздела и скопировать их блоки в свободное пространство. В результате этого освободится непрерывный участок ближе к началу раздела, в который могут быть целиком помещены исходные или какие-нибудь другие файлы. Затем процесс может быть повторен для следующего участка дискового пространства и т. д.

Некоторые файлы не могут быть перемещены; к ним относятся файлы, используемые в страничной организации памяти и реализации спящего режима, а также файлы журналирования, поскольку выигрыш от этого не оправдывает затрат, необходимых на администрирование. В некоторых системах такие файлы все равно занимают непрерывные участки фиксированного размера и не нуждаются в дефрагментации. Один из случаев, когда недостаток их подвижности становится проблемой, связан с их размещением близко к концу раздела и желанием пользователя сократить размер этого раздела. Единственный способ решения этой проблемы состоит в их полном удалении, изменении размера раздела, а затем их повторном создании.

Файловые системы Linux (особенно ext2 и ext3) обычно меньше страдают от дефрагментации, чем системы, используемые в Windows, благодаря способу выбора дисковых блоков, поэтому принудительная дефрагментация требуется довольно редко. Кроме того, SSD-накопители вообще не страдают от фрагментации. Фактически дефрагментация SSD-накопителя является контрпродуктивной. Она не только не дает никакого выигрыша в производительности, но и приводит к износу, сокращая время их жизни.

4.5. Примеры файловых систем

В следующих разделах будут рассмотрены несколько примеров файловых систем, от самых простых до более сложных. Поскольку современные файловые системы UNIX и собственная файловая система Windows 8 рассмотрены в главе 10, посвященной UNIX, и в главе 11, посвященной Windows 8, здесь эти системы рассматриваться не будут. Зато будут рассмотрены их предшественники.

4.5.1. Файловая система MS-DOS

Файловая система MS-DOS — одна из тех систем, которые применялись на первых персональных компьютерах. До появления Windows 98 и Windows ME она была основной файловой системой. Она все еще поддерживается на Windows 2000, Windows XP и Windows Vista, но теперь уже не является стандартом для новых персональных компьютеров, за исключением тех случаев, когда на них используются гибкие диски. Тем не менее она и ее расширение (FAT-32) нашли широкое применение во многих встраиваемых системах. Ее используют большинство цифровых камер. Многие MP3-плееры используют только эту систему. Популярное устройство Apple iPod использует ее в качестве исходной файловой системы, хотя искушенные хакеры могут переформатировать iPod и установить другую файловую систему. Таким образом, электронных устройств, использующих файловую систему MS-DOS, стало намного больше, чем

когда-либо в прежние времена, и их несомненно больше, чем устройств, использующих более современную файловую систему NTFS. Уже только по этой причине стоит рассмотреть эту систему более подробно.

Чтобы прочитать файл, программа MS-DOS должна сначала сделать системный вызов *open*, чтобы получить его дескриптор. Системному вызову *open* указывается путь, который может быть как абсолютным, так относительным (от текущего рабочего каталога). В пути ведется покомпонентный поиск до тех пор, пока не будет определено местоположение последнего каталога, после чего происходит его чтение в память. Затем в нем ведется поиск открываемого файла.

Хотя каталоги в файловой системе MS-DOS переменного размера, в них используются записи фиксированного размера — 32 байта. Формат записи каталога системы MS-DOS показан на рис. 4.26. В этой записи содержатся имя файла, его атрибуты, дата и время создания, номер начального блока и точный размер файла. Имена файлов короче 8 + 3 символов выравниваются по левому краю полей, и каждое поле по отдельности дополняется пробелами. Поле Attributes (атрибуты) представляет собой новое поле, содержащее биты, указывающие, что для файла разрешено только чтение, файл должен быть заархивирован, файл является системным или скрытым. Запись в файл, для которого разрешено только чтение, не разрешается. Таким образом осуществляется защита файлов от случайных повреждений. Бит *archived* (архивный) не играет для операционной системы никакой роли (то есть MS-DOS его не проверяет и не устанавливает). Он предназначен для того, чтобы пользовательские программы архивирования его сбрасывали при создании резервной копии файла, а остальные программы его устанавливали, если файл подвергся модификации. Таким образом, программа резервного копирования получает возможность просто просмотреть состояние этого бита у каждого файла, чтобы определить, какие именно файлы следует архивировать. Бит *hidden* (скрытый файл) может быть установлен для предотвращения появления файла при отображении содержимого каталога. В основном он используется для того, чтобы не смущать новичков присутствием файлов, назначение которых им непонятно. И наконец, бит *system* (системный) также скрывает файлы и защищает их от случайного удаления командой *del*. Этот бит установлен у всех файлов, содержащих основные компоненты системы MS-DOS.

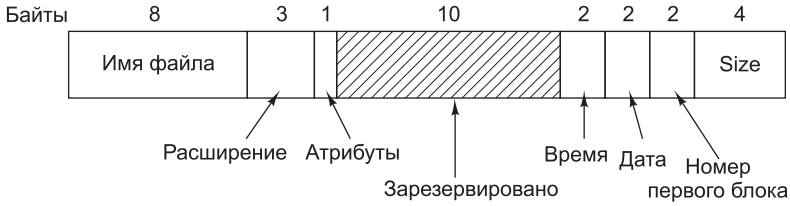

Рис. 4.26. Запись каталога файловой системы MS-DOS

В записи каталога содержатся также дата и время создания или последнего изменения файла. Точность показания времени составляет ±2 с, поскольку под него отводится 2-байтовое поле, способное хранить только 65 536 уникальных значений (а в сутках 86 400 с). Это поле времени подразделяется на секунды (5 бит), минуты (6 бит) и часы (5 бит). Счетчики даты в днях также используют три подполя: день (5 бит), месяц (4 бита) и год — 1980 (7 бит). При использовании 7-разрядного числа для года и отсчета

времени с 1980 года самым большим отображаемым годом будет 2107-й. Это означает, что файловой системе MS-DOS присуща проблема 2108 года.

Чтобы избежать катастрофы, пользователи системы MS-DOS должны как можно раньше начать подготовку к 2108 году. Если бы в MS-DOS использовались объединенные поля даты и времени в виде 32-разрядного счетчика секунд, то удалось бы добиться точности до секунды, а катастрофу можно было бы отложить до 2116 года.

В MS-DOS размер файла хранится в виде 32-разрядного числа, поэтому теоретически файл может иметь размер до 4 Гбайт. Но другие ограничения, рассматриваемые далее, приводят к тому, что максимальный размер файла равен 2 Гбайт или еще меньше. Как ни удивительно, но значительная часть записи (10 байт) остается неиспользуемой.

В MS-DOS файловые блоки отслеживаются через таблицу размещения файлов (FAT — file allocation table), содержащуюся в оперативной памяти. В записи каталога хранится номер первого блока файла. Этот номер используется в качестве индекса к одному из 64 К элементов FAT в оперативной памяти[1]. Следуя по цепочке, можно найти все блоки. Работа с FAT показана на рис. 4.9.

Существуют три версии файловой системы, использующей FAT: FAT-12, FAT-16 и FAT-32 в зависимости от разрядности дискового адреса. Вообще-то название FAT-32 дано несколько неверно, поскольку для адресов диска используются только 28 бит младшего разряда. Ее следовало бы назвать FAT-28, но число, являющееся степенью двойки, куда более благозвучно.

Еще одним вариантом файловой системы FAT является exFAT, введенная компанией Microsoft для больших съемных устройств. Компания Apple лицензировала exFAT, поэтому она является одной из современных файловых систем, которая может использоваться для переноса файлов в обе стороны между компьютерами Windows и компьютерами OS X. Поскольку на exFAT распространяется право собственности и компания Microsoft не выпустила ее спецификацию, далее она рассматриваться не будет.

Для всех вариантов FAT размер дискового блока может быть кратен 512 байтам (это число может быть разным для каждого раздела) и браться из набора разрешенных размеров блока (которые Microsoft называет размерами кластера), разных для каждого варианта. В первой версии MS-DOS использовалась FAT-12 с блоками по 512 байт, задавая максимальный размер раздела $2^{12} \cdot 512$ байт (на самом деле только $4086 \cdot 512$ байт, поскольку 10 дисковых адресов задействовано в качестве специальных маркеров, например конца файла, плохого блока и т. д.). При этом максимальный размер дискового раздела составлял около 2 Мбайт, а размер таблицы FAT в памяти был 4096 записей по 2 байта каждая, поскольку при использовании в таблице 12-разрядных записей система работала бы слишком медленно.

Эта система хорошо работает с гибкими дисками, но с появлением жестких дисков возникла проблема. Microsoft решила эту проблему, разрешив дополнительные размеры блоков в 1, 2 и 4 Кбайт. Такое изменение сохраняет структуру и размер таблицы FAT-12, но допускает использование размера дисковых разделов вплоть до 16 Мбайт.

---

[1] Следует заметить, что, во-первых, FAT в памяти формируется на основе одноименной структуры данных на диске и должна своевременно обновляться, а во-вторых, примерный объем FAT 64 К элементов справедлив только для FAT-16. Для FAT-12 он меньше, для FAT-32 соответственно больше. — *Примеч. ред.*

Так как MS-DOS поддерживала четыре дисковых раздела на каждый привод, новая файловая система FAT-12 работала с дисками объемом до 64 Мбайт. Но кроме этого нужно было что-нибудь другое. Поэтому была представлена FAT-16 с 16-разрядными дисковыми указателями. Дополнительно были разрешены размеры блоков 8, 16 и 32 Кбайт. (32 768 — это наибольшее число, кратное степени двойки, которое может быть представлено 16 разрядами.) Таблица FAT-16 теперь все время занимала 128 Кбайт оперативной памяти, но с ростом доступного объема памяти она получила широкое распространение и быстро вытеснила файловую систему FAT-12. Объем наибольшего дискового раздела, поддерживаемого FAT-16, стал равен 2 Гбайт (64 K записей по 32 Кбайт каждая), а наибольший объем диска — 8 Гбайт, то есть четыре раздела по 2 Гбайт каждый. Долгое время этого было вполне достаточно.

Но такая ситуация сохранилась не навсегда. Для деловых писем такое ограничение не играло существенной роли, а вот при хранении цифрового видео, использующего DV-стандарт, в файле размером 2 Гбайт умещался видеофрагмент продолжительностью чуть больше 9 минут. Как следствие того, что на диске персонального компьютера поддерживаются только четыре раздела, самый большой видеофрагмент, который можно было сохранить на диске, продолжался около 38 минут независимо от того, насколько объемным был сам диск. Это ограничение также означало, что продолжительность наибольшего видеофрагмента, который можно было подвергнуть линейному монтажу, мог быть менее 19 минут, поскольку для этого требовался как входной, так и выходной файлы.

Начиная со второго выпуска Windows 95 была представлена файловая система FAT-32, использующая 28-разрядные дисковые адреса, а версия MS-DOS, положенная в основу Windows 95, была приспособлена для поддержки FAT-32. В этой системе разделы теоретически могли быть по $2^{28} \cdot 2^{15}$ байт, но на самом деле они ограничивались размером 2 Тбайт (2048 Гбайт), поскольку система ведет учет размеров разделов секторами по 512 байт, используя 32-разрядные числа, а $2^9 \cdot 2^{32} = 2$ Тбайт. Максимальный размер раздела для различных размеров блока и всех трех типов FAT показан в табл. 4.4.

Кроме поддержки дисков большого объема файловая система FAT-32 имеет два других преимущества над FAT-16. Во-первых, диск объемом 8 Гбайт, на котором используется FAT-32, может быть отформатирован одним разделом. При использовании FAT-16 он должен был иметь четыре раздела, которые появлялись бы для пользователя Windows как логические диски C:, D:, E: и F:. Пользователь должен был сам решать, какой файл на какой диск поместить, и следить за тем, что где находится.

Таблица 4.4. Максимальный размер раздела для различных размеров блока (пустые клетки означают запрещенные комбинации)

Размер блока, Кбайт	FAT-12, Мбайт	FAT-16, Мбайт	FAT-32, Тбайт
0,5	2		
1	4		
2	8	128	
4	16	256	1
8		512	2
16		1024	2
32		2048	2

Другое преимущество FAT-32 над FAT-16 заключается в том, что для дискового раздела заданного размера могут использоваться блоки меньшего размера. Например, для 2-гигабайтного дискового раздела система FAT-16 должна использовать 32-килобайтные блоки, иначе при наличии всего 64 К доступных дисковых адресов она не смогла бы покрыть весь раздел. В отличие от нее FAT-32 может использовать, к примеру, блоки размером 4 Кбайт для 2-гигабайтного дискового раздела. Преимущество блоков меньшего размера заключается в том, что длина большинства файлов менее 32 Кбайт. При размере блока 32 Кбайт даже 10-байтовый файл будет занимать на диске 32 Кбайт. Если средний размер файлов, скажем, равен 8 Кбайт, то при использовании 32-килобайтных блоков около 3/4 дискового пространства будет теряться впустую, то есть эффективность использования диска будет очень низкой. При 8-килобайтных файлах и 4-килобайтных блоках потерь дискового пространства не будет, но зато для хранения таблицы FAT потребуется значительно больше оперативной памяти. При 4-килобайтных блоках 2-гигабайтный раздел будет состоять из 512 К блоков, поэтому таблица FAT должна состоять из 512 К элементов (занимая 2 Мбайт ОЗУ).

Файловая система MS-DOS использует FAT для учета свободных блоков. Любой нераспределенный на данный момент блок помечается специальным кодом. Когда системе MS-DOS требуется новый блок на диске, она ищет в таблице FAT элемент, содержащий этот код. Поэтому битовый массив или список свободных блоков не нужен.

4.5.2. Файловая система UNIX V7

Даже в ранних версиях системы UNIX применялась довольно сложная многопользовательская файловая система, так как в основе этой системы лежала операционная система MULTICS. Далее будет рассмотрена файловая система V7, разработанная для компьютера PDP-11, сделавшего систему UNIX знаменитой. Современная файловая система UNIX будет рассмотрена в контексте операционной системы Linux в главе 10.

Файловая система представляет собой дерево, начинающееся в корневом каталоге, с добавлением связей, формирующих направленный ациклический граф. Имена файлов могут содержать до 14 любых символов ASCII, кроме слеша (поскольку он служит разделителем компонентов пути) и символа NUL (поскольку он используется для дополнения имен короче 14 символов). Символ NUL имеет числовое значение 0.

Каталог UNIX содержит по одной записи для каждого файла этого каталога. Каждая запись каталога максимально проста, так как в системе UNIX используется система i-узлов (см. рис. 4.10). Запись каталога, как показано на рис. 4.27, состоит всего из двух полей: имени файла (14 байт) и номера i-узла для этого файла (2 байта). Эти параметры ограничивают количество файлов в файловой системе до 64 К.

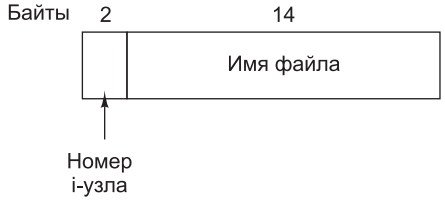

Рис. 4.27. Запись каталога файловой системы UNIX V7

Подобно i-узлу на рис. 4.10 i-узлы системы UNIX содержат некоторые атрибуты, которые содержат размер файла, три указателя времени (создания, последнего доступа и последнего изменения), идентификатор владельца, группы, информацию о защите и счетчик указывающих на этот i-узел записей в каталогах. Последнее поле необходимо для установления связей. При добавлении к i-узлу новой связи счетчик в i-узле увеличивается на единицу. При удалении связи счетчик в i-узле уменьшается на единицу. Когда значение счетчика достигает нуля, i-узел освобождается, а дисковые блоки возвращаются в список свободных блоков.

Для учета дисковых блоков файла используется общий принцип, показанный на рис. 4.10, позволяющий работать с очень большими файлами. Первые 10 дисковых адресов хранятся в самом i-узле, поэтому для небольших файлов вся необходимая информация содержится непосредственно в i-узле, считываемом с диска в оперативную память при открытии файла. Для файлов большего размера один из адресов в i-узле представляет собой адрес блока диска, называемого **однократным косвенным блоком**. Этот блок содержит дополнительные дисковые адреса. Если и этого недостаточно, используется другой адрес в i-узле, называемый **двукратным косвенным блоком** и содержащий адрес блока, в котором хранятся адреса однократных косвенных блоков. Если и этого мало, используется **трехкратный косвенный блок**. Полная схема показана на рис. 4.28.

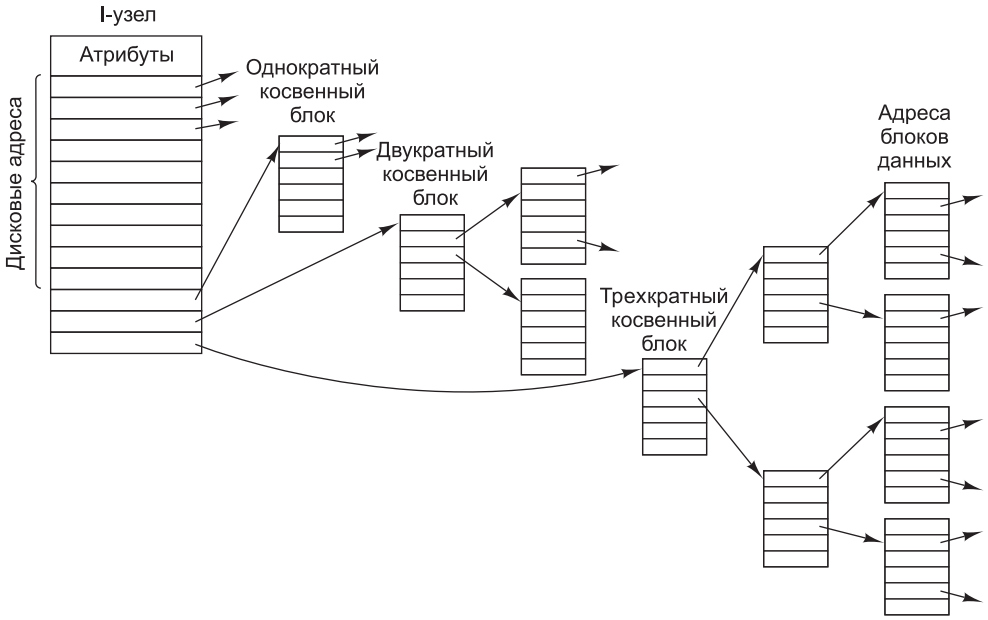

Рис. 4.28. I-узел UNIX

При открытии файла файловая система по предоставленному имени файла должна найти его блоки на диске. Рассмотрим, как происходит поиск по абсолютному имени /usr/ast/mbox. В этом примере будет использоваться файловая система UNIX, хотя для всех иерархических каталоговых систем применяется в основном такой же алгоритм. Сначала файловая система определяет местоположение корневого каталога. В системе UNIX его i-узел размещается в фиксированном месте на диске. По этому i-узлу система

определяет местоположение корневого каталога, который может находиться в любом месте диска, в данном примере — в блоке 1.

Затем файловая система считывает корневой каталог и ищет в нем первый компонент пути, usr, чтобы определить номер i-узла файла /usr. Определить местоположение i-узла по его номеру несложно, поскольку у каждого из них есть свое фиксированное место на диске. По этому i-узлу файловая система определяет местоположение каталога для /usr и ищет в нем следующий компонент, ast. Найдя описатель ast, файловая система получает i-узел для каталога /usr/ast. По этому i-узлу она может найти сам каталог и искать в нем файл mbox. При этом i-узел файла mbox считывается в память и остается там, пока файл не будет закрыт. Процесс поиска показан на рис. 4.29.

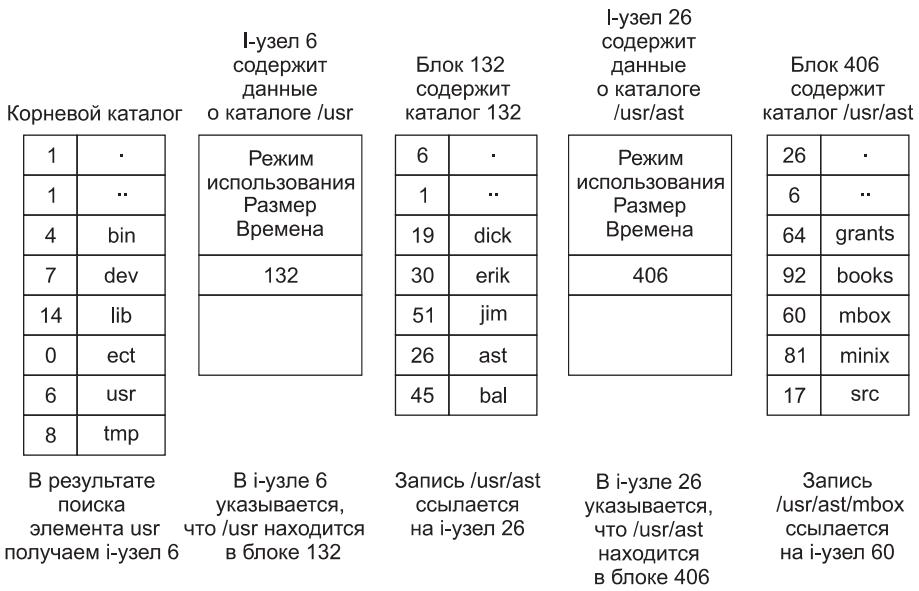

Рис. 4.29. Этапы поиска /usr/ast/mbox

Поиск по относительным именам путей ведется так же, как и по абсолютным, с той лишь разницей, что алгоритм начинает работу не с корневого, а с рабочего каталога. В каждом каталоге есть элементы «.» и «..», помещаемые в каталог в момент его создания. Элемент «.» содержит номер i-узла текущего каталога, а элемент «..» — номер i-узла родительского каталога. Таким образом, процедура, ведущая поиск файла ../dick/prog.c, просто находит «..» в рабочем каталоге, разыскивает в нем номер i-узла родительского каталога, в котором ищет описатель каталога dick. Для обработки этих имен не требуется никакого специального механизма. Что касается системы каталогов, она представляет собой обыкновенные ASCII-строки, ничем не отличающиеся от любых других имен. Единственная тонкость в том, что элемент «..» в корневом каталоге указывает на сам этот каталог.

4.5.3. Файловые системы компакт-дисков

Давайте в качестве последнего примера файловой системы рассмотрим системы, которые используются на компакт-дисках. Это очень простые системы, поскольку они были разработаны для носителей, предназначенных только для чтения данных. Кроме

всего прочего, в них не отслеживаются свободные блоки, поскольку файлы на компакт-дисках не могут освобождаться или добавляться после того, как диск был произведен. Далее мы рассмотрим основной тип файловой системы компакт-дисков и два расширения этого типа. Хотя компакт-диски в настоящее время уже устарели, они все же отличаются простотой, и файловые системы, используемые на DVD- и Blu-ray-дисках, основаны на тех файловых системах, которые использовались на компакт-дисках.

Через несколько лет после появления первого компакт-диска был представлен записываемый компакт-диск — CD-R (CD Recordable). В отличие от простого компакт-диска, на него можно было добавлять файлы после первой записи, но они просто добавлялись к концу записываемого компакт-диска. Файлы никогда не удалялись (хотя каталог мог быть обновлен, чтобы скрыть существующие файлы). С появлением этой файловой системы «только для добавления» основные свойства не изменились. В частности, все свободное пространство представляло собой один непрерывный участок в конце компакт-диска.

Файловая система ISO 9660

Наиболее распространенный международный стандарт **ISO 9660** для файловых систем компакт-дисков был принят в 1988 году. По сути, каждый компакт-диск, имеющийся сейчас на рынке, совместим с этим стандартом, а иногда и с теми расширениями, которые будут рассмотрены чуть позже. Одна из целей принятия этого стандарта — сделать каждый компакт-диск читаемым на любом компьютере, независимо от использующегося порядка следования байтов и независимо от используемой операционной системы. Это привело к тому, что на файловую систему были наложены некоторые ограничения, чтобы она могла читаться в среде слабых операционных систем, использовавшихся в то время (например, в MS-DOS).

У компакт-диска отсутствуют концентрические цилиндры, имеющиеся у магнитных дисков. Вместо них используется одна протяженная спираль с записью битов в линейной последовательности (хотя возможность перемещения головок поперек спирали сохранилась). Биты на протяжении этой спирали разбиты на логические блоки (которые называют также логическими секторами) по 2352 байта. Некоторые из них предназначены для преамбул, коррекции ошибок и других служебных данных. Полезная часть каждого логического блока занимает 2048 байт. Когда компакт-диск используется для музыкальных записей, на нем имеются начальные, конечные и промежуточные пустые места, которые не используются для компакт-дисков с данными. Зачастую позиция блока на спирали приводится в минутах и секундах. Ее можно преобразовать в номер линейного блока, используя соотношение 1 с = 75 блоков.

Стандарт ISO 9660 поддерживает также наборы компакт дисков до $2^{16} - 1$ компакт-диска в наборе. Отдельный компакт-диск также может быть разбит на несколько логических томов (разделов). Но далее мы сконцентрируемся на стандарте ISO 9660 для одного не разбитого на разделы компакт-диска.

Каждый компакт-диск начинается с 16 блоков, чья функция не определена стандартом ISO 9660. Производитель компакт-диска может использовать эту область для программы самозагрузки, позволяющей компьютерам запускаться с компакт-диска, или в каких-нибудь других целях. Далее следует один блок, содержащий **основной описатель тома**, в котором хранится некоторая общая информация о компакт-диске. В нее включены идентификатор системы (32 байта), идентификатор тома (32 байта), идентификатор

издателя (128 байт) и идентификатор того, кто подготовил данные (128 байт). Производитель диска может заполнить эти поля по своему усмотрению, за исключением того, что в них для обеспечения совместимости с различными платформами должны быть буквы в верхнем регистре, цифры и весьма ограниченное количество знаков препинания.

Основной описатель тома содержит также имена трех файлов, в которых могут храниться краткий обзор, уведомление об авторских правах и библиографическая информация соответственно. Кроме того, в этом блоке содержатся определенные ключевые числа, включающие размер логического блока (как правило, 2048, однако в определенных случаях могут использоваться более крупные блоки, размер которых равен степеням числа 2, например 4096, 8192 и т. д.), количество блоков на компакт-диске, а также дата создания и дата окончания срока службы диска. И наконец, основной описатель тома также содержит описатель корневого каталога, что позволяет найти этот каталог на компакт-диске (то есть определить номер блока, содержащего начало каталога). Из этого каталога можно получить местоположение всех остальных элементов файловой системы.

Помимо основного описателя тома компакт-диск может содержать дополнительный описатель тома. В нем хранится информация, подобная той, что хранится в основном описателе, но сейчас нас это интересовать не будет.

Что касается корневого и всех остальных каталогов, то они состоят из переменного количества записей, последняя из которых содержит бит, который помечает ее последней. Сами по себе записи каталогов также имеют переменную длину. Каждая запись каталога состоит из 10–12 полей, некоторые из них имеют ASCII-формат, а остальные — формат двоичного числа. Двоичные поля кодируются дважды, один раз в формате прямого порядка байтов (используемого, к примеру, на машинах Pentium), а второй — в формате обратного порядка байтов (используемого, к примеру, на машинах SPARC). Таким образом, 16-разрядное число использует 4 байта, а 32-разрядное — 8 байт.

Использование подобного избыточного кодирования было обусловлено стремлением не ущемить при разработке стандарта ничьих интересов. Если бы стандарт навязывал прямой порядок байтов, то представители компаний, в чьей продукции использовался обратный порядок байтов, почувствовали бы себя гражданами второго сорта и не приняли бы этот стандарт. Таким образом, эмоциональная составляющая компакт-дисков может быть подсчитана и измерена в килобайтах потерянного пространства.

Формат записи каталога стандарта ISO 9660 показан на рис. 4.30. Поскольку запись каталога имеет переменную длину, первое поле представлено байтом, сообщающим о длине записи. Чтобы избежать любой неопределенности, установлено, что в этом байте старший бит размещается слева.

Записи каталогов могут дополнительно иметь расширенные атрибуты. Если такая возможность используется, то во втором байте указывается длина расширенных атрибутов.

Затем следует номер начального блока самого файла. Файлы хранятся в виде непрерывной череды блоков, поэтому размещение файла полностью определяется начальным блоком и размером, который содержится в следующем поле.

Дата и время записи компакт-диска хранятся в следующем поле в отдельных байтах для года, месяца, дня, часа, минуты, секунды и часового пояса[1]. Летоисчисление для

---

[1] Точнее, дата и время создания файла. — *Примеч. ред.*

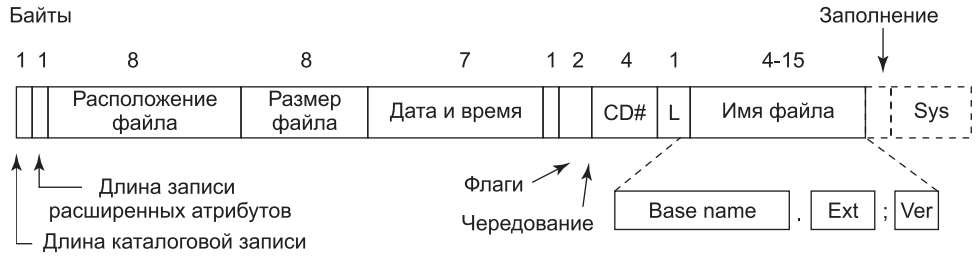

Рис. 4.30. Запись каталога в стандарте ISO 9660

компакт-дисков начинается с 1900 года, а это значит, что компакт-диски подвержены проблеме 2156 года, поскольку за 2155 годом для них последует год 1900. Если бы отсчет велся с 1988 года (когда был принят стандарт), то проблема была бы отложена до 2244 года и в запасе было бы еще 88 лет.

Поле флажков содержит несколько битов разного назначения, включая бит скрытия записи в выводах каталога (свойство, позаимствованное у MS-DOS), бит, позволяющий отличить запись, относящуюся к файлу, от записи, относящейся к каталогу, бит, позволяющий использовать расширенные атрибуты, и бит, помечающий последнюю запись в каталоге. В этом поле имеются и несколько других битов, но здесь они рассматриваться не будут. Следующее поле относится к чередующимся частям файлов, но в простейшей версии ISO 9660 это свойство не используется, поэтому далее оно рассматриваться не будет.

Следующее поле сообщает, на каком компакт-диске расположен файл. Допускается, чтобы запись каталога на одном компакт-диске ссылалась на файл, расположенный на другом компакт-диске набора. Таким образом, появляется возможность создания главного каталога на первом компакт-диске набора, в котором содержится список всех файлов на всех компакт-дисках всего набора.

Поле, помеченное на рис. 4.26 буквой *L*, задает размер имени файла в байтах. Сразу за ним следует само имя файла, которое состоит из основного имени, точки, расширения, точки с запятой и двоичного номера версии (1 или 2 байта). В основном имени и расширении могут использоваться буквы в верхнем регистре, цифры от 0 до 9 и символ подчеркивания. Все остальные символы запрещены, чтобы обеспечить возможность обработки любого имени файла на любом компьютере. Основное имя может содержать до восьми символов, а расширение — до трех. Этот выбор был продиктован необходимостью сохранения совместимости с MS-DOS. Имя файла может фигурировать в каталоге несколько раз, если только каждый экземпляр отличается номером версии.

Последние два поля присутствуют не всегда. Поле *Дополнение* используется для того, чтобы каждая запись каталога составляла четное количество байтов и можно было выровнять числовые поля последующих записей по двухбайтовым границам. Если требуется дополнение, то используется нулевой байт. И наконец, у нас есть поле, используемое системой. Его функции и размер не определены, за исключением того, что в нем должно быть четное количество байтов. В разных системах оно используется по-разному. К примеру, в системах Macintosh в нем хранятся флажки Finder.

Записи в каталоге, за исключением первых двух, идут в алфавитном порядке. Первая запись предназначена для самого каталога. А вторая — для его родительского каталога.

В этом отношении эти записи аналогичны записям «.» и «..» каталога UNIX. Самих файлов для этих записей быть не должно.

Явного ограничения на количество записей в каталоге не установлено. Но есть ограничение на глубину вложенности каталогов. Максимальная глубина вложенности каталогов равна восьми каталогам. Это ограничение было установлено произвольным образом, чтобы упростить реализацию файловой системы.

Стандартом ISO 9660 определены так называемые три уровня. На уровне 1 применяются самые жесткие ограничения и определяется, что имена файлов ограничиваются уже рассмотренной схемой 8 + 3 символа, а также требуется, чтобы все файлы были непрерывными согласно ранее данному описанию. Кроме того, на этом уровне определено, что имена каталогов ограничены восемью символами и не могут иметь расширений. Использование этого уровня предоставляет максимальные возможности того, что компакт-диск будет читаться на любом компьютере.

На уровне 2 делаются послабления ограничений по длине. На нем именам файлов и каталогов позволено иметь длину до 31 символа, но при сохранении того же набора символов.

На уровне 3 используются те же ограничения имен, что и на уровне 2, но ослабляется требование к непрерывности файлов. При применении этого уровня файл может состоять из нескольких секций, каждая из которых представляет собой непрерывную последовательность блоков. Одна и та же секция (последовательность блоков) может встречаться в файле несколько раз и даже входить в несколько различных файлов. Если большие фрагменты данных повторяются в нескольких файлах, применение уровня 3 позволяет каким-то образом оптимизировать использование пространства диска, не требуя, чтобы одни и те же данные присутствовали на нем несколько раз.

Расширения Rock Ridge

Мы уже убедились в том, что стандарт ISO 9660 накладывает целый ряд строгих ограничений. Вскоре после его выпуска представители сообщества UNIX приступили к работе над расширением, позволяющим реализовать файловые системы UNIX на компакт-диске. Эти расширения были названы Rock Ridge, по названию города в фильме Мэла Брукса «Сверкающие седла» (Blazing Saddles), возможно, только потому, что кому-то из членов комиссии нравился этот фильм.

Расширение использует поле, предназначенное для использования системой, чтобы компакт-диск формата Rock Ridge мог читаться на любом компьютере. Все остальные поля сохраняют свое назначение согласно обычному стандарту ISO 9660. Любая система, не работающая с расширениями Rock Ridge, просто игнорирует их и видит обычный компакт-диск.

Расширения делятся на следующие поля:

- PX — атрибуты POSIX;
- PN — старший и младший номера устройств;
- SL — символическая ссылка;
- NM — альтернативное имя;
- CL — расположение дочернего каталога;

- PL — расположение родительского каталога;
- RE — перемещение;
- TF — отметки времени.

Поле PX содержит стандартные биты разрешений *rwxrwxrwx* системы UNIX для владельца, группы и всех остальных. Оно также содержит остальные биты слова режима использования, такие как *SETUID*, *SETGID* и т. п.

Поле PN предназначено для представления на компакт-диске обычных устройств. Оно содержит старший и младший номера устройств, связанных с файлом. Это дает возможность записать на компакт-диск содержимое каталога /dev и впоследствии правильно воссоздать его на целевой системе.

Поле SL предназначено для символических ссылок. Оно позволяет файлу из одной файловой системы ссылаться на файл из другой файловой системы.

Самым важным является поле NM. Оно позволяет связать с файлом второе имя. На это имя не распространяются ограничения по набору используемых символов или длине, накладываемые стандартом ISO 9660, что дает возможность представлять на компакт-диске произвольные имена файлов, принятые в UNIX.

Следующие три поля используются вместе для того, чтобы обойти ограничение ISO 9660 на вложенность каталогов, допускающее только восемь вложений. Использование этих полей позволяет указать на то, что каталог был перемещен, и указать его место в иерархии. Этот способ позволяет эффективно обойти искусственно заданное ограничение на глубину вложенности.

И наконец, поле TF содержит три отметки времени, включенные в каждый i-узел UNIX: время создания файла, время его последней модификации и время последнего доступа к этому файлу. Все вместе эти расширения позволяют скопировать файловую систему UNIX на компакт-диск, а затем правильно восстановить ее на другой системе.

Расширения Joliet

Расширить ISO 9660 стремилось не только сообщество UNIX. Компания Microsoft также сочла этот стандарт слишком усеченным (хотя MS-DOS, из-за которой в первую очередь и были введены эти ограничения, принадлежала самой Microsoft). Поэтому Microsoft изобрела ряд расширений — **Joliet**. Они были разработаны, чтобы позволить файловой системе Windows быть скопированной на компакт-диск с последующим восстановлением, — точно с такой же целью, с которой расширения Rock Ridge были разработаны для UNIX. По сути, все программы, запускаемые под Windows и использующие компакт-диски, поддерживают Joliet, включая программы, которые копируют данные на записываемые компакт-диски. Обычно такие программы предоставляют выбор между различными уровнями ISO 9660 и Joliet.

К основным расширениям, предоставляемым Joliet, относятся:

- длинные имена файлов;
- набор символов Unicode;
- вложенность каталогов глубже восьми уровней;
- имена каталогов, имеющие расширения.

Первое расширение допускает использование в именах файлов до 64 символов. Второе расширение допускает использование в именах файлов набора символов Unicode. Это расширение играет важную роль для программного обеспечения, предназначенного для использования в тех странах, где не применяется латинский алфавит, например в Японии, Израиле и Греции[1]. Поскольку символы Unicode занимают 2 байта, имя файла в Joliet занимает максимум 128 байт.

Из Joliet, как и из Rock Ridge, удалено ограничение на вложенность каталогов. Каталоги могут быть вложенными настолько глубоко, насколько это нужно. И наконец, имена каталогов могут иметь расширения. Не вполне понятно, зачем именно были включены эти расширения, поскольку каталоги Windows вообще-то никогда не используют расширения, но, возможно, когда-нибудь и станут использовать[2].

4.6. Исследования в области файловых систем

Файловые системы всегда привлекали и продолжают привлекать исследователей намного больше, чем другие части операционной системы. Файловым системам и системам хранения данных посвящаются в основном такие тематические конференции, как FAST, MSST и NAS. Хотя стандартные файловые системы довольно хорошо продуманы, все же еще проводятся исследования в отношении резервного копирования (Smaldone et al., 2013; Wallace et al., 2012), кеширования (Koller et al.; Oh, 2012; Zhang et al., 2013a), безопасного стирания данных (Wei et al., 2011), сжатия файлов (Harnik et al., 2013), файловых систем на флеш-накопителях (No, 2012; Park and Shen, 2012; Narayanan, 2009), производительности (Leventhal, 2013; Schindler et al., 2011), RAID (Moon and Reddy, 2013), надежности и восстановления после ошибок (Chidambaram et al., 2013; Ma et. al, 2013; McKusick, 2012; Van Moolenbroek et al., 2012), файловых систем на уровне пользователя (Rajgarhia and Gehani, 2010), проверки согласованности (Fryer et al., 2012) и управления версиями файловых систем (Mashtizadeh et al., 2013). Темой исследования являются также простые измерения того, что происходит в файловой системе (Harter et al., 2012).

Постоянной темой является безопасность (Botelho et al., 2013; Li et al., 2013c; Lorch et al., 2013). В отличие от нее новой актуальной темой стали облачные файловые системы (Mazurek et al., 2012; Vrable et al., 2012). Другой областью, привлекающей внимание в последнее время, является происхождение — отслеживание истории данных, включая то, откуда они взялись, кто их владелец и как они были преобразованы (Ghoshal and Plale, 2013; Sultana and Bertino, 2013). Сохранение данных безопасными и полезными на десятилетия также интересует компании, у которых есть для этого вполне законные требования (Baker et al., 2006). И наконец, другие исследователи занимаются переосмыслением стека файловой системы (Appuswamy et al., 2011).

---

[1] И, разумеется, в России. А также во всех остальных странах, в которых используются системы письменности, не основанные на латинице. — *Примеч. ред.*

[2] Если считать Internet Explorer частью операционной системы, то он уже в Windows XP использует расширения как минимум для каталогов со своими временными файлами. Да и зачем ограничивать пользователей в выразительных возможностях при создании своих каталогов. Например, можно для каталога с резервной копией своего сайта использовать его доменное имя вида my.site.city.net. — *Примеч. ред.*

4.7. Краткие выводы

С точки зрения внешнего наблюдателя файловая система представляется коллекцией файлов и каталогов и совокупностью действий над ними. Файлы могут быть считаны и записаны, каталоги могут быть созданы и удалены, а файлы могут быть перемещены из каталога в каталог. Многие современные файловые системы поддерживают иерархическую систему каталогов, в которой каталоги могут иметь подкаталоги, те, в свою очередь, также могут иметь подкаталоги, и так до бесконечности.

Изнутри файловая система выглядит совершенно иначе. Разработчики файловых систем должны заботиться о том, как распределяется пространство носителя и как система ведет учет распределения блоков файлам. При этом есть возможность использования непрерывных файлов, связанных списков, таблиц размещения файлов и i-узлов. Разные системы имеют различные структуры каталогов. Атрибуты могут помещаться в каталогах или где-либо еще (например, в i-узле). Дисковым пространством можно управлять при помощи списков свободных блоков или битовых массивов. Надежность файловой системы увеличивается за счет осуществления инкрементного архивирования и использования программы, способной устранять дефекты файловых систем. Важную роль играет производительность файловой системы, которая может быть улучшена несколькими способами, включая кэширование, опережающее чтение и размещение блоков одного файла близко друг к другу. Повысить производительность позволяют также файловые системы с журнальной структурой, которые ведут запись на диск большими порциями.

Примерами файловых систем могут послужить ISO 9660, MS-DOS и UNIX. Они во многом отличаются друг от друга, включая способ учета распределения блоков между файлами, структуру каталогов и управление свободным дисковым пространством.

Вопросы

1. Дайте пять разных путевых имен для файла /etc/passwd.

   Подсказка: подумайте о записях каталогов «.» и «..».

2. Когда в системе Windows пользователь щелкает на файле, находящемся в списке Windows Explorer, запускается программа и этот файл передается ей в качестве аргумента. Назовите два разных способа, позволяющие операционной системе узнать, какую именно программу следует запускать.

3. В ранних UNIX-системах исполняемые файлы (файлы a.out) начинались с особого «магического» числа, выбираемого далеко не случайным образом. Эти файлы начинались с заголовка, за которым следовали сегменты с текстом программы и данными. Как вы думаете, почему для исполняемых файлов было выбрано особое «магическое» число, тогда как для файлов других типов в качестве первых слов использовались более или менее произвольные «магические» числа?

4. Является ли системный вызов *open* неотъемлемой частью UNIX? Какими будут последствия, если этого системного вызова в ней не станет?

5. У систем, поддерживающих последовательные файлы, всегда есть операция для их «перемотки». Нужна ли эта операция системе, поддерживающей файлы произвольного доступа?

6. В некоторых операционных системах для присваивания файлу нового имени предоставляется системный вызов *rename*. Есть ли какая-нибудь разница между использованием этого системного вызова для переименования файла и копированием файла в новый файл с новым именем с последующим удалением старого файла?
7. Некоторые системы позволяют отображать часть файла на память. Какие ограничения должны накладывать такие системы? Как реализуется такое частичное отображение?
8. Простая операционная система поддерживает только один каталог, но позволяет хранить в нем произвольное количество файлов с именами произвольной длины. Можно ли на такой системе сымитировать что-либо подобное иерархической файловой системе? Как это сделать?
9. В UNIX и Windows произвольный доступ осуществляется с помощью специальных системных вызовов, перемещающих связанный с файлом указатель текущей позиции на заданное количество байтов в файле. Предложите альтернативный способ осуществления произвольного доступа без использования этого системного вызова.
10. Рассмотрим дерево каталогов, показанное на рис. 4.5. Если каталог /usr/jim является рабочим, то каким будет абсолютное имя пути для файла, чье относительное имя пути ../ast/x?
11. В тексте главы упоминалось, что последовательное размещение файлов ведет к фрагментации диска, поскольку часть пространства в последнем блоке файла будет потрачена впустую в тех файлах, чья длина не укладывается в целое число блоков. К какому типу фрагментации это относится, внутренней или внешней? Проведите аналогию с примерами, рассмотренными в предыдущей главе.
12. Опишите последствия повреждения блока данных для заданного файла: а) для непрерывных, б) связанных, в) индексированных (или основанных на использовании таблицы) схем размещения блоков.
13. Одним из способов распределения дискового пространства непрерывными областями без ущерба от наличия пустующих мест является уплотнение диска при каждом удалении файла. Поскольку все файлы располагаются одной непрерывной областью, то при копировании файла его нужно прочитать, для чего приходится тратить определенное время на позиционирование блока головок на нужный цилиндр и на ожидание подхода нужного сектора, после чего осуществляется перенос данных на полной скорости. Запись файла на диск требует тех же действий. Предположим, что время позиционирования блока головок на нужный цилиндр занимает 5 мс, ожидание подхода под головку нужного сектора — 4 мс, скорость передачи данных равна 8 Мбайт/с, а средний размер файла 8 Кбайт. Сколько времени понадобится для того, чтобы считать файл в оперативную память, а затем записать его обратно на новое место на диске? Сколько понадобится времени при тех же параметрах для уплотнения половины 16-гигабайтного диска?
14. Ответьте, взяв за основу тему предыдущего вопроса, есть ли вообще какой-нибудь смысл в уплотнении диска?
15. Некоторые цифровые потребительские устройства нуждаются в хранении данных, например в виде файлов. Назовите современные устройства, требующие хранения файлов, для которых хорошо подошло бы непрерывное размещение файлов.

16. Рассмотрим i-узел, показанный на рис. 4.10. Каков может быть максимальный размер файла, если этот узел содержит 10 прямых адресов по 8 байтов каждый, а все дисковые блоки имеют размер 1024 Кбайт?
17. Записи студентов для заданного класса хранятся в файле. Доступ к записям и их обновление происходят в произвольном порядке. Предположим, что запись каждого студента имеет фиксированный размер. Какая из трех схем размещения (непрерывная, связанная или табличная индексированная) будет наиболее подходящей?
18. Представьте себе файл, чей размер варьируется за время его существования между 4 Кбайт и 4 Мбайт. Какая из трех схем размещения (непрерывная, связанная или табличная индексированная) будет для него наиболее подходящей?
19. Поступило предложение для увеличения эффективности использования и экономии дискового пространства хранить данные коротких файлов прямо в i-узлах. Сколько байтов данных может храниться в i-узле, показанном на рис. 4.10?
20. Две изучающие информатику студентки, Кэролин и Элинор, обсуждают i-узлы. Кэролин утверждает, что оперативная память стала настолько большой по объему и настолько дешевой, что при открытии файла проще и быстрее считать новую копию i-узла в таблицу i-узлов, чем искать этот i-узел по всей таблице. Элинор не согласна. Кто из них прав?
21. Назовите одно преимущество жестких связей над символическими ссылками и одно преимущество символических ссылок над жесткими связями.
22. Объясните, чем жесткие ссылки отличаются от символьных ссылок в соответствии с распределениями i-узлов.
23. Возьмем диск объемом 4 Тбайт, который использует блоки размером 4 Кбайт и метод списка свободных блоков. Сколько адресов блоков может храниться в одном блоке?
24. Учет свободного дискового пространства может осуществляться с помощью связанных списков или битовых массивов. Дисковые адреса состоят из D бит. При каком условии для диска из B блоков, F из которых свободны, список свободных блоков займет меньше места, чем битовый массив? Выразите ваш ответ в процентах от дискового пространства, которое должно быть свободным, для $D = 16$ бит.
25. После первого форматирования дискового раздела начало битового массива учета свободных блоков выглядит так: 1000 0000 0000 0000 (первый блок используется для корневого каталога). Система всегда ищет свободные блоки от начала раздела, поэтому после записи файла A, занимающего 6 блоков, битовый массив принимает следующий вид: 1111 1110 0000 0000. Покажите, как будет выглядеть битовый массив после каждого из следующих действий:

    а) записи файла B размером 5 блоков;

    б) удаления файла A;

    в) записи файла C размером 8 блоков;

    г) удаления файла B.
26. Что произойдет, если битовый массив или список свободных блоков окажется полностью потерян в результате сбоя? Есть ли способ восстановления системы после такого сбоя или с диском можно уже попрощаться? Дайте ответ отдельно для файловых систем UNIX и FAT-16.

27. Ночная работа Оливера «Совы» в университете состоит в смене лент, используемых для архивации данных. Ожидая окончания записи на каждую ленту, Оливер пишет статью, в которой доказывает, что пьесы Шекспира были созданы инопланетными пришельцами. За неимением ничего другого его текстовый процессор работает прямо в архивируемой системе. Возникают ли проблемы, связанные с этими обстоятельствами?

28. В этой главе рассматривались некоторые аспекты инкрементного архивирования. В системе Windows вопрос о том, нужно ли архивировать файл, решить довольно просто, поскольку у каждого файла есть специальный архивный бит. А вот в системе UNIX такого бита нет. Как программа архивации в системе UNIX определяет, для какого файла следует создать резервную копию?

29. Допустим, что файл 21 на рис. 4.22 не изменялся со времени последней архивации. Как при этом изменятся четыре битовых массива на рис. 4.23?

30. Было внесено предложение, чтобы первая часть каждого файла в UNIX хранилась в том же дисковом блоке, что и его i-узел. В чем польза этого предложения?

31. Посмотрите на рис. 4.23. Может ли быть такое, чтобы для какого-то конкретного номера блока счетчики в *обоих* списках имели значение 2? Как избавиться от этой проблемы?

32. Производительность файловой системы зависит от степени удачных обращений к кэшу (доли блоков, найденных в кэше). Если на удовлетворение запроса к кэшу уходит 1 мс, а на удовлетворение запроса к диску, если потребуется чтение с диска, — 40 мс, выведите формулу для вычисления среднего времени удовлетворения запроса, если степень удачных обращений равна h. Нарисуйте график этой функции для значений h от 0 до 1,0.

33. Что больше подойдет для внешнего жесткого USB-диска, подключенного к компьютеру, — кэш со сквозной записью или блочное кэширование?

34. Рассмотрим приложение, в котором записи студентов сохраняются в файле. Приложение получает в качестве ввода идентификатор студента, а затем последовательно проводит чтение, обновление и запись соответствующей записи студента, и все это повторяется до завершения работы приложения. Пригодится ли здесь технология опережающего чтения блока?

35. Предположим, что имеется диск, у которого есть 10 блоков данных, начиная с блока 14 и заканчивая блоком 23. Пускай на диске будет два файла, f1 и f2. В структуре каталога показано, что первыми блоками данных f1 и f2 являются соответственно 22 и 16. Используя показанные далее записи таблицы FAT, покажите, какие блоки данных выделены f1 и f2.

    (14,18); (15,17); (16,23); (17,21); (18,20); (19,15); (20, −1); (21, −1); (22,19); (23,14).

    Показанная выше система записи (x,y) означает, что значение, сохраненное в записи таблицы x, указывает на блок данных y.

36. Воспользуйтесь графиком на рис. 4.17 применительно к диску, имеющему среднее время поиска цилиндра 6 мс, скорость вращения диска 15 000 об/мин и дорожки по 1 048 576 байт. Чему равна скорость передачи данных для блоков, имеющих размер 1, 2 и 4 Кбайт?

37. Некая файловая система использует 4-килобайтные дисковые блоки. Средний размер файлов составляет 1 Кбайт. Если бы все файлы имели размер 1 Кбайт, какая часть диска терялась бы понапрасну? Как вы думаете, потери в реальной системе выше этого числа или ниже? Обоснуйте ответ.

38. Какой самый большой размер файла (в байтах) может быть доступен с использованием 10 прямых адресов и одного косвенного блока, если размер дискового блока составляет 4 Кбайт, а значение адреса указателя блока составляет 4 байта?

39. В системе MS-DOS файлам приходится бороться за место в таблице FAT-16, хранящейся в оперативной памяти. Если один файл использует k элементов, то существуют k элементов, недоступных никаким другим файлам. Какие ограничения это накладывает на общую длину всех остальных файлов?

40. Файловая система UNIX имеет блоки размером 4 Кбайт и 4-байтовые дисковые адреса. Чему равен максимальный размер файла, если i-узлы содержат 10 прямых адресов и по одному из адресов однократного, двукратного и трехкратного косвенных блоков?

41. Сколько понадобится дисковых операций для считывания i-узла файла /usr/ast/courses/os/handout.t? Предположим, что кроме i-узла корневого каталога в оперативной памяти больше нет ничего относящегося к этому пути. Предположим также, что все каталоги занимают по одному дисковому блоку.

42. Во многих версиях системы UNIX i-узлы хранятся в начале диска. Альтернативный вариант предусматривает выделение i-узла при создании файла и помещение его в начале первого блока файла. Приведите все аргументы за и против альтернативного варианта.

43. Напишите программу, меняющую порядок байтов в файле на обратный так, чтобы последний байт стал первым, а первый — последним. Программа должна работать с файлами произвольной длины, но постарайтесь добиться от нее эффективной работы.

44. Напишите программу, которая начинает работу в заданном каталоге и спускается по дереву каталогов, записывая размеры всех найденных ею файлов. Когда программа выполнит эту задачу, она должна вывести на печать гистограмму размеров файлов, используя в качестве параметра шаг столбца (например, при шаге 1024 файлы размером от 0 до 1023 попадают в один столбец, от 1024 до 2047 — в другой столбец и т. д.).

45. Напишите программу, сканирующую все каталоги файловой системы UNIX, отыскивающую и определяющую местонахождение всех i-узлов с двумя и более жесткими связями. Для каждого файла, фигурирующего в жестких связях, программа должна собрать в единый список все имена файлов, указывающих на этот файл.

46. Напишите новую версию программы ls для системы UNIX. Эта версия должна принимать в качестве аргумента одно или несколько имен каталогов и для каждого каталога выдавать список всех файлов, содержащихся в этом каталоге, по одной строке на файл. Каждое поле должно форматироваться в соответствии с его типом. Укажите в списке только первый дисковый адрес или не указывайте вообще никаких адресов.

47. Создайте программу для измерения влияния размеров буфера на уровне приложения на процесс чтения. Эта программа использует запись в большой файл

(скажем, размером 2 Гбайт) и чтение из него. Также в ней изменяется размер буфера приложения (скажем, от 64 байт до 4 Кбайт). Используйте процедуры измерения времени (например, *gettimeofday* и *getitimer* в UNIX), чтобы измерить время, затрачиваемое при различных размерах буфера. Проанализируйте результаты и сообщите о своих изысканиях: вносит ли размер буфера разницу в общее время записи и во время каждой записи?

48. Создайте смоделированную файловую систему, которая полностью помещалась бы в отдельный обычный файл, сохраненный на диске. В этом дисковом файле должны содержаться каталоги, i-узлы, информация о свободных блоках, блоки файловых данных и т. д. Выберите подходящий алгоритм для сохранения информации о свободных блоках и для размещения блоков данных (непрерывного, индексированного, связанного). Ваша программа должна воспринимать поступающие от пользователя системные команды на создание и удаление каталогов, создание, удаление и открытие файлов, чтение выбранного файла и записи его на диск, а также вывод содержимого каталога.