## Управление файловой системой и ее оптимизация

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

### 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.

#### 1.1. 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, снижающих 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 Э́ндрю Стюартом Таненба́умом на кафедре информатики крупного исследовательского университета (VU, Амстердамский свободный университет) в 1984 году, а затем повторно изучен в 2005 году, исследовался также коммерческий веб-сервер, предоставляющий хостинг политическому веб-сайту ([www.](http://www/)electoral-vote.com). Результаты показаны в табл.1, 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.

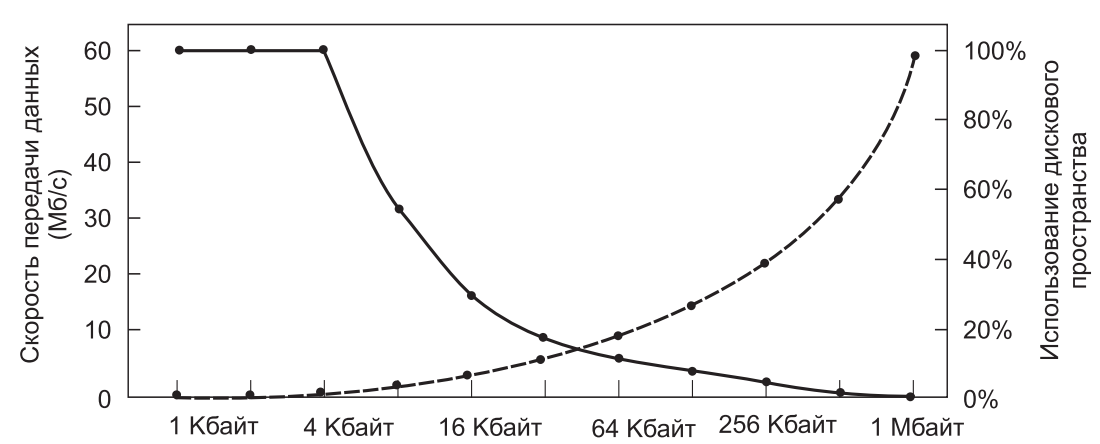
**Таблица 1.** Процент файлов меньше заданного размера

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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) 8,33.

Пунктирная кривая на рис.1 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 (Амстердамском свободном университете), 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.1 показывает зависимость эффективности использования дискового пространства от размера блока.

**Рис.1.** 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:

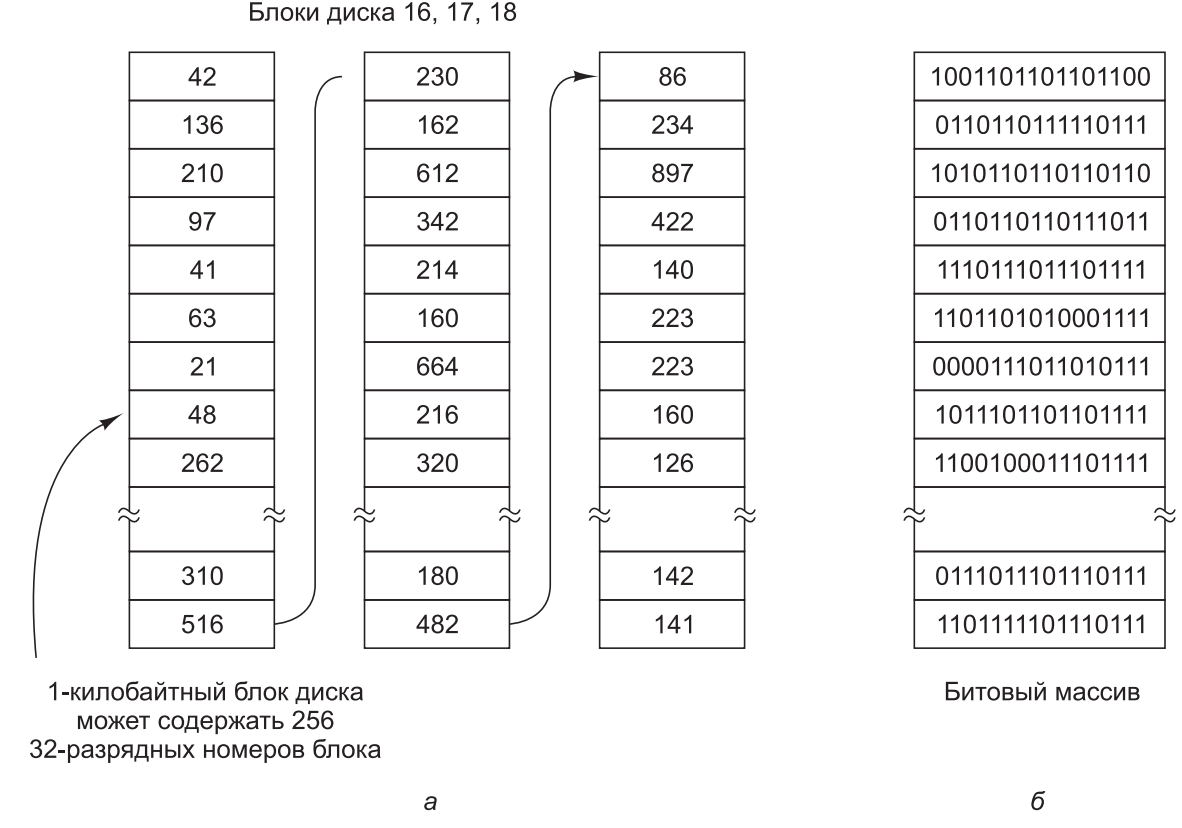
«Набор в Блокноте нескольких символов с последующим сохранением в файле приводит к 26 системным вызовам, включая 3 неудачные попытки открытия файла, 1 переписывание файла и 4 дополнительные последовательности его открытия и закрытия».

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.

#### 1.2. Отслеживание свободных блоков

После выбора размера блока возникает следующий вопрос: как отслеживать свободные блоки? На рис.2 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.2.** Хранение сведений о свободных блоках:

*а* — в связанном списке;

*б* — в битовом массиве

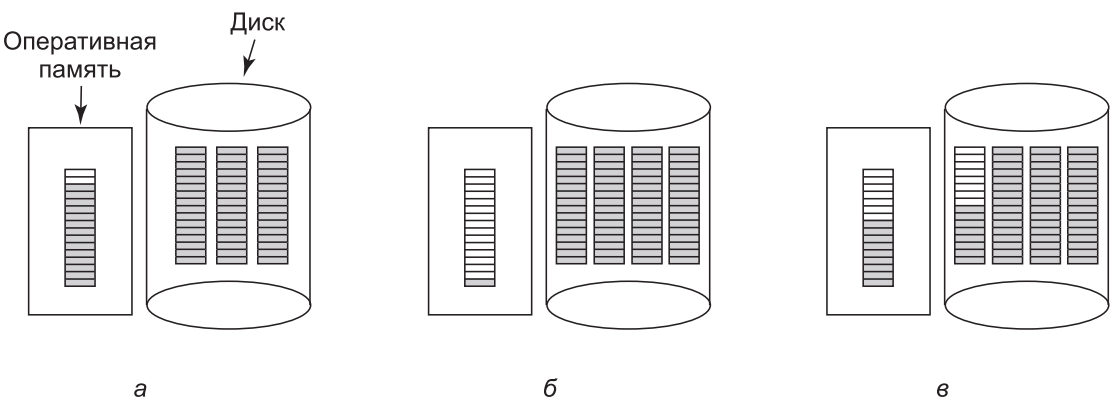
Если свободные блоки выстраиваются в длинные непрерывные последовательности блоков, система, использующая список свободных блоков, может быть модифицирована на отслеживание последовательности блоков, а не отдельных блоков. С каждым блоком, дающим номер последовательных свободных блоков, может быть связан 8-, 16или 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.3,*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.3, *б*. Если теперь записывается файл из трех блоков, опять должен быть считан полный блок указателей, возвращающий нас к ситуации, изображенной на рис.3, *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.3, *а*, к ситуации, проиллюстрированной на рис.3, *б*, мы переходим от ситуации, показанной на рис.3, *а*, к ситуации, которую видим на рис.3, *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 наполовину заполненный 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 опустошении хранящегося в памяти блока.

**Рис.3.** 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

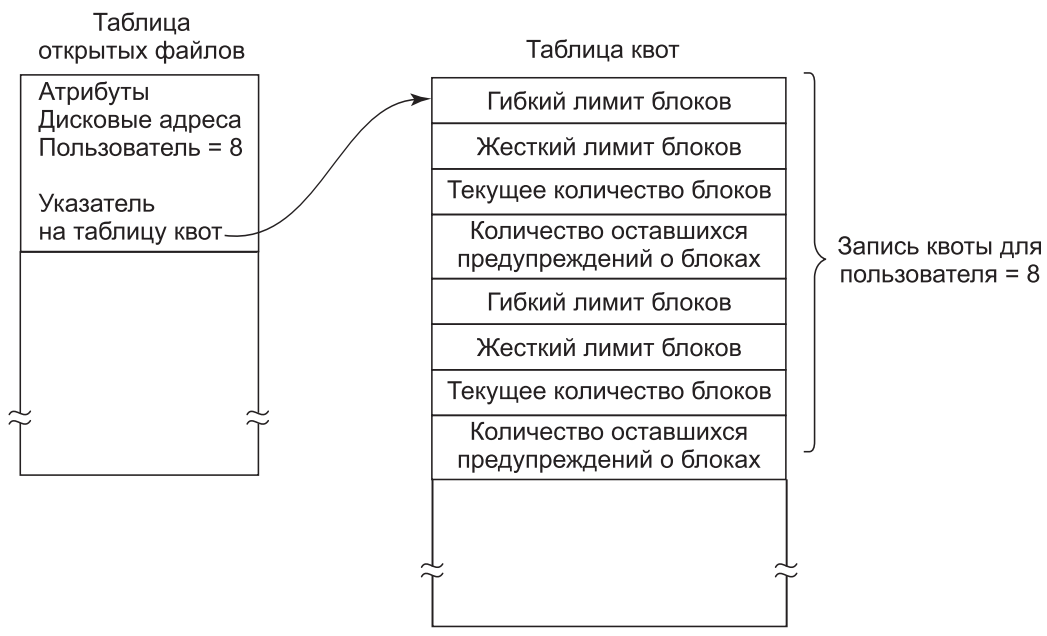
Дополнительное преимущество такого подхода состоит в том, что при осуществлении всех распределений из одного блока битового массива дисковые блоки будут находиться близко друг к другу, сводя к минимуму перемещения блока головок. Поскольку битовый массив относится к структурам данных с фиксированным размером,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

#### 1.3. 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. 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.** 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.

### 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. 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.



Рис.5. Ленточный накопитель IBM TS1150 Физическая емкость до 10 ТБ с картриджем IBM 3592 JD/JZ

Рис.6. Кассеты магнитной ленты IBM 3592. Вес: 239 г. Оценочный срок сохранности архива: до 30 лет. Загрузка: 20000 циклов загрузки, выгрузки и инициализации. Размеры: 24,5 мм высота × 109 мм ширина × 125 мм длина. Длина магнитной ленты: JD/JZ: 1072 м; JK: 146 м; JL: 281 м; JJ/JR: 246 м; JB/JX: 825 м; JC/JY: 880 м; JA/JW: 610 м

Несмотря на это создание резервной копии — не такое уж простое дело, поэтому далее мы рассмотрим ряд вопросов, связанных с этим процессом.

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).

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 нечитаемые.

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 находящихся в эксплуатации 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.7). 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.7.** 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



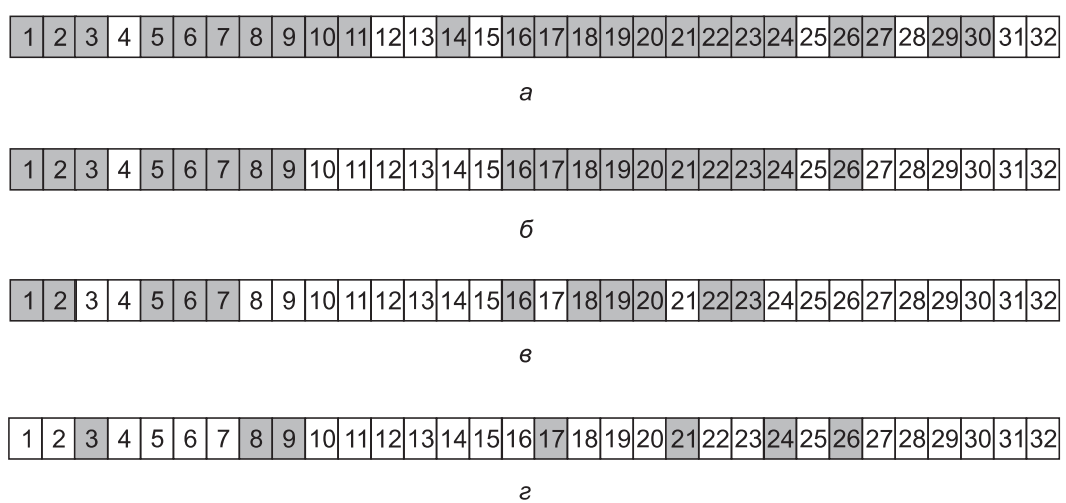
Согласно алгоритму архивации создается битовый массив, проиндексированный по номеру i-узла, в котором на каждый i-узел отводится несколько битов. При реализации алгоритма эти биты будут устанавливаться и сбрасываться. Реализация проходит в четыре этапа. Первый этап начинается с исследования всех элементов 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.8, *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.8,*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.8, *б*. Третий этап состоит из сканирования i-узлов в порядке их нумерации и архивирования всех каталогов, помеченных для архивации. Эти каталоги показаны на рис.8,*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.8,*г*, в префиксах которых также содержатся их атрибуты. На этом архивация завершается.



**Рис.8.** 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, 232 байт или, хуже того, 264 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).

### 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).

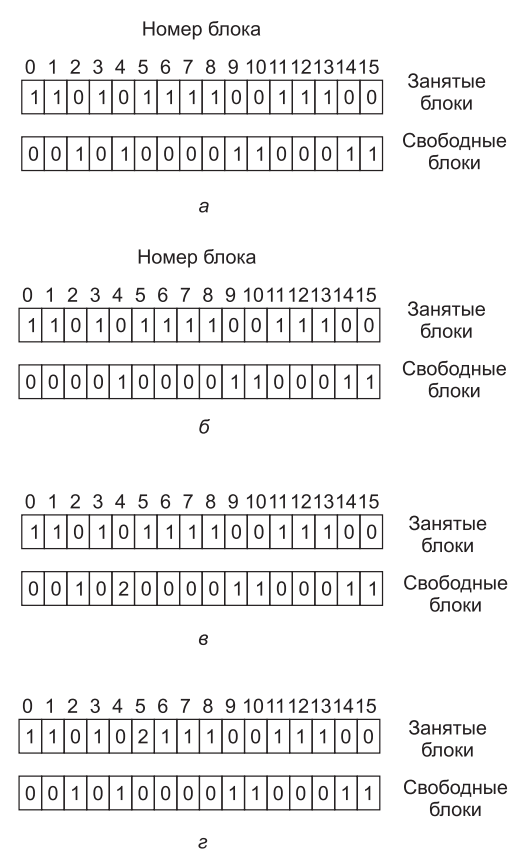
Затем программа считывает все i-узлы, используя непосредственно само устройство, при этом файловая структура игнорируется, и возвращаются все дисковые блоки, начиная с нулевого блока. Начиная с i-узла можно построить список всех номеров блоков, используемых в соответствующем файле. Как только будет считан 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 (см. рис.9, *а*). Но в результате отказа таблицы могут принять вид, показанный на рис.9, *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.

Другая возможная ситуация показана на рис.9, 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.9, 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.

**Рис.9.** 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

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 никакого 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.

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.

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.

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.

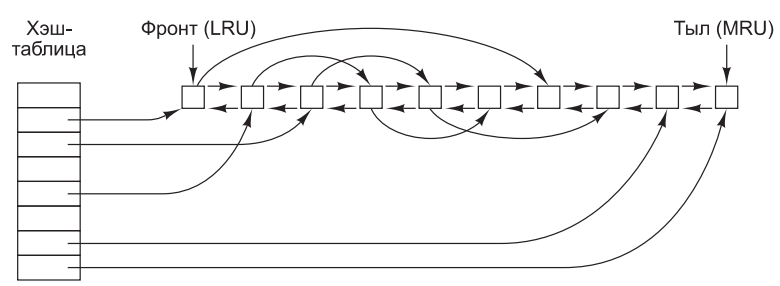
Разумеется, пока они не будут реально удалены из этого каталога, пространство запоминающего устройства возвращено не будет.

**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 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.10. 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.10.** 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.10 показано, что в дополнение к цепочкам коллизий, начинающихся в хэштаблице, 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.

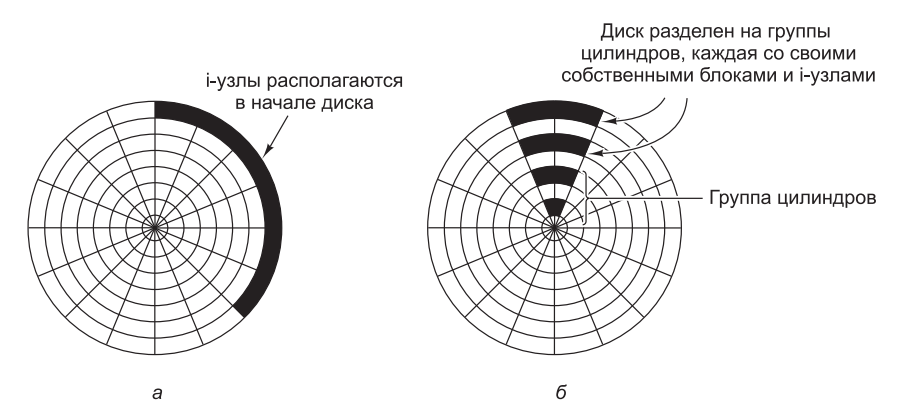
Разумеется, стратегия опережающего чтения работает только для тех файлов, которые считываются последовательно. При произвольном обращении к файлу опережающее чтение не поможет. Будет досадно, если скорость передачи данных с диска снизится из-за чтения бесполезных блоков и ради них придется удалять полезные 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. Передача данных с диска будет вестись блоками размером 1 Кбайт, но последовательное чтение файла на ни чем 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.11, *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.

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.11, *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.



**Рис.11.** Диск: *а* — i-узлы расположены в его начале; *б* — разделен на группы цилиндров, каждая со своими собственными блоками и i-узлами

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.

### 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.