**№ занятия: 9. Лекция № 7**

**Тема: Базовые файловые системы.**

**Вопрос 1. Типы файловых систем, поддерживаемых ОС МСВС.**

Операционная система МСВС разработана на платформе *RED HAT Linux*. Первоначально файловой системой в *Linux* была *MINIX 1*. Однако из-за того обстоятельства, что имена файлов были ограничены в ней 14 символами, а максимальный размер файла составлял 64 Мбайт, интерес к более совершенным файловым системам появился сразу же после начала разработки системы *Linux*.

Первым улучшением стала файловая система ext, которая позволяла использовать имена файлов длиной в 255 символов и размер файлов 2 Гбайт (однако она была медленнее, чем файловая система *MINIX 1*, так что поиски продолжались еще некоторое время). В итоге была изобретена файловая система *ext*2 (с длинными именами файлов, большими файлами и более высокой производительностью), которая и стала основной файловой системой. Однако Linux поддерживает несколько десятков файловых систем – при помощи уровня виртуальной файловой системы *Virtual File System* (*VFS*), описанного ниже.

Файл в ФС *ext*2 – это последовательность байтов произвольной длины (от 0 до некоторого максимума), содержащая произвольную информацию. Не делается различия между текстовыми (*ASCII*) файлами, двоичными файлами и любыми другими типами файлов. Значение битов в файле целиком определяется владельцем файла. Системе это безразлично. Имена файлов ограничены 255 символами.

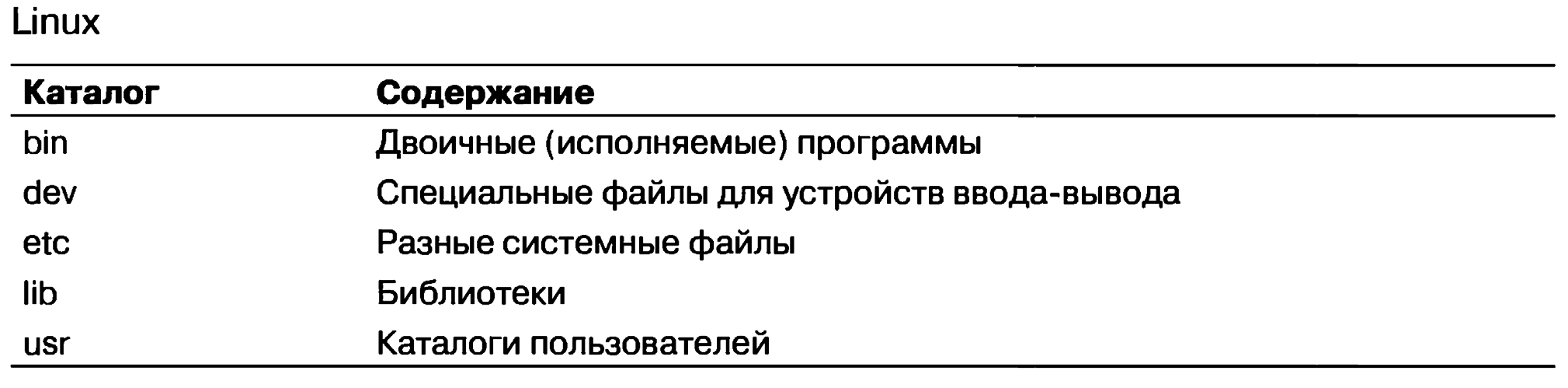
Для удобства файлы могут группироваться в каталоги. Каталоги хранятся на диске в виде файлов, и в значительной степени с ними можно работать как с файлами. Каталоги могут содержать подкаталоги, что приводит к иерархической файловой системе. Корневой каталог называется / и, как правило, содержит несколько подкаталогов. Символ / также используется для разделения имен каталогов, поэтому имя /*usr/ast/x* обозначает файл *х*, расположенный в каталоге *ast*, который в свою очередь находится в каталоге *usr*. Некоторые основные каталоги (находящиеся у вершины дерева каталогов) показаны в табл.1.

Существует два способа задания имени файла в системе *Linux*. Первый способ заключается в использовании абсолютного пути (*absolute path*), указывающего, как найти файл от корневого каталога. Пример абсолютного пути: */usr/ast/books/mos3/chap-10*. Он сообщает системе, что в корневом каталоге следует найти каталог usr, затем в нем найти каталог *ast*, который содержит каталог *books*, в котором содержится каталог mos3, а в нем расположен файл *chap-10*.

Абсолютные имена путей часто бывают длинными и неудобными. По этой причине операционная система *Linux* позволяет пользователям и процессам обозначить каталог, в котором они работают в данный момент, как рабочий каталог (*working directory*). Имена путей могут указываться относительно рабочего каталога. Путь, заданный относительно рабочего каталога, называется относительным путем (*relative path*).

*Таблица 1*

**Некоторые важные каталоги, существующие в большинстве систем *Linux***

****

Пользователям часто бывает необходимо обратиться к файлу, принадлежащему другому пользователю, или к своему файлу, расположенному в другом месте дерева файлов. Например, если два пользователя совместно используют один файл, то он будет находиться в каталоге, принадлежащем одному из них, – поэтому другому пользователю понадобится для обращения к этому файлу использовать абсолютное имя пути (или изменить свой рабочий каталог). Если абсолютный путь достаточно длинный, то необходимость вводить его каждый раз может весьма сильно раздражать. В системе *Linux* эта проблема решается при помощи так называемых ссылок (*link*), представляющих собой записи каталога, указывающие на существующие файлы.

В качестве примера рассмотрим ситуацию на рис.1,а. Фред и Лиза вместе работают над одним проектом, и каждому из них нужен доступ к файлам другого. Если рабочий каталог Фреда /*usr/jred*, то он может обращаться к файлу *х* в каталоге Лизы как /*usr/lisa/x*. Однако Фред может также создать новую запись в своем каталоге (рис.1,б), после чего он сможет обращаться к этому файлу просто как к *х*.

В только что обсуждавшемся примере, мы предположили, что до создания ссылки единственный способ, которым Фред мог обратиться к файлу *х* Лизы, заключался в использовании абсолютного пути.

В действительности это не совсем так. При создании каталога в нем автоматически создаются две записи, «.» и «..». Первая запись обозначает сам каталог. Вторая является ссылкой на родительский каталог, то есть каталог, в котором данный каталог числится как запись.

Таким образом, из каталога /*usr/fred* к файлу Лизы *х* можно обратиться еще и при помощи использования пути../*lisa/x*. Кроме обычных файлов, *Linux* также поддерживает символьные специальные файлы и блочные специальные файлы. Символьные специальные файлы используются для моделирования последовательных устройств ввода-вывода, таких как клавиатуры и принтеры. Если процесс откроет файл /*dev/tty* и прочитает из него, то он получит введенные с клавиатуры символы. Если открыть файл/*dev/lp* и записать в него данные, то эти данные будут распечатаны на принтере. Блочные специальные файлы (обычно с такими именами, как /*dev/hdl*) могут использоваться для чтения и записи необработанных дисковых разделов, минуя файловую систему. При этом поиск байта номер *k*, за которым последует чтение, приведет к чтению *k*-го байта из соответствующего дискового раздела, игнорируя *i*-узел и файловую структуру.

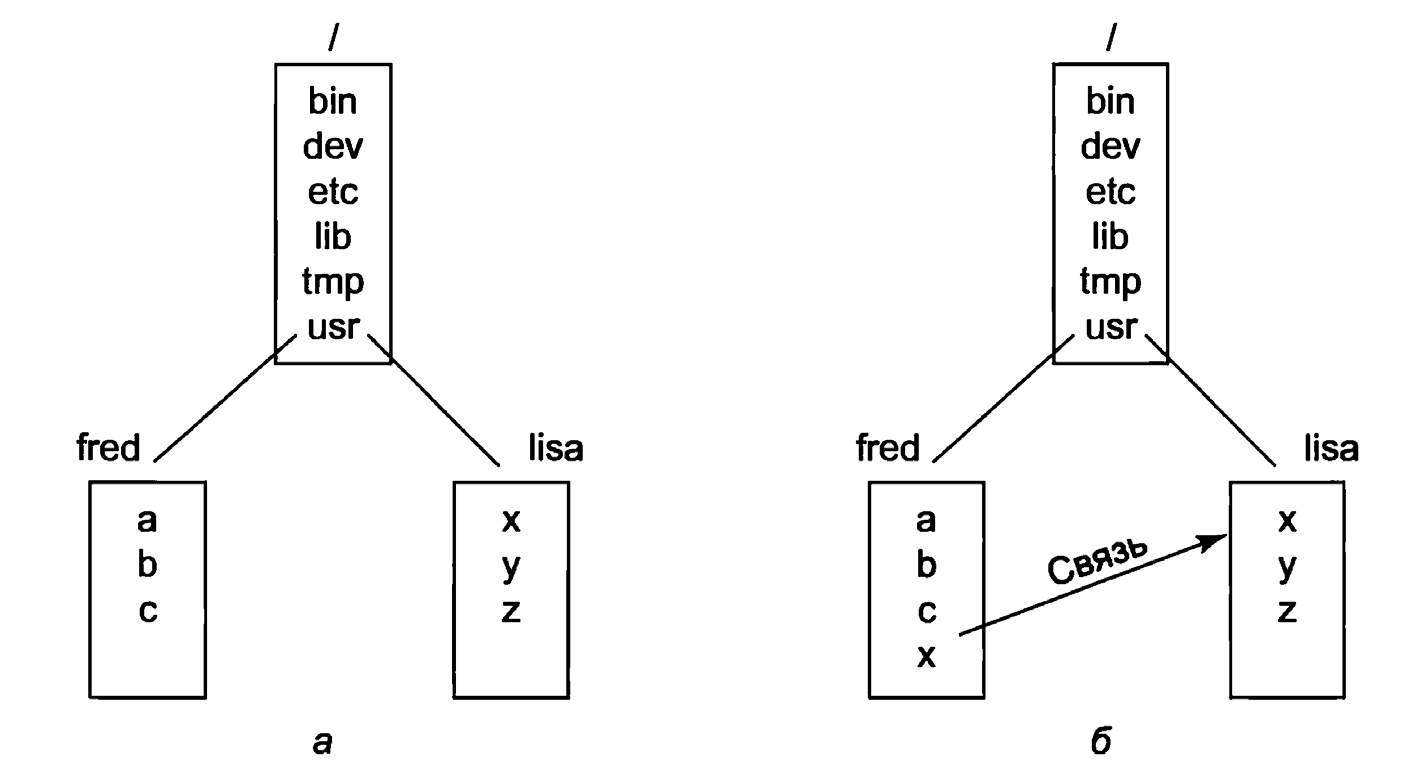


Рис.1. Примеры проектов: а - каталог создания ссылки; б – после создания ссылки

Необработанные блочные устройства используются для страничной подкачки и свопинга программами установки файловой системы (например, *mkfs*) и программами, исправляющими поврежденные файловые системы.

На многих компьютерах установлено по два и более жестких диска. Например, на мэйнфреймах в банках часто бывает необходимо иметь по сто и более дисков (чтобы хранить огромные базы данных). Даже у персональных компьютеров обычно есть, по меньшей мере, два диска – жесткий диск и дисковод для оптических дисков. При наличии у компьютера нескольких дисков возникает вопрос управления ими.

Одно из решений заключается в том, чтобы создать отдельную файловую систему на каждом диске и управлять ими по отдельности.

Например, рассмотрим ситуацию, изображенную на рис.2,а. На данном рисунке показан жесткий диск, который мы будем называть *С*:, а также *DVD*, который мы будем называть *D*:. У каждого есть собственный корневой каталог и файлы. При таком решении пользователь должен помимо каталогов указывать также и устройство (если оно отличается от используемого по умолчанию).

Например, чтобы скопировать файл *х* в каталог *d* (предполагая, что по умолчанию выбирается диск *С*:), следует ввести команду ср *D*:/*x /a/d/x*.

Такой подход применяется в операционных системах *Windows* и *VMS*. Применяемое в операционной системе *Linux* решение заключается в том, чтобы позволить смонтировать один диск в дерево файлов другого диска. В нашем примере мы можем смонтировать ДВД в каталог /*b*, получая в результате файловую систему, показанную на рис. 2,б. Теперь пользователь видит единое дерево файлов и уже не должен думать о том, какой файл на каком устройстве находится.

В результате приведенная выше команда примет вид *ср /b/x /a/d/x*, то есть все будет выглядеть так, как если бы файл копировался из одного каталога жесткого диска в другой каталог того же диска.

Другое интересное свойство файловой системы *Linux* – блокировка (*locking*). В некоторых приложениях два и более процессов могут одновременно использовать один и тот же файл, что может привести к условиям гонки. Одно из решений данной проблемы заключается в том, чтобы создать в приложении критические области. Однако если эти процессы принадлежат независимым пользователям, которые даже не знакомы друг с другом, то такой способ координации действий, как правило, очень неудобен.

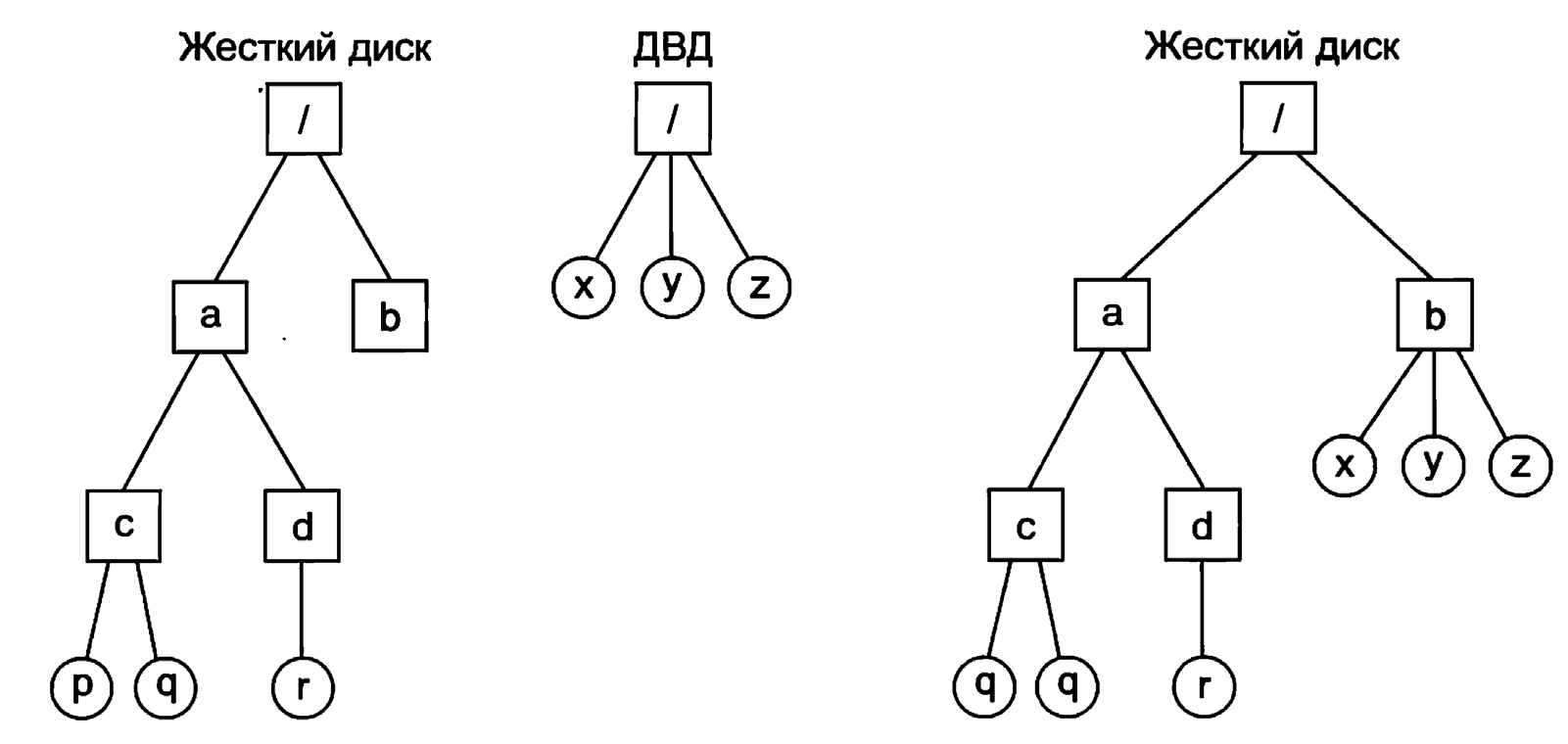


Рис. 2. Пример монтирования жесткого диска и DVD привода: а - раздельные файловые системы; б - после монтирования

Рассмотрим, например, базу данных, состоящую из многих файлов в одном или нескольких каталогах, доступ к которым выполняют никак не связанные между собой пользователи. С каждым каталогом или файлом можно связать семафор и достичь взаимного исключения, заставляя процессы выполнять операцию *down* на соответствующем семафоре перед доступом к данным. Недостаток такого решения заключается в том, что при этом недоступным становится весь каталог или файл – даже если нужна всего одна запись.

По этой причине стандарт *POSIX* предоставляет гибкий и детальный механизм, позволяющий процессам за одну неделимую операцию блокировать даже единственный байт файла (или целый файл). Механизм блокировки требует от вызывающей стороны указать блокируемый файл, начальный байт и количество байтов. Если операция завершается успешно, то система создает запись в таблице, в которой указывается, что определенные байты файла (например, запись базы данных) заблокированы.

Стандартом определены два типа блокировки: блокировка с монополизацией (*exclusive locks*) и блокировка без монополизации (*shared locks*). Если часть файла уже имеет блокировку без монополизации, то повторная попытка установка блокировки без монополизации на это место файла разрешается, но попытка установить блокировку с монополизацией будет отвергнута. Если же какая-либо область файла содержит блокировку с монополизацией, то любые попытки заблокировать любую часть этой области файла будут отвергаться, пока не будет снята блокировка. Для успешной установки блокировки необходимо, чтобы каждый байт в блокируемой области был доступен.

При установке блокировки процесс должен указать, будет ли он блокироваться в том случае, когда блокировку установить не удается. Если процесс выбрал блокирование, то он блокируется до тех пор, пока с запрашиваемой области файла не будет снята блокировка, после чего процесс активизируется и блокировка устанавливается. Если процесс решил не блокироваться при невозможности установить блокировку, то системный вызов немедленно делает возврат, а в коде состояния указывается, была блокировка успешной или нет. Если нет, то вызывающая сторона должна решить, что делать дальше (например, подождать и попробовать опять). Заблокированные области могут перекрываться.

На рис.3,а мы видим, что процесс *А* установил блокировку без монополизации на байты с 4-го по 7-й в некотором файле. Затем процесс *В* устанавливает блокировку без монополизации на байты с 6-го по 9-й (рис.3,б). Наконец, процесс *С* блокирует байты со 2-го по 11-й. Пока это блокировки без монополизации, они могут существовать одновременно. Теперь посмотрим, что произойдет, если процесс попытается получить блокировку с монополизацией на байт 9 (рис.3,в), блокируясь при неудаче блокировки.

Две предыдущие блокировки перекрываются с этой блокировкой. Поэтому вызывающая сторона будет заблокирована и останется заблокированной до тех пор, пока оба процесса (*В* и *С*) не снимут свои блокировки.

Многие системные вызовы имеют отношение к файлам и файловой системе. Сначала мы рассмотрим системные вызовы, работающие с отдельными файлами. Затем мы изучим те системные вызовы, которые оперируют каталогами или всей файловой системой в целом. Для создания нового файла можно использовать системный вызов creat (когда Кена Томпсона однажды спросили, что бы он поменял, если бы у него была возможность во второй раз разработать операционную систему *Unix*, он ответил, что на этот раз вместо creat он назвал бы этот системный вызов create).

В качестве параметров этому системному вызову следует задать имя файла и режим защиты. Системный вызов *creat* не только создает новый файл, но также и открывает его для записи. Чтобы последующие системные вызовы могли получить доступ к файлу, успешный системный вызов creat возвращает небольшое неотрицательное целое число, называемое дескриптором файла (*file descriptor*) (*fd* в приведенном выше примере).

Если системный вызов выполняется с уже существующим файлом, то длина этого файла уменьшается до 0, а все его содержимое теряется. Файлы можно также создавать при помощи вызова *open* с соответствующими аргументами.

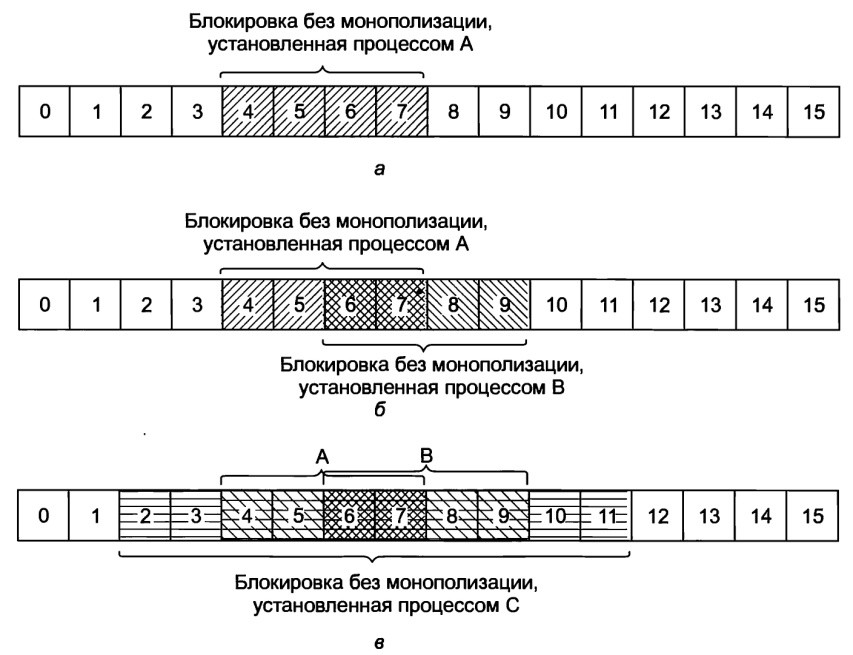
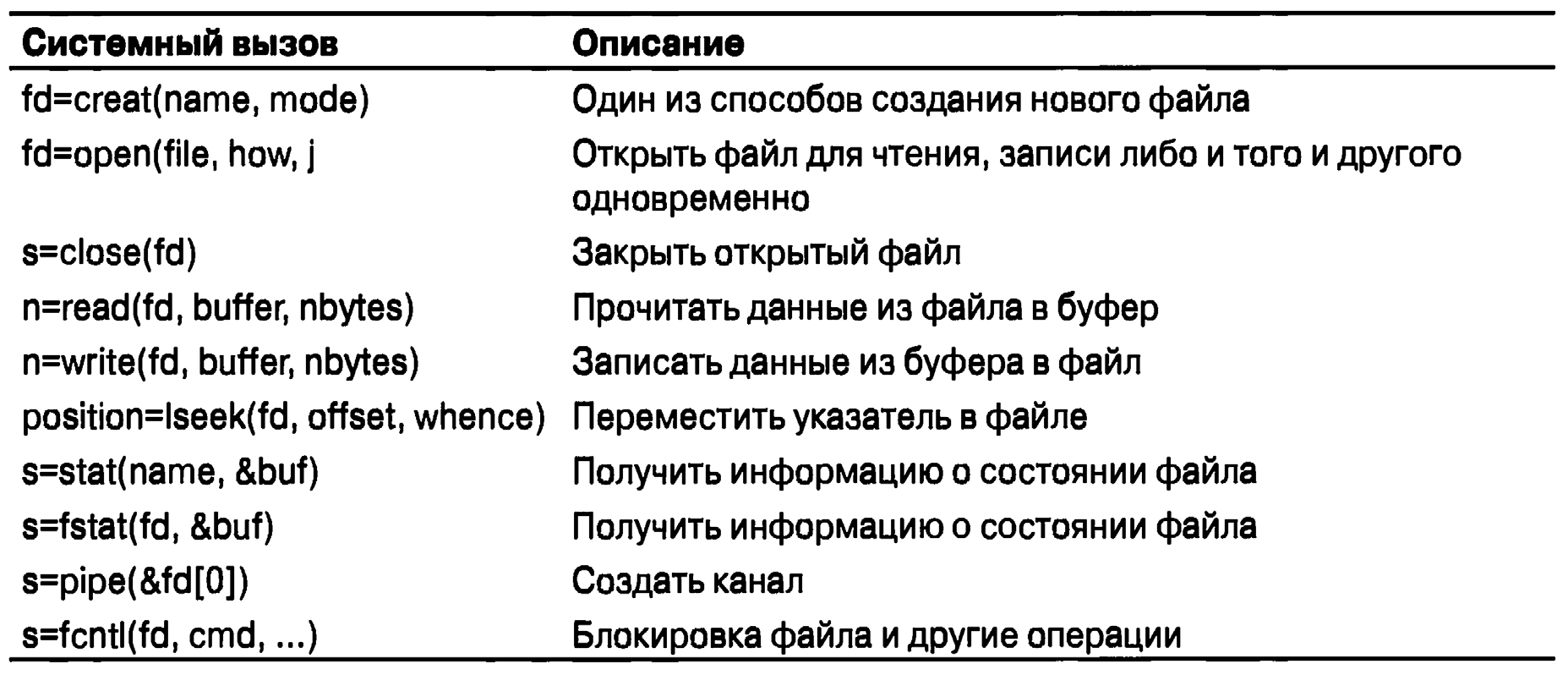


Рис. 3. Примеры блокировки процессов: а – файл с одной блокировкой; б – файл со второй блокировкой; в – третьей блокировкой

*Таблица 2*

****

Теперь продолжим изучение основных вызовов файловых систем, перечисленных в табл.2, чтобы прочитать данные из существующего файла или записать данные в существующий файл, его нужно сначала открыть с помощью open.

Этому системному вызову следует указать имя файла, а также режим, в котором он должен быть открыт: для чтения, для записи либо и для того и для другого. Также можно указать различные дополнительные параметры.

Как и *creat*, системный вызов open возвращает дескриптор файла, который может быть использован для чтения или записи.

Затем файл может быть закрыт при помощи вызова *close*, после чего дескриптор файла можно использовать повторно (для последующего *creat* или *open*). Системные вызовы *creat* и *open* всегда возвращают наименьший неиспользуемый в данный момент дескриптор файла.

**Некоторые системные вызовы для работы с файлами.** В случае ошибки возвращаемое значение s равно -1, fd – дескриптор файла, position – смещение в файле. Параметры должны быть понятны без пояснений. Когда программа начинает выполнение стандартным образом, файловые дескрипторы 0, 1 и 2 уже открыты для стандартного ввода, стандартного вывода и стандартного потока сообщений об ошибках соответственно.

Таким образом, фильтр (например, программа *sort*) может просто читать свои входные данные из файла с дескриптором 0, а писать выходные данные в файл с дескриптором 1, не заботясь о том, что это за файлы. Работа этого механизма обеспечивается оболочкой, которая проверяет, чтобы эти дескрипторы соответствовали нужным файлам (прежде чем программа начнет свою работу).

Чаще всего программы используют системные вызовы *read* и *write*. У обоих вызовов по три параметра: дескриптор файла (указывающий, с каким из открытых файлов будет производиться операция чтения или записи), адрес буфера (сообщающий, куда положить данные или откуда их взять), а также счетчик (указывающий, сколько байтов следует передать).

Пример типичного вызова:

*n = read* (*fd. buffer, nbytes*).

Хотя большинство программ читают и записывают файлы последовательно, некоторым программам бывает необходимо иметь доступ к произвольной части файла. С каждым открытым файлом связан указатель, который обозначает текущую позицию в файле. При последовательном чтении (или записи) он указывает на следующий байт, который будет прочитан (или записан).

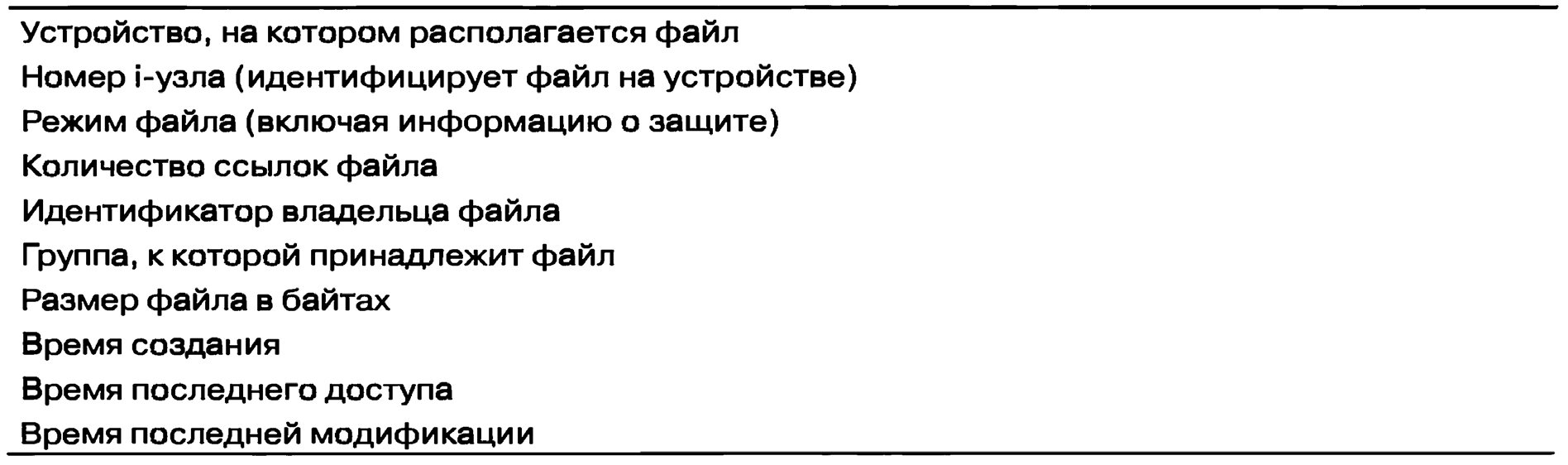
Например, если перед чтением 1024 байтов указатель был установлен на 4096-й байт, то после успешного системного вызова read он будет автоматически перемещен на 5120-й байт. Указатель в файле можно переместить с помощью системного вызова *Iseek*, что позволяет при последующих системных вызовах *read* (или *write*) читать данные из файла (или писать их в файл) в произвольной позиции файла и даже за концом файла.

Этот системный вызов назван *Iseek*, чтобы не путать его с теперь уже устаревшим, использовавшимся ранее на 16-разрядных компьютерах системным вызовом *seek*. У системного вызова *Iseek* три параметра: первый – это дескриптор файла, второй – позиция в файле, а третий сообщает, указывается ли эта позиция относительно начала файла, текущей позиции или конца файла. Возвращаемое системным вызовом *Iseek* значение представляет собой абсолютную позицию в файле после того, как указатель был изменен.

Забавно, что системный вызов *Iseek* – это единственный системный вызов из относящихся к файловой системе, который никогда не вызывает перемещения блока головок диска, так как все, что он делает – это обновление текущей позиции в файле, представляющей собой просто число в памяти.

Для каждого файла операционная система *Linux* хранит такие сведения, как тип (режим) файла (обычный, каталог, специальный файл), его размер, время последней модификации и другую информацию. Программы могут получить эту информацию при помощи системного вызова *stat*. Первый параметр представляет собой имя файла. Второй является указателем на структуру, в которую следует поместить запрошенную информацию. Поля этой структуры перечислены в табл.3.

*Таблица 3*

**Поля структуры, возвращаемой системным вызовом stat**

Системный вызов *fstat* – это то же самое, что и системный вызов *stat*, с той лишь разницей, что он работает с уже открытым файлом (имя которого может быть неизвестно), а не с путем.

Системный вызов *pipe* используется для создания каналов оболочки. Он создает псевдофайл для буферизации данных, которыми обмениваются компоненты канала, и возвращает дескрипторы файлов для чтения и записи буфера.

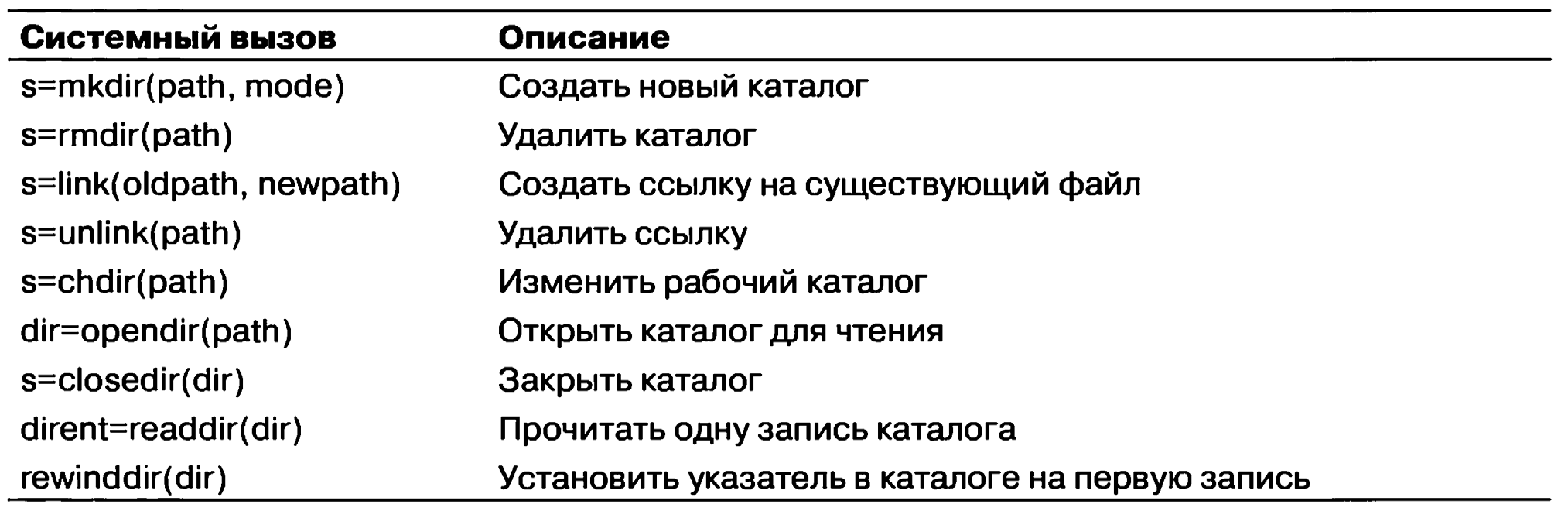
В канале *sort* <*in | head* -30 дескриптор файла 1 (стандартный вывод) в процессе, выполняющем программу *sort*, будет настроен оболочкой на запись в канал, а дескриптор файла 0 (стандартный ввод) в процессе, выполняющем программу *head*, будет настроен на чтение из канала. Программа *sort* просто читает из файла с дескриптором 0 (установлен на файл *in*) и пишет в файл с дескриптором 1 (канал), даже не зная о том, что оба эти файла перенаправлены. Если бы ввод и вывод не были перенаправлены, программа *sort* автоматически читала бы данные с клавиатуры и выводила бы их на экран (устройства по умолчанию). Подобным же образом, когда программа *head* считывает входные данные из файла с дескриптором 0, она получает данные, которые программа *sort* поместила в буфер канала (даже не зная о том, что используется канал). Вот хороший пример того, как простая концепция (перенаправление) плюс простая реализация (файлы с дескрипторами 0 и 1) дают мощный инструмент.

Последний системный вызов в табл.2 – это *fcntl* Он используется для блокировки и разблокирования файлов, а также некоторых других специфических для файлов операций.

Рассмотрим теперь некоторые системные вызовы, относящиеся скорее к каталогам или файловой системе в целом, нежели к одному конкретному файлу. Наиболее часто употребляемые системные вызовы перечислены в табл. 4.

*Таблица 4*

**Часто употребляемые системные вызовы**

****

Каталоги создаются и удаляются при помощи системных вызовов *mkdir*

и *rmdir* соответственно. Каталог может быть уничтожен только когда он пуст. Некоторые системные вызовы, относящиеся к работе с каталогами. В случае ошибки возвращаемое значение s равно -1; *of/'r* – идентифицирует каталог; a *dirent* – представляет собой запись каталога. Параметры должны быть понятны без пояснений.

Как было показано на рис.3, при создании ссылки на файл создается новая запись в каталоге, указывающая на существующий файл. Ссылка создается при помощи системного вызова *link*. В параметрах этого системного вызова указываются исходное и новое имя. Записи в каталоге удаляются системным вызовом *unlink*.

Когда удаляется последняя ссылка на файл, файл также автоматически удаляется. Если для файла не было создано ни одной ссылки, то при первом же обращении к системному вызову *unlink* файл исчезнет.

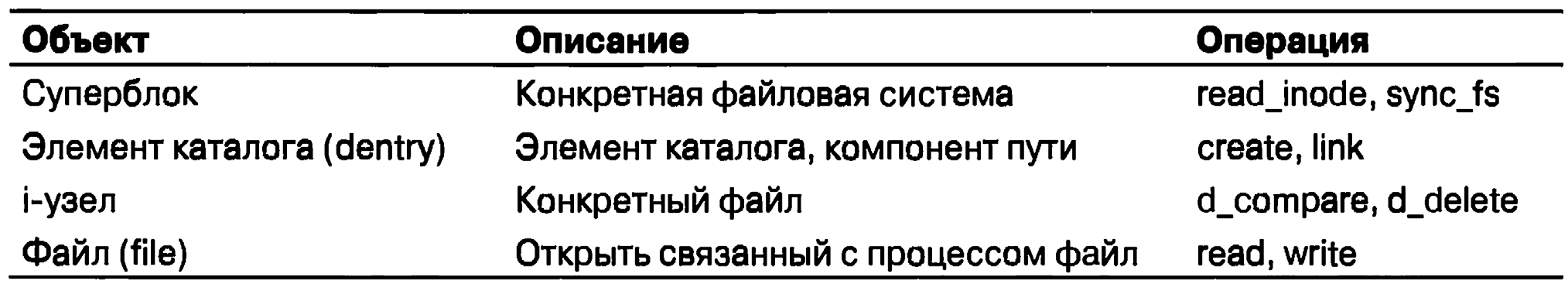
Рабочий каталог можно изменить при помощи системного вызова *chdir*. После выполнения этого системного вызова будут по-другому интерпретироваться относительные имена путей. Последние четыре системных вызова в табл.4 предназначены для чтения каталогов. Каталоги могут открываться, закрываться и читаться аналогично обычным файлам. Каждое обращение к системному вызову *readdir* возвращает ровно одну запись каталога (в фиксированном формате). Пользователям запрещено писать в каталоги (это делается, чтобы пользователи случайно не нарушили целостности системы). Файлы могут добавляться к каталогу при помощи системных вызовов creat и *link*, а удаляться с помощью системного вызова *unlink*. В операционной системе *Linux* нет способа перейти к конкретному файлу в каталоге, но есть системный вызов *rewinddir*, позволяющий начать читать открытый каталог с начала.

**Реализация файловой системы *Linux***. В этом разделе мы сначала рассмотрим поддерживаемые уровнем виртуальной файловой системы *Virtual File System* абстракции. *VFS* скрывает (от процессов и приложений верхнего уровня) отличия поддерживаемых в *Linux* файловых систем. Доступ к устройствам и другим специальным файлам также производится через уровень *VFS*. Затем мы опишем реализацию первой получившей широкое распространение файловой системы *Linux* под названием *ext*2 (вторая расширенная файловая система). После этого мы обсудим улучшения файловой системы *ext*3. Используется также множество других файловых систем. Все *Linux*-системы могут работать с большим количеством дисковых разделов, причем в каждом может быть своя файловая система.

**Виртуальная файловая система *Linux***. Для того чтобы приложения могли взаимодействовать с разными файловыми системами, реализованными на разных типах локальных или удаленных устройств, в *Linux* принят использованный в других *Unix*-системах подход – виртуальная файловая система *Virtual File System* (*VFS*). *VFS* определяет набор основных абстракций файловой системы и разрешенные с этими абстракциями операции. Описанные в предыдущем разделе системные вызовы обращаются к структурам данных *VFS*, определяют тип файловой системы (к которой принадлежит нужный файл) и при помощи хранящихся в структурах данных *VFS* указателей на функции запускают соответствующую операцию в указанной файловой системе.

*Таблица 5*

**Поддерживаемые в *VFS* абстракции файловой системы**

****

В табл.5 даны четыре основные структуры файловой системы, поддерживаемые *VFS*. Суперблок содержит критичную информацию о компоновке файловой системы. Разрушение суперблока делает файловую систему нечитаемой *i*-узел (сокращение от «индекс-узлы», никто их так не называет) описывает один файл. Обратите внимание, что в Linux каталоги и устройства также представлены файлами, так что они тоже имеют соответствующие *i*-узлы. И суперблок, и *i*-узлы имеют соответствующие структуры на том физическом диске, где находится файловая система.

Чтобы улучшить некоторые операции с каталогами и перемещение по путям (таким, как /*usr/ast/bin*), *VFS* поддерживает структуру данных *dentry*, которая представляет элемент каталога. Эта структура данных создается файловой системой «на ходу».

Элементы каталога кэшируются в *dentry\_cache*. Например, *dentry\_cache* будет содержать элементы для /, /*nsr, /usr/ast* и т. д. Если несколько процессов обращаются к одному и тому же файлу при помощи одной и той же жесткой ссылки (то есть одного и того же пути), то их объект файла будет указывать на один и тот же элемент в этом кэше.

И наконец, структура данных *file* является представлением открытого файла в памяти, она создается в ответ на системный вызов open. Она поддерживает такие операции, как *read*, *write*, *sendfile*, *lock* (и прочие описанные в предыдущем разделе системные вызовы).

Реализованные под уровнем VFS реальные файловые системы не обязаны использовать внутри себя точно такие же абстракции и операции. Однако они должны реализовать семантически эквивалентные операции файловой системы (такие же, как указанные для объектов *VFS*). Элементы структур данных operations для каждого из четырех объектов *VFS* – это указатели на функции в нижележащей файловой системе.

**Файловая система /ргос**. Еще одна файловая система *Linux* – это файловая система /*ргос* (*process* – процесс). Идея этой файловой системы изначально была реализована в 8-й редакции операционной системы *Unix*, созданной лабораторией *Bell Labs*, а позднее скопирована в версиях *4.4BSD* и *System V*. Однако в операционной системе *Linux* данная идея получила дальнейшее развитие. Основная концепция этой файловой системы заключается в том, что для каждого процесса системы создается каталог в каталоге /*ргос*. Имя каталога формируется из *PID* процесса в десятичном формате. Например, /*ргос*/*619* – это каталог, соответствующий процессу с *PID 619*.

В этом каталоге находятся файлы, которые хранят информацию о процессе – его командную строку, строки окружения и маски сигналов. В действительности этих файлов на диске нет. Когда они считываются, система получает информацию от реального процесса и возвращает ее в стандартном формате. Многие расширения, реализованные в операционной системе *Linux*, относятся к другим файлам и каталогам, расположенным в каталоге /*ргос*. Они содержат информацию о центральном процессоре, дисковых разделах, устройствах, векторах прерывания, счетчиках ядра, файловых системах, подгружаемых модулях и о многом другом. Непривилегированные программы пользователя могут читать большую часть этой информации, что позволяет им узнать о поведении системы (безопасным способом). Некоторые из этих файлов могут записываться в каталог */ргос*, чтобы изменить параметры системы.

**Файловая система *Ext*2 в *Linux***. Теперь мы опишем наиболее популярную дисковую файловую систему *Linux – ext*2. Первый выпуск *Linux* использовал файловую систему *MINIX*, которая имела короткие имена файлов и максимальный размер файла в 64 Мбайт. Файловая система *MINIX* была в итоге заменена первой расширенной файловой системой ext, которая позволяла использовать более длинные имена файлов и более крупные размеры файлов. Вследствие своей низкой эффективности (в смысле производительности) система ext была заменена своей последовательницей *ext*2, которая до сих пор широко используется.

Дисковый раздел с *ext*2 содержит файловую систему с показанной на рис.4 компоновкой.

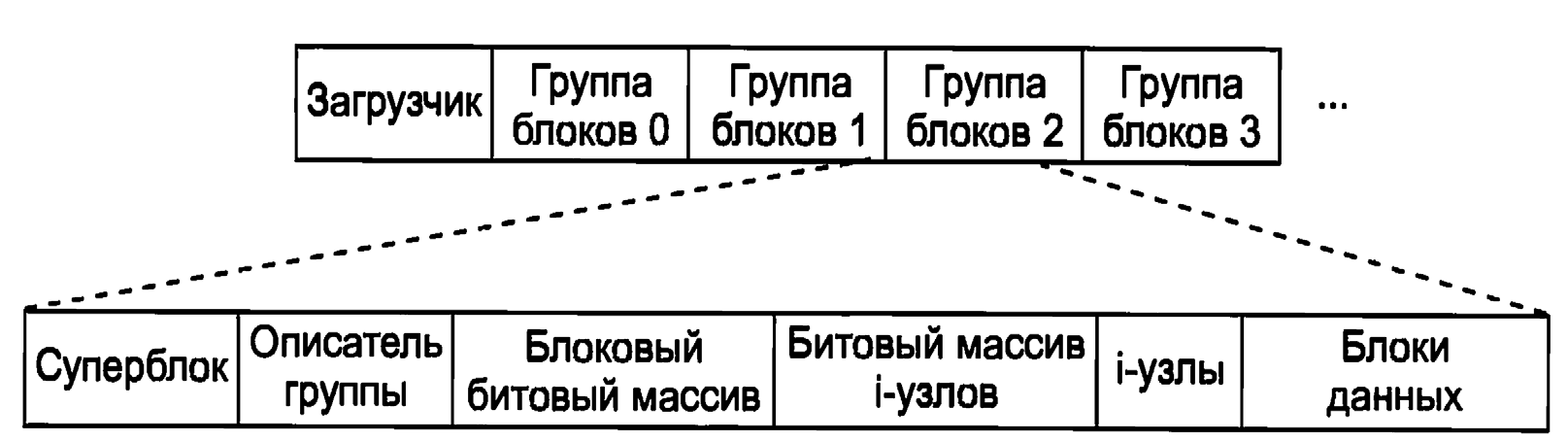


Рис.4. Размещение файловой системы ext2 на диске

Блок 0 не используется системой *Linux* и часто содержит код загрузки компьютера. Следом за блоком 0 дисковый раздел разделен на группы блоков. Каждая группа организована следующим образом.

Первый блок – это *суперблок* (*superblock*), в котором хранится информация о компоновке файловой системы, включая количество *i*-узлов, количество дисковых блоков, начало списка свободных дисковых блоков.

Затем следует *дескриптор группы*, содержащий информацию о расположении битовых массивов, количестве свободных блоков и *i*-узлов в группе, а также о количестве каталогов в группе. Эта информация важна, так как файловая система *ext*2 пытается распределить каталоги равномерно по всему диску. В двух битовых массивах ведется учет свободных блоков и свободных *i*-узлов (это тоже унаследовано из файловой системы *MINIX 1* и отличается от большинства файловых систем *Unix*, в которых для свободных используется список).

Размер каждого битового массива равен одному блоку. При размере блока в 1 Кбайт такая схема ограничивает размер группы блоков 8192 блоками и 8192 *i*узлами. Первое число является реальным ограничением, а второе – практически нет. Затем располагаются сами *i*-узлы. Они нумеруются от 1 до некоторого максимума. Размер каждого *i*-узла – 128 байт, и описывает он ровно один файл, iузел содержит учетную информацию (в том числе всю возвращаемую вызовом *stat*> который просто берет ее из *i*-узла), а также достаточное количество информации для определения местоположения всех дисковых блоков, которые содержат данные файла.

Следом за *i*-узлами идут *блоки данных*. Здесь хранятся все файлы и каталоги. Если файл или каталог состоит более чем из одного блока, то эти блоки не обязаны быть непрерывными на диске. В действительности блоки большого файла, скорее всего, будут разбросаны по всему диску. Соответствующие каталогам *i*-узлы разбросаны по всем группам дисковых блоков. *Ext*2 пытается расположить обычные файлы в той же самой группе блоков, что и родительский каталог, а файлы данных в том же блоке, что и *i*-узел исходного файла (при условии, что там имеется достаточно места). Эта идея была позаимствована из файловой системы *Berkeley Fast File System*. Битовые массивы используются для того, чтобы принимать быстрые решения относительно выделения места для новых данных файловой системы. Когда выделяются новые блоки файлов, то *ext*2 также делает упреждающее выделение (*preallocates*) нескольких (восьми) дополнительных блоков для этого же файла.

Эта схема распределяет файловую систему по всему диску. Она также имеет хорошую производительность (благодаря ее тенденции к смежному расположению и пониженной фрагментации). Для доступа к файлу нужно сначала использовать один из системных вызовов *Linux* (такой, как *open)*, для которого нужно указать путь к файлу. Этот путь разбирается, и из него извлекаются составляющие его каталоги. Если указан относительный путь, то поиск начинается с текущего каталога процесса, в противном случае – с корневого каталога.

В любом случае, *i*-узел для первого каталога найти легко: в дескрипторе процесса есть указатель на него либо (в случае корневого каталога) он обычно хранится в определенном блоке на диске. Каталог позволяет использовать имена файлов до 255 символов (показан на рис.5). В каталоге элементы для файлов и каталогов находятся в несортированном порядке. Элементы не могут пересекать границы блоков, поэтому в конце каждого дискового блока обычно имеется некоторое количество неиспользуемых байтов.

Каждая запись каталога на рис.5 состоит из четырех полей фиксированной длины и одного поля переменной длины. Первое поле представляет собой номер *i*-узла, равный 19 для файла *colossal*, 42 для файла *voluminous* и 88 для каталога *bigdir*. Следом идет поле *rес\_1еп*, сообщающее размер всей записи каталога в байтах (возможно, вместе с дополнительными байтами заполнителями после имени). Это поле необходимо, чтобы найти следующую запись.

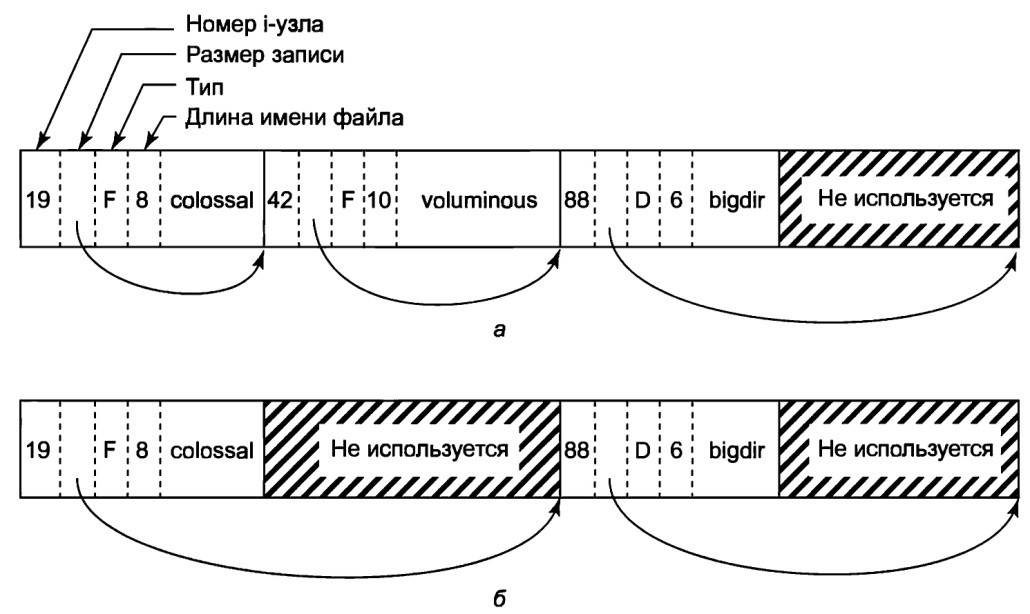


Рис.5. Каталог *Linux* с тремя файлами (а); каталог *Linux* после удаления файла *voluminous* (б)

На рисунке это поле обозначено стрелкой. Затем располагается поле типа: файл, каталог и т. д. Последнее поле фиксированной длины содержит длину имени файла в байтах (8, 10 и 6 для данного примера). Наконец, идет само имя файла, заканчивающееся нулевым байтом и дополненное до 32-битной границы. За ним могут следовать дополнительные байты-заполнители.

На рис.5,б показан тот же самый каталог после того, как элемент для *voluminous* был удален. Все, что при этом делается в каталоге – увеличивается число в поле размера записи предыдущего файла *colossal*, а байты записи каталога для удаленного файла *voluminous* превращаются в заполнители первой записи. Впоследствии эти байты могут использоваться для записи при создании нового файла.

Поскольку поиск в каталогах производится линейно, то поиск записи, которая находится в конце большого каталога, может занять много времени. Поэтому система поддерживает кэш каталогов, к которым недавно производился доступ. Поиск в кэше производится по имени файла – и если оно найдено, то дорогой линейный поиск уже не нужен. Объект dentry вводится в кэш элементов каталога для каждого из компонентов пути и (через его *i*-узел) делается поиск в каталоге последующих элементов пути (до тех пор, пока не будет найден фактический *i*-узел файла).

Например, чтобы найти файл, указанный абсолютным путем (таким, как ***/usr/ast/file***), необходимо выполнить следующие шаги. Прежде всего, система находит корневой каталог, который обычно использует *i*-узел с номером 2 (особенно когда *i*-узел с номером 1 зарезервирован для работы с плохими блоками). Она помещает в кэш элементов каталога соответствующий элемент (для будущих поисков корневого каталога). Затем она ищет в корневом каталоге строку «*usr*», чтобы получить номер *i*-узла для каталога /*usr* (который также вносится в кэш элементов каталога). Этот *i*-узел затем читается, и из него извлекаются дисковые блоки – так что можно читать каталог /*usr* и искать в нем строку «*ast*». После того как соответствующий элемент найден, из него можно определить номер *i*-узла для каталога. Имея этот номер *i*-узла, его можно прочитать и найти блоки каталога. И наконец, мы ищем «*file*» и находим номер его *i*-узла.

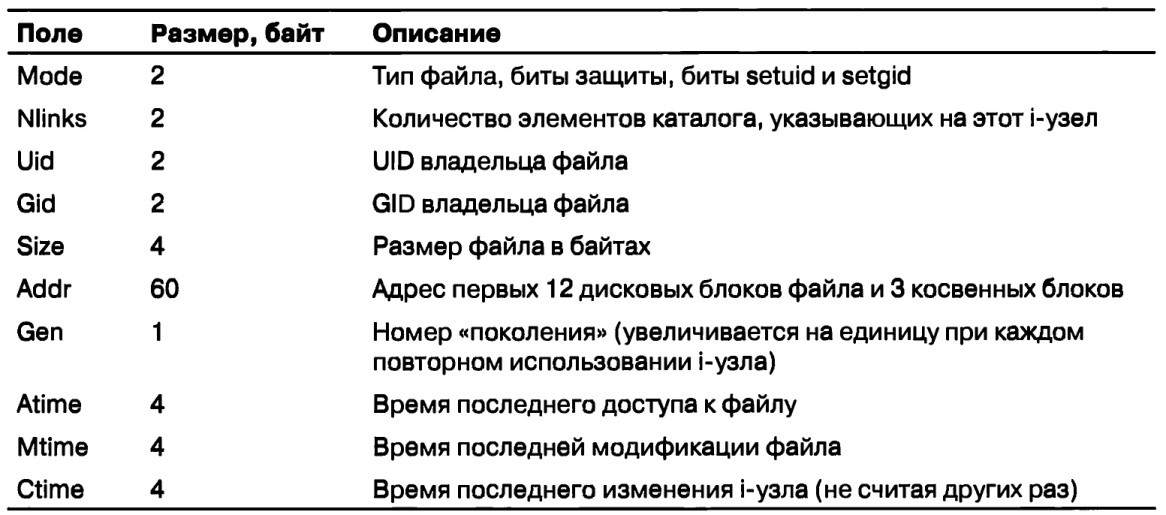
Таким образом , использование относительного пути не только более удобно для пользователя, но и сокращает количество работы для системы.

Если файл имеется в наличии, то система извлекает номер *i*-узла и использует его как индекс таблицы *i*-узлов (на диске) для поиска соответствующего *i*узла и считывания его в память. Этот *i*-узел помещается в таблицу *i*-узлов (*inode table*) – структуру данных ядра, которая содержит все *i*-узлы для открытых в данный момент файлов и каталогов. Формат элементов *i*-узлов должен содержать (как минимум) все поля, которые возвращает системный вызов *stat*, чтобы вызов *stat* мог работать.

В табл.6 показаны некоторые из полей структуры *i*-узла, поддерживаемой в файловой системе *Linux*. Реальная структура *i*-узла содержит гораздо больше полей, поскольку эта же структура используется и для представления каталогов, устройств и прочих специальных файлов.

Структура *i*-узла содержит также зарезервированные для будущего использования поля. История показала, что неиспользованные биты недолго остаются без дела. Теперь давайте посмотрим, как система читает файл. Вы помните, что типичный вызов библиотечной процедуры для запуска системного вызова *read* выглядит следующим образом: *n = read*(*fd, buffer, nbytes*).

*Таблица 6*

**Структура *i*-узла в *Linux***

Один из элементов этих внутренних таблиц – массив файловых дескрипторов. Он индексирован по файловым дескрипторам и содержит по одному элементу на каждый открытый файл (до некоторого максимального количества, по умолчанию это обычно 32).

Идея состоит в том, чтобы начать с этого дескриптора файла и закончить соответствующим *i*-узлом. Давайте рассмотрим одну вполне возможную схему: поместим указатель на *i*-узел в таблицу дескрипторов файлов. Несмотря на простоту, данный метод (к сожалению) не работает. Проблема заключается в следующем. С каждым дескриптором файла должен быть связан указатель в файле, определяющий тот байт в файле, с которого начнется следующая операция чтения или записи. Где следует хранить этот указатель? Один вариант состоит в размещении его в таблице *i*-узлов. Однако такой подход не сможет работать, если несколько не связанных друг с другом процессов одновременно откроют один и тот же файл – поскольку у каждого процесса должен быть свой собственный указатель.

Второй вариант решения заключается в размещении указателя в таблице дескрипторов файлов. При этом каждый открывающий файл процесс имеет свою собственную позицию в файле. К сожалению, такая схема также не работает, но причина неудачи в данном случае не столь очевидна и имеет отношение к природе совместного использования файлов в системе *Linux*. Рассмотрим сценарий оболочки s, состоящий из двух команд (*р*1 и *р*2), которые должны выполняться по очереди. Если сценарий вызывается командной строкой *S* > *Х*, то ожидается, что команда *р1* будет писать свои выходные данные в файл х, а затем команда *р*2 будет также писать свои выходные данные в файл *х*, начиная с того места, на котором остановилась команда *p1*.

Когда оболочка запустит процесс *р1* – файл *х* будет сначала пустым, поэтому команда *р1* просто начнет запись в файл в позиции 0. Однако когда *р1* закончит свою работу, то потребуется некий механизм, который гарантирует, что процесс *р2* увидит в качестве начальной позиции не 0, а то значение, на котором остановился *р1*.То, как это делается, показано на рис.6.

Фокус состоит в том, чтобы ввести новую таблицу – таблицу описания открытых файлов (*open file description table*) – между таблицей дескрипторов файлов и таблицей *i*-узлов и хранить в ней указатель в файле (а также бит чтения/записи). На рисунке родительским процессом является оболочка, а дочерним сначала является процесс *р1*, а затем процес *р2*. Когда оболочка создает процесс *р2*, то его пользовательская структура (включая таблицу дескрипторов файлов) представляет собой точную копию такой же структуры оболочки, поэтому обе они содержат указатели на одну и ту же таблицу описания открытых файлов. Когда процесс *р1* завершает свою работу, дескриптор файла оболочки продолжает указывать на таблицу описания открытых файлов, в которой содержится позиция процесса *р1* в файле. Когда теперь оболочка создает процесс *р2*, то новый дочерний процесс автоматически наследует позицию в файле – при этом ни новый процесс, ни оболочка даже не обязаны знать текущее значение этой позиции.

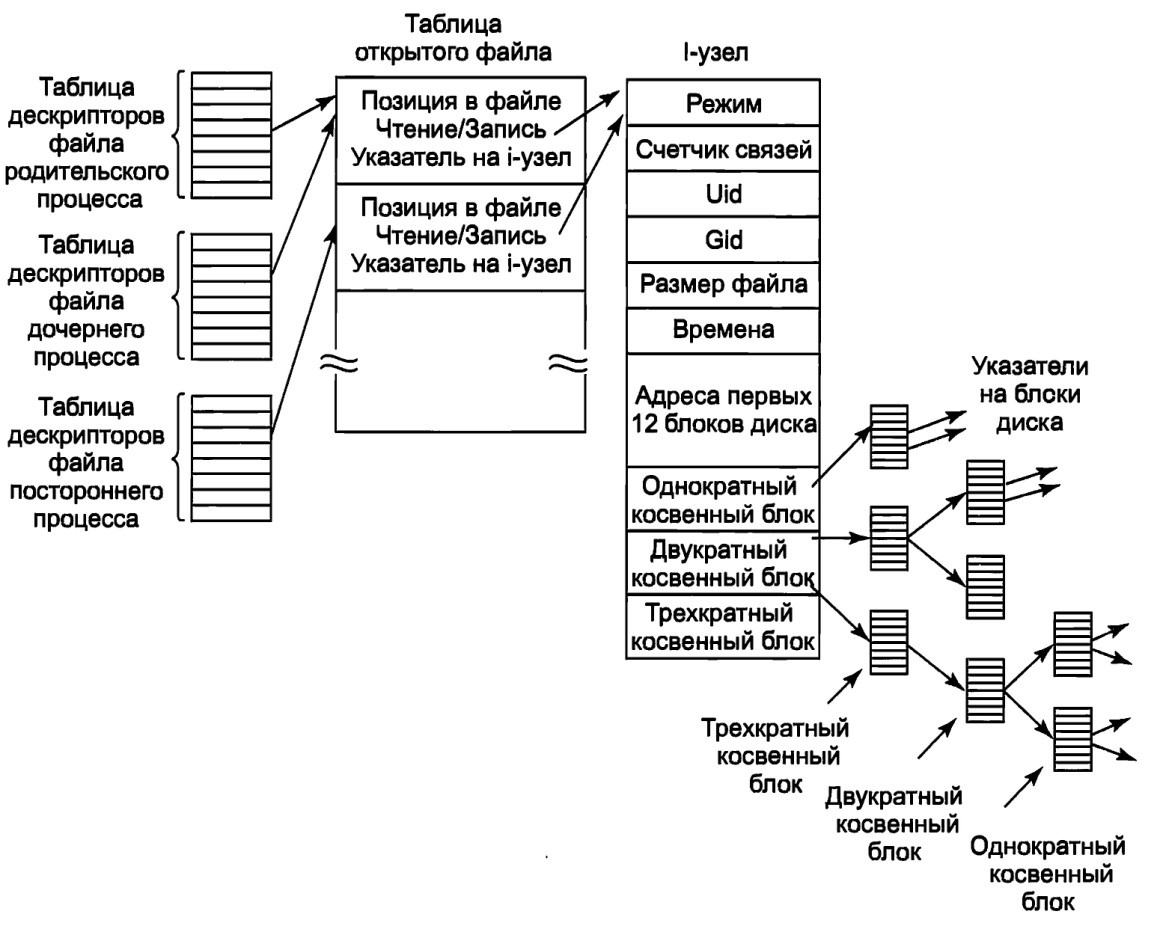


Рис.6. Связь между таблицей дескрипторов файлов, таблицей описания открытых файлов и таблицей *i*-узлов

Если какой-нибудь посторонний процесс откроет файл, то он получит свою собственную запись в таблице описания открытых файлов со своей позицией в файле – а именно это и нужно.

Таким образом, задача таблицы описания открытых файлов заключается в том, чтобы позволить родительскому и дочернему процессам совместно использовать один указатель в файле, но для посторонних процессов выделять персональные указатели.

Итак (возвращаясь к проблеме выполнения чтения *read*), мы показали, как определяются позиция в файле и *i*-узел. 1-узел содержит дисковые адреса первых 12 блоков файла. Если позиция в файле попадает в его первые 12 блоков, то считывается нужный блок файла и данные копируются пользователю. Для файлов, длина которых превышает 12 блоков, в *i*-узле содержится дисковый адрес одинарного косвенного блока (*single indirect block*). Этот блок содержит дисковые адреса дополнительных дисковых блоков.

Например, если размер блока составляет 1 Кбайт, а дисковый адрес занимает 4 байта, то одинарный косвенный блок может хранить до 256 дисковых адресов. Такая схема позволяет поддержать файлы размером до 268 Кбайт.

Для более крупных размеров используется двойной косвенный блок (*double indirect block*). Он содержит адреса 256 одинарных косвенных блоков, каждый из которых содержит адреса 256 блоков данных. Такая схема позволяет поддерживать файлы размером до 10+216 блоков F7 119 104 байт). Если и этого оказывается недостаточно, то в *i*-узле есть место для тройного косвенного блока (*triple indirect block*). Его указатели показывают на множество двойных косвенных блоков. Такая схема адресации позволяет работать с размерами файлов до 224 блоков по 1 Кбайт (это 16 Гбайт). При размере блоков в 8 Кбайт такая схема адресации поддерживает файлы размером до 64 Тбайт.

**Файловая система *Ext*3 в *Linux***. Для предотвращения потерь данных после сбоев системы и отказов электропитания файловой системе *ext*2 пришлось бы записывать каждый блок данных на диск немедленно после его создания. Вызванные перемещением головок записи задержки были бы такими значительными, что производительность стала бы недопустимо низкой. Поэтому записи откладываются, и изменения могут находиться в незафиксированном на диске состоянии до 30 с – что является очень длинным промежутком времени.

Для повышения живучести файловой системы *Linux* использует журналируемые файловые системы (*journaling file systems*). Примером такой системы является *ext*3 – последовательница файловой системы *ext*2.

Основное отличие *ext*3 от *ext*2 состоит в том, что *ext*3 журналируема, то есть в ней предусмотрена запись некоторых данных, позволяющих восстановить файловую систему при сбоях в работе компьютера.

Основная идея такого типа файловой системы состоит в поддержке журнала, который в последовательном порядке описывает все операции файловой системы.

В конечном итоге изменения записываются (фиксируются) в соответствующее место на диске – и соответствующие им записи журнала можно удалить. Если же до фиксации изменений происходит системный сбой или отказ электропитания, то при последующем запуске система обнаружит, что файловая система не была должным образом размонтирована, просмотрит журнал и выполнит все (описанные в журнале) изменения в файловой системе.

*Ext3* спроектирована таким образом, чтобы быть в значительной степени совместимой с *ext2*, и фактически все основные структуры данных и компоновка диска в обеих системах одинаковы. Боле того, размонтированная файловая система *ext2* может быть затем смонтирована как система *ext3* и обеспечивать журналирование.

***Журнал***– это файл, с которым работают как с кольцевым буфером. Журнал может храниться как на том же устройстве, что и основная файловая система, так и на другом. Поскольку операции с журналом не журналируются, файловая система *ext3* с ними не работает. Для выполнения операций чтения/записи в журнал используется отдельное блочное устройство журналирования JBD (*Journaling Block Device*).

JBD поддерживает три основные структуры данных: запись журнала (*log record*), описатель атомарной операции (*atomic operation handle*), транзакция (*transaction*). Запись журнала описывает операцию низкого уровня в файловой системе (которая обычно приводит к изменениям внутри блока). Поскольку системный вызов (такой, как *write*) обычно приводит к изменениям во многих местах – *i*-узлах, блоках существующих файлов, новых блоках файлов, списке свободных блоков и т. д., – соответствующие записи журнала группируются в атомарные операции. *Ext*3 уведомляет *JBD* о начале и конце обработки системного вызова (чтобы устройство *JBD* могло обеспечить фиксацию либо всех записей журнала данной атомарной операции, либо никаких). И наконец, в основном из соображений эффективности, *JBD* обрабатывает коллекции атомарных операций как транзакции. В транзакции записи журнала хранятся последовательно. JBD позволяет удалять фрагменты файла журнала только после того, как все принадлежащие к транзакции записи журнала надежно зафиксированы на диске.

В *ext3* предусмотрено три режима журналирования:

* *writeback*: в журнал записываются только метаданные файловой системы, то есть информация о ее изменении. Не может гарантировать целостности данных, но уже заметно сокращает время проверки по сравнению с *ext*2;
* *ordered*: то же, что и *writeback*, но запись данных в файл производится гарантированно до записи информации об изменении этого файла. Немного снижает производительность, также не может гарантировать целостности данных (хотя и увеличивает вероятность их сохранности при дописывании в конец существующего файла);
* *journal*: полное журналирование как метаданных ФС, так и пользовательских данных. Самый медленный, но и самый безопасный режим; может гарантировать целостность данных при хранении журнала на отдельном разделе (а лучше на отдельном жестком диске).

Журналирование только метаданных снижает издержки системы и повышает производительность, но не гарантирует от повреждения данных в файлах. Некоторые другие журнальные файловые системы ведут журналы только для операций с метаданными (например, *XFS* в *SGI*).

Файловая система *ext*3 может поддерживать файлы размером до 1 ТБ. С *Linux*-ядром 2.4 объем файловой системы ограничен максимальным размером блочного устройства, что составляет 2 ТБ. В *Linux* 2.6 (для 32-разрядных процессоров) максимальный размер блочных устройств составляет 16 ТБ, однако *ext*3 поддерживает только до 4 ТБ [3].

**Файловая система *Ext4* в *Linux***. Последней версией расширенной файловой системы на сегодняшней день является файловая система *ext*4, обратно совместимая с *ext2* и *ext3*. По сравнению с *ext3* в *ext4* реализован ряд улучшений, в основном касающихся скорости и надежности. Файловая система *ext*4 имеется в *Linux* с версией ядра 2.6.28 и выше. В табл.7 представлена эволюция расширенной файловой системы *Linux*.

*Таблица 7*

**Эволюция расширенной файловой системы**

|  |  |  |
| --- | --- | --- |
| Файловая система | Дата представления, г. | Характеристика |
| *Extended file system* | 1991 | Самая ранняя файловая система *Linux*. Недостатком этой файловой системы является чрезмерная фрагментация |
| *Ext*2 | 1993 | Эта файловая система обладает высокой надежностью, но в ней отсутствует журналирование. После внезапной перезагрузки или сбоя системы для всей файловой системы запускается команда *fsck* |
| *Ext*3 | 2001 | Эта файловая система может содержать 32 000 поддиректорий, поддерживает журналирование и обратно совместима с файловой системой *ext*2 |
| *Ext*4 | 2008 | Эта файловая система может содержать 64 000 поддиректорий, позволяет полностью отключить журналирование (в отличие от *ext*3) и обратно совместима с файловыми системами *ext*2 и *ext*3 |

Основными изменениями *ext4* по сравнению с *ext3* являются:

* увеличение максимального объема одного раздела диска до 1 эксбибайта (260 байт) при размере блока 4 кибибайт;
* увеличение размера одного файла до 16 тебибайт (240 байт);
* введение механизма пространственной (*extent*) записи файлов, уменьшающего фрагментацию и повышающего производительность.

Суть механизма заключается в том, что новая информация добавляется в конец области диска, выделенной заранее по соседству с областью, занятой содержимым файла.

Основными особенностями *ext*4 являются:

* использование экстентов (англ. *extent*). В ФС *ext3* адресация данных выполнялась традиционным образом − поблочно. Такой способ адресации стал менее эффективным с ростом размера файлов. Экстенты позволяют адресовать большое количество (до 128 MB) последовательно идущих блоков одним дескриптором. До 4-х указателей на экстенты может размещаться непосредственно в inode, этого достаточно для файлов маленького и среднего размера;
* 48-и битные номера блоков. При размере блока 4 KB это позволяет адресовать до одного эксбибайта (248 (4 KB) = 248·(22)·(210) B = 260 B = 1 EB);
* выделение блоков группами (англ. *multiblock allocation*). ФС хранит информацию не только о местоположении свободных блоков, но и о количестве свободных блоков, расположенных друг за другом. При выделении места ФС находит такой фрагмент, в который данные можно записать без фрагментации. Использование этой техники позволяет снизить уровень фрагментации ФС;
* отложенное выделение блоков (англ. *delayed allocation*). Выделение блоков для хранения содержимого файла происходит непосредственно перед записью на диск (например, при вызове *sync*), а не при вызове *write*. Из-за этого блоки можно выделять не по одному, а группами, что в свою очередь минимизирует фрагментацию и ускоряет процесс выделения блоков. С другой стороны, увеличивается риск потери данных в случае внезапного пропадания питания;
* поднято ограничение на число вложенных каталогов *ext*3 позволяла размещать в одном каталоге не более 32 000 подкаталогов (до 65 535, если изменить константы ядра);
* резервирование *inode-*ов при создании каталога (англ. *directory inodes reservation*). При создании каталога резервируется несколько *inode-*ов. Впоследствии, при создании файлов в этом каталоге сначала используются зарезервированные *inode*-ы, и если таких не осталось, выполняется обычная процедура выделения inode;
* размер *inode*. Размер *inode* (по умолчанию) увеличен с 128 (*ext3*) до 256 байтов. Это позволило реализовать перечисленное ниже;
* временные метки с наносекундной точностью (англ. *nanosecond timestamps*). Точность временных меток, хранящихся в inode, повышена до наносекунд. Диапазон значений тоже расширен: у *ext3* верхней границей хранимого времени было 18 января 2038 года, а у *ext*4 − 25 апреля 2514 года;
* версия *inode*. В структуре inode появилось поле, хранящее номер версии. Номер может увеличиваться при каждом изменении inode, если ФС монтирована с опцией *iversion*[3]. Это используется демонами сетефой файловой системы *NFS* версии 4 (*NFSv4*) для отслеживания изменений файлов;
* хранение расширенных атрибутов в структуре *inode* (англ. *extended attributes* (*EA*) *in inode*). Производительность файловой системы повышается изза исключения операции поиска атрибутов в каком-либо другом месте диска. К расширенным атрибутам относятся списки контроля доступа *ACL*, атрибуты *SELinux* и другие. Атрибуты, для которых недостаточно места в структуре inode, хранятся в отдельном блоке размером 4 KB. В будущем планируется снять это ограничение;
* вычисление контрольных сумм для записей журнала (англ. *journal checksumming*). Использование контрольных сумм для транзакций журнала позволяет быстрее находить и (иногда) исправлять ошибки системы после сбоя (при проверке целостности);
* предварительное выделение (англ. *persistent preallocation*). Для выделения места на *ext2* и *ext3* приложению приходилось записываться в файл нулевые байты. В *ext*4 появилась возможность резервирования блоков. Теперь не нужно тратить время на запись нулевых байт, достаточно использовать системный вызов *fallocate*. Вызов *fallocate* выделяет для файла блоки и устанавливает для них флаг «заполнены нулевыми байтами». При чтении из файла приложение получит нулевые байты (так же, как при чтении sparse файла). При записи в файл флаг «заполнены нулевыми байтами» будет снят. В отличие от sparse-файлов запись в такой файл никогда не прервется из-за нехватки свободного места;
* дефрагментация без размонтирования (англ. *online defragmentation*). Дефрагментация выполняется утилитой *e4defrag*, поставляемой в составе пакета *e2fsprogs* с 2011 года;
* неинициализированные блоки (англ. *uninitialised groups*). Возможность пока не реализована и предназначена для ускорения проверки целостности ФС утилитой *fsck*. Блоки, отмеченные как неиспользуемые, будут проверяться группами, и детальная проверка будет производится только если проверка группы показала наличие повреждений. Предполагается, что время проверки будет составлять от 1/2 до 1/10 от нынешнего в зависимости от способа размещения данных.

**2. Типы файловых систем, поддерживаемые ОС *Windows***

Windows поддерживает несколько файловых систем, самыми важными из которых являются *FAT-32* и *NTFS* (*NT File System*).

*FAT-32* использует 32-битные дисковые адреса и поддерживает дисковые разделы размером до двух терабайтов. *FAT-32* не имеет никакой системы безопасности и на сегодняшний день она фактически используется только для переносных носителей (таких, как флэш-диски). Файловая система *NTFS* была разработана специально для версии *Windows NT*. Начиная с *Windows ХР* ее по умолчанию устанавливает большинство производителей компьютеров, она существенно увеличивает безопасность и функциональность *Windows. NTFS* использует 64-битные дисковые адреса и теоретически может поддерживать дисковые разделы размером до 264 байт (однако некоторые соображения ограничивают этот размер до более низких значений).

Мы изучим файловую систему *NTFS*, так как это современная файловая система со многими интересными функциональными возможностями и конструктивными новшествами.

Имена файлов в *NTFS* ограничены 255 символами; размер полного маршрута ограничен 32 767 символами. Имена файлов хранятся к кодировке *Unicode*, что позволяет в тех странах, где не используется латинский алфавит (например, Греция, Япония, Индия, Россия и Израиль), писать имена файлов на своем языке. Например, *cpita* – это допустимое имя файла. *NTFS* полностью поддерживает чувствительные к регистру имена (так что *foo* отличается от *Foo* и *FOO*). Интерфейс прикладного программирования *Win 32* не полностью поддерживает чувствительность к регистру имен файлов и вовсе не поддерживает ее для имен каталогов.

Поддержка чувствительности к регистру имеется при работе подсистемы POSIX (для совместимости с *Unix*). Win32 не является чувствительным к регистру, но он сохраняет регистр – так что имена файлов могут иметь в своем составе буквы разных регистров. Несмотря на то что чувствительность к регистру хорошо знакома пользователям *Unix*, она очень неудобна для обычных пользователей (которые обычно не делают таких различий). Например, почти весь современный Интернет не имеет чувствительности к регистру.

Файл в *NTFS* – это не просто линейная последовательность байтов (как файлы в *FAT-32* и в *Unix*). Файл состоит из множества атрибутов, каждый из которых представлен потоком байтов. Большинство файлов имеет несколько коротких потоков (таких, как название файла и его 64-битный идентификатор объекта) плюс один длинный (не именованный) поток с данными. Однако файл может иметь также два или более (длинных) потока данных. Каждый поток имеет имя, состоящее из имени файла, двоеточия и имени потока. Каждый поток имеет свой размер и может блокироваться независимо от всех остальных потоков. Идея множества потоков в *NTFS* не является новой. Файловая система компьютеров *Apple Macintosh* использует два потока на файл (ветвь данных и ветвь ресурсов). Первоначально потоки в *NTFS* применялись для того, чтобы файловый сервер NT мог обслуживать клиентов *Macintosh*.

Множественность потоков данных используется также и для того, чтобы представлять метаданные файлов – такие, как контрольные картинки изображений в формате JPEG (которые есть в графическом интерфейсе пользователя *Windows*). Однако, к сожалению, множественные потоки данных уязвимы, и они часто теряются при переносе в другие файловые системы или по сети (и даже при резервном копировании и последующем восстановлении, поскольку многие утилиты игнорируют их).

*NTFS* – это иерархическая файловая система (похожая на файловую систему *Unix*). Однако разделителем компонентов имени является знак «\», а не «/». В отличие от *Unix*, концепции текущего рабочего каталога, жестких ссылок на текущий каталог (.) и родительский каталог (..) реализованы как соглашения (а не как фундаментальная часть файловой системы). Жесткие ссылки поддерживаются, но используются только для подсистемы POSIX, так же, как и поддержка проверки обхода каталогов (разрешение Y в *Unix*).

До *Windows Vista* символические ссылки в *NTFS* не поддерживались. Создание символических ссылок обычно разрешается только администраторам (во избежание проблем с безопасностью типа спуфинга, которые появились в *Unix*, когда символические ссылки были введены в версии *BSD 4.2*). Реализация символических ссылок в *Vista* использует функциональную возможность *NTFS* под названием «точка повторной обработки» (*reparse points*), которая обсуждается далее в этом разделе. Кроме того, поддерживаются также сжатие, шифрование, отказоустойчивость, журналирование и разреженные файлы. Эти функциональные возможности и их реализацию мы скоро обсудим.

**Реализация файловой системы *NTFS****. NTFS* – это очень сложная файловая система, которая была разработана специально для *NT* как альтернатива файловой системе *HPFS*.

Далее мы изучим функциональные возможности *NTFS* (начиная с ее структуры), затем перейдем к поиску имен файлов, сжатию файлов, журналированию и шифрованию файлов.

**Структура файловой системы**. Каждый том *NTFS* (например, дисковый раздел) содержит файлы, каталоги, битовые массивы и другие структуры данных. Каждый том организован как линейная последовательность блоков (которые в терминологии компании *Microsoft* называются кластерами), причем размер блоков для каждого тома фиксирован (в зависимости от размера тома он

может изменяться от 512 байт до 64 Кбайт). Большинство дисков *NTFS* использует блоки размером 4 Кбайт – это компромисс между применением больших блоков (для эффективной передачи данных) и использованием маленьких блоков (для снижения внутренней фрагментации). Ссылки на блоки делаются с использованием смещения от начала тома (при помощи 64-битных чисел).

Главная структура данных каждого тома – это *MFT* (*Master File Table* – главная таблица файлов), которая является линейной последовательностью записей фиксированного размера A Кбайт). Каждая запись *MFT* описывает один файл или один каталог. Она содержит атрибуты файла (такие, как его имя и временная метка), а также список дисковых адресов (где расположены его блоки). Если файл очень большой, то иногда приходится использовать две или более записей *MFT* (чтобы разместить в них список всех блоков) – в этом случае первая запись в *MFT*, называемая основной записью (*base record*), указывает на остальные записи в *MFT*.

Такая схема переполнения ведет свое начало из *СР/М*, где каждый элемент каталога назывался экстентом. Битовый массив отслеживает свободные элементы *MFT*. Сама *MFT* также является файлом и в качестве такового может быть размещена в любом месте тома (таким образом устраняется проблема наличия дефектных секторов на первой дорожке). Более того, при необходимости этот файл может расти (до максимального размера в 248 записей). *MFT* показана на рис.7.

Каждая запись *MFT* состоит из последовательности пар (заголовок атрибута – значение). Каждый атрибут начинается с заголовка, рассказывающего о том, что это за атрибут и какую длину имеет его значение. Некоторые значения атрибутов имеют переменную длину (такие, как имя файла и его данные). Если значение атрибута достаточно короткое для того, чтобы уместиться в записи *MFT*, то оно помещается именно туда. Это называется непосредственным файлом – *immediate file*. Если же значение слишком длинное, то оно размещается на диске, а в запись *MFT* помещается указатель на него. Это делает систему *NTFS* очень эффективной для небольших полей, которые могут разместиться в самой записи *MFT*.

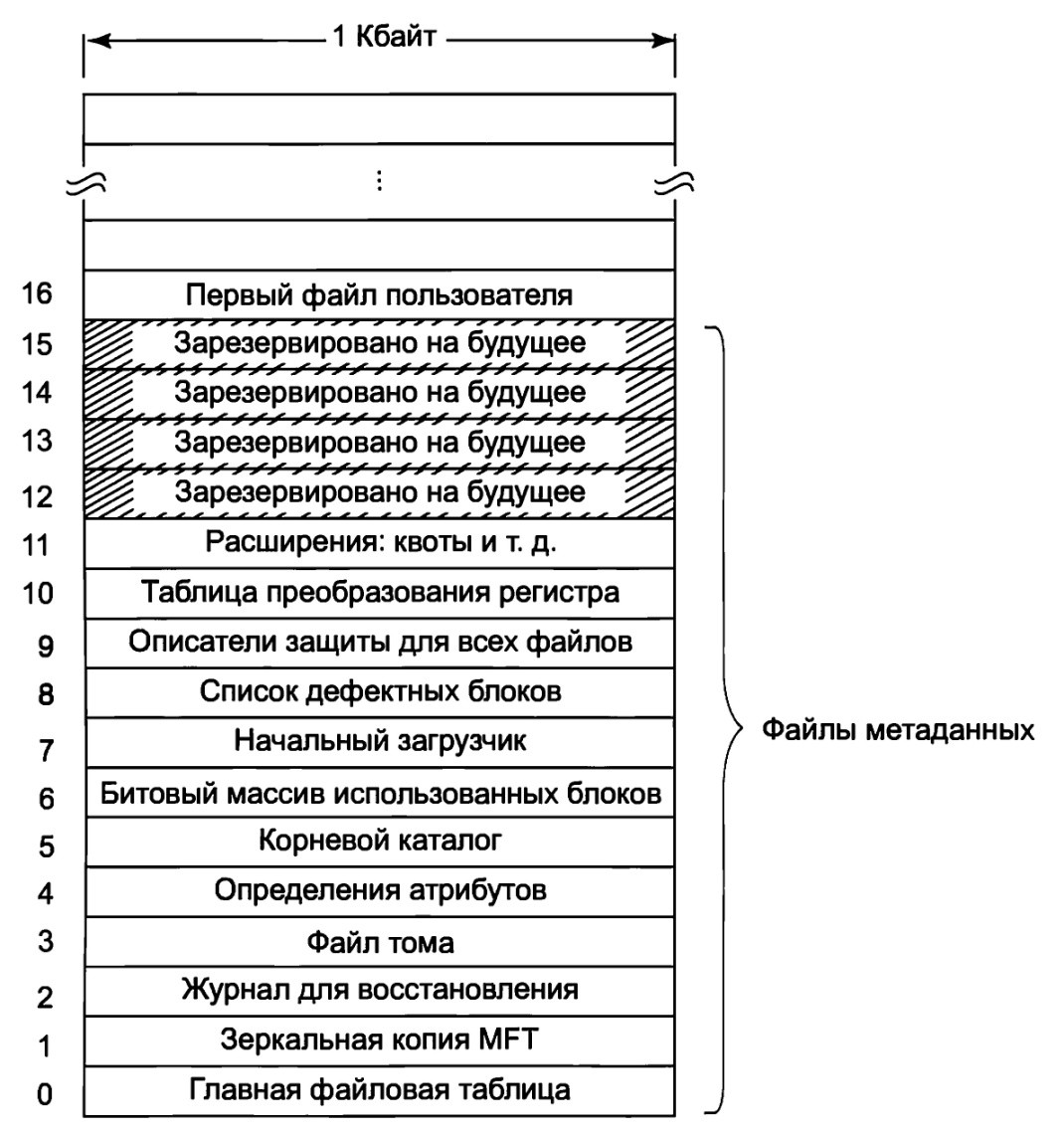


Рис.7. Главная таблица файлов NTFS

Первые 16 записей *MFT* резервируются для файлов метаданных *NTFS* (см. рис.7). Каждая из этих записей описывает нормальный файл, который имеет атрибуты и блоки данных (как и любой другой файл). Каждый из этих файлов имеет имя, которое начинается со знака доллара (чтобы обозначить его как файл метаданных).

Первая запись описывает сам файл *MFT*. В частности, в ней говорится, где находятся блоки файла *MFT* (чтобы система могла найти файл *MFT*). Очевидно, что *Windows* нужен способ нахождения первого блока файла *MFT*, чтобы найти остальную информацию по файловой системе.

*Windows* смотрит в загрузочном блоке – именно туда записывается адрес первого блока файла *MFT* при форматировании тома.

Запись 1 является дубликатом начала файла *MFT*. Эта информация настолько ценная, что наличие второй копии может быть просто критическим (в том случае, если один из первых блоков *MFT* испортится).

Вторая запись – файл журнала. Когда в файловой системе делаются структурные изменения (такие, как добавление нового или удаление существующего каталога), то такое действие журналируется здесь до его выполнения (чтобы повысить вероятность корректного восстановления в случае сбоя во время операции – например такого, как отказ системы). Здесь также журналируются и изменения в файловых атрибутах. Фактически не журналируются здесь только изменения в пользовательских данных. Запись 3 содержит информацию о томе (такую, как его размер, метка и версия).

Как уже упоминалось выше, каждая запись *MFT* содержит последовательность пар (заголовок атрибута – значение). Атрибуты определяются в файле *AttrDef*. Информация об этом файле содержится в *MFT* (в записи 4). Затем идет корневой каталог, который сам является файлом и может расти до произвольного размера. Он описывается записью номер 5 в *MFT*.

Свободное пространство тома отслеживается при помощи битового массива. Сам битовый массив – тоже файл, его атрибуты и дисковые адреса даны в записи 6 в *MFT*. Следующая запись *MFT* указывает на файл начального загрузчика. Запись 8 используется для того, чтобы связать вместе все плохие блоки (чтобы обеспечить невозможность их использования для файлов). Запись 9 содержит информацию безопасности. Запись 10 используется для установления соответствия регистра. Для латинских букв *A-Z* соответствие регистра очевидно (по крайней мере для тех, кто разговаривает на романских языках). Однако соответствие регистров для других языков (таких, как греческий, армянский или грузинский) для говорящих на романских языках не столь очевидно – поэтому данный файл рассказывает, как это сделать.

И наконец, запись «И» – это каталог, содержащий различные файлы для таких вещей, как дисковые квоты, идентификаторы объектов, точки повторной обработки и т. д. Последние четыре записи *MFT* зарезервированы для использования в будущем.

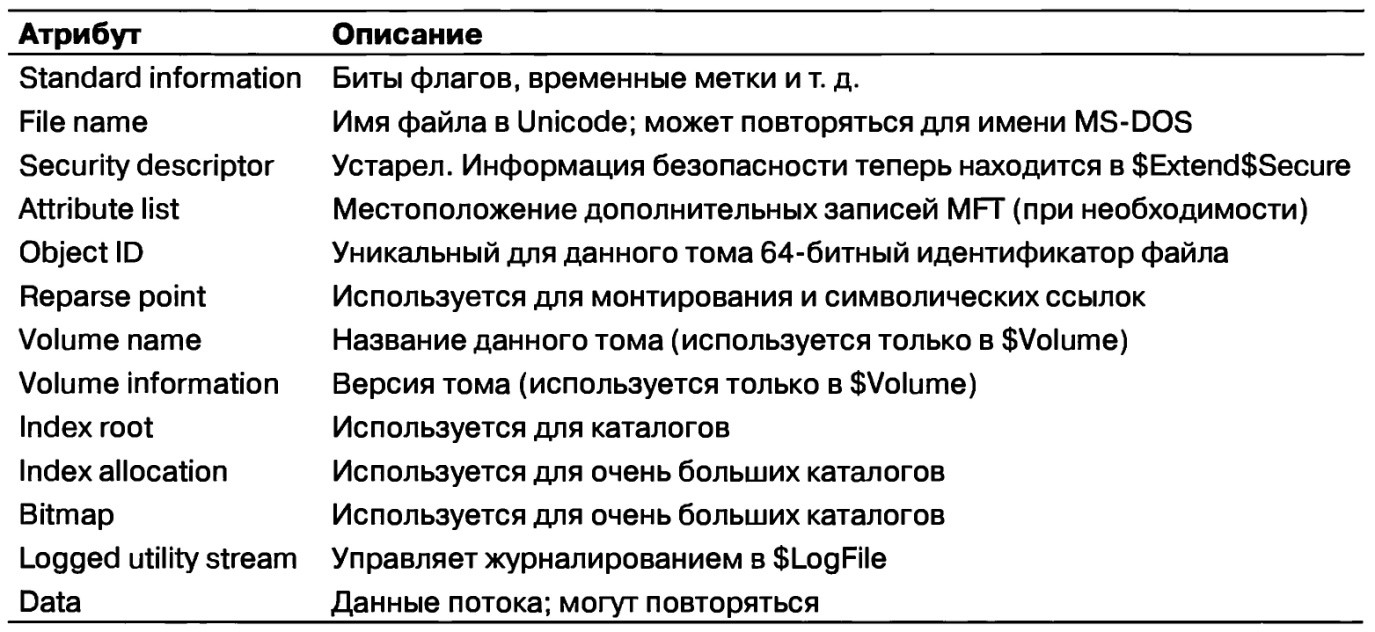
Каждая запись *MFT* состоит из заголовка записи, за которым следуют пары «заголовок атрибута – значение». Заголовок записи содержит системный код, используемый для проверки достоверности, последовательный номер (обновляемый каждый раз, когда запись используется для нового файла), счетчик количества ссылок на файл, фактическое количество использованных в записи байтов, идентификатор (индекс, порядковый номер) основной записи (используется только для записей расширения), а также некоторые другие поля.

*NTFS* определяет 13 атрибутов, которые могут появиться в записях *MFT*. Они перечислены в табл.8. Каждый заголовок атрибута идентифицирует атрибут и содержит длину и местоположение поля значения, а также разнообразные флаги и прочую информацию.

Обычно значения атрибутов следуют непосредственно за своими заголовками атрибутов, но если значение слишком длинное для того, чтобы поместиться в запись *MFT*, то оно может быть размещено в отдельных дисковых блоках. Такой атрибут называется нерезидентным атрибутом (*nonresident attribute*). Очевидно, что таким атрибутом является атрибут данных.

Некоторые атрибуты (такие, как имя) могут повторяться, но все атрибуты должны присутствовать в записи *MFT* в определенном порядке. Заголовки резидентных атрибутов имеют длину 24 байта; заголовки нерезидентных атрибутов длиннее (поскольку они содержат информацию о том, где нужно искать атрибут на диске).

*Таблица 8*

**Используемые в записях *MFT* атрибуты**

Стандартное информационное поле содержит: сведения о владельце файла; информацию безопасности; нужные для *POSIX* временные метки; количество жестких ссылок; биты архивирования и т. д. Это поле имеет фиксированную длину и присутствует всегда. Имя файла – это строка переменной длины в коде *Unicode*.

В *NT 4.0* информация безопасности размещалась в атрибуте, но в *Windows* 2000 и более поздних версиях вся информация безопасности размещается в одном файле (чтобы она могла совместно использоваться многими файлами). Это приводит к существенной экономии места во многих записях *MFT* и в файловой системе в целом, поскольку информация безопасности идентична для большого количества принадлежащих одному пользователю файлов.

Список атрибутов нужен в том случае, когда атрибуты не помещаются в запись *MFT*. Из этого атрибута можно узнать, где искать записи расширения. Каждый элемент списка содержит 48-битный индекс по *MFT* (который говорит о том, где находится запись расширения) и 16-битный порядковый.

Файлы *NTFS* имеют связанный с ними идентификатор, который подобен номеру узла *i-node* в *Unix*. Файлы можно открывать по идентификатору, но присваиваемый файловой системой *NTFS* идентификатор не всегда можно использовать, поскольку он основан на записи *MFT* и может измениться при перемещении записи для данного файла (например, если файл восстанавливается из резервной копии). *NTFS* позволяет использовать отдельный атрибут «идентификатор объекта», который может быть установлен для файла и который нет необходимости изменять. Его можно сохранить вместе с файлом (например, если он копируется на новый том).

Точка повторной обработки сообщает разбирающей имя файла процедуре о необходимости сделать что-то особенное. Этот механизм используется для явного монтирования файловых систем и для символических ссылок. Два атрибута тома используются только для идентификации томов. Следующие три атрибута работают с реализацией каталогов. Маленькие каталоги – это просто списки файлов, а большие реализованы как деревья *В*+. Атрибут *logged utility stream* используется шифрующей файловой системой.

И наконец, мы подошли к атрибуту, который важнее всех: поток данных (или потоки). Файл в *NTFS* имеет один (или несколько) связанных с ним потоков данных. Именно здесь находится его полезное содержание. Поток данных по умолчанию (*default data stream*) названия не имеет (например, *dirpath\filename:: DATA*), но альтернативные потоки данных (alternate data stream) имеют свои имена, например: *dirpath\filename:streamname: DATA*.

Для каждого потока его имя (если оно имеется) находится в заголовке этого атрибута. Следом за заголовком идет либо список дисковых адресов, либо сам поток. Размещенные в записи *MFT* реальные данные потока называются «непосредственный файл».

Конечно, в основном данные не помещаются в запись *MFT*, поэтому данный атрибут обычно нерезидентный. Теперь давайте рассмотрим, как *NTFS* отслеживает местоположение нерезидентных атрибутов.

**Выделение дискового пространства**. Модель отслеживания дисковых блоков состоит в том, что они выделяются последовательными участками, насколько это возможно (из соображений эффективности). Например, если первый логический блок потока помещен в блок 20 диска, то система будет очень стараться поместить второй логический блок в блок 21, третий логический блок в блок 22 и т. д. Одним из способов достижения непрерывности этих участков является выделение дискового пространства по нескольку блоков за один раз.

Блоки потока описываются последовательностью записей, каждая их которых описывает последовательность логически смежных блоков. Для потока без пропусков будет только одна такая запись. К этой категории принадлежат такие потоки, которые записаны по порядку с начала и до конца. Для потока с одним пропуском (например, определены только блоки 0-49 и блоки 60-79) будет две записи. Такой поток может быть получен при помощи записи первых 50 блоков, а затем пропуском до 60 блока и записи еще 20 блоков. Когда такой пропуск считывается, то все недостающие байты – нулевые. Файлы с пропусками называются разреженными файлами (*sparse files*).

Каждая запись начинается с заголовка, в котором дается смещение первого блока потока. Затем идет смещение первого не описанного данной записью блока. В приведенном выше примере в первой записи будет заголовок @,50) и будут даны дисковые адреса этих 50 блоков. Во второй записи будет заголовок F0,80) и дисковые адреса этих 20 блоков.

За заголовком записи следует одна или несколько пар (в каждой дается дисковый адрес и длина участка). Дисковый адрес – это смещение дискового блока от начала раздела; длина участка – это количество блоков в участке. В записи участка может быть столько пар, сколько необходимо. Использование этой схемы для потока из трех участков и девяти блоков показано на рис.8.

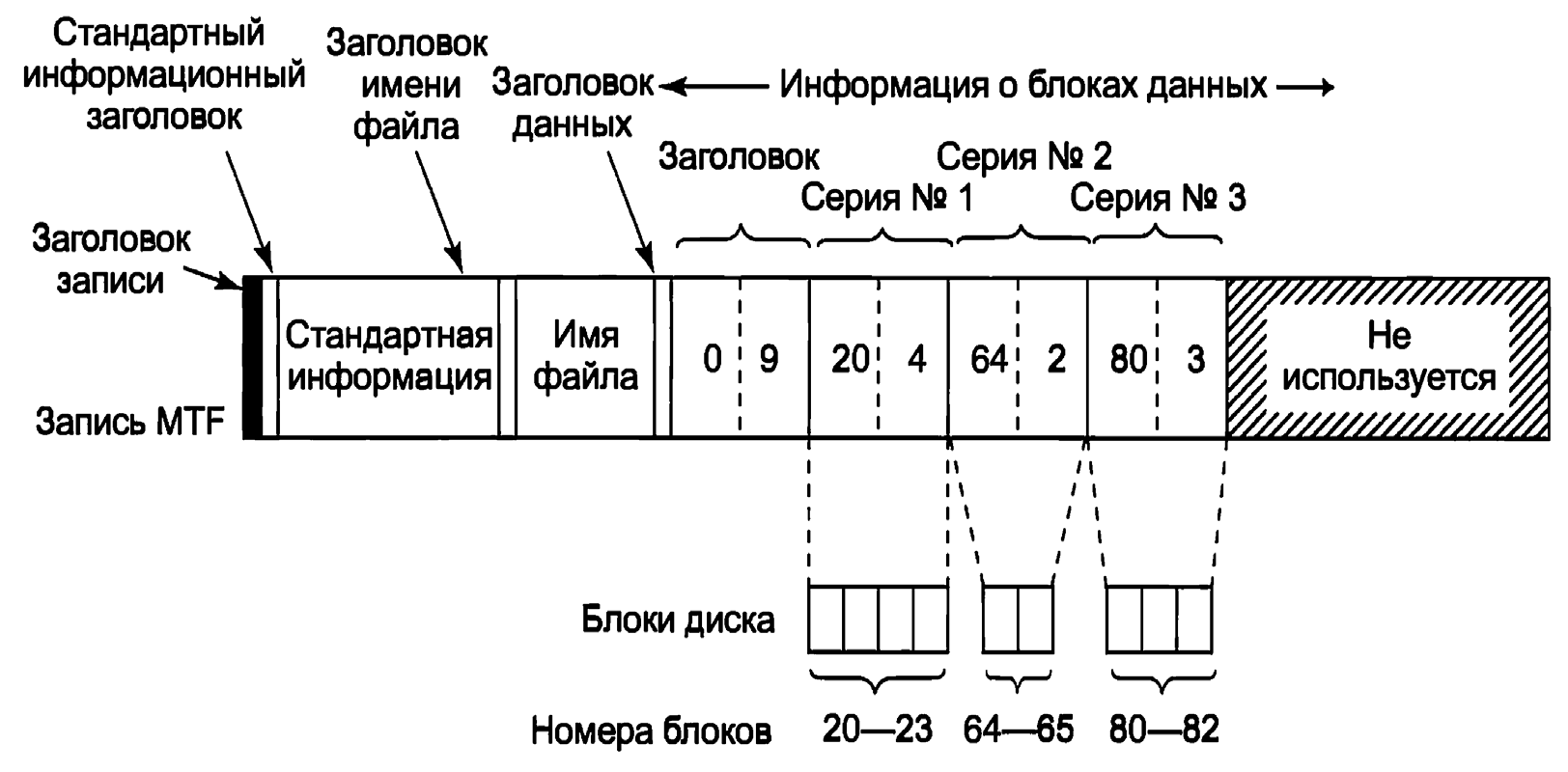


Рис.8. Запись *MFT* для потока из трех участков и девяти блоков

На этом рисунке у нас есть запись *MFT* для короткого потока из девяти блоков (заголовок 0-8). Он состоит из трех участков последовательных блоков на диске. Первый участок – блоки 20-23, второй – блоки 64-65, третий – блоки 80-82. Каждый из этих участков заносится в запись *MFT* как пара (дисковый адрес, количество блоков). Количество участков зависит от того, насколько хорошо справился со своей работой модуль выделения блоков при создании потока. Для потока из п блоков количество участков может составлять от 1 до *n*. Здесь нужно сделать несколько замечаний.

*Во-первых*, для представленных таким способом потоков нет верхнего ограничения размера. Если не использовать сжатие адресов, то для каждой пары требуется два 64-битных числа (всего 16 байт). Однако пара может представлять 1 миллион (или более) смежных дисковых блоков. Фактически состоящий из 20 отдельных участков (каждый – по одному миллиону блоков размером в 1 Кбайт) поток размером в 20 Мбайт легко помещается в одну запись *MFT*, а разбросанный по 60 изолированным блокам поток размером 60 Кбайт в одну запись MFT не помещается.

*Во-вторых*, в то время как самый простой способ представления каждой пары требует 2x8 байт, имеется также и метод сжатия, который уменьшает размер пары меньше чем до 16 байт. Многие дисковые адреса имеют нулевые старшие байты. Их можно опустить. Заголовок данных сообщает о том, сколько их было опущено (то есть, сколько байтов реально используется на один адрес). Используются также и другие виды сжатия. На практике пара часто занимает только 4 байта. Наш первый пример был простым: вся информация файла уместилась в одной записи *MFT*. Что произойдет, если файл настолько большой или так сильно фрагментирован, что информация о блоках не помещается в одну запись *MFT*? Ответ простой: используются две или более записей *MFT*.

На рис.9 мы видим файл, основная запись которого находится в записи 102 в *MFT*. Он имеет слишком много (для одной записи *MFT*) участков, поэтому вычисляется количество нужных записей расширения (например, две) и их индексы помещаются в основную запись. Остальная часть записи используется для первых *k* участков данных.

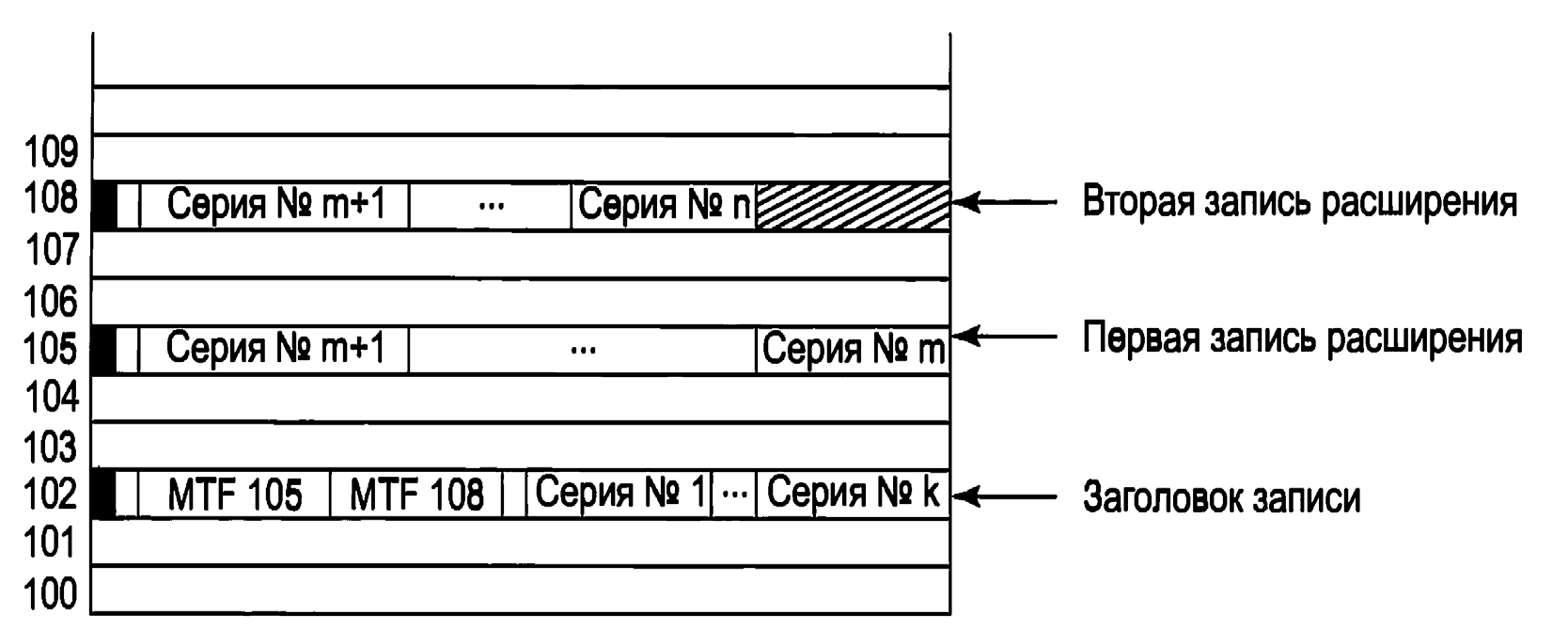


Рис.9. Структура файла (три записи *MFT*)

Обратите внимание, что на рис.9 имеется некоторая избыточность. В теории не должно быть необходимости указывать конец последовательности участков, поскольку эту информацию можно вычислить по парам для участков. Цель избыточного указания этой информации в том, чтобы сделать более эффективным поиск: чтобы найти блок по заданному смещению в файле, нужно обследовать только заголовки записей (а не пары для участков).

Когда все пространство записи 102 будет использовано, сохранение участков продолжится в записи 105 в *MFT*. В нее будет записано столько участков, сколько поместится. Когда эта запись также заполнится, остальные участки попадут в запись 108 в *MFT*. Таким образом можно использовать много записей *MFT* (для работы с большими фрагментированными файлами). Если нужно очень много записей *MFT*, то появляется проблема: может не хватить места в основной *MFT* для размещения всех их индексов. Для этой проблемы также есть решение: список записей расширения *MFT* делается нерезидентным – то есть хранится в других дисковых блоках (вместо основной записи *MFT*). В этом случае он может увеличиваться настолько, насколько это нужно. Элемент *MFT* для небольшого каталога показан на рис.10.

Запись содержит некоторое количество элементов каталога, каждый из которых описывает один файл или каталог. Каждый элемент содержит структуру фиксированной длины, за которой следует имя файла (переменной длины). Фиксированная часть содержит индекс элемента *MFT* для данного файла, длину имени файла, а также разнообразные прочие поля и флаги. Поиск элемента каталога состоит из опроса всех имен файлов по очереди. Большие каталоги используют другой формат. Теперь у нас достаточно информации, чтобы закончить описание поиска имен файлов для файла *\??\C:\foo\bar*.

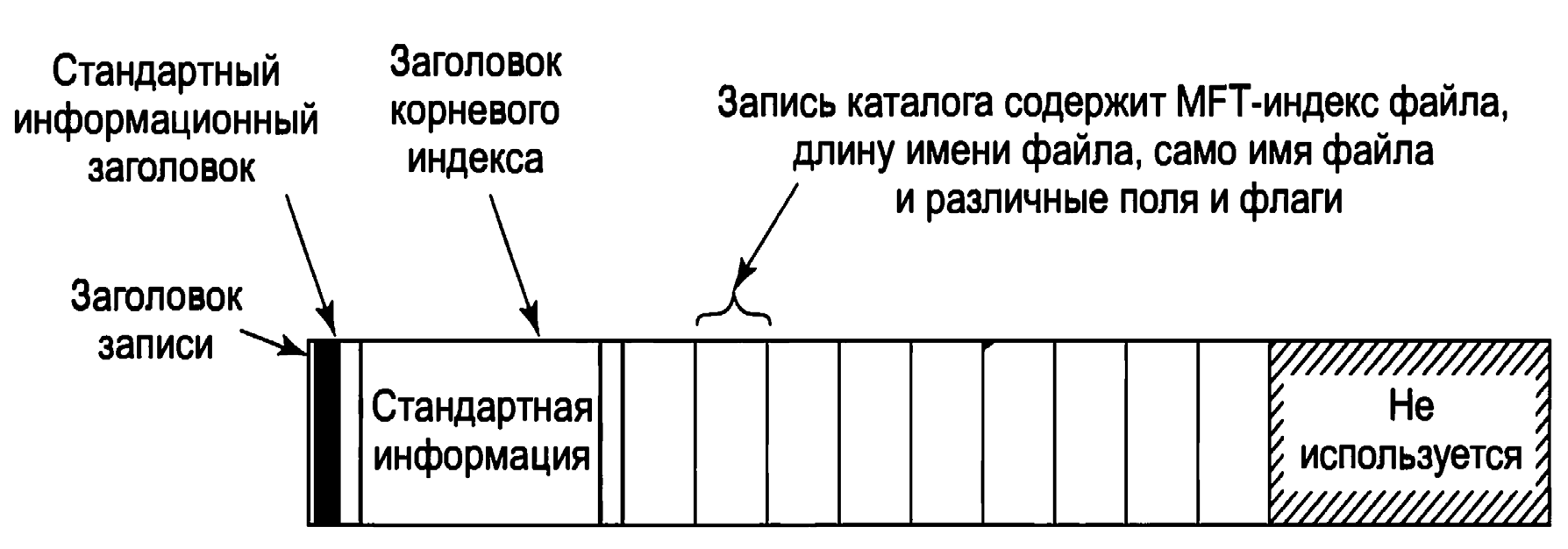


Рис.10. Запись *MFT* для небольшого каталога

На рис.10 мы видели, как *Win32* (интерфейс собственных системных вызовов NT), а также диспетчеры объектов и ввода-вывода совместно открывали файл: для этого посылался запрос ввода-вывода в стек устройства *NTFS* (для тома С:). Запрос ввода-вывода просит *NTFS* заполнить объект файла для маршрута *\foo\bar*.

Разбор маршрута \*foo\bar* начинается в корневом каталоге С: блоки которого можно определить из элемента 5 в *MFT*. Строка «*foo*» ищется в корневом каталоге, который возвращает индекс в *MFT* для каталога *foo*. В этом каталоге затем выполняется поиск строки «*bar*», которая ссылается на запись *MFT* для данного файла*. NTFS* выполняет проверки доступа (обращаясь к монитору безопасности), и если все в порядке – ищет запись MFT для атрибута ::*DATA*, который является потоком данных по умолчанию.

Обнаружив файл *bar*, *NTFS* установит указатели на свои метаданные в объекте файла, переданном из диспетчера ввода-вывода. Метаданные включают указатель на запись *MFT*, информацию по сжатию и блокировке диапазонов, различные подробности о совместном использовании и т. д. Большинство этих метаданных содержится в структурах данных, совместно используемых всеми ссылающимися на этот файл объектами файлов. Несколько полей специфичны только для текущего открытого файла – например, следует ли файл удалить после его закрытия.

После того как открытие успешно произошло, *NTFS* вызывает *IoCompleteRequest* для передачи *IRP* обратно вверх по стеку ввода-вывода (в диспетчеры ввода-вывода и объектов). В конечном итоге описатель для объекта файла помещается в таблицу описателей для текущего процесса, и управление передается обратно в пользовательский режим. При последующих вызовах *ReadFile* приложение может предоставлять описатель, указывая, что этот объект файла для *C:\foo\bar* следует включать в запрос чтения, который передается вниз по стеку устройства С: в *NTFS*.

В дополнение к обычным файлам и каталогам NTFS поддерживает и жесткие ссылки (в *Unix*-смысле), а также символические ссылки (с использованием механизма под названием «точка повторной обработки» – *reparse points*). NTFS поддерживает пометку файла или каталога как точки повторной обработки и ассоциирование с ней блока данных. Когда такой файл или каталог встречается во время разбора имени файла, то операция заканчивается неудачей и диспетчеру объектов возвращается этот блок данных.

Диспетчер объектов может интерпретировать данные как представляющие альтернативный маршрут, после чего он обновляет строку для разбора и повторяет операцию ввода-вывода. Этот механизм используется для поддержки как символических ссылок, так и смонтированных файловых систем, выполняя перенаправление поиска в другую часть иерархии каталогов или даже в другой раздел диска.

Точки повторной обработки используются также для пометки отдельных файлов для драйверов-фильтров файловой системы. На рис.11 мы показали, как фильтры файловой системы можно установить между диспетчером ввода-вывода и файловой системой.

Запросы ввода-вывода завершаются при помощи вызова *IoCompleteRequest*, передающего управление процедурам завершения, которые каждый драйвер представил в стеке устройств (вставленном в IRP во время запроса).

Желающий пометить файл драйвер ассоциирует тег повторной обработки, а затем отслеживает закончившиеся неудачно (потому что им встретилась точка повторной обработки) запросы на завершение операций открывания файлов.

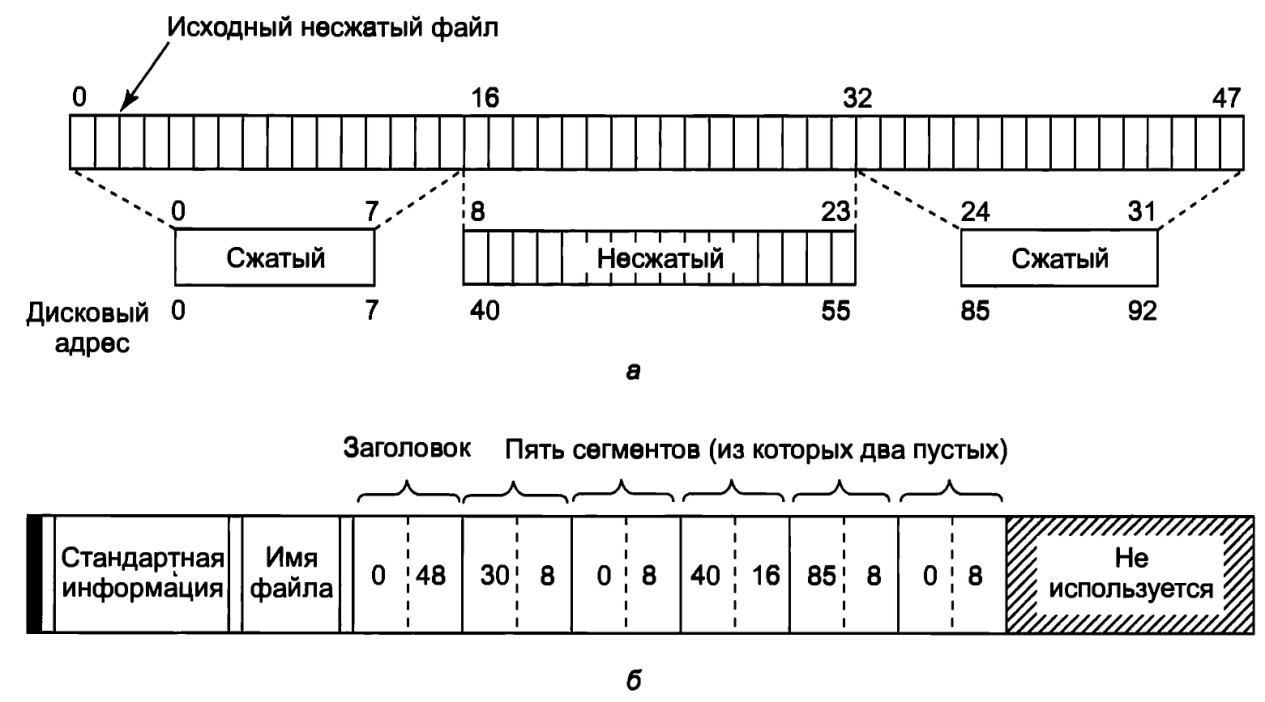


Рис.11. Пример сжатия файла из 48 блоков до размера в 32 блока (а); запись MFT для файла после сжатия (б)

Из блока (передаваемых обратно с *IRP*) данных драйвер может определить, тот ли это блок данных, который сам драйвер ассоциировал с файлом. Если это так, то драйвер останавливает обработку завершения и продолжает обработку исходного запроса ввода-вывода. Обычно при этом продолжается выполнение запроса на открытие, но имеется флаг, который дает указание *NTFS* игнорировать точку повторной обработки и открыть файл.

*NTFS* поддерживает прозрачное сжатие файлов. Файл может создаваться в сжатом режиме, а это означает, что *NTFS* пытается автоматически сжать блоки при их записи на диск и автоматически распаковывает их при чтении обратно. Те процессы, которые читают или пишут сжатые файлы, совершенно не в курсе того факта, что происходит сжатие или распаковка.

Сжатие работает следующим образом. Когда *NTFS* пишет файл (помеченный как сжатый) на диск, то она изучает первые 16 (логических) блоков файла – независимо от того, сколько участков они занимают. Затем она запускает по ним алгоритм сжатия. Если полученные данные можно записать в 15 или менее блоков, то сжатые данные записываются на диск (по возможности – одним участком). Если сжатые данные по-прежнему занимают 16 блоков, то эти 16 блоков записываются в несжатом виде. Затем исследуются блоки 16-31, чтобы узнать, можно ли их сжать до размера 15 блоков (или менее), и т. д.

На рис.11,а показан файл, в котором первые 16 блоков успешно сжались до 8 блоков, вторые 16 блоков не сжались, а третьи 16 блоков также сжались на 50 процентов. Эти три части были записаны как три участка и сохранены в записи *MFT*. Отсутствующие блоки хранятся в элементе *MFT* с дисковым адресом 0 (рис.11,б). Когда файл считывается обратно, *NTFS* должна знать, какие участки сжаты, а какие – нет. Она может определить это по дисковым адресам. Дисковый блок 0 не может использоваться для хранения данных.

Произвольный доступ к сжатым файлам возможен, но это сложно. Предположим, что процесс делает поиск блока 35. Как *NTFS* найдет блок 35 в сжатом файле? Сначала ей нужно прочитать и распаковать весь участок. Затем она узнает, где находится блок 35, и сможет передать его в любой процесс (который его прочитает). Выбор в качестве единицы сжатия 16 блоков – это компромисс. Меньший размер снизил бы эффективность сжатия. Больший размер сделал бы произвольный доступ еще более дорогим.

**Журналирование***. NTFS* поддерживает два механизма, при помощи которых программы могут обнаружить изменения в файлах и каталогах тома.

*Первый* механизм это операция ввода-вывода под названием *NtNotifyChangeDirectoryFile*, передающая системе буфер, который возвращается после обнаружения изменения в каталоге или подкаталоге. В результате ввода-вывода буфер заполняется списком записей об изменениях. Хорошо, если буфер достаточно большой – потому что в противном случае лишние записи теряются. *Второй* механизм – это журнал изменений *NTFS. NTFS* содержит список всех записей об изменениях для каталогов и файлов тома в специальном файле, который программы могут читать при помощи специальных операций управления файловой системой. Файл журнала обычно очень большой и поэтому вероятность затирания записей до того, как они будут изучены, очень мала.