Файловые системы. Часть 1

Лекция

Ревизия: 0.2

Последняя модификация: Vlad Kovtun

Дата последней модификации: 03.04.2015 14:44:00

История изменений

05.04.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

05.08.2014 – Версия 0.2. Внесены изменения в разделы Файловые системы, Типы файлов, Атрибуты файлов, I-узлы, Реализация каталогов, Организация дискового пространства, Резервные копии, Оперативное чтение блока, Файловые системы с журнальной структурой LFS. Ковтун. В.Ю.

Содержание

Лекция 7. Файловые системы

Вопросы

1. Файловые системы: файлы.

2. Файловые системы: каталоги.

3. Разработка файловых систем.

Файловые системы

Файловая система - это часть ОС, назначение которой состоит в том, чтобы обеспечить пользователю удобный интерфейс при работе с данными, хранящимися на диске, и обеспечить совместное использование файлов несколькими пользователями и процессами.

В широком смысле понятие "файловая система" включает:

- совокупность всех файлов на диске,
- наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске,
- комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, именование, поиск и другие операции над файлами.

Приложениям необходимо сохранять собственное состояние: во время работы процесс может хранить ограниченное количество данных в собственном адресном пространстве (ОЗУ), однако емкость такого хранилища ограничена размерами виртуального адресного пространства.

Таким образом, к долговременным устройствам хранения информации предъявляются три следующих важных требования:

1. Устройства должны позволять хранить очень большие объемы данных.

2. Информация должна сохраняться после прекращения работы процесса, использующего ее.

3. Несколько процессов должны иметь возможность получения одновременного доступа к информации.

Обычное решение всех этих проблем состоит в хранении информации на дисках и других внешних хранителях в модулях (сущностях), называемых **файлами**. Процессы могут по мере надобности читать их и создавать новые файлы. Информация, хранящаяся в файлах, должна обладать **устойчивостью** (в данном контексте иногда применяется термин **персистентность**), то есть на нее не должны оказывать влияния создание или прекращение работы какого-либо процесса. Файл должен исчезать только тогда, когда его владелец дает команду удаления файла.

Файловые системы: файлы

Именование файлов

Файлы относятся к абстрактному механизму. Они предоставляют собой способ сохранять информацию на диске и считывать ее снова позднее. При этом от пользователя должны скрываться такие детали, как способ и место хранения информации, а также детали работы дисков.

Наиболее важной характеристикой любого механизма абстракции является то, как именуются управляемые объекты - именование файлов. При создании файла процесс дает файлу имя. Когда процесс завершает работу, файл продолжает свое существование и по его имени к нему могут получить доступ другие процессы.

Точные правила именования файлов варьируются от системы к системе, но все современные ОС поддерживают в качестве имен файлов 8-символьные текстовые строки: допускается использование буквенно-символьные наборы, за исключением некоторых спецсимволов. Современные ФС поддерживают имена файлов длиной до 255 символов.

В некоторых ФС, например UNIX, различаются прописные и строчные символы, тогда как в других, таких как MS-DOS, нет. Таким образом, имена файлов maria, Maria и MARIA будут означать в системе UNIX три различных файла, тогда как в MS-DOS все эти имена будут соответствовать одному файлу.

ОС Windows 2000 и более поздние, поддерживают ФС MS-DOS и наследуют ее свойства. В современных ОС Windows получила развитие своя ФС – NTFS 5.0, начавшая свое существование с Windows 2000.

Во многих ОС имя файла может состоять из двух частей, разделенных точкой: prog.c. Часть имени файла после точки называется **расширением файла** и обычно означает тип файла. Так, в MS-DOS имя файла может содержать от 1 до 8 символов плюс расширение от 0 до 3 символов. В системе UNIX размер расширения файла зависит от пользователя. Кроме того, у файла может быть несколько расширений, например prog.c.Z, где .Z обычно используется, чтобы указать, что файл (prog.c) был сжат с помощью алгоритма ZIP.

Структура файла

Файлы могут быть структурированы несколькими различными способами. Три типа структур показаны на Рис. 1. Файл на Рис. 1(а) представляет собой неструктурированную последовательность байтов. В этом случае ОС не интересуется содержимым файла. Все, что она видит — это байты. Значения этим байтам придается программами уровня пользователя. Такой подход используется как в системе UNIX, так и в Windows.

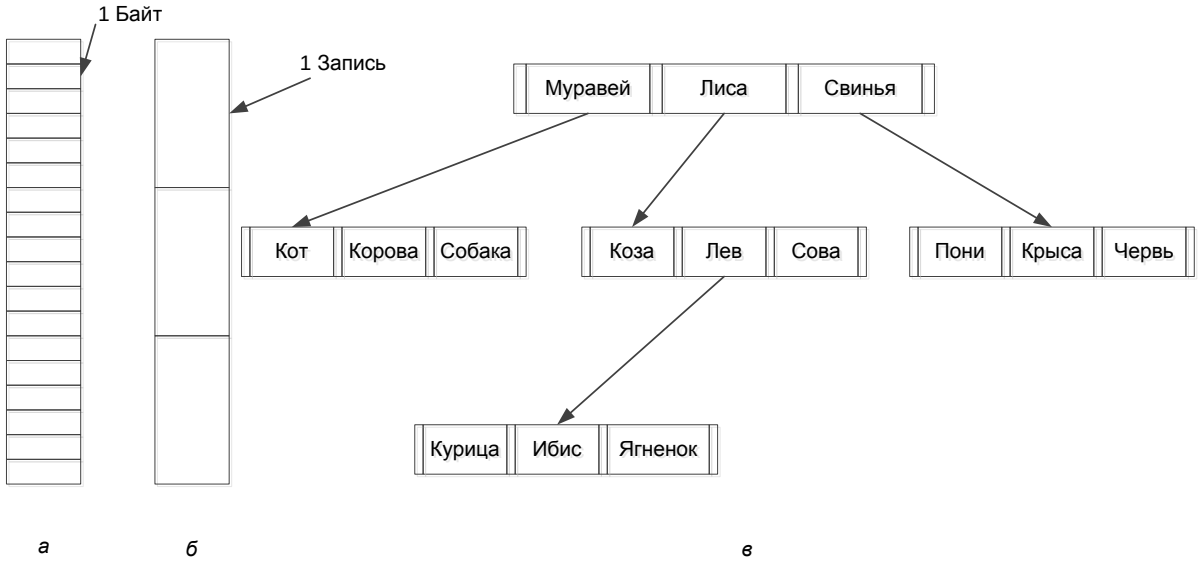

Рис. 1. Три типа файлов: последовательность байтов (а); последовательность записей (б); дерево (в)

Рассмотрение ОС файлов как просто последовательностей байтов обеспечивает максимальную гибкость. Программы пользователя могут помещать в файлы все что угодно и именовать их любым удобным для них способом. ОС не вмешивается в этот процесс, что может быть особенно ценно для специализированных приложений.

На Рис. 1(б) показан, который файл представляет собой последовательность записей фиксированной длины, каждая со своей внутренней структурой. Для файлов, состоящих из записей, важным является то, что операция чтения возвращает одну запись, а операция записи перезаписывает или дополняет одну запись.

Существуют другие ФС, которые оперируют файлами, представленными в виде 80-символьных записей, представляющими собой образы перфокарт. Этими ОС поддерживались также файлы, состоящие из 132-символьных записей, предназначающихся для строковых принтеров (которые в те дни печатали по 132 символа в строке). Программы читали из входных файлов 80-символьные блоки и записывали их в виде 132-символьных блоков, хотя остальные 52 символа могут быть пробелами. Ни одна современная универсальная ОС не работает подобным образом.

Третий вариант файловой структуры показан на Рис. 1(в). При такой организации файл представляет собой дерево записей, не обязательно одной и той же длины. Каждая

запись в фиксированной позиции содержит поле ключа. Дерево сортировано по ключевому полю, что обеспечивает быстрый поиск заданного ключа.

Основной ФС здесь является не получение следующей записи, хотя это также возможно, а получение записи с указанным значением ключа.

Типы файлов

Файлы бывают разных типов: обычные файлы, специальные файлы, файлы-каталоги.

Обычные файлы в свою очередь подразделяются на текстовые и двоичные. **Текстовые** файлы состоят из строк символов, представленных в ASCII-коде. Это могут быть документы, исходные тексты программ и т.п. Текстовые файлы можно прочитать на экране и распечатать на принтере. **Двоичные** файлы не используют ASCII-коды, они часто имеют сложную внутреннюю структуру, например, объектный код программы или архивный файл. Все ОС должны уметь распознавать хотя бы один тип файлов - их собственные исполняемые файлы.

Специальные файлы - это файлы, ассоциированные с устройствами ввода-вывода, которые позволяют пользователю выполнять операции ввода-вывода, используя обычные команды записи в файл или чтения из файла. Эти команды обрабатываются вначале программами ФС, а затем на некотором этапе выполнения запроса преобразуются ОС в команды управления соответствующим устройством. Специальные файлы, так же как и устройства ввода-вывода, делятся на блок-ориентированные и байт-ориентированные.

Каталог - это системные файлы, обеспечивающие поддержку структуры ФС. В каталоге содержится список файлов, входящих в него, и устанавливается соответствие между файлами и их характеристиками (атрибутами).

Например, на Рис. 2(а) показан простой исполняемый двоичный файл одной из версий системы UNIX. Хотя технически файл представляет собой всего лишь последовательность байтов, ОС станет исполнять файл только в том случае, если этот файл имеет соответствующий формат.

Файл состоит из пяти разделов:

- Заголовка. Заголовок начинается с идентификатора, идентифицирующего файл как исполняемый (чтобы предотвратить случайное исполнение файла другого формата). Следом за идентификатором в заголовке располагаются размеры различных частей файла, адрес начала исполнения файла и некоторые флаговые биты.
- Двоичный код программы (машинные инструкции).
- Двоичные данные.
- Реалокационные биты.
- Таблица символов.

Двоичный код программы и данные загружаются в ОЗУ и настраиваются на работу по адресу загрузки при помощи битов реалокации. Таблица символов используется для отладки – позволяет показать соответствие именами сущностей и адресами памяти (т.е. при отладке можно увидеть имя переменной либо название функции).

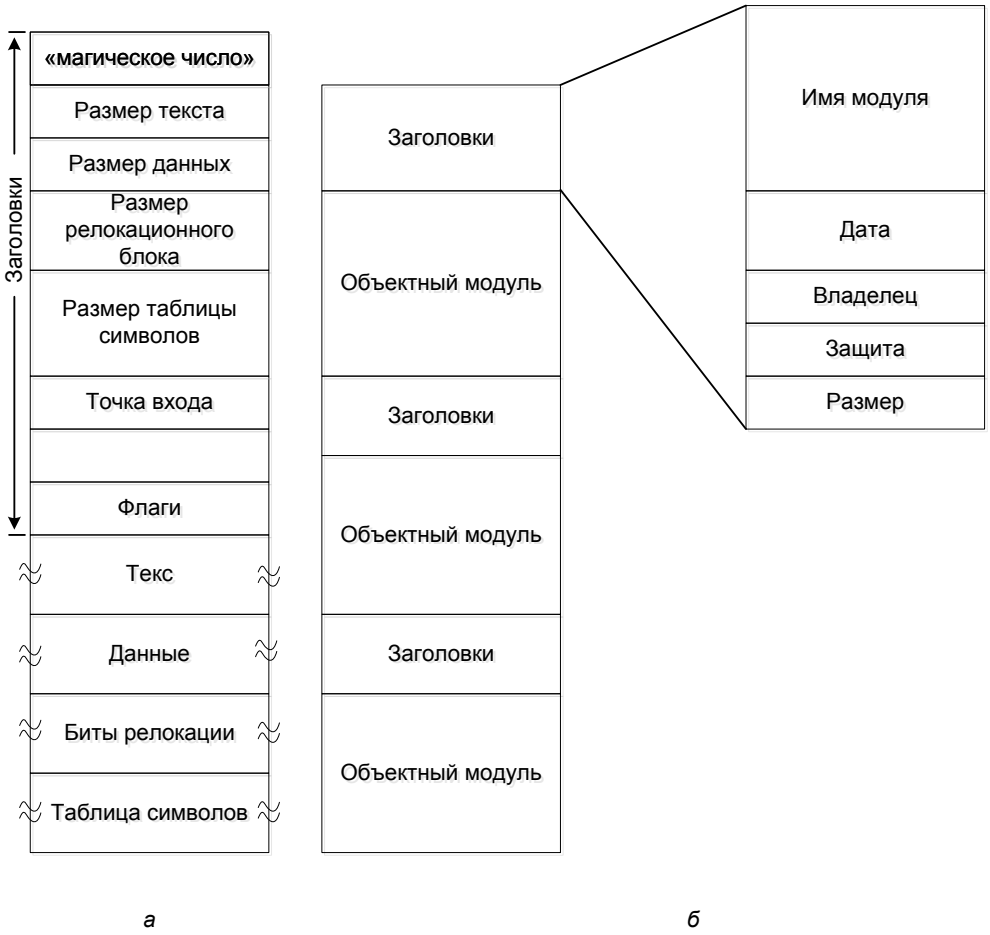

Рис. 2. Исполняемый файл (а); архив (б)

Доступ к файлам

В старых ОС предоставлялся только один тип доступа к файлам — **последовательный доступ**. В этих ОС процесс мог читать байты или записи файла только по порядку от начала к концу. Такой доступ к файлам появился, когда дисков еще не было и компьютеры оснащались стримерами. Поэтому даже в дисковых ОС при последовательном доступе к файлу имитировалось его чтение или запись на накопителе на магнитной ленте с возможностью многократной перемотки назад.

С появлением дисков стало возможным читать байты или записи файла в произвольном порядке или получать доступ к записям по ключу. Файлы, байты которых могут быть прочитаны в произвольном порядке, называются файлами **произвольного доступа**.

Для указания места начала чтения используются два метода:

- **В первом** случае каждая операция `read` задает позицию в файле.
- **При втором** способе используется **специальная операция поиска** `seek`, устанавливающая текущую позицию. После выполнения операции `seek` файл может читаться последовательно с текущей позиции.

В современных ОС все файлы автоматически являются файлами произвольного доступа.

В **Ошибка! Источник ссылки не найден.** представлены функции по работе с файлами в ОС Windows.

Выделим основные категории функций:

- Функции по управлению файлами.
- Функции для файлового ввода-вывода.
- Функции по работе с файлами, отображаемыми в память.
- Функции по работе с шифрованной ФС.
- Функции по работе с редиректором ФС (зарезервировано для 64 разрядных ОС Windows).
- Функции по работе с ZIP файлами.

Таблица 1. Основные функции по работе с файлами в ОС Windows (Win32 API)

Функция	Описание
AreFileApisANSI	Определяет, используются ли функции для файлового ввода-вывода используя кодовую страницу ANSI или OEM.
CheckNameLegalDOS8Dot3	Проверяет, может ли быть использовано имя для создания файла в ФС FAT.
CloseHandle	Закрывает дескриптор открытого объекта.
CopyFile	Копирует существующий файл в новый.
CopyFileEx	Копирует существующий файл в новый, и при этом, уведомляет приложение о процессе копирования посредством callback функции.
CreateFile	Открывает либо создает файл, директорию, физический диск, том, буфер консоли, стример, коммуникационные ресурсы, mailslot, или именованные каналы.
CreateHardLink	Устанавливает жесткие связи между существующим и новым файлом.
DeleteFile	Удаляет существующий файл.
FindClose	Закрывает дескриптор поиска файла, который был открыт функциями FindFirstFile, FindFirstFileEx или FindFirstStreamW.
FindFirstFile	Ищет поддиректорию, файл, которые по имени удовлетворяют шаблону (фильтру) поиска.
FindFirstFileEx	Ищет поддиректорию, файл, которые по имени и атрибутам удовлетворяют шаблону (фильтру) поиска.
FindFirstStreamW	Перечисляет первый файловый поток в указанном файле либо директории.
FindNextFile	Продолжает поиск файла либо директории.
FindNextStreamW	Продолжает поиск потока.
GetBinaryType	Определяет является ли файл исполняемым. Если файл исполняемый, функция указывает тип исполняемого файла.
GetCompressedFileSize	Возвращает реальный размер байт необходимый для хранения файла на диске.
GetFileAttributes	Возвращает набор атрибутов ФС FAT для указанного файла либо директории.
GetFileAttributesEx	Возвращает атрибуты для файла либо директории.
GetFileInformationByHandle	Возвращает информацию для указанного файла.
GetFileSize	Возвращает размер файла. Размер файла ограничен размером DWORD.
GetFileSizeEx	Возвращает размер файла.
GetFileTime	Возвращает дату и время создания, последнего доступа, последней модификации файла.

GetFileType	Возвращает тип указанного файла.
GetFullPathName	Возвращает полный путь и имя файлае.
GetLongPathName	Преобразует указанный путь к его длинной форме.
GetShortPathName	Возвращает путь в короткой форме к файлу.
GetTempFileName	Создает имя для временного файла.
GetTempPath	Возвращает путь к директории предназначенной для хранения временных файлов.
MoveFile	Перемещает существующий файл либо директорию и ее детей.
MoveFileEx	Переносит существующий файл либо директорию.
MoveFileWithProgress	Переносит существующий файл либо директорию. Возможно получить уведомление о процессе переноса посредством callback функции.
ReOpenFile	Пере открывает указанный файл с другими правами доступа, режимом разделения доступа и флагами.
ReplaceFile	Заменяет один файл другим и создает резервную копию оригинала.
SearchPath	Ищет указанный файл по указанному пути.
SetFileApisToANSI	Указывает использовать кодовую страницу символов ANSI для файлового ввода-вывода.
SetFileApisToOEM	Указывает использовать кодовую страницу символов OEM для файлового ввода-вывода.
SetFileAttributes	Устанавливает атрибуты файла.
SetFileSecurity	Устанавливает атрибуты безопасности для файла либо директории.
SetFileShortName	Устанавливает короткое имя для указанного файла.
SetFileTime	Устанавливает дату и время создания, последнего доступа и последней модификации для файла.
SetFileValidData	Устанавливает корректный размер (длину) данных файла.

Атрибуты файлов

У каждого файла есть имя и данные. Помимо этого все ОС связывают с каждым файлом также и другую информацию, например дату и время создания файла, а также его размер, эти и дополнительные сведения называют **атрибутами файла**.

В разных файловых системах могут использоваться в качестве атрибутов разные характеристики, например:

- информация о разрешенном доступе,
- пароль для доступа к файлу,
- владелец файла,
- создатель файла,
- признак "только для чтения",
- признак "скрытый файл",
- признак "системный файл",
- признак "архивный файл",

- признак "двоичный/символьный",
- признак "временный" (удалить после завершения процесса),
- признак блокировки,
- длина записи,
- указатель на ключевое поле в записи,
- длина ключа,
- времена создания, последнего доступа и последнего изменения,
- текущий размер файла,
- максимальный размер файла.

Кроме атрибутов с файлами ассоциируется и другая информация, которая приведена в Таблица 2. Эта информация доступна посредством класса `FileInfo` в Microsoft .NET, либо же через целый набор функций Win32 API.**Ошибка! Источник ссылки не найден.**

Таблица 2. Информация о файлах в ОС Windows

Информация	Описание
CreationTime	Время создания файла.
CreationTimeUtc	Время создания файла в формате UTC (coordinated universal time).
Directory	Экземпляр родительской директории.
DirectoryName	Имя родительской директории.
Extention	Расширение файла.
LastAccessTime	Время последнего доступа к файлу или директории.
LastAccessTimeUtc	Время последнего доступа к файлу или директории в формате UTC.
LastWriteTime	Время последней записи в файл или директорию.
LastWriteTimeUtc	Время последней записи в файл или директорию в формате UTC.
Length	Размер файла.
Name	Имя файла.

Операции с файлами

Перечислим наиболее часто встречающиеся системные вызовы для работы с файлами:

1. **Create (создание)**. Файл создается без данных. Этот системный вызов объявляет о появлении нового файла и позволяет установить некоторые его атрибуты.

2. **Delete (удаление)**. Когда файл уже более не нужен, его удаляют, чтобы освободить пространство на диске. Этот системный вызов присутствует в каждой ОС.

3. **Open (открытие)**. Прежде чем использовать файл, процесс должен его открыть. Системный вызов open позволяет системе прочитать в ОЗУ атрибуты файла и список дисковых адресов для быстрого доступа к содержимому файла при последующих вызовах.

4. **Close (закрытие)**. Когда все операции с файлом закончены, атрибуты и дисковые адреса более не нужны, поэтому файл следует закрыть, чтобы освободить пространство во внутренней таблице. Многие ОС позволяют одновременно открыть ограниченное количество файлов. Запись на диск производится поблочно, а закрытие файла вызывает запись последнего блока файла, даже если этот блок еще не заполнен до конца.

5. **Read (чтение)**. Чтение данных из файла. Обычно байты поступают с текущей позиции в файле. Вызывающий процесс должен указать количество требуемых данных и предоставить для них буфер.

6. **Write (запись)**. Запись данных в файл, также в текущую позицию в файле. Если текущая позиция находится в конце файла, размер файла автоматически увеличивается. В противном случае запись производится поверх существующих данных, которые теряются навсегда.

7. **Append (добавление)**. Этот системный вызов представляет собой усеченную форму вызова write. Он может только добавлять данные к концу файла. В ОС с минимальным набором системных вызовов может не быть данного системного вызова.

8. **Seek (поиск)**. Для файлов произвольного доступа требуется способ указать, где располагаются данные в файле. Данный системный вызов устанавливает файловый указатель в определенную позицию в файле. После выполнения данного системного вызова данные могут читаться или записываться в этой позиции.

9. **Get\Set attributes (получение\установка атрибутов)**. Процессам часто для выполнения их работы бывает необходимо получить\установить атрибуты файла. Например, для сборки программ, состоящих из большого числа отдельных исходных файлов, в системе UNIX часто используется программа `make`. Эта программа исследует время изменения всех исходных и объектных файлов, благодаря чему обходится компиляцией минимального количества файлов. Для выполнения этой работы ей требуется получить атрибуты файлов.

Пример приложения

Данное консольное .NET приложение выводит список файлов, которые находятся в указанной директории.

```
using System;
using System.IO;
class DirectoryLister
{
    public static void Main(string[] args)
    {
        string path = ".";
        if (args.Length > 0)
        {
            if (File.Exists(args[0])
            {
                path = args[0];
            }
            else
            {
                Console.WriteLine("{0} not found; using current directory:",
                    args[0]);
            }

        DirectoryInfo dir = new DirectoryInfo(path);
        foreach (FileInfo f in dir.GetFiles("*.exe"))
        {
            string name = f.Name;
            long size = f.Length;
            DateTime creationTime = f.CreationTime;
            Console.WriteLine("{0,-12:N0} {1,-20:g} {2}", size,
                creationTime, name);
        }
    }
```

```
}
```

Результат работы приложения будет выглядеть так:

```
953          7/20/2000 10:42 AM   C:\MyDir\Bin\paramatt.exe
664          7/27/2000 3:11 PM    C:\MyDir\Bin\tst.exe
403          8/8/2000 10:25 AM    C:\MyDir\Bin\dirlist.exe
```

Исходный код приложения C++, с использованием Win32 API для демонстрации операции: копирования, установки и получения атрибутов файла.

```
#include <windows.h>
#include <stdio.h>
void main()
{
   WIN32_FIND_DATA FileData;
   HANDLE hSearch;
   DWORD dwAttrs;
   TCHAR szDirPath[] = TEXT("c:\\TextRO\\");
   TCHAR szNewPath[MAX_PATH];
   BOOL fFinished = FALSE;
// Create a new directory.
   if (!CreateDirectory(szDirPath, NULL))
   {
      printf("Could not create new directory.\n");
      return;
   }
// Start searching for text files in the current directory.
   hSearch = FindFirstFile(TEXT("*.txt"), &FileData);
   if (hSearch == INVALID_HANDLE_VALUE)
   {
      printf("No text files found.\n");
      return;
   }
// Copy each .TXT file to the new directory
// and change it to read only, if not already.
   while (!fFinished)
   {
      lstrcpy(szNewPath, szDirPath);
      lstrcat(szNewPath, FileData.cFileName);
      if (CopyFile(FileData.cFileName, szNewPath, FALSE))
      {
         dwAttrs = GetFileAttributes(FileData.cFileName);
         if (dwAttrs==INVALID_FILE_ATTRIBUTES) return;
         if (!(dwAttrs & FILE_ATTRIBUTE_READONLY))
         {
            SetFileAttributes(szNewPath,
                dwAttrs | FILE_ATTRIBUTE_READONLY);
         }
      }
```

```
      else
      {
         printf("Could not copy file.\n");
         return;
      }
      if (!FindNextFile(hSearch, &FileData))
      {
         if (GetLastError() == ERROR_NO_MORE_FILES)
         {
            printf("Copied all text files.\n");
            fFinished = TRUE;
         }
         else
         {
            printf("Could not find next file.\n");
            return;
         }
      }
   }
// Close the search handle.
   FindClose(hSearch);
}
```

Файлы, отображаемые на адресное пространство памяти

Многие программисты считают описанный в предыдущем примере доступ неудобным, особенно по сравнению с доступом к обычной памяти. По этой причине в некоторых ОС, начиная с системы MULTICS, был предоставлен способ отображения файлов на адресное пространство работающего процесса. Концептуально можно представить себе два новых системного вызова: `map` и `unmap`. Первый системный вызов принимает на входе два параметра: имя файла и виртуальный адрес памяти, по которому ОС отображает указанный файл.

Таблица 3. Функции в ОС Windows по работе с файлами, отображаемыми в адресное пространство

Функция	Описание
CreateFileMapping	Создает или открывает именованный или неименованный объект отображения для указанного файла.
FlushViewOfFile	Пишет на диск диапазон байтов из отображения файла.
MapViewOfFile	Отображает вид (отображение) файла в адресное пространство вызывающего процесса.
MapViewOfFileEx	Отображает вид (образ) файла в адресное пространство вызывающего процесса. В дополнение, можно указать адрес памяти, куда можно отобразить файл.
OpenFileMapping	Открывает именованный объект отображения.
UnmapViewOfFile	Убирает отображения для образа файла из адресного пространства вызывающего процесса.

Пример исходного кода

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define BUF_SIZE 256
TCHAR szName[] = TEXT("MyFileMappingObject");
TCHAR szMsg[] = TEXT("Message from first process");
void main()
{
   HANDLE hMapFile;
   LPCTSTR pBuf;
   hMapFile = CreateFileMapping(
                 INVALID_HANDLE_VALUE,    // use paging file
                 NULL,                    // default security
                 PAGE_READWRITE,          // read/write access
                 0,                       // max. object size
                 BUF_SIZE,                // buffer size
                 szName);                 // name of mapping object
   if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE)
   {
      printf("Could not create file mapping object (%d).\n",
             GetLastError());
      return;
   }
   pBuf = (LPTSTR) MapViewOfFile(hMapFile,   // handle to map object
                        FILE_MAP_ALL_ACCESS, // read/write permission
                        0,
                        0,
                        BUF_SIZE);
   if (pBuf == NULL)
   {
      printf("Could not map view of file (%d).\n",
             GetLastError());
      return;
   }
   CopyMemory((PVOID)pBuf, szMsg, strlen(szMsg));
   getch();
   UnmapViewOfFile(pBuf);
   CloseHandle(hMapFile);
}
```

Предположим, например, что некий файл размером 64 Кбайт отображается в виртуальную память, начиная с адреса 512 К. После этого любая команда CPU, читающая байт по адресу 512 К, получит байт 0 этого файла и т. д. Запись по адресу 512 К+21000 изменит байт 21000 файла. Когда процесс завершает свою работу, модифицированный файл остается на диске, как если бы он был изменен системными вызовами `seek` и `write`.

Для реализации отображения файлов на память изменяются системные внутренние таблицы. При обращении к памяти по адресу от 512 до 576 К происходит прерывание из-за отсутствия страницы, обработчик которого предоставляет считанную в память страницу 0 файла. При записи происходит приблизительно тоже самое, но страница памяти, на которую отображается страница файла, помечается как модифицированная. Если потом эта страница удаляется из памяти алгоритмом замены страниц, она записывается в соответствующее место файла. После завершения процесса все модифицированные страницы сохраняются в соответствующих файлах.

Отображение файлов на память лучше всего работает в ОС, поддерживающей сегментацию. В такой системе каждый файл может быть отображен на свой собственный сегмент, так чтобы байт k файла был также байтом k сегмента. На Рис. 3(а) показан процесс с двумя сегментами, исполняемым кодом программы и данными. Предположим, что процесс копирует файлы. Сначала он отображает на сегмент исходный файл, например abc. Затем он создает пустой сегмент и отображает его на выходной файл, xyz. В результате получается ситуация, показанная на Рис. 3(б).

Рис. 3. Сегментированный процесс до отображения файлов на адресное пространство (а); процесс после отображения существующего файла abc на один сегмент и создания нового сегмента для файла xyz (б)

В этот момент процесс может скопировать сегмент-источник в сегмент-приемник с помощью обычного цикла копирования памяти. При этом не требуются системные вызовы `read` и `write`. Когда копирование закончено, процесс может выполнить системный вызов `UnmapViewOfFile`, чтобы удалить эти файлы из адресного пространства, и завершить свою работу. Выходной файл, xyz, теперь будет существовать на диске, как если бы он был создан обычным путем.

Хотя отображение на память устраняет необходимость обращения к системным вызовам ввода-вывода, вместе с ним появляются новые проблемы. Во-первых, в нашем примере ОС **трудно определить длину выходного файла**.

ОС знает номер максимальной модифицированной страницы, но она не может определить, сколько байтов было записано в эту страницу. Предположим, программа использует только страницу 0 и после выполнения цикла копирования все байты остались равны 0 (их исходному значению). Возможно, файл xyz состоит из 10 нулей. Может быть, он должен состоять из 100 нулей. Кто знает? ОС не может этого сказать. Все, что она может сделать — это создать файл, длина которого равна размеру страницы.

Вторая проблема может возникнуть **при попытке одного процесса открыть файл**, уже отображенный на адресное пространство другого процесса. Если первый процесс модифицирует страницу, это изменение не отразится на файле до тех пор, пока эта страница не будет сохранена в файле. ОС должна прилагать особые усилия, чтобы гарантировать, что оба процесса не работают с устаревшими версиями файла.

Третья проблема, связанная с отображением файлов на память, вызвана тем, что **файл может оказаться больше сегмента памяти и даже больше чем все виртуальное адресное пространство**. При этом единственный способ работы системного вызова `MapViewOfFile` состоит в отображении на память части файла. Хотя такой метод и работает, он все же менее удобен, чем отображение всего файла целиком.

Файловые системы: каталоги

Исторически сложилась следующая последовательность упорядочивания файлов:

- Одноуровневые каталоговые системы. Простейшая система – единственный каталог, который называется **корневым**. См. Рис. 4.
- Двухуровневые каталоговые системы. См. Рис. 5.
- Многоуровневые (иерархические) каталоговые системы. См. Рис. 6.

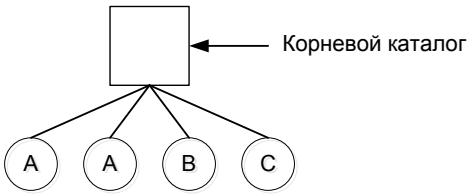

Рис. 4. Одноуровневая каталоговая система, содержащая 4 файла

Недостаток системы с одним каталогом и несколькими пользователями состоит в том, **что различные пользователи могут случайно использовать для своих файлов одинаковые имена**. Поэтому такая схема больше не используется в многопользовательских системах, но может применяться в небольших встроенных системах, например автомобильной системе, предназначенной для хранения профилей пользователей для небольшого количества водителей.

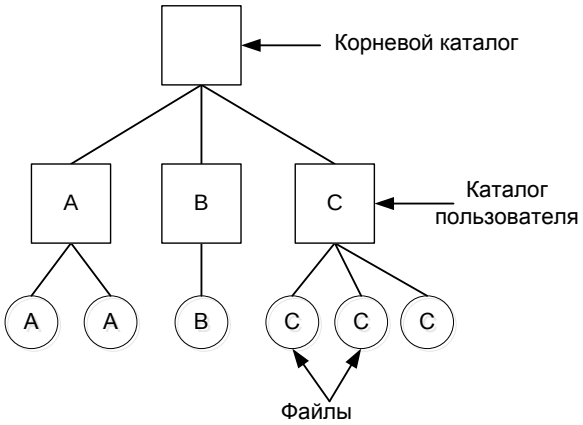

Рис. 5. Двухуровневая каталоговая система

Первым этапом в деле решения проблемы одинаковых имен файлов, созданных различными пользователями, можно считать систему, в которой каждому пользователю выделяется один каталог. При этом имена файлов, созданных одним пользователем, не конфликтуют с именами файлов другого пользователя. Схематично такая двухуровневая каталоговая система проиллюстрирована на Рис. 5. Такая организация могла, например, использоваться на многопользовательском компьютере или в простой сети персональных компьютеров, соединенных с общим файловым сервером локальной сетью.

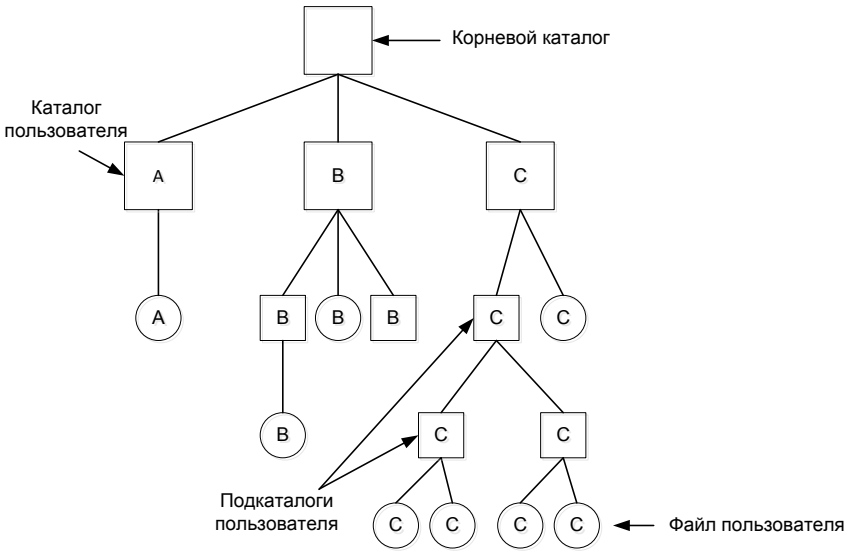

Рис. 6. Иерархия каталогов

Благодаря двухуровневой иерархии исчезают конфликты имен файлов между различными пользователями, но ее недостаточно для пользователей с большим числом файлов. Обычно пользователям бывает необходимо логически группировать свои файлы. Требуется некий гибкий способ, позволяющий объединять эти файлы в группы.

Следовательно, нужна некая общая иерархия (то есть дерево каталогов). При таком подходе каждый пользователь может сам создать себе столько каталогов, сколько ему нужно, группируя свои файлы естественным образом. Этот подход проиллюстрирован на Рис. 6. Здесь каталоги A, B и C, содержащиеся в корневом каталоге, принадлежат различным пользователям, два из которых создали подкаталоги для проектов, над которыми они работают.

Имя пути

При организации ФС в виде дерева каталогов требуется некоторый способ указания файла. Для этого обычно используются два различных метода:

В первом случае каждому файлу дается **абсолютное имя пути**, состоящее из имен всех каталогов от корневого до того, в котором содержится файл, и имени самого файла. Например, путь `/usr/ast/mailbox` означает, что корневой каталог содержит подкаталог `usr`, который, в свою очередь, содержит подкаталог `ast`, где находится файл `mailbox`. Абсолютные имена путей всегда начинаются от корневого каталога и являются уникальными. В системе UNIX компоненты пути разделяются косой чертой /. В Windows в качестве разделителя используется обратная косая черта \.. Таким образом, одно и то же имя пути в этих ОС будет выглядеть следующим образом:

Windows: c:\usr\ast\mailbox

UNIX: /usr/ast/mailbox

Если первой буквой имени пути был разделитель, это означало, независимо от используемого в качестве разделителя символа, что путь абсолютный.

Во втором случае - **относительное имя пути**. Оно используется вместе с концепцией **рабочего каталога** (также называемого **текущим каталогом**). Пользователь может назначить один из каталогов текущим рабочим каталогом. В этом случае все имена путей, не начинающиеся с символа разделителя, считаются относительными и отсчитываются относительно текущего каталога. Например, если текущим каталогом является `/usr/ast`, тогда к файлу с абсолютным путем `/usr/ast/mailbox` можно обратиться просто как к `mailbox`. Другими словами, команда UNIX:

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

и команда

```
cp mailbox mail box.bak
```

выполнят одно и то же действие, если рабочим каталогом является `/usr/ast`. Относительная форма часто оказывается более удобной, но она выполняет то же самое, что и абсолютная.

Некоторым программам бывает нужно получить доступ к файлам независимо от того, какой каталог является в данный момент текущим. В этом случае они всегда должны использовать абсолютные имена. Например, программе проверки правописания может понадобиться для выполнения работы прочитать файл `/usr/lib/dictionary`. В этом случае она должна использовать полное, абсолютное имя файла, так как она не знает, каким будет рабочий каталог при ее вызове. Абсолютное имя файла будет работать всегда, независимо от того, какой каталог является текущим в данный момент.

Операции с каталогами

Системные вызовы, управляющие каталогами, значительно менее схожи в различных системах, чем системные вызовы для работы с файлами. Чтобы дать представление о том, что они собой представляют и как работают, приведем следующий пример (взятый из UNIX):

1. **Create. Создать каталог.** Только что созданный каталог пуст и не содержит других элементов, кроме «> и «..», автоматически помещаемых в каталог операционной системой.

2. **Delete. Удалить каталог.** Может быть удален только пустой каталог. Элементы «.» и «..» файлами не являются и удалены быть не могут.

3. **Opendir. Открыть каталог.** После этой операции каталог может быть прочитан. Например, для распечатки всех файлов, содержащихся в каталоге, программа, создающая листинг, открывает каталог, чтобы прочитать имена всех содержащихся в нем файлов. Прежде чем каталог может быть прочитан, его следует открыть, подобно открытию и чтению файла.

4. **Closedir. Закрыть каталог.** Когда каталог прочитан, его следует закрыть, чтобы освободить место во внутренней таблице.

5. **Readdir. Прочитать следующий элемент открытого каталога.** В прежние времена было возможно читать каталоги с помощью обычного системного вызова `read`, но такой подход был небезопасен, так как требовал от программиста умения работать с внутренней структурой каталогов. Поэтому был создан отдельный системный вызов `readdir`, всегда возвращающий одну запись каталога стандартного формата независимо от используемой структуры каталогов.

6. **Rename. Переименование каталога.** Во многих отношениях каталоги аналогичны файлам и могут переименовываться так же, как и файлы.

7. **Link. Связывание представляет собой технику, позволяющую файлу появляться сразу в нескольких каталогах.** Этот системный вызов принимает в качестве входных параметров имя файла и имя пути и создает связь между ними. Таким образом, один и тот же файл может появляться сразу в нескольких каталогах. Подобная связь, увеличивающая на единицу счетчик `i`-узла файла (для учета количества каталогов со ссылками на этот файл), иногда называется жесткой связью.

8. **Unlink. Удаление ссылки на файл из каталога.** Если файл присутствует только в одном каталоге, то данный системный вызов удалит его из файловой системы. Если существует несколько ссылок на этот файл, то будет удалена только указанная ссылка, а остальные останутся. Этот системный вызов применяется для удаления файла в операционной системе UNIX.

Приведенный выше список содержит наиболее важные системные вызовы, но существует также множество других; например, для управления защитой информации.

Рассмотрим функции по управлению каталогами в ОС Windows посредством Win32 API.

Таблица 4. Функции в ОС по работе с каталогами

Функция	Описание
CreateDirectory	Создать новую директорию.
CreateDirectoryEx	Создать новую директорию с атрибутами по указанной директории-шаблону.
CreateFile	Открыть либо создать файловый объект.
DeleteFile	Удалить существующий файл.
FindCloseChangeNotification	Прекратить мониторинг уведомлений об изменениях объекта дескриптора.
FindFirstChangeNotification	Создать дескриптор уведомлений.
FindFirstFile	Поиск в директории файла либо поддиректории по совпадению имени.
FindFirstFileEx	Поиск в директории файла либо поддиректории по совпадению имени и атрибутов.
FindNextChangeNotification	Запрашивает у ОС формировать сигнал посредством дескриптора уведомления об изменениях в следующий

	раз, когда будут выявлены соответствующие изменения.
FindNextFile	Продолжает поиск файла.
GetCurrentDirectory	Получает текущую директорию для текущего процесса.
MoveFile	Переносит существующий файл либо директорию.
MoveFileEx	Переносит существующий файл либо директорию.
ReadDirectoryChangesW	Получает информацию об изменениях произошедших внутри директории.
RemoveDirectory	Удаляет существующую пустую директорию.
SetCurrentDirectory	Устанавливает текущую директорию для текущего процесса.

Реализация файловых систем

Структура файловой системы

ФС хранятся на дисках. Большинство дисков делятся на несколько разделов с независимой ФС на каждом разделе. Сектор 0 диска называется **главной загрузочной записью (MBR, Master Boot Record)** и используется для загрузки компьютера. В конце главной загрузочной записи содержится **таблица разделов**. В этой таблице хранятся начальные и конечные адреса (номера блоков) каждого раздела. Один из разделов помечен в таблице как **активный**. При загрузке компьютера BIOS считывает и исполняет MBR-запись, после чего загрузчик в MBR-записи определяет активный раздел диска, считывает его первый блок, называемый **загрузочным**, и исполняет его. Программа, находящаяся в загрузочном блоке, загружает ОС, содержащуюся в этом разделе.

Для единообразия каждый дисковый раздел начинается с загрузочного блока, даже если в нем не содержится загружаемой ОС. К тому же в этом разделе может быть в дальнейшем установлена ОС, поэтому зарезервированный загрузочный блок оказывается полезным.

Во всем остальном строение раздела диска меняется от системы к системе. Часто ФС содержат некоторые из элементов, показанных на Рис. 7. Один из таких элементов, называемый **суперблоком**, содержит ключевые параметры ФС и считывается в память при загрузке компьютера или при первом обращении к ФС. Типичная информация, хранящаяся в суперблоке, включает идентификатор, позволяющее различать системные файлы, количество блоков в ФС, а также другую ключевую административную информацию.

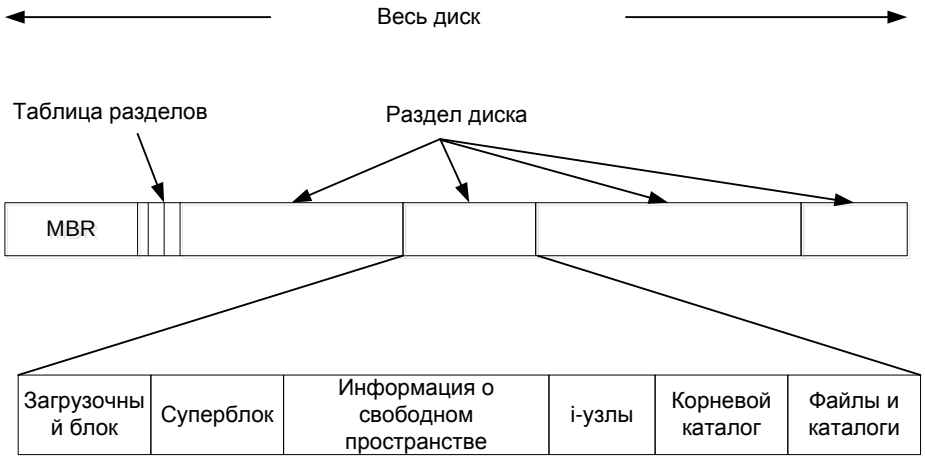

Рис. 7. Возможная структура ФС

Работа с дисками

В ОС Windows используются следующие функции по работе с дисками.

Таблица 5. Функции в ОС Windows по работе с дисками

Функция	Описание
CreateFile	Создает или открывает файловый объект.
DeleteFile	Удаляет существующий файл.
GetDiskFreeSpace	Получает информацию об указанном диске, включая количество свободного дискового пространства.
GetDiskFreeSpaceEx	Получает информацию об указанном диске, включая количество свободного дискового пространства.

Пример исходного кода

Приложение получает информацию об указанном диске.

```
#include <windows.h>
#include <stdio.h>
typedef BOOL (WINAPI *PGETDISKFREESPACEEX)(LPCSTR,
   PULARGE_INTEGER, PULARGE_INTEGER, PULARGE_INTEGER);
BOOL MyGetDiskFreeSpaceEx(LPCSTR pszDrive)
{
   PGETDISKFREESPACEEX pGetDiskFreeSpaceEx;
   __int64 i64FreeBytesToCaller, i64TotalBytes, i64FreeBytes;
   DWORD dwSectPerClust,
         dwBytesPerSect,
         dwFreeClusters,
         dwTotalClusters;
   BOOL fResult;
   pGetDiskFreeSpaceEx = (PGETDISKFREESPACEEX) GetProcAddress(
                          GetModuleHandle("kernel32.dll"),
                         "GetDiskFreeSpaceExA");
   if (pGetDiskFreeSpaceEx)
   {
      fResult = pGetDiskFreeSpaceEx (pszDrive,
                (PULARGE_INTEGER)&i64FreeBytesToCaller,
                (PULARGE_INTEGER)&i64TotalBytes,
                (PULARGE_INTEGER)&i64FreeBytes);
   // Process GetDiskFreeSpaceEx results.
      if(fResult)
      {
         printf("Total free bytes = %I64d\n", i64FreeBytes);
      }
      return fResult;
   }
```

```
   else
   {
      fResult = GetDiskFreeSpaceA (pszDrive,
                 &dwSectPerClust,
                 &dwBytesPerSect,
                 &dwFreeClusters,
                 &dwTotalClusters);
   // Process GetDiskFreeSpace results.
      if(fResult)
      {
         printf("Total free bytes = I64d\n",
              dwFreeClusters*dwSectPerClust*dwBytesPerSect);
      }
      return fResult;
   }
}
int main(int argc, char *argv[])
{
      MyGetDiskFreeSpaceEx ("C");
}
```

Работа с разделами

В ОС Windows используются следующие функции по работе с разделами.

Таблица 6. Функции в ОС Windows по работе с разделами

Функция	Описание
DefineDosDevice	Определяет, переопределяет или удаляет MS-DOS имя устройства.
GetDriveType	Определяет является ли дисковое устройство извлекаемым, фиксированным, CD-ROM, RAM диском либо сетевым диском.
GetLogicalDrives	Возвращает битовую маску представляющую список доступных на текущий момент дисковых устройств.
GetLogicalDriveStrings	Заполняет буфер со строками, что указывают допустимые приводы в системе.
GetVolumeInformation	Получает информацию о ФС и томе.
QueryDosDevice	Получает информацию о MS-DOS именах устройств.
SetVolumeLabel	Устанавливает имя для тома ФС.

Также существуют функции для монтирования разделов, более детально о них можно почерпнуть информацию из [5].

Что бы получить доступ к физическому диску и разделу следует использовать функцию Win32 API `CreateFile` с параметрами, показаннями в таблице 7.

После вызова `CreateFile` необходимо воспользоваться полученным дескриптором для функции `DeviceIoControl`. Это позволит произвести доступ к таблице разделов, хотя это и является потенциально опасным действием, т.к. любая запись на диск может привести к потере ФС. Для успешного вызова следует выполнить следующие условия:

- Вызов следует производить с административными привилегиями.
- Параметр `dwCreationDisposition` следует установить в `OPEN_EXISTING`.
- В случае открытия тома либо гибкого диска, параметр `dwShareMode` следует установить `FILE_SHARE_WRITE`.

В случае открытия физического устройства `x`, строка `lpFileName` строка должна иметь вид: \\.*PHYSICALDRIVE<x>*. Нумерация физических дисков начинается с 0.

Таблица 7. Примеры правильного наименования физических дисковых устройств

Строка	Значение
\\.\PHYSICALDRIVE0	Открыть первый физический привод.
\\.\PHYSICALDRIVE2	Открыть третий физический привод.

Пример исходного кода

Пример позволяет получить информацию о первом физическом приводе.

```
/* The code of interest is in the subroutine GetDriveGeometry. The
   code in main shows how to interpret the results of the call. */
#include <windows.h>
#include <winioctl.h>
#include <stdio.h>
BOOL GetDriveGeometry(DISK_GEOMETRY *pdg)
{
  HANDLE hDevice;               // handle to the drive to be examined
  BOOL bResult;                 // results flag
  DWORD junk;                   // discard results
  hDevice = CreateFile("\\\\.\\PhysicalDrive0",  // drive to open
                    0,                // no access to the drive
                    FILE_SHARE_READ | // share mode
                    FILE_SHARE_WRITE,
                    NULL,             // default security attributes
                    OPEN_EXISTING,    // disposition
                    0,                // file attributes
                    NULL);            // do not copy file attributes
  if (hDevice == INVALID_HANDLE_VALUE) // cannot open the drive
  {
    return (FALSE);
  }
  bResult = DeviceIoControl(hDevice,  // device to be queried
      IOCTL_DISK_GET_DRIVE_GEOMETRY,  // operation to perform
                             NULL, 0, // no input buffer
                            pdg, sizeof(*pdg),     // output buffer
                            &junk,                 // # bytes returned
                            (LPOVERLAPPED) NULL);  // synchronous I/O
  CloseHandle(hDevice);
  return (bResult);
}
```

```
int main(int argc, char *argv[])
{
  DISK_GEOMETRY pdg;            // disk drive geometry structure
  BOOL bResult;                 // generic results flag
  ULONGLONG DiskSize;           // size of the drive, in bytes
  bResult = GetDriveGeometry (&pdg);
  if (bResult)
  {
    printf("Cylinders = %I64d\n", pdg.Cylinders);
    printf("Tracks/cylinder = %ld\n", (ULONG) pdg.TracksPerCylinder);
    printf("Sectors/track = %ld\n", (ULONG) pdg.SectorsPerTrack);
    printf("Bytes/sector = %ld\n", (ULONG) pdg.BytesPerSector);
    DiskSize = pdg.Cylinders.QuadPart * (ULONG)pdg.TracksPerCylinder *
      (ULONG)pdg.SectorsPerTrack * (ULONG)pdg.BytesPerSector;
    printf("Disk size = %I64d (Bytes) = %I64d (Gb)\n", DiskSize,
           DiskSize / (1024 * 1024 * 1024));
  }
  else
  {
    printf ("GetDriveGeometry failed. Error %ld.\n", GetLastError ());
  }
  return ((int)bResult);
}
```

Реализация файлов

Вероятно, наиболее важным моментом в реализации хранения файлов является учет соответствия блоков диска файлам. Для определения того, какой блок, какому файлу принадлежит, в различных ОС применяются различные методы.

Непрерывные файлы

Простейшей схемой выделения файлам определенных блоков на диске является система, в которой файлы представляют собой непрерывные наборы соседних блоков диска. Тогда на диске, состоящем из блоков по 1 Кбайт, файл размером в 50 Кбайт будет занимать 50 последовательных блоков. При 2-килобайтных блоках такой файл займет 25 соседних блоков.

Пример непрерывных файлов показан на Рис. 8(а). Здесь показаны первые 40 блоков диска, начиная с блока 0, слева. Вначале диск был пуст. Затем на диск, начиная с блока 0, был записан файл A длиной в четыре блока. После него был записан шестиблочный файл B, впритык к файлу A. Каждый файл начинается с нового блока, так что если длина файла A была равна 3,5 блока, некоторое место в конце последнего блока файла пропадает. На рисунке всего показано семь файлов. Каждый следующий файл начинается с блока, следующего за последним блоком предыдущего файла. Затенение используется только для того, чтобы было легче различать отдельные файлы.

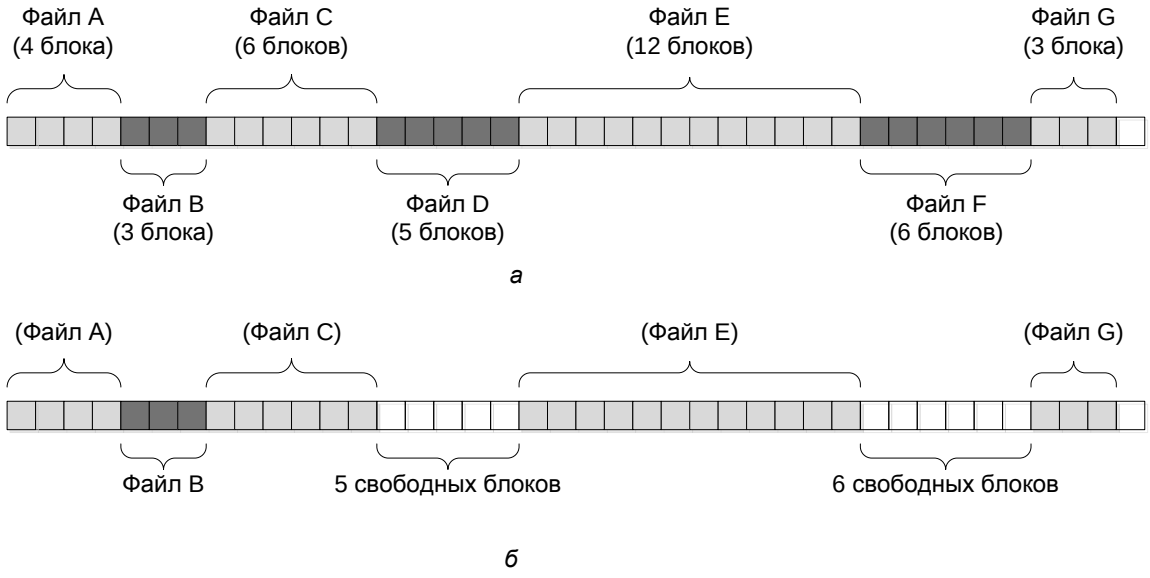

Рис. 8. Семь непрерывных файлов на диске (а); состояние диска после удаления двух файлов (б)

У непрерывных файлов есть два существенных преимущества:

Во-первых, такую **систему легко реализовать**, так как системе, чтобы определить, какие блоки принадлежат тому, или иному файлу, нужно следить всего лишь за двумя числами: номером первого блока файла и числом блоков в файле. Зная первый блок файла, любой другой его блок легко получить при помощи простой операции сложения.

Во-вторых, при работе с **непрерывными файлами производительность просто превосходна**, так как весь файл может быть прочитан с диска за одну операцию. Требуется только одна операция поиска (для первого блока). После этого более не нужно искать цилиндры и тратить время на ожидания вращения диска, поэтому данные могут считываться с максимальной скоростью, на которую способен диск. **Таким образом, непрерывные файлы легко реализуются и обладают высокой производительностью.**

К сожалению, у такого способа распределения дискового пространства имеется **серьезный недостаток**: **со временем диск становится фрагментированным**. Чтобы понять, как это происходит, рассмотрим Рис. 8(б). Два файла, D и F, были удалены. Когда файл удаляется, его блоки освобождаются, оставляя промежутки свободных блоков на диске. По мере удаления файлов диск становится все более «дырявым».

Вначале эта фрагментация не представляет проблемы, так как каждый новый файл может быть записан в конец диска, вслед за предыдущим файлом. Однако, в конце концов, диск заполнится и либо потребуется специальная операция по уплотнению используемого пространства диска, либо надо будет изыскать способ использовать свободное пространство на месте удаленных файлов. Для повторного использования освободившегося пространства потребуется содержать список пустых участков, что в принципе выполнимо. Однако при создании нового файла будет необходимо знать его окончательный размер, чтобы выбрать для него участок подходящего размера.

Связные списки

Второй метод размещения файлов состоит в представлении каждого файла в виде связного списка из блоков диска, как показано на Рис. 9. Первое слово каждого блока используется как указатель на следующий блок. В остальной части блока хранятся данные.

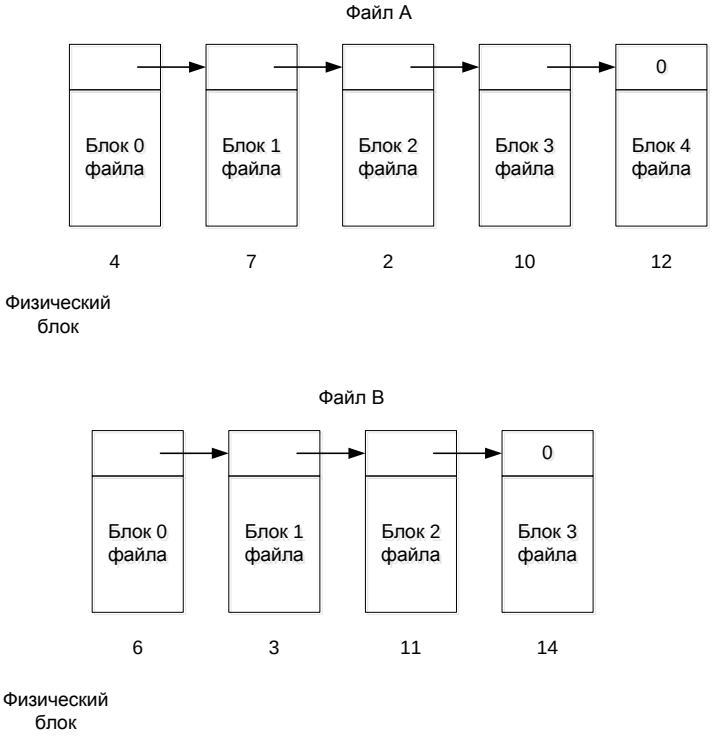

Рис. 9. Размещение файла в виде связного списка блоков диска

В отличие от систем с непрерывными файлами, такой метод позволяет использовать каждый блок диска. Нет потерь дискового пространства на фрагментацию (кроме потерь в последних блоках файла). **Кроме того, в каталоге нужно хранить только адрес первого блока файла.** Всю остальную информацию можно найти там.

С другой стороны, хотя последовательный доступ к такому файлу несложен, **произвольный доступ будет довольно медленным**. Чтобы получить доступ к блоку n, ОС должна сначала прочитать первые $n-1$ блоков по очереди. Очевидно, такая схема оказывается очень медленной.

Кроме того, размер блока уменьшается на несколько байтов, требуемых для хранения указателя. Хотя это и не смертельно, но размер блока, не являющийся степенью двух, будет менее эффективным, так как многие программы читают и пишут блоками по 512, 1024, 2048 и т. д. байтов. Если первые несколько байтов каждого блока будут заняты указателем на следующий блок, то для чтения блока полного размера придется считывать и объединять два соседних блока диска, для чего потребуется выполнение дополнительных операций.

Связный список при помощи таблицы в памяти

Оба недостатка предыдущей схемы организации файлов в виде связных списков могут быть устранены, если указатели на следующие блоки хранить не прямо в блоках, а в отдельной таблице, загружаемой в память. На Рис. 10 показан внешний вид такой таблицы для файлов с Рис. 9. На обоих рисунках показаны два файла. Файл А использует блоки диска 4, 7, 2, 10 и 12, а файл В использует блоки диска 6, 3, 11 и 14. С помощью таблицы, показанной на Рис. 10, можем начать с блока 4 и следовать по цепочке до конца файла. То же может быть сделано для второго файла, если начать с блока 6. Обе цепочки **завершаются специальным маркером** (например -1), не являющимся допустимым номером блока. Такая таблица, загружаемая в оперативную память, называется **FAT-таблицей (File Allocation Table — таблица размещения файлов)**.

Рис. 10. Связный список с таблице в памяти

Эта схема позволяет использовать для данных весь блок. Кроме того, случайный доступ при этом становится намного проще. Хотя для получения доступа к какому-либо блоку файла все равно понадобится проследовать по цепочке по всем ссылкам вплоть до ссылки на требуемый блок, однако в данном случае вся цепочка ссылок уже хранится в памяти, поэтому для следования по ней не требуются дополнительные дисковые операции. Как и в предыдущем случае, в каталоге достаточно хранить одно целое число (номер начального блока файла) для обеспечения доступа ко всему файлу.

Основной недостаток этого метода состоит в том, что вся таблица должна постоянно находиться в памяти.

I-узлы

Последний метод отслеживания принадлежности блоков диска файлам состоит в связывании с каждым файлом структуры данных, называемой **i-узлом (index node — индекс-узел)**, содержащей атрибуты файла и адреса блоков файла. Простой пример i-узла показан на Рис.11. **Большое преимущество** такой схемы перед хранящейся в памяти таблицей из связных списков заключается в том, что **каждый конкретный i-узел должен находиться в памяти только тогда, когда соответствующий ему файл открыт**. Если каждый i-узел занимает n байт, а одновременно открыто может быть k файлов, то для массива i-узлов потребуется в памяти всего kn байтов.

Рис.11. Пример i-узла

Для диска из n блоков потребуется n записей в таблице. Таким образом, размер таблицы линейно растет с ростом размера диска. Для схемы i-узлов, напротив, требуется массив в памяти с размером, пропорциональным максимальному количеству файлов, которые могут быть открыты одновременно.

С такой схемой связана проблема, заключающаяся в том, что при выделении каждому файлу фиксированного количества дисковых адресов этого количества может не хватить. Одно из решений заключается в резервировании последнего дискового адреса не для блока данных, а для следующего адресного блока, как показано на Рис.11. Более того, можно создавать целые цепочки и даже деревья адресных блоков.

Реализация каталогов

При открытии файла используется имя пути, чтобы найти запись в каталоге. Запись в каталоге указывает на адреса блоков диска.

В зависимости от системы это может быть:

- дисковый адрес всего файла (для непрерывных файлов);
- номер первого блока (связные списки);
- номер i-узла.

Одна из основных задач каталоговой системы преобразование ASCII-имени в информацию, необходимую для нахождения данных.

Также она хранит атрибуты файлов.

Варианты хранения атрибутов:

- В каталоговой записи (MS-DOS), Рис. 12(а);
- В i-узлах (UNIX), Рис. 12(б).

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 12. Простой каталог, содержащий записи фиксированной длины с атрибутами и дисковыми адресами (а); каталог, в котором каждая запись является просто ссылкой на i-узел (б)

Реализация длинных имен файлов

До сих пор предполагалось, что файлы имеют короткие имена фиксированной длины:

- В системе MS-DOS файл может иметь имя длиной от 1 до 8 символов, а также расширение длиной до 3 символов.
- В системе UNIX v7 имена файлов могут быть от одного до 14 символов, включая расширение.
- Однако почти всеми современными ОС поддерживаются более длинные имена переменной длины.

Простейший способ состоит в установке ограничения на длину имени файла, обычно 255 символов, и использовании одной из схем, показанных на Рис. 13. Такой способ прост, но он расходует много места в каталоге, так как длинные имена обычно бывают далеко не у всех файлов. Следовательно, для более эффективного использования дискового пространства желательно использовать другую структуру.

Один из альтернативных подходов состоит в отказе от предположения о том, что все записи в каталоге должны иметь один и тот же размер. При таком подходе каждая запись в каталоге начинается с порции фиксированного размера, обычно начинающейся с длины записи, за которой следуют данные в фиксированном формате — идентификатор владельца, дата создания, информация о защите и прочие атрибуты. Следом за заголовком фиксированной длины идет часть записи переменной длины, содержащая имя файла, Рис. 13(а). На рисунке показаны три описателя файла, `project-budget`, `personnel` и `foo`. Имя каждого файла завершается специальным символом (обычно 0), обозначенным на рисунке перечеркнутыми квадратиками. Чтобы каждая запись в каталоге могла начинаться с границы слова, имя каждого файла дополняется до целого числа слов байтами, показанными на рисунке затенёнными прямоугольниками.

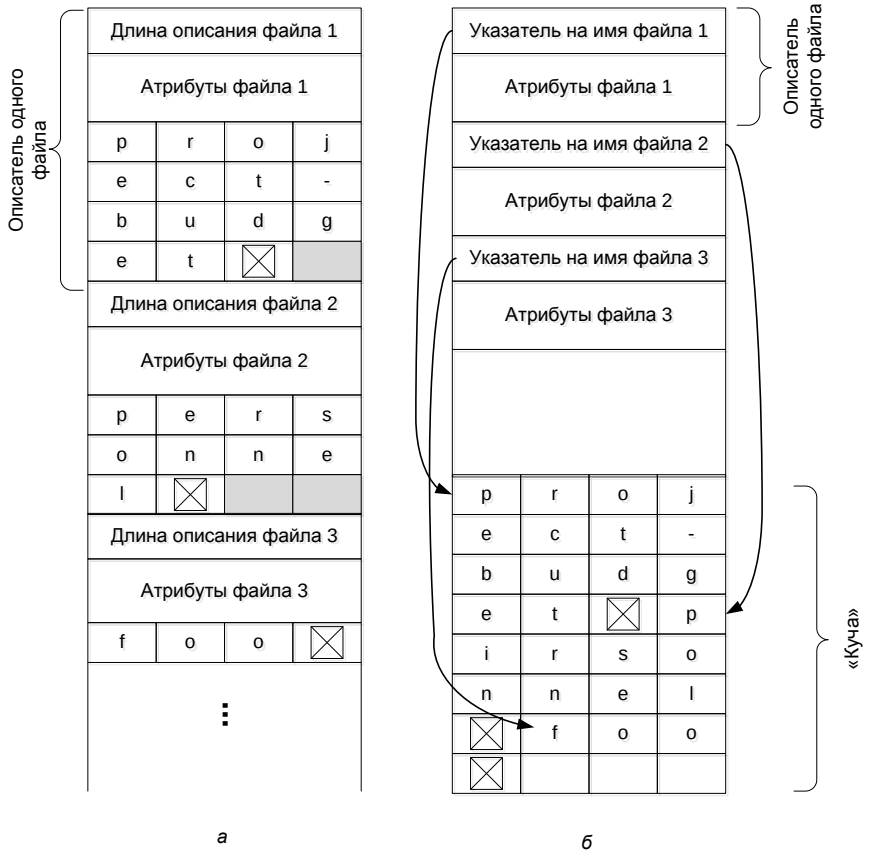

Рис. 13. Два варианта реализации длинных имен: прямо в записи каталога (а); в «куче» (б)

Недостаток этого метода состоит в том, что при удалении файла в каталоге остается промежуток переменной длины, в который описатель следующего файла может не поместиться. Эта проблема аналогична проблеме хранения на диске непрерывных файлов, хотя уплотнить каталог значительно легче, чем весь диск. Другая проблема связана с тем, что каталоговые записи переменной длины могут занимать сразу две страницы памяти. При чтении такой каталоговой записи может возникнуть прерывание из-за отсутствия в ОЗУ следующей страницы.

Другой метод реализации длинных имен файлов заключается в том, чтобы сделать все записи каталога фиксированной (равной) длины и хранить в них только указатели на имена, а сами имена хранить отдельно в «куче», в конце каталога, как показано на Рис. 13(б). Преимущество этого метода состоит в том, что при удалении файла освободившееся место в каталоге точно подойдет для нового описателя файла. Тем не менее «кучу» придется все так же «разгребать», и при чтении длинного имени из нее также может возникнуть прерывание из-за отсутствия страницы памяти. Однако имена файлов уже не должны начинаться с границы слов, поэтому символы-заполнители не потребуются.

Для очень больших каталогов, содержащих много тысяч файлов, такой поиск может занять довольно много времени. **Один из способов ускорить поиск** файла состоит в **использовании хэш-таблицы в каждом каталоге.**

Алгоритм записи файла:

- Создается хэш-таблица в начале каталога, с размером n (n записей).
- Для каждого имени файла применяется хэш-функция, такая, чтобы при хэшировании получалось число от 0 до n-1.
- Исследуется элемент таблицы соответствующий хэш-коду.
- Если элемент не используется, туда помещается указатель на описатель файла (описатели размещены вслед за хэш-таблицей).
- Если используется, то создается связный список, объединяющие все описатели файлов с одинаковым хэш-кодом.

Алгоритм поиска файла:

- Имя файла хэшируется
- По хэш-коду определяется элемент таблицы

- Затем проверяются все описатели файла из связного списка и сравниваются с искомым именем файла
- Если имени файла в связном списке нет, это значит, что файла нет в каталоге.

Такой метод очень сложен в реализации, поэтому используется в тех системах, в которых ожидается, что каталоги будут содержать тысячи файлов.

Организация дискового пространства

Обычно файлы хранятся на диске, поэтому организация дискового пространства является основной заботой разработчиков ФС. Для хранения файла из n байтов возможно использование двух стратегий:

- выделение на диске n последовательных байтов
- или разбиение файла на несколько непрерывных блоков.

Та же дилемма присутствует в системах управления памяти, где имеется выбор между чистой сегментацией и страничной организацией памяти.

Как уже было показано, при хранении файла в виде непрерывной последовательности байтов возникает проблема, связанная с увеличением его размеров. Единственный способ увеличить непрерывный файл состоит в перемещении его на новое место на диске. Проблема существенна и для управления сегментами памяти, с той разницей, что перемещение сегмента в памяти является более быстрой операцией по сравнению с перемещением файла на диске. По этой причине почти все ФС хранят файлы в виде блоков фиксированного размера, расположенных в различных частях диска.

Размер блока

Как только принято решение хранить файлы блоками фиксированного размера, возникает вопрос о размере этих блоков. Учитывая организацию дисков, очевидными кандидатами на роль сегментов файлов являются сектор, дорожка и цилиндр диска (минусом такого выбора является зависимость этих параметров от устройств). В системе управления страницами памяти размер страницы также входит в число основных претендентов.

Если выбрать большую единицу хранения, такую как цилиндр, это будет означать, что любой файл, даже состоящий из одного байта, займет как минимум целый цилиндр. Исследования показали, что средний размер файла в системе UNIX около 1 Кбайт, так что при выделении каждому файлу 32-килобайтного блока будет расходоваться понапрасну 31/32 или 97% общего дискового пространства.

С другой стороны, при использовании маленьких единиц хранения каждый файл будет состоять из большого числа блоков. Для чтения каждого блока файла обычно требуется операция поиска нужного цилиндра и задержка вращения диска, поэтому чтение файла, состоящего из большого числа блоков, будет медленным.

Например, представьте себе диск с 128 Кбайт на дорожку, периодом вращения 8,33 мс и средним временем поиска 10 мс. При этом время, требующееся для чтения блока из k байтов, будет равно сумме времени поиска, задержки вращения и времени переноса данных:

$$10 + 4{,}165 + (k/131072)8.33$$

Жирная ломаная линия на Рис. 14 показывает зависимость скорости передачи данных от размера блока. Чтобы вычислить эффективность использования дискового пространства, сделаем предположение о среднем размере файла. Медианный размер файла в ОС Windows равен 1680 байт. Это означает, что половина файлов меньше 1680 байт, а другая половина больше этого размера. Следует заметить, что медианный размер представляет собой лучшую единицу измерения, чем средний, так как небольшое количество файлов может очень сильно повлиять на среднее значение, но не на медиану. Для простоты предположим, что все файлы имеют размер, равный 2 Кбайт, в результате чего получим штриховую линию, изображающую на Рис. 14 эффективность использования дискового пространства.

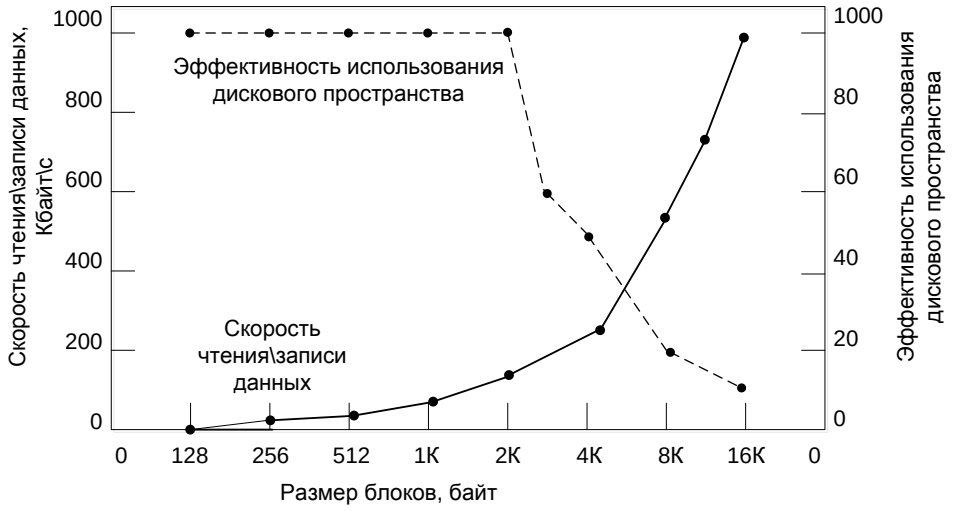

Рис. 14. Зависимость скорости чтения/записи данных диска (жирная линия, левая шкала) и эффективности использования дискового пространства (штриховая линия, правая шкала) от размера блоков. Все файлы по 2 Кбайт

Эти две кривые можно понимать следующим образом. **Время доступа к блоку практически полностью определяется временем поиска и задержкой вращения, поэтому**, если считать, что для доступа к блоку требуется 14 мс, чем больше данных удастся считать за одну такую операцию, тем лучше. **Таким образом, скорость чтения/записи данных растет пропорционально размеру блока до тех пор, пока блок не вырастет настолько, что важнее окажется время передачи блока.** При использовании небольших блоков, размер которых составляет степень числа два, и 2-килобайтных файлов потерь дискового пространства нет. Однако при хранении 2-килобайтных файлов в 4-килобайтных блоках теряется уже 50 % дискового пространства. В действительности мало файлов имеют длину, кратную размеру блока, поэтому какая-то часть дискового пространства всегда теряется в последнем блоке файла.

Учет свободных блоков

Выбора размера блоков, следует определить, как учитывать свободные и занятые блоки. Широкое применение получили два метода, показанные на Рис. 15.

Первый метод представляет собой использование **связного списка блоков диска**. При этом в каждом блоке списка содержится столько номеров свободных блоков, сколько может поместиться в один блок. При размере блока, равном 1 Кбайт, и 32-разрядных номерах блоков каждый блок списка свободных блоков может содержать номера 255 свободных блоков. (Одно 32-разрядное слово нужно для указателя на следующий блок списка). Для 16-гигабайтного диска потребуется список свободных блоков, состоящий из 16 794 блоков, чтобы содержать все 2^{24} номера дисковых блоков. Часто для списка резервируется нужное число блоков в начале диска.

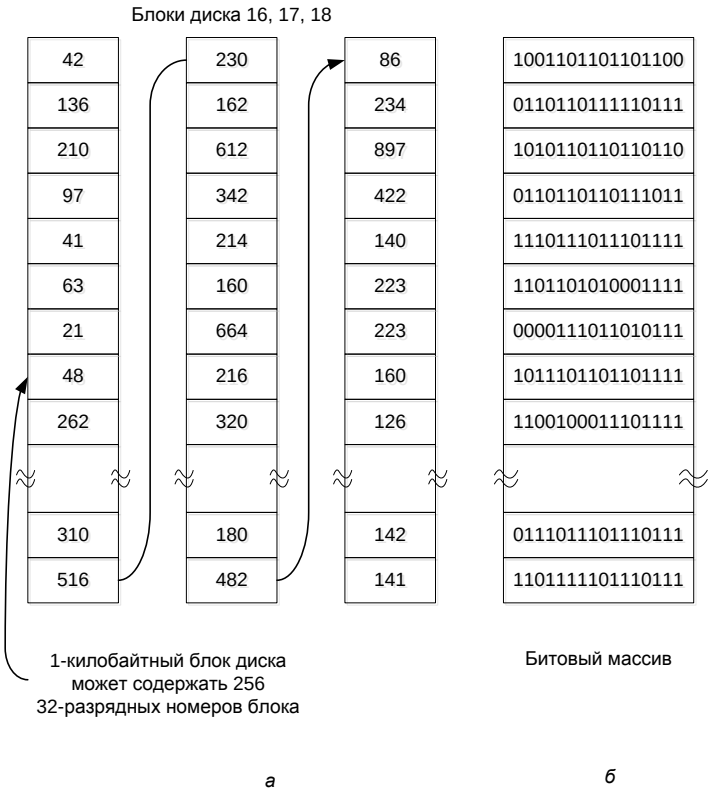

Рис. 15. Хранение информации о свободных блоках в виде списка (а); битовый массив (б)

Другой метод учёта свободного дискового пространства состоит в хранении этой информации в **виде битового массива** (иногда называемого бит-картой). При этом на каждый блок приходится всего по одному биту вместо 32. Свободные блоки обозначаются в массиве единицами, а занятые — нулями (или наоборот). 16 Гб диск состоит из 2^{24} килобайтных блоков, таким образом, для него требуется массив размером 2^{24} бит, то есть 2048 блоков.

При использовании метода учета свободных блоков при помощи списка в ОЗУ требуется хранить только один блок указателей. Когда создается файл, нужные блоки берутся из блока указателей. Когда указатели в этом блоке заканчиваются, читается новый блок с диска. Соответственно, при удалении файла его блоки освобождаются, а их номера добавляются в список, хранящийся в ОЗУ. Когда блок указателей наполняется, он записывается на диск.

При определенных обстоятельствах такой метод приводит к излишним операциям ввода-вывода. Рассмотрим ситуацию на Рис. 16(а), в которой в блоке указателей есть место только для еще двух записей (затененные прямоугольники обозначают указатели в блоке). При удалении трехблочного файла блок указателей переполняется и его нужно записать на диск, что приводит к ситуации, показанной на Рис. 16(б). Если теперь снова создается трехблочный файл, полный блок указателей должен быть снова прочитан с диска, что нас возвращает к ситуации на Рис. 16(а). Если этот трехблочный файл был временным файлом, то он вскоре опять удаляется, при этом опять блок указателей пишется на диск. Таким образом, когда указатели в блоке оказываются на границе блока, возникает необходимость в дополнительных операциях ввода-вывода.

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 16. Почти полный блок указателей свободных блоков в памяти и три блока указателей на диске (а); результат удаления трехблочного файла (б); альтернативная стратегия (в)

Существует метод, позволяющий избежать лишних операций ввода-вывода: полный блок указателей расщепляется на два. При этом из ситуации на Рис. 16(а) вместо ситуации на Рис. 16(б), переходим к ситуации на Рис. 16(в). При заполнении блока указателей в памяти этот блок записывается на диск не в виде заполненного до конца блока, а как блок, заполненный наполовину, то есть записывается, по сути,

половина указателей блока. В памяти остается блок с остальными указателями, то есть также заполненный наполовину блок. Таким образом, хранящийся в ОЗУ блок указателей максимально готов как к добавлению в него новых указателей, так и к освобождению хранящихся в нем.

При использовании битового массива также возможно хранение в памяти всего одного блока с обращением к диску за другим блоком, когда текущий блок переполняется или, наоборот, становится пустым. Дополнительное преимущество такого подхода состоит в том, что выделяемые файлу блоки будут располагаться близко друг к другу, в результате чего для доступа к файлу будет затрачено меньше времени на перемещение блока головок. Поскольку битовый массив представляет собой структуру данных фиксированного размера, то при (частичной) постраничной организации ядра битовый массив может быть помещен в виртуальную память, откуда можно получать страницы по мере надобности.

Дисковые квоты

Чтобы не допустить захвата пользователями слишком больших участков дискового пространства, многопользовательские ОС часто содержат механизм предоставления дисковых квот. Суть в том, что администратор назначает каждому пользователю максимальное количество файлов и максимальный суммарных используемый размер пространства, а ОС гарантирует, что пользователи не превышают выделенных им квот. Типичный механизм реализации этой идеи описан ниже.

Когда пользователь открывает файл, ОС находит его атрибуты и дисковые адреса и помещает их в таблицу в ОЗУ. Среди атрибутов находится элемент, сообщающий, кто является владельцем данного файла. Любые увеличения размеров файла будут учитываться в записи квоты для этого пользователя.

Вторая таблица содержит запись квоты для каждого пользователя, файл которого открыт в данный момент, даже если этот файл был открыт кем-либо другим.

Эта таблица показана на Рис. 17. Она извлекается из файла квот, хранящегося на диске. Когда все файлы закрыты, запись записывается обратно в файл.

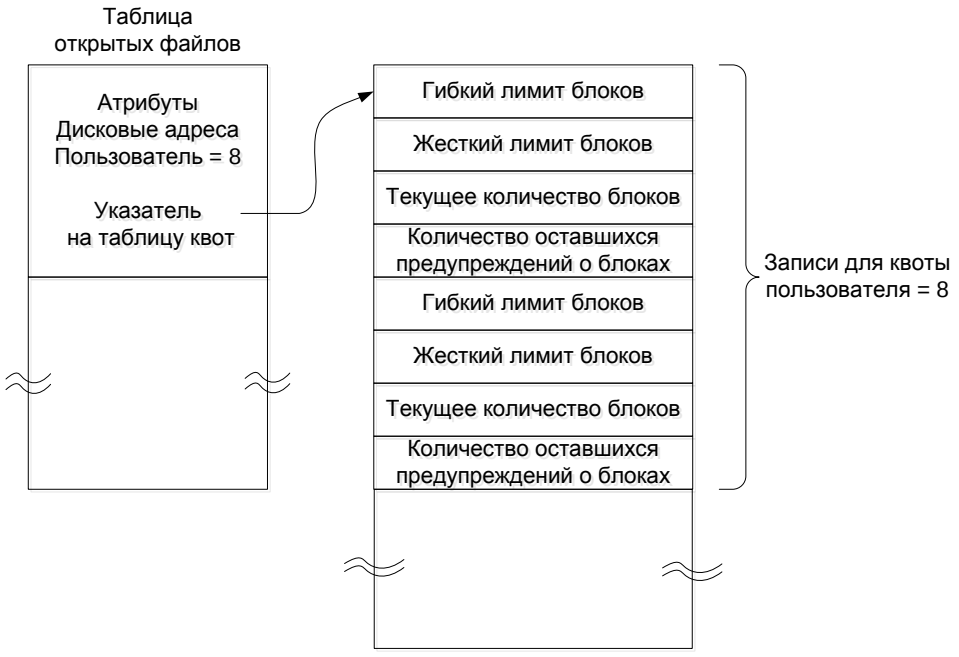

Рис. 17. Квоты учитываются для каждого пользователя в таблице квот

Когда в таблице открытых файлов создается новый элемент, в него помещается указатель на запись квоты владельца файла. При каждом добавлении нового блока к файлу общее количество блоков, числящееся за пользователем, увеличивается и сравнивается с гибким и жестким лимитом. Гибкий лимит может быть превышен, но жесткий нет. При достижении жесткого лимита любая попытка добавить блок к файлу будет завершаться ошибкой. Аналогично проверяется количество файлов пользователя.

Когда пользователь пытается зарегистрироваться в системе, ОС считывает его файл квот, проверяя, не превысил ли пользователь гибкие пределы по числу блоков или числу файлов. Если один из пределов превышен, ОС выдает предупреждение, а счетчик оставшихся предупреждений уменьшается на единицу. Когда счетчик предупреждений достигает нуля, ОС отказывает пользователю в регистрации. Чтобы снова получить возможность входить в систему, пользователю необходимо обратиться к системному администратору.

Таким образом, пользователь может превысить свои гибкие лимиты, при условии, что перед выходом из системы он удалит лишние файлы, превышающие эти пределы. Жесткие лимиты превышены быть не могут.

Надежность файловой системы

Резервные копии

Случаи, для которых необходимо резервное копирование:

- **Аварийные ситуации, приводящие к потере данных на диске.** К авариям можно отнести такие события, как выход из строя жесткого диска, пожары, потопы и другие природные катаклизмы. На практике такое случается нечасто, поэтому пользователи редко беспокоятся о создании резервных копий. Как правило, эти пользователи по той же причине не страхуют свое имущество от пожаров и прочих стихийных бедствий.
- **Случайное удаление или программная порча файлов.** Эта проблема возникает столь часто, что в ОС Windows при обычном удалении файла он на самом деле не удаляется, а помещается в специальный каталог, называемый мусорной корзиной, откуда его при необходимости несложно восстановить.

Основные принципы создания резервных копий:

- Создавать несколько копий - ежедневные, еженедельные, ежемесячные, ежеквартальные.
- Как правило, необходимо сохранять не весь диск, а только выборочные каталоги.
- Применять **инкрементные резервные копии** - сохраняются только измененные файлы.
- Сжимать резервные копии для экономии места.
- Фиксировать систему при создании резервной копии, чтобы вовремя резервирования система не менялась.
- Хранить резервные копии в защищенном месте, не доступном для посторонних.

Существует две стратегии:

Физическая архивация - поблочное копирование диска (копируются блоки, а не файлы).

Недостатки:

- копирование пустых блоков;
- проблемы с дефектными блоками;
- не возможно применять инкрементное копирование;
- не возможно копировать отдельные каталоги и файлы.

Преимущества:

- высокая скорость копирования;
- простота реализации.

Логическая архивация сканирует один или несколько указанных каталогов со всеми их подкаталогами и копирует на ленту все содержащиеся в них файлы и каталоги, изменившиеся с указанной даты (например, с момента последней архивации). Таким образом, при логической архивации на магнитную ленту записываются последовательности детально идентифицированных каталогов и файлов, что позволяет восстановить отдельный файл или каталог.

Поскольку логическая архивация применяется на практике чаще физической, познакомимся более детально с алгоритмом архивации на примере, показанном на Рис. 18. Этот алгоритм используется в большинстве систем UNIX. На рисунке показано дерево файлов с каталогами (квадраты) и файлами (кружки). Затенённые элементы были изменены с момента последней архивации и поэтому следует создать их

резервную копию. Светлые элементы не нуждаются в архивации. Каждый каталог и файл помечены номером своего i-узла.

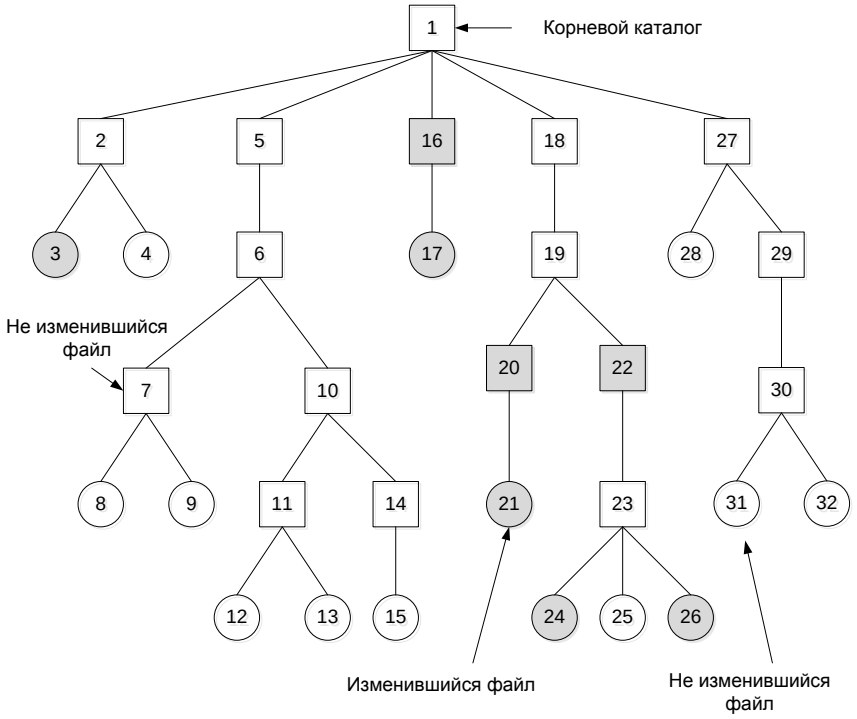

Рис. 18. Архивируемая ФС

Этот алгоритм также создает резервные копии всех каталогов (даже не модифицированных), расположенных на пути к модифицированному файлу или каталогу. Делается это по двум причинам:

Во-первых, чтобы все сохраненные файлы и каталоги можно было восстановить на другом компьютере. Благодаря этому программа архивации может использоваться для переноса всей ФС с одного компьютера на другой.

Во-вторых, сохранение резервных копий всех каталогов, расположенных на пути к модифицированному файлу, позволяет восстановить один отдельный файл (например, удаленный по ошибке). Представьте, что полная архивация системы произведена в воскресенье, а в понедельник выполнена инкрементная архивация. Во вторник удаляется каталог /usr/jhs/proj/nr3 со всеми содержащимися в нем каталогами и файлами. В среду утром пользователь хочет восстановить файл /usr/jhs/proj/nr3/plans/summary. Однако нельзя просто восстановить файл summary, так как каталоги /usr/jhs/proj/nr3/plans и /usr/jhs/proj/nr3 удалены и файл summary некуда поместить. Сначала нужно восстановить каталоги nr3 и plans. Чтобы ОС смогла корректно установить такие параметры этих каталогов, как идентификатор владельца, режимы доступа, время создания, время изменения и т.д., эти каталоги должны присутствовать на архивной ленте, несмотря на то, что сами каталоги не модифицировались с момента последней архивации.

Алгоритм архивации создает битовый массив, индексированный по номеру i-узла, в котором на каждый i-узел отводится несколько битов. Работа алгоритма протекает в четыре этапа:

Первый этап начинается в начальном каталоге (например, в корневом), в котором исследуются все элементы. Для каждого модифицированного файла его i-узел помечается в битовом массиве. Каждый каталог также помечается (независимо от того, был он изменен или нет), после чего он открывается и рекурсивно исследуется все его содержимое.

К концу этапа все модифицированные файлы и все каталоги помечаются в битовом массиве, как показано (затенением) на Рис. 19(а).

На втором этапе алгоритм снова рекурсивно проходит весь каталог, снимая отметку со всех каталогов, в которых нет модифицированных файлов или каталогов. В результате этих действий битовый массив принимает вид, показанный на Рис. 19(б). Обратите внимание, что каталоги 10, 11, 14, 27, 29 и 30 теперь уже не помечены, так

как в них и в их подкаталогах не содержится ничего модифицированного. Для этих каталогов не будет создаваться резервная копия. Напротив, каталоги 5 и 6 будут заархивированы, несмотря на то, что сами они не были модифицированы. Однако они потребуются, чтобы сегодняшние изменения можно было восстановить на новой машине. Для большей эффективности фазы 1 и 2 можно объединить, выполнив их за один проход.

Рис. 19. Битовые массивы из алгоритма логической архивации

К этому моменту алгоритму известно, какие каталоги и файлы должны архивироваться, помеченные на Рис. 19(б).

Этап 3 состоит из сканирования i-узлов по порядку номеров и создания резервных копий всех каталогов, помеченных для архивации. Они помечены на Рис. 19(в). Перед каждым каталогом записываются его атрибуты (владелец, времена и т.д.), необходимые для восстановления.

Наконец, на **четвертом этапе файлы**, помеченные на Рис. 19(г), также архивируются со своими атрибутами, записываемыми перед ними. На этом архивация завершается.

Восстановление ФС происходит достаточно просто. Сначала на диске создается пустая ФС. Затем восстанавливаются данные последней полной архивации. Сначала, восстанавливаются каталоги, создавая скелет ФС, после чего восстанавливаются все файлы. Затем весь процесс повторяется со всеми инкрементными архивациями по очереди.

Хотя алгоритм логической архивации несложен, в этом деле имеется несколько хитростей:

Во-первых, поскольку список свободных блоков не является файлом, он не архивируется, следовательно, его приходится восстанавливать с нуля, после того как были восстановлены все архивы. Это всегда является выполнимой задачей, так как множество свободных блоков представляет собой всего лишь совокупность всех блоков, содержащихся во всех файлах.

Второй проблемой являются связи файлов. Если у файла имеются связи в двух или более каталогах, важно, чтобы этот файл был восстановлен только один раз, и чтобы во всех каталогах, в которых были ссылки на этот файл, появились именно ссылки.

В ОС Windows предусмотрены следующие функции по работе с архивами в формате ZIP.

Функция	Описание
GetExpandedName	Получает оригинальное имя сжатого файла.
LZClose	Закрывает файл открытый с помощью функции LZOpenFile, сжатый файл.

LZCopy	Копирует исходный файл в указанный.
LZInit	Выделяет память для внутренних структур данных, необходимых для распаковки файлов.
LZOpenFile	Создает, открывает, переоткрывает и удаляет указанный файл.
LZRead	Читает указанное число байт из файла и копирует их в заданный буфер.
LZSeek	Перемещает файловый указатель на заданное количество байт от текущей позиции.

Производительность файловой системы

Доступ к диску производится значительно медленнее, чем к ОЗУ. Чтение слова из памяти может занять около 10 нс. Чтение с HDD может выполняться со скоростью 10 Мбайт/с, что в сорок раз медленнее, но к этому следует добавить 5-10 мс на поиск нужного цилиндра и задержку вращения HDD. Если требуется прочитать или записать всего одно слово, то ОЗУ оказывается примерно в миллион раз быстрее жесткого диска. Поэтому во многих ФС применяются различные методы оптимизации, увеличивающие производительность.

Кэширование

Для минимизации количества обращений к HDD применяется блочный кэш или буферный кэш. (Термин «кэш» происходит от французского слова `cacher`, что значит «скрывать»). В данном контексте кэшем называется набор блоков, логически принадлежащих диску, но хранящихся в оперативной памяти по соображениям производительности.

Существуют различные алгоритмы управления кэшем. Обычная практика заключается в перехвате всех запросов чтения к диску и проверке наличия требующихся блоков в кэше. Если блок присутствует в кэше, то запрос чтения блока может быть удовлетворен без обращения к диску. В противном случае блок сначала считывается с диска в кэш, а оттуда копируется по нужному адресу памяти. Последующие обращения к тому же блоку могут удовлетворяться из кэша.

Работа кэша показана на Рис. 20. Поскольку в кэше хранится большое количество (часто тысячи) блоков, требуется некий быстрый способ определения наличия или отсутствия блока в кэше. Обычно для этого используется хэширование номера устройства и дискового адреса (номера блока) и поиск результата в хэш-таблице. Все блоки с одинаковыми хэш-кодами сцепляются вместе в связный список.

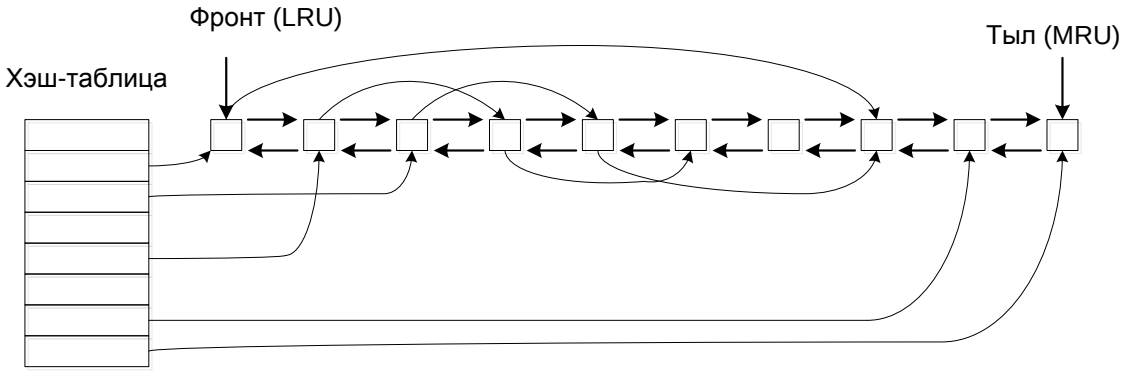

Рис. 20. Структура данных блочного кеша

Когда требуется загрузить блок в заполненный до предела кэш, какой-либо другой блок должен быть удален из кэша (и записан на диск, если он был модифицирован в кэше). Эта ситуация очень похожа на страничную организацию памяти, и к ней

применимы все обычные алгоритмы замены, описанные в лекции 5, такие как FIFO (First in First Out—первым прибыл—первым обслужен), «вторая попытка» и LRU (Least Recently Used — с наиболее давним использованием). Одно приятное отличие кэширования от страничной организации памяти состоит в том, что обращения к кэшу производятся относительно нечасто, что позволяет хранить все блоки в точном LRU-порядке со связными списками.

На Рис. 20 видим, что в дополнение к цепям, начинающимся в хэш-таблице, используется также и двунаправленный список, в котором содержатся номера всех блоков в порядке их использования. При этом самый старый блок помещается в начало списка, а самый новый блок — в его конец. При обращении к блоку блок может перемещаться со своей текущей позиции в конец двунаправленного списка. Таким образом, может поддерживаться точный LRU-порядок.

К сожалению, здесь есть одна загвоздка. Теперь, когда можно реализовать точное выполнение алгоритма LRU, оказывается, что алгоритм LRU является нежелательным. Вызвано это тем, что буквальное применение алгоритма LRU снижает надёжность ФС и угрожает ее непротиворечивости. Если в кэш считывается и модифицируется критический блок, например блок i-узла, но не записывается тут же на диск, то компьютерный сбой может привести к тому, что ФС окажется в противоречивом состоянии. Если блок i-узла поместить в конец цепочки LRU, то может пройти довольно много времени, прежде чем этот блок попадёт в ее начало и будет записан на диск.

Более того, к некоторым блокам, таким как блоки i-узлов, программы редко обращаются дважды в течение короткого интервала времени. Исходя из этих соображений, приходим к модифицированной схеме LRU, принимая во внимание два следующих фактора:

1. Насколько велика вероятность того, что данный блок скоро снова понадобится?

2. Важен ли данный блок для непротиворечивости ФС?

Для ответа на каждый из этих вопросов блоки можно разделить на такие категории:

- как блоки i-узлов;
- косвенные блоки;
- блоки каталогов;
- блоки полные данных;
- блоки частично заполненные данными.

Блоки, которые, вероятно, не потребуются снова в ближайшее время, помещаются в начало списка LRU, чтобы занимаемые ими буферы могли вскоре освободиться. Блоки, вероятность повторного использования которых в ближайшее время высока (например, записываемые блоки, частично заполненные данными), помещаются в конец списка LRU, что позволяет им оставаться в кэше более долгое время.

Второй вопрос не связан с первым. Если блок представляет важность для непротиворечивости ФС (обычно это все блоки, кроме блоков данных) и такой блок модифицируется, то его следует немедленно сохранить на диске независимо от его положения в списке LRU. Быстро записывая критические блоки, значительно снижаем вероятность того, что сбой компьютера повредит ФС. Пользователь вряд ли будет рад потере одного из своих файлов из-за сбоя компьютера. Еще более он огорчится, если при этом испорченной окажется вся ФС.

Эта ситуация случается не слишком часто, если только с очень невезучими пользователями. Для решения данной проблемы обычно применяется два метода:

В системе UNIX есть системный вызов `sync`, принуждающий сохранение всех модифицированных блоков кэша на диске. При загрузке ОС запускается фоновая задача, обычно называющаяся `update`, вся работа которой заключается в периодическом (обычно через каждые 30 с) обращении к системному вызову `sync`. В результате при любом сбое будет потеряно не более 30 с работы.

В системе MS-DOS используется другой подход, состоящий в том, что каждый модифицированный блок записывается на диск сразу же. Кэш, в котором все модифицированные блоки немедленно записываются на диск, называются **сквозным кэшем или кэшем со сквозной записью**. При использовании сквозного кэша количество обращений ввода-вывода к диску больше, чем при применении обычного кэша. Чтобы лучше понять разницу в этих двух подходах, представьте себе программу, записывающую блок размером в 1 Кбайт по одному символу. Система UNIX будет собирать все символы в кэше и записывать этот блок на диск каждые 30 с или когда

блок будет удален из кэша. Система MS-DOS будет обращаться к диску при каждом записываемом символе. Конечно, большинством программ применяется внутренняя буферизация, поэтому обычно они обращаются к системному вызову write не с одним символом, а с целыми строками или большими единицами данных.

Опережающее чтение блока

Второй метод увеличения производительности ФС состоит в попытке получить блоки диска в кэш прежде, чем они потребуются. В частности, многие файлы считываются последовательно. Когда ФС получает запрос на чтение блока k файла, она выполняет его, но после этого сразу проверяет, есть ли в кэше блок $k+1$. Если этого блока в кэше нет, ФС читает его в надежде, что к тому моменту, когда он понадобится, этот блок уже будет считан в кэш. В крайнем случае, он уже будет на пути туда.

Такая стратегия работает только для тех файлов, которые считываются последовательно. Если обращения к блокам файла производятся в случайном порядке, опережающее чтение не помогает. Чтобы определить, следует ли использовать опережающее чтение блоков, ФС может вести учет доступа к блокам каждого открытого файла. Например, для каждого открытого файла один бит может означать «режим последовательного доступа» или «режим произвольного доступа». Вначале каждому открываемому файлу в соответствии с принципом презумпции невиновности назначается режим последовательного доступа. Однако при перемещении указателя в файле этот бит сбрасывается. Если к этому файлу опять будут обращаться с запросами последовательного чтения, бит будет установлен снова.

Снижение времени перемещения блока головок

Кэширование и опережающее чтение являются не единственными способами увеличения производительности ФС. Другой важный метод состоит в уменьшении затрат времени на перемещение блока головок. Достигается это помещением блоков, к которым высока вероятность доступа в течение короткого интервала времени, близко друг к другу, желательно на одном цилиндре. Когда записывается выходной файл, ФС должна зарезервировать место для чтения таких блоков за одну операцию. Если свободные блоки учитываются в битовом массиве, а весь битовый массив помещается в ОЗУ, то довольно легко выбрать свободный блок как можно ближе к предыдущему блоку. В случае, когда свободные блоки хранятся в списке, часть которого в ОЗУ, а часть на диске, сделать это значительно труднее.

Однако даже при использовании списка свободных блоков может быть выполнена определенная **кластеризация блоков**, которая заключается в том, чтобы учитывать место на диске не в блоках, а в группах последовательных блоков - **кластеров**. Если сектор состоит из 512 байт, система может использовать блоки размером в 1 Кбайт (2 сектора), но выделять пространство на диске в единицах по 2 блока (4 сектора). Это не то же самое, что использование 2-килобайтных дисковых блоков, так как кэш по-прежнему будет использовать килобайтные блоки и дисковые операции чтения и записи будут по-прежнему работать с килобайтными блоками. Однако при последовательном чтении файла количество операций поиска цилиндра уменьшится вдвое, что значительно увеличит производительность. Вариация этой же темы состоит в попытке системы учесть позицию блока в цилиндре.

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 21. I-узлы, размещенные в начале диска (а); диск, разделенный на группы цилиндров, каждая со своими собственными блоками и i-узлами (б)

Еще один фактор, снижающий производительность ФС, связан с тем, что при использовании i-узлов или чего-либо эквивалентного им, особенно при чтении коротких файлов, **требуется два обращения к диску вместо одного**: одно для i-узла и одно для блока данных. Обычное размещение i-узлов на диске показано на Рис. 21(а). Здесь все i-узлы располагаются в начале диска, так что среднее расстояние между i-узлом и его блоками будет составлять около половины количества цилиндров, то есть при доступе практически к каждому файлу потребуются значительные перемещения блока головок.

Один из способов увеличения производительности состоит в помещении i-узлов в середину диска, уменьшая, таким образом, среднее расстояние перемещения блоков

головок в два раза. Другая идея, показанная на Рис. 21(б), заключается в разбиении диска на группы цилиндров, каждая со своими i-узлами, блоками и списком свободных блоков. Когда создается новый файл, может быть выбран любой i-узел, но предпринимается попытка найти блок в той же группе цилиндров, что и i-узел. Если эта попытка заканчивается неудачей, используется блок в соседней группе цилиндров.

Файловые системы с журнальной структурой LFS

Log-structured File System. Идея в том, что по мере увеличения скорости процессоров и объёма ОЗУ кэширование становится все выгоднее. Становится возможным удовлетворить существенную часть всех дисковых запросов непосредственно из кэша ФС без обращения к диску. Следовательно, большинство обращений к диску будут обращениями на запись, а не на чтение. Поэтому алгоритм опережающего чтения становится малоэффективным. Далее, в большинстве ФС запись выполняется небольшими блоками данных, что также неэффективно, поскольку помимо собственно записи выполняется еще и относительно длинный поиск цилиндра. Например, в UNIX для записи в файл требуется: выполнить операции записи в i-узел каталога, блок каталога, i-узел файла и блок самого файла. Система LFS пытается учесть эти особенности. Идея в том, что диск используется как журнал. Периодически, когда возникает необходимость, все буферизированные в памяти блоки, которые должны быть записаны, собираются в единый сегмент, и он записывается на диск единым блоком в конец журнала. Этот сегмент может содержать i-узлы, блоки каталогов, блоки данных, перемешанные друг с другом. В начале каждого сегмента создаётся оглавление сегмента. Если средний размер сегмента довести до 1 Мб, пропускная способность диска может быть использована практически на 100%. Для быстрого поиска i-узлов, поскольку теперь они могут располагаться в произвольной области, создаётся массив, j элемент которого хранит указатель на j i-узел. Этот массив хранится на диске и в кэше. Поскольку постепенно весь объем диска будет использован журналом, то в такой ФС определён чистящий поток, постоянно сканирующий журнал, чтобы делать его более компактным. Поток считывает содержимое сегмента журнала, определяя какие i-узлы и файлы в нем находятся. Далее проверяется текущий массив i-узлов, чтобы определить, являются ли i-узлы текущими и используются ли все еще блоки файлов. Если нет, то такая информация отбрасывается, а все еще используемые узлы и блоки считываются в память, чтобы быть записанными в следующий сегмент. Исходный сегмент отмечается как свободный, и может быть использован для новых данных. Чистильщик двигается по журналу от сегмента к сегменту, а диск фактически представляет большой кольцевой буфер, в котором пишущий поток добавляет сегменты с одного конца, а чистильщик удаляет их с другого.

Домашнее задание

Используя материалы [1, 4], следует изучить строение ФС:

- CD-ROM.
- MS-DOS.
- Unix v7.
- XFS.
- EXT4.

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. –360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. –680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

5. Microsoft Development Network. URL: http://msdn.com