MODERN OPERATING SYSTEMS

Fifth Edition

Andrew S. Tanenbaum
Herbert Bos

MODERN OPERATING SYSTEMS

FIFTH EDITION

ANDREW S. TANENBAUM
HERBERT BOS

Vrije Universiteit
Amsterdam, The Netherlands

4

FILE SYSTEMS

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information in physical RAM. For many applications, the amount of memory is far too small and some even need many terabytes of storage.

A second problem with keeping information in RAM is that when the process terminates, the information is lost. For many applications (e.g., for databases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process or power goes off during an electrical storm.

A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. If we have an online telephone directory stored inside the address space of a single process, only that process can access it, unless it is shared explicitly. The way to solve this problem is to make the information itself independent of any one process.

Thus, we have three essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information at once.

Magnetic disks have been used for years for this long-term storage. While such disks are still used extensively, solid-state drives (SSDs) have also become hugely

popular, complementing or replacing their magnetic counterparts. Compared to hard disks, they do not have any moving parts that may break, and offer fast random access. Tapes and optical disks are no longer as popular as they used to be and have much lower performance. Nowadays, if they are used at all, it is typically for backups. We will study magnetic hard disks and SSDs more in Chap. 5. For the moment, you can think of both as *disk-like*, even though strictly speaking an SSD is not a disk at all. Here, "disk-like" means that it supports an interface that appears to be a linear sequence of fixed-size blocks and supporting two operations:

1. Read block *k*
2. Write block *k*

In reality there are more, but with these two operations one could, in principle, solve the long-term storage problem.

However, these are very inconvenient operations, especially on large systems used by many applications and possibly multiple users (e.g., on a server). Just a few of the questions that quickly arise are:

1. How do you find information?
2. How do you keep one user from reading another user's data?
3. How do you know which blocks are free?

and there are many more.

Just as we saw how the operating system abstracted away the concept of the processor to create the abstraction of a process and how it abstracted away the concept of physical memory to offer processes (virtual) address spaces, we can solve this problem with a new abstraction: the file. Together, the abstractions of processes (and threads), address spaces, and files are the most important concepts relating to operating systems. If you really understand these three concepts from beginning to end, you are well on your way to becoming an operating systems expert.

Files are logical units of information created by processes. A disk will usually contain thousands or even millions of them, each one independent of the others. In fact, if you think of each file as a kind of address space, you are not that far off, except that they are used to model the disk instead of modeling the RAM.

Processes can read existing files and create new ones if need be. Information stored in files must be **persistent**, that is, not be affected by process creation and termination. A file should disappear only when its owner explicitly removes it. Although operations for reading and writing files are the most common ones, there exist many others, some of which we will examine below.

Files are managed by the operating system. How they are structured, named, accessed, used, protected, implemented, and managed are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the **file system** and is the subject of this chapter.

From the user's standpoint, the most important aspect of a file system is how it appears, in other words, what constitutes a file, how files are named and protected, what operations are allowed on files, and so on. The details of whether linked lists or bitmaps are used to keep track of free storage and how many sectors there are in a logical disk block are of no interest, although they are of great importance to the designers of the file system. For this reason, we have structured the chapter as several sections. The first two are concerned with the user interface to files and directories, respectively. Then comes a detailed discussion of how the file system is implemented and managed. Finally, we give some examples of real file systems.

4.1 FILES

In the following pages, we will look at files from the user's point of view, that is, how they are used and what properties they have.

4.1.1 File Naming

A file is an abstraction mechanism. It provides a way to store information on the disk and read it back later. This must be done in a way that shields the user from the details of how and where the information is stored, and how the disks actually work.

Probably the most important characteristic of any abstraction mechanism is the way the objects being managed are named, so we will start our examination of file systems with the subject of file naming. When a process creates a file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.

The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of letters as legal file names. Thus *andrea*, *bruce*, and *cathy* are possible file names. Frequently digits and special characters are also permitted, so names like *2*, *urgent!*, and *Fig.2-14* are often valid as well. Some older file systems, such as the one that was used in MS-DOS in a century long ago, limit file names to eight letters maximum, but most modern systems support file names of up to 255 characters or more.

Some file systems distinguish between upper- and lowercase letters, whereas others do not. UNIX falls in the first category; the old MS-DOS falls in the second. Thus, a UNIX system can have all of the following as three distinct files: *maria*, *Maria*, and *MARIA*. In MS-DOS, all these names refer to the same file.

An aside on file systems is probably in order here. Older versions of Windows (such as Windows 95 and Windows 98) used the MS-DOS file system, called **FAT-16**, and thus inherited many of its properties, such as how file names are constructed. Admittedly, Windows 98 introduced some extensions to FAT-16, leading to **FAT-32**, but these two are quite similar. Modern versions of Windows all still support the FAT file systems, even though they also have a much more advanced

native file system (NTFS) that has different properties (such as file names in Unicode). We will discuss NTFS in Chap. 11. There is also a second file system for Windows, known as **ReFS** (**Resilient File System**), but that one is targeted at the server version of Windows. In this chapter, when we refer to the MS-DOS or FAT file systems, we mean FAT-16 and FAT-32 as used on Windows unless specified otherwise. We will discuss the FAT file systems later in this chapter and NTFS in Chap. 12, where we will examine Windows 10 in detail. Incidentally, there is also an even newer FAT-like file system, known as **exFAT** file system, a Microsoft extension to FAT-32 that is optimized for flash drives and large file systems.

Many operating systems support two-part file names, with the two parts separated by a period, as in *prog.c*. The part following the period is called the **file extension** and usually indicates something about the file. In MS-DOS, for example, file names were 1–8 characters, plus an optional extension of 1–3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in *homepage.html.zip*, where *.html* indicates a Web page in HTML and *.zip* indicates that the file (*homepage.html*) has been compressed using the *zip* program. Some of the more common file extensions and their meanings are shown in Fig. 4-1.

Extension	Meaning
.bak	Backup file
.c	C source program
.gif	Compuserve Graphical Interchange Format image
.html	World Wide Web HyperText Markup Language document
.jpg	Still picture encoded with the JPEG standard
.mp3	Music encoded in MPEG layer 3 audio format
.mpg	Movie encoded with the MPEG standard
.o	Object file (compiler output, not yet linked)
.pdf	Portable Document Format file
.ps	PostScript file
.tex	Input for the TEX formatting program
.txt	General text file
.zip	Compressed archive

Figure 4-1. Some typical file extensions.

In some systems (e.g., all flavors of UNIX), file extensions are just conventions and are not enforced by the operating system. A file named *file.txt* might be some kind of text file, but that name is more to remind the owner than to convey any actual information to the computer. On the other hand, a C compiler may actually insist that files it is to compile end in *.c*, and it may refuse to compile them if they do not. However, the operating system does not care.

Conventions like this are especially useful when the same program can handle several different kinds of files. The C compiler, for example, can be given a list of several files to compile and link together, some of them C files and some of them assembly-language files. The extension then becomes essential for the compiler to tell which are C files, which are assembly files, and which are other files.

In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system and specify for each one which program "owns" that extension. When a user double clicks on a file name, the program assigned to its file extension is launched with the file as parameter. For example, double clicking on *file.docx* starts Microsoft *Word* with *file.docx* as the initial file to edit. In contrast, *Photoshop* will not open file ending in *.docx*, no matter how often or hard you click on the file name, because it knows that *.docx* files are not image files.

4.1.2 File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 4-2. The file in Fig. 4-2(a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.

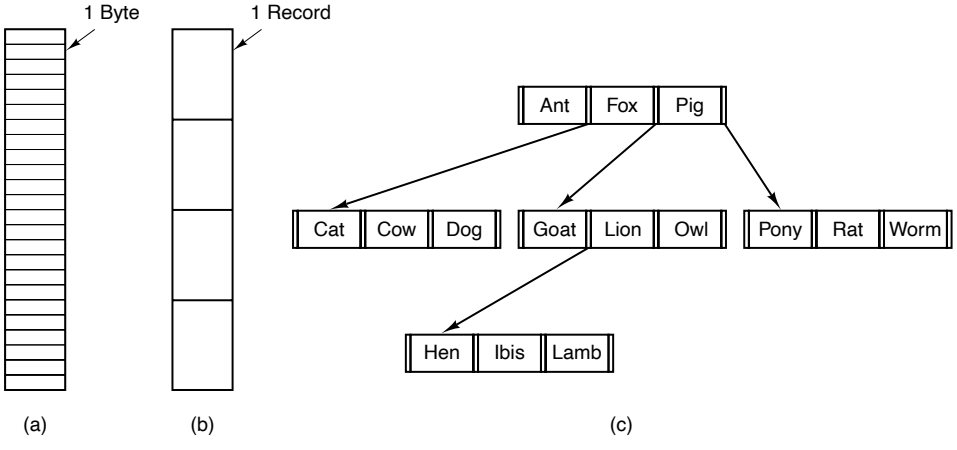

Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Having the operating system regard files as nothing more than byte sequences provides the maximum amount of flexibility. User programs can put anything they want in their files and name them any way that they find convenient. The operating system does not help, but it also does not get in the way. For users who want to do

unusual things, the latter can be very important. All versions of UNIX (including Linux and MacOS) as well as Windows use this file model. It is worth noting that in this chapter when we talk about UNIX the text usually applies MacOS (which was based on Berkeley UNIX) and Linux (which was carefully designed to be compatible with UNIX).

The first step up in structure is illustrated in Fig. 4-2(b). In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record. As a historical note, in decades gone by, when the 80-column punched card was pretty much the only input medium available, many (mainframe) operating systems based their file systems on files consisting of 80-character records, in effect, card images. These systems also supported files of 132-character records, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of 80 characters and wrote it in units of 132 characters, although the final 52 could be spaces, of course. No current general-purpose system uses this model as its primary file system any more, but back in the days of 80-column punched cards and 132-character line printer paper, this was a common model on mainframe computers.

The third kind of file structure is shown in Fig. 4-2(c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a **key** field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

The basic operation here is not to get the "next" record, although that is also possible, but to get the record with a specific key. For the zoo file of Fig. 4-2(c), one could ask the system to get the record whose key is *pony*, for example, without worrying about its exact position in the file. Furthermore, new records can be added to the file, with the operating system, and not the user, deciding where to place them. This type of file is clearly quite different from the unstructured byte streams used in UNIX and Windows and is used on some large mainframe computers for commercial data processing.

4.1.3 File Types

Many operating systems support several types of files. UNIX (again, including MacOS and Linux) and Windows, for example, have regular files and directories. UNIX also has character and block special files. **Regular files** are the ones that contain user information. All the files of Fig. 4-2 are regular files since these are the files most users deal with. **Directories** are system files for maintaining the structure of the file system. We will study directories below. **Character special files** are related to input/output and used to model serial I/O devices, such as terminals, printers, and networks. **Block special files** are used to model disks. In this chapter, we will be primarily interested in regular files.

Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text. In some systems, each line is terminated by a carriage return character. In others, the line feed character is used. Some systems (e.g., Windows) use both. Lines need not all be of the same length.

The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines. (The interprocess plumbing is not any easier, but interpreting the information certainly is if a standard convention, such as ASCII, is used for expressing it.)

Other files are binary, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of random junk. Usually, they have some internal structure known to programs that use them.

For example, in Fig. 4-3(a) we see a simple executable binary file taken from an early version of UNIX. Although technically the file is just a sequence of bytes, the operating system will execute a file only if it has the proper format. It has five sections: header, text, data, relocation bits, and symbol table. The header starts with a **magic number**, identifying the file as an executable file (to prevent the accidental execution of a file not in this format). Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits. After the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is for debugging.

Our second example of a binary file is an archive, also from UNIX. It consists of a collection of library procedures (modules) compiled but not linked. Each one is prefaced by a header telling its name, creation date, owner, protection code, and size. Just as with the executable file, the module headers are full of binary numbers. Copying them to the printer would produce complete gibberish.

Every operating system must recognize at least one file type: its own executable file; some recognize more. The old TOPS-20 system (for the DECsystem 20) went so far as to examine the creation time of any file to be executed. Then it located the source file and saw whether the source had been modified since the binary was made. If it had been, it automatically recompiled the source. In UNIX terms, the *make* program had been built into the shell. The file extensions were mandatory, so it could tell which binary program was derived from which source.

Having strongly typed files like this causes problems whenever the user does anything that the system designers did not expect. Consider, as an example, a system in which program output files have extension *.dat* (data files). If a user writes a program formatter that reads a *.c* file (C program), transforms it (e.g., by converting it to a standard indentation layout), and then writes the transformed file as output, the output file will be of type *.dat*. If the user tries to offer this to the C compiler to compile it, the system will refuse because it has the wrong extension. Attempts to copy *file.dat* to *file.c* will be rejected by the system as invalid (to protect the user against mistakes).

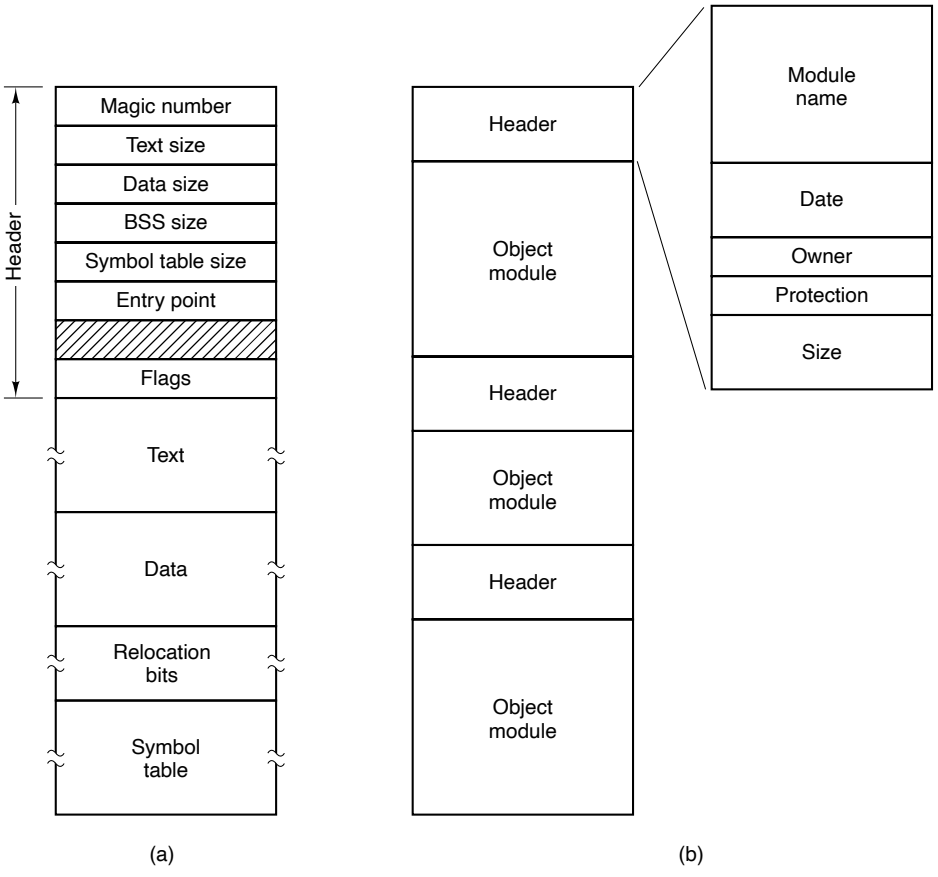

Figure 4-3. (a) An executable file. (b) An archive.

While this kind of "user friendliness" may help novices, it drives experienced users up the wall since they have to devote considerable effort to circumventing the operating system's idea of what is reasonable and what is not.

Most operating systems offer a slew of tools to examine files. For instance, on UNIX you can use the *file* utility to examine the type of files. It uses heuristics to determine that something is a text file, a directory, an executable, etc. Examples of its use can be found in Fig. 4-4.

4.1.4 File Access

Early operating systems provided only one kind of file access: **sequential access**. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of

Command	Result
file README.txt	UTF-8 Unicode text
file hjb.sh	POSIX shell script, ASCII text executable
file Makefile	makefile script, ASCII text
file /usr/bin/less	symbolic link to /bin/less
file /bin/	directory
file /bin/less	ELF 64-bit LSB shared object, x86-64 [...more information...]

Figure 4-4. Finding out file types.

order. Sequential files could be rewound, however, so they could be read as often as needed. Sequential files were convenient when the storage medium was magnetic tape rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key rather than by position. Files whose bytes or records can be read in any order are called **random-access files**. They are required by many applications.

Random access files are essential for many applications, for example, database systems. If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first.

Two methods can be used for specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. After a seek, the file can be read sequentially from the now-current position. The latter method is used in UNIX and Windows.

4.1.5 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was last modified and the file's size. We will call these extra items the file's **attributes**. Some people call them **metadata**. The list of attributes varies considerably from system to system. The table of Fig. 4-5 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.

The first four attributes relate to the file's protection and tell who may access it and who may not. All kinds of schemes are possible, some of which we will study later. In some systems the user must present a password to access a file, in which case the password must be one of the attributes.

The flags are bits or short fields that control or enable some specific property. Hidden files, for example, do not appear in listings of all the files. The archive flag

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 4-5. Some possible file attributes.

is a bit that keeps track of whether the file has been backed up recently. The backup program clears it, and the operating system sets it whenever a file is changed. In this way, the backup program can tell which files need backing up. The temporary flag allows a file to be marked for automatic deletion when the process that created it terminates.

The record-length, key-position, and key-length fields are only present in files whose records can be looked up using a key. They provide the information required to find the keys.

The times keep track of when the file was created, most recently accessed, and most recently modified. These are useful for a variety of purposes. For example, a source file that has been modified after the creation of the corresponding object file needs to be recompiled. These fields provide the necessary information.

The current size tells how big the file is at present. Some old mainframe operating systems required the maximum size to be specified when the file was created, in order to let the operating system reserve the maximum amount of storage in advance. Personal-computer operating systems are thankfully clever enough to do without this feature nowadays.

4.1.6 File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.

1. Create. The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
2. Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. Open. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.
5. Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.
6. Write. Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
7. Append. This call is a restricted form of write. It can add data only to the end of the file. Systems that provide a minimal set of system calls rarely have append, but some systems have this call.
8. Seek. For random-access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.
9. Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When *make* is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. Set attributes. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection-mode information is an obvious example. Most of the flags also fall in this category.

11. Rename. This call is not essential because a file that needs to be renamed can be copied and then the original file deleted. However, renaming a 50-GB movie by copying it and then deleting the original will take a long time.

4.1.7 An Example Program Using File-System Calls

In this section, we will examine a simple UNIX program that copies one file from its source file to a destination file. It is listed in Fig. 4-6. The program has minimal functionality and even worse error reporting, but it gives a reasonable idea of how some of the system calls related to files work.

The program, *copyfile*, can be called, for example, by the command line

```
copyfile abc xyz
```

to copy the file *abc* to *xyz*. If *xyz* already exists, it will be overwritten. Otherwise, it will be created. The program must be called with exactly two arguments, both legal file names. The first is the source; the second is the output file.

The four *#include* statements near the top of the program cause a large number of definitions and function prototypes to be included in the program. These are needed to make the program conformant to the relevant international standards, but will not concern us further. The next line is a function prototype for *main*, something required by ANSI C, but also not important for our purposes.

The first *#define* statement is a macro definition that defines the character string *BUF_SIZE* as a macro that expands into the number 4096. The program will read and write in chunks of 4096 bytes. It is considered good programming practice to give names to constants like this. The second *#define* statement determines who can access the output file.

The main program is called *main*, and it has two arguments, *argc* and *argv*. These are supplied by the operating system when the program is called. The first one tells how many strings were present on the command line that invoked the program, including the program name. It should be 3. The second one is an array of pointers to the arguments. In the example call given above, the elements of this array would contain pointers to the following values:

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

It is via this array that the program accesses its arguments.

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

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

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break;       /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

Figure 4-6. A simple program to copy a file.

SP 1v

Five variables are declared. The first two, *in_fd* and *out_fd*, will hold the **file descriptors**, small integers returned when a file is opened. The next two, *rd_count* and *wt_count*, are the byte counts returned by the read and write system calls, respectively. The last one, *buffer*, is the buffer used to hold the data read and supply the data to be written.

The first actual statement checks *argc* to see if it is 3. If not, it exits with status code 1. Any status code other than 0 means that an error has occurred. The status code is the only error reporting present in this program. A production version would normally print error messages as well.

Then we try to open the source file and create the destination file. If the source file is successfully opened, the system assigns a small integer to *in_fd*, to identify the file. Subsequent calls must include this integer so that the system knows which file it wants. Similarly, if the destination is successfully created, *out_fd* is given a value to identify it. The second argument to *creat* sets the protection mode. If either the open or the create fails, the corresponding file descriptor is set to −1, and the program exits with an error code.

Now comes the copy loop. It starts by trying to read in 4 KB of data to *buffer*. It does this by calling the library procedure *read*, which actually invokes the read system call. The first parameter identifies the file, the second gives the buffer, and the third tells how many bytes to read. The value assigned to *rd_count* gives the number of bytes actually read. Normally, this will be 4096, except if fewer bytes are remaining in the file. When the end of the file has been reached, it will be 0. If *rd_count* is ever zero or negative, the copying cannot continue, so the *break* statement is executed to terminate the (otherwise endless) loop.

The call to *write* outputs the buffer to the destination file. The first parameter identifies the file, the second gives the buffer, and the third tells how many bytes to write, analogous to *read*. Note that the byte count is the number of bytes actually read, not *BUF_SIZE*. This point is important because the last *read* will not return 4096 unless the file just happens to be a multiple of 4 KB.

When the entire file has been processed, the first call beyond the end of file will return 0 to *rd_count*, which will make it exit the loop. At this point, the two files are closed and the program exits with a status indicating normal termination.

Although the Windows system calls are different from those of UNIX, the general structure of a command-line Windows program to copy a file is moderately similar to that of Fig. 4-6. We will examine the Windows calls in Chap. 11.

4.2 DIRECTORIES

To keep track of files, file systems normally have **directories** or **folders**, which are themselves files. In this section, we will discuss directories, their organization, their properties, and the operations that can be performed on them.

4.2.1 Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory**, but since it is the only one, the name does not matter much. On the first personal computers, this system was

common, in part because there was only one user. Interestingly enough, the world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once. This decision was no doubt made to keep the software design simple.

An example of a system with one directory is given in Fig. 4-7. Here the directory contains four files. The advantages of this scheme are its simplicity and the ability to locate files quickly—there is only one place to look, after all. It is sometimes still used on simple embedded devices such as digital cameras and some portable music players.

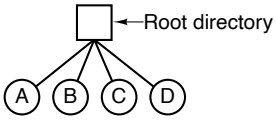

Figure 4-7. A single-level directory system containing four files.

Biologist Ernst Haeckel once said "ontogeny recapitulates phylogeny." It's not entirely accurate, but there is a grain of truth in it. Something analogous happens in the computer world. Some concept was in vogue on, say, mainframe computers, then discarded as they got more powerful, but picked up later on minicomputers. Then it was discarded there and later picked up on personal computers. Then it was discarded there and later picked up further down the food chain.

So we frequently see concepts (like having one directory for all files) no longer used on powerful computers, but now being used on simple embedded devices like digital cameras and portable music players. For this reason through this chapter (and, indeed, the entire book), we will often discuss ideas that were once popular on mainframes, minicomputers, or personal computers, but have since been discarded. Not only is this a good historical lesson, but often these ideas make perfect sense on yet lower-end devices. The chip on your credit card really does not need the full-blown hierarchical directory system we are about to explore. The simple file system used on the CDC 6600 supercomputer in the 1960s will do just fine, thank you. So when you read about some old concept here, do not think "how old-fashioned." Think: Would that work on an RFID (Radio Frequency IDentification) chip? that costs 5 cents and is used on a public-transit payment card? It just might.

4.2.2 Hierarchical Directory Systems

The single level is adequate for very simple dedicated applications (and was even used on the first personal computers), but for modern users with thousands of files, it would be impossible to find anything if all files were in a single directory. Consequently, a way is needed to group related files together. A professor, for example, might have a collection of files that together form a book that she is writing, a second collection containing student programs submitted for another course,

a third group containing the code of an advanced compiler-writing system she is building, a fourth group containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers she is writing, games, and so on.

What is needed is a hierarchy (i.e., a tree of directories). With this approach, there can be as many directories as are needed to group the files in natural ways. Furthermore, if multiple users share a common file server, as is the case on many company networks, each user can have a private root directory for his or her own hierarchy. This approach is shown in Fig. 4-8. Here, the directories *A*, *B*, and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

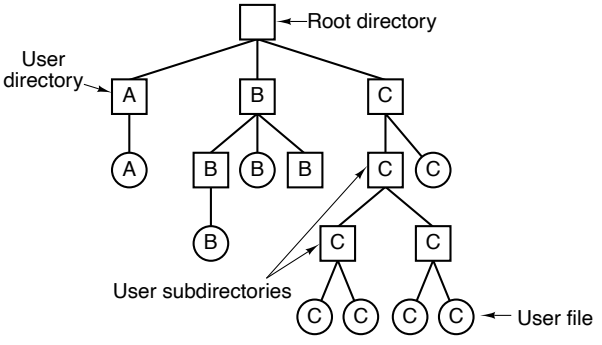

Figure 4-8. A hierarchical directory system.

The ability for users to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work. For this reason, all modern file systems are organized in this manner. It is worth noting that an hierarchical file system is one of many things that was pioneered by Multics in the 1960s.

4.2.3 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox* means that the root directory contains a subdirectory *usr*, which in turn contains a subdirectory *ast*, which contains the file *mailbox*. Absolute path names always start at the root directory and are unique. In UNIX the components of the path are separated by /. In Windows the separator is \. In MULTICS it was >. Thus, the same path name would be written as follows in these three systems:

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

No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

The other kind of name is the **relative path name**. This is used in conjunction with the concept of the **working directory** (also called the **current directory**). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is */usr/hjb*, then the file whose absolute path is */usr/hjb/mailbox* can be referenced simply as *mailbox*. In other words, the UNIX command

```
cp /usr/hjb/mailbox /usr/hjb/mailbox.bak
```

and the command

```
cp mailbox mailbox.bak
```

do exactly the same thing if the working directory is */usr/hjb*. The relative form is often more convenient, but it does the same thing as the absolute form.

Some programs need to access a specific file without regard to what the working directory is. In that case, they should always use absolute path names. For example, a spelling checker might need to read */usr/lib/dictionary* to do its work. It should use the full, absolute path name in this case because it does not know what the working directory will be when it is called. The absolute path name will always work, no matter what the working directory is.

Of course, if the spelling checker needs a large number of files from */usr/lib*, an alternative approach is for it to issue a system call to change its working directory to */usr/lib*, and then use just *dictionary* as the first parameter to open. By explicitly changing the working directory, it knows for sure where it is in the directory tree, so it can then use relative paths.

Each process has its own working directory. When it changes its working directory and later exits, no other processes are affected and no traces of the change are left behind. In this way, a process can change its working directory whenever it is convenient. On the other hand, if a *library procedure* changes the working directory and does not change back to where it was when it is finished, the rest of the program may not work since its assumption about where it is may now suddenly be invalid. For this reason, library procedures rarely change the working directory, and when they must, they always change it back again before returning.

Most operating systems that support a hierarchical directory system have two special entries in every directory, "." and "..", generally pronounced "dot" and "dotdot." Dot refers to the current directory; dotdot refers to its parent (except in the root directory, where it refers to itself). To see how these are used, consider the UNIX file tree of Fig. 4-9. A certain process has */usr/ast* as its working directory. It can use .. to go higher up the tree. For example, it can copy the file */usr/lib/dictionary* to its own directory using the command

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

The first path instructs the system to go upward (to the *usr* directory), then to go down to the directory *lib* to find the file *dictionary*.

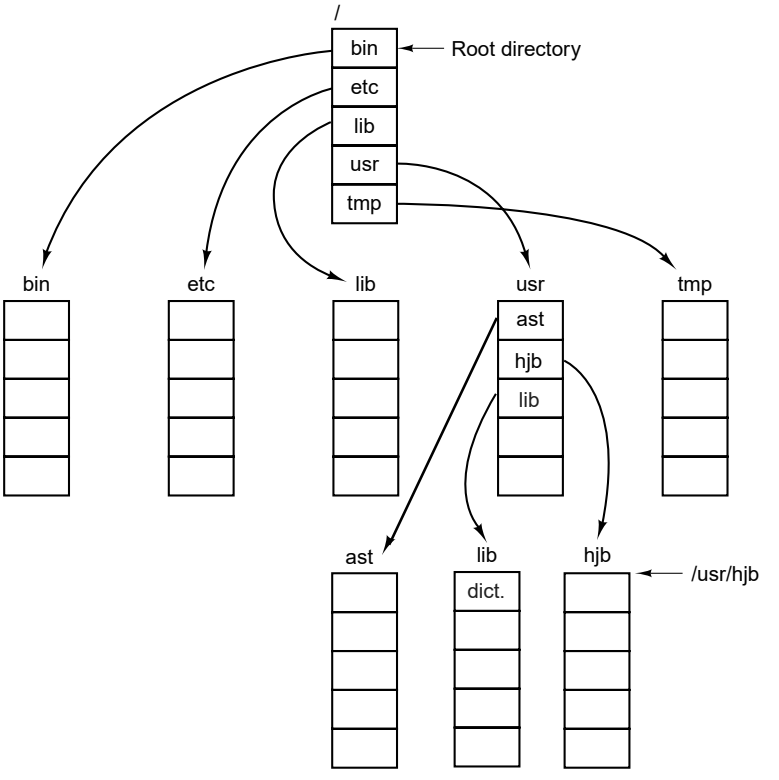

Figure 4-9. A UNIX directory tree.

The second argument (dot) names the current directory. When the *cp* command gets a directory name (including dot) as its last argument, it copies all the files to that directory. Of course, a more normal way to do the copy would be to use the full absolute path name of the source file:

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

Here the use of dot saves the user the trouble of typing *dictionary* a second time. Nevertheless, typing

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

also works fine, as does

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

All of these do exactly the same thing.

4.2.4 Directory Operations

The allowed system calls for managing directories exhibit more variation from system to system than system calls for files. To give an impression of what they are and how they work, we will give a sample (taken from UNIX).

1. Create. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the *mkdir* program.
2. Delete. A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot be deleted.
3. Opendir. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
4. Closedir. When a directory has been read, it should be closed to free up internal table space.
5. Readdir. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.
6. Rename. In many respects, directories are just like files and can be renamed the same way files can be.
7. Link. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a **hard link**.
8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory.

A variant on the idea of linking files is the **symbolic link** (sometimes called a **shortcut** or **alias**). Instead, of having two names point to the same internal data structure representing a file, a name can be created that points to a tiny file naming another file. When the first file is used, for example, opened, the file system follows the path and finds the name at the end. Then it starts the lookup process all over using the new name. Symbolic links have the advantage that they can cross disk boundaries and even name files on remote computers. Their implementation is somewhat less efficient than hard links though.

4.3 FILE-SYSTEM IMPLEMENTATION

Now it is time to turn from the user's view of the file system to the implementor's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementers are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. In the following sections, we will examine a number of these areas to see what the issues and trade-offs are.

4.3.1 File-System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. The layout depends on whether you have an old computer with a BIOS and a master boot record, or a modern UEFI-based system.

Old School: The Master Boot Record

On older systems, sector 0 of the disk is called the **MBR** (**Master Boot Record**) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, which is called the **boot block**, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the future.

Other than starting with a boot block, the layout of a disk partition varies a lot from file system to file system. Often the file system will contain some of the items shown in Fig. 4-10. The first one is the **superblock**. It contains all the key parameters about the file system and is read into memory when the computer is booted or

the file system is first touched. Typical information in the superblock includes a magic number to identify the file-system type, the number of blocks in the file system, and other key administrative information.

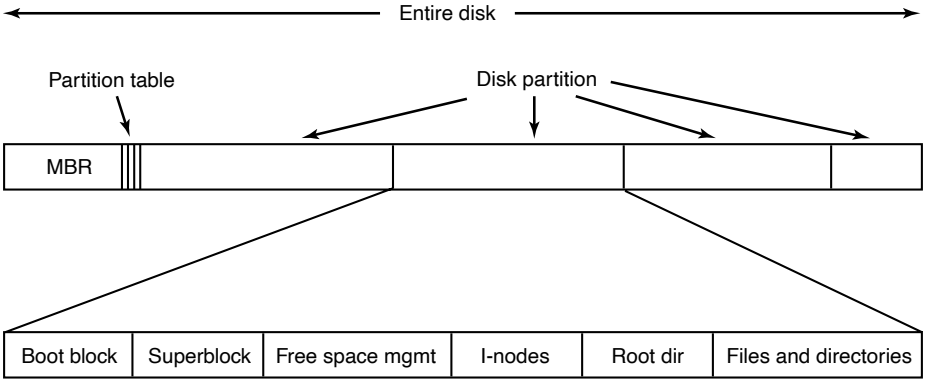

Figure 4-10. A possible file-system layout.

Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers. This might be followed by the i-nodes, an array of data structures, one per file, telling all about the file. After that might come the root directory, which contains the top of the file-system tree. Finally, the remainder of the disk contains all the other directories and files.

New School: Unified Extensible Firmware Interface

Unfortunately, booting in the way described above is slow, architecture-dependent, and limited to smaller disks (up to 2 TB) and Intel therefore proposed the **UEFI** (**Unified Extensible Firmware Interface**) as a replacement. is now the most popular way to boot personal computer systems. It fixes many of the problems of the old-style BIOS and MBR: fast booting, different architectures, and disk sizes up to 8 ZiB. It is also quite complex.

Rather than relying on a Master Boot Record residing in sector 0 of the boot device, UEFI looks for the location of the **partition table** in the second block of the device. It reserves the first block as a special marker for software that expects an MBR here. The marker essentially says: No MBR here!

The **GPT** (**GUID Partition Table**), meanwhile, contains information about the location of the various partitions on the disk. **GUID** stands for globally unique identifiers. As shown in Fig. 4-11, UEFI keeps a backup of the GPT in the last block. A GPT contains the start and end of each partition. Once the GPT is found, the firmware has enough functionality to read file systems of specific types. According to the UEFI standard the firmware should support at least FAT file system types. One such file system is placed in a special disk partition, known as the

EFI system partition (ESP). Rather than a single magic boot sector, the boot process can now use a proper file system containing programs, configuration files, and anything else that may be useful during boot. Moreover, UEFI expects the firmware to be able to execute programs in a specific format, called PE (Portable Executable). In other words, the firmware under UEFI looks like a small operating system itself with an understanding of disk partitions, file systems, executables, etc.

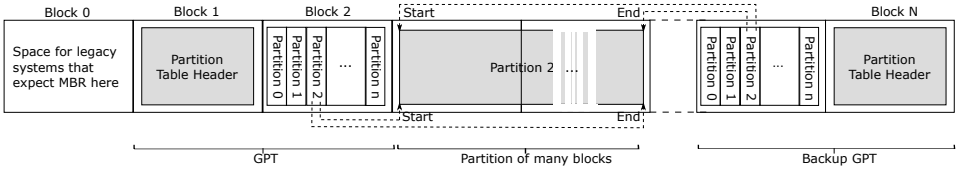

Figure 4-11. Layout for UEFI with partition table.

4.3.2 Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. In this section, we will examine a few of them.

Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.

We see an example of contiguous storage allocation in Fig. 4-12(a). Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file *A*, of length four blocks, was written to disk starting at the beginning (block 0). After that a six-block file, *B*, was written starting right after the end of file *A*.

Note that each file begins at the start of a new block, so that if file *A* was really 3½ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart. It has no actual significance in terms of storage.

Contiguous disk-space allocation has two significant advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.

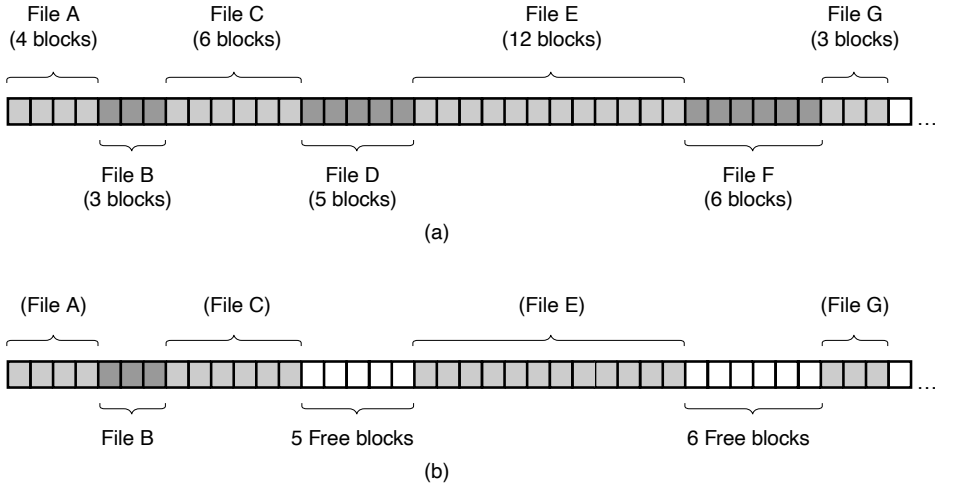

Figure 4-12. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

Second, the read performance is excellent even on a magnetic disk because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed, so data come in at the full bandwidth of the disk. Thus contiguous allocation is simple to implement and has high performance. We will talk about sequential versus random accesses on SSDs later.

Unfortunately, contiguous allocation also has a very serious drawback: over the course of time, the disk becomes fragmented. To see how this comes about, examine Fig. 4-12(b). Here two files, *D* and F, have been removed. When a file is removed, its blocks are naturally freed, leaving a run of free blocks on the disk. The disk is not compacted on the spot to squeeze out the hole, since that would involve copying all the blocks following the hole, potentially millions of blocks, which would take hours or even days with large disks. As a result, the disk ultimately consists of files and holes, as illustrated in the figure.

Initially, this fragmentation is not a problem, since each new file can be written at the end of disk, following the previous one. However, eventually the disk will fill up and it will become necessary to either compact the disk, which is prohibitively expensive, or to reuse the free space in the holes. Reusing the space requires maintaining a list of holes, which is doable. However, when a new file is to be created, it is necessary to know its final size in order to choose a hole that is big enough.

Imagine the consequences of such a design. The user starts a recording application in order to create a video. The first thing the program asks is how many bytes the final video will be. The question must be answered or the program will not continue. If the number given ultimately proves too small, the program has to

terminate prematurely because the disk hole is full and there is no place to put the rest of the file. If the user tries to avoid this problem by giving an unrealistically large number as the final size, say, 100 GB, the editor may be unable to find such a large hole and announce that the file cannot be created. Of course, the user would be free to start the program again, say 50 GB this time, and so on until a suitable hole was located. Still, this scheme is not likely to lead to happy users.

Linked-List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 4-13. The first part of each block is used as a pointer to the next one. The rest of the block is for data.

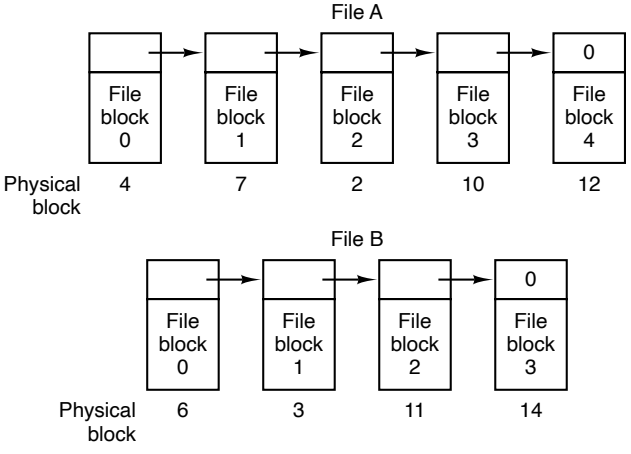

Figure 4-13. Storing a file as a linked list of disk blocks.

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. To get to block n, the operating system has to start at the beginning and read the $n - 1$ blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow.

Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied by a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

Linked-List Allocation Using a Table in Memory

Both disadvantages of the linked-list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Figure 4-14 shows what the table looks like for the example of Fig. 4-13. In both figures, we have two files. File *A* uses disk blocks 4, 7, 2, 10, and 12, in that order, and file *B* uses disk blocks 6, 3, 11, and 14, in that order. Using the table of Fig. 4-14, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., −1) that is not a valid block number. Such a table in main memory is called a **FAT** (**File Allocation Table**).

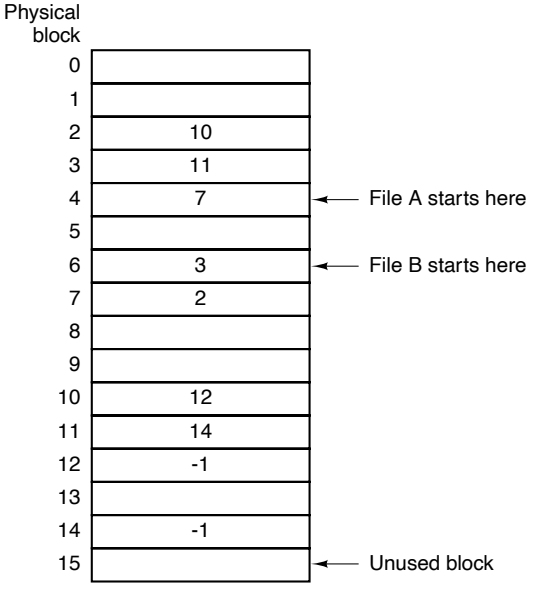

Figure 4-14. Linked-list allocation using a file-allocation table in main memory.

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is.

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 1-TB disk and a 1-KB block size, the table needs 1 billion entries, one for each of the 1 billion disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus

the table will take up 3 GB or 2.4 GB of main memory all the time, depending on whether the system is optimized for space or time. Not wildly practical. Clearly the FAT idea does not scale well to large disks. Nevertheless, it was the original MS-DOS file system and is still fully supported by all versions of Windows though (and UEFI). Versions of the FAT file system are still commonly used on the SD cards used in digital cameras, electronic picture frames, music players, and other portable electronic devices, as well as in other embedded applications.

I-nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node** (**index-node**), which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Fig. 4-15. Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node needs to be in memory only when the corresponding file is open. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only kn bytes. Only this much space need be reserved in advance.

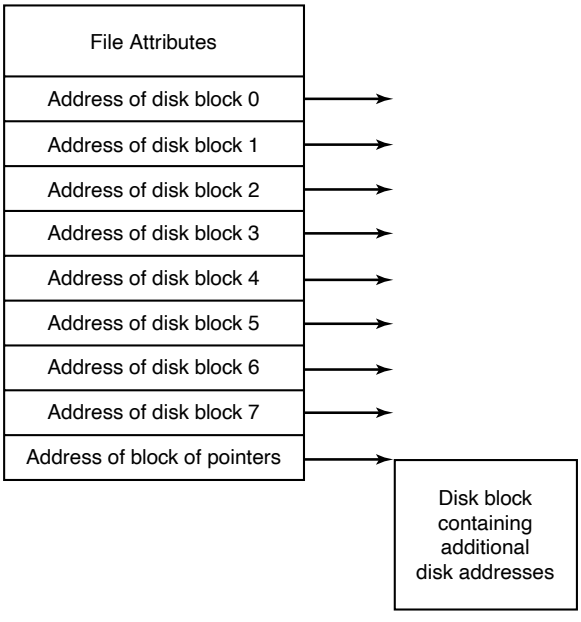

Figure 4-15. An example i-node.

This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the

linked list of all disk blocks is proportional in size to the disk itself. If the disk has n blocks, the table needs n entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 500 GB, 500 TB, or 500 PB.

One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disk-block addresses, as shown in Fig. 4-15. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses. We will come back to i-nodes when studying UNIX in Chap. 10. Similarly, the Windows NTFS file system uses a similar idea, only with bigger i-nodes that can also contain small files.

4.3.3 Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry on the disk. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (with contiguous allocation), the number of the first block (both linked-list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

A closely related issue is where the attributes should be stored. Every file system maintains various file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Some systems do precisely that. This option is shown in Fig. 4-16(a). In this simple design, a directory consists of a list of fixed-size entries, one per file, containing a (fixed-length) file name, a structure of the file attributes, and one or more disk addresses (up to some maximum) telling where the disk blocks are.

For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number. This approach is illustrated in Fig. 4-16(b). As we shall see later, this method has some advantages over putting them in the directory entry.

So far we have made the implicit assumption that files have short, fixed-length names. In MS-DOS files have a 1–8 character base name and an optional extension of 1–3 characters. In UNIX Version 7, file names were 1–14 characters, including any extensions. However, nearly all modern operating systems support longer, variable-length file names. How can these be implemented?

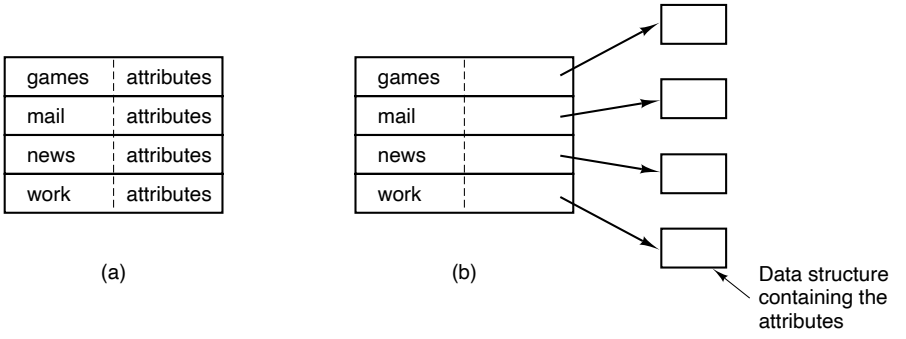

Figure 4-16. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

The simplest approach is to set a limit on file-name length, typically 255 characters, and then use one of the designs of Fig. 4-16 with 255 characters reserved for each file name. This approach is simple, but wastes a great deal of directory space, since few files have such long names. For efficiency reasons, a different structure is desirable.

One alternative is to give up the idea that all directory entries are the same size. With this method, each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes. This fixed-length header is followed by the actual file name, however long it may be, as shown in Fig. 4-17(a) in big-endian format (as used by some CPUs). In this example we have three files, *project-budget*, *personnel*, and *foo*. Each file name is terminated by a special character (usually 0), which is represented in the figure by a box with a cross in it. To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words, shown by shaded boxes in the figure.

A disadvantage of this method is that when a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit. This problem is essentially the same one we saw with contiguous disk files, only now compacting the directory is feasible because it is entirely in memory. Another problem is that a single directory entry may span multiple pages, so a page fault may occur while reading a file name.

Another way to handle variable-length names is to make the directory entries themselves all fixed length and keep the file names together in a heap at the end of the directory, as shown in Fig. 4-17(b). This method has the advantage that when an entry is removed, the next file entered will always fit there. Of course, the heap must be managed and page faults can still occur while processing file names. One very minor win here is that there is no longer any real need for file names to begin

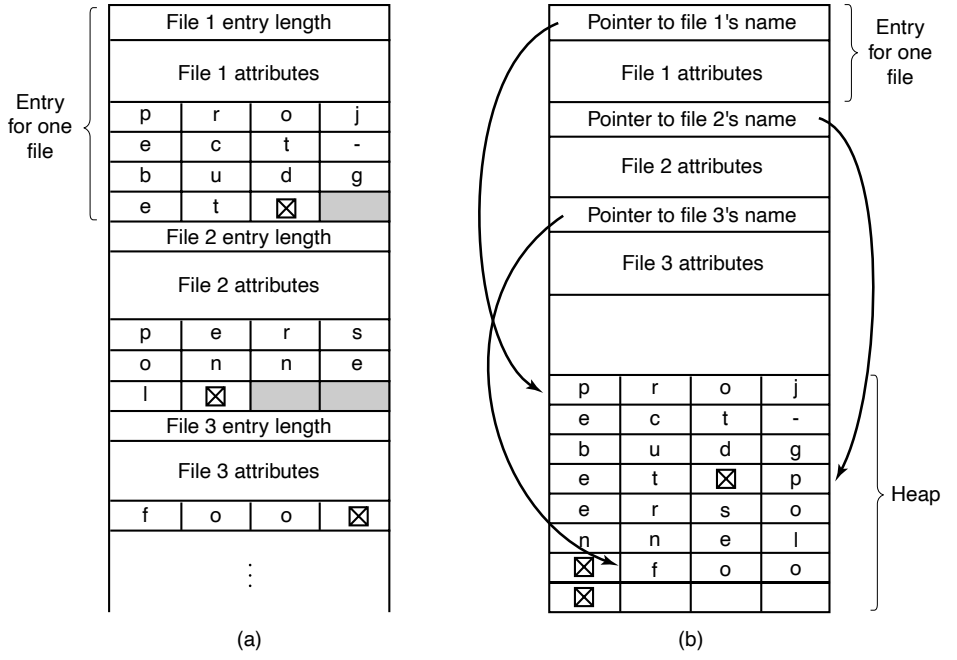

Figure 4-17. Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

at word boundaries, so no filler characters are needed after file names in Fig. 4-17(b) as they are in Fig. 4-17(a).

In all of the designs so far, directories are searched linearly from beginning to end when a file name has to be looked up. For extremely long directories, linear searching can be slow. One way to speed up the search is to use a hash table in each directory. Call the size of the table n. To enter a file name, the name is hashed onto a value between 0 and $n - 1$, for example, by dividing it by n and taking the remainder. Alternatively, the words comprising the file name can be added up and this quantity divided by n, or something similar.

Either way, the table entry corresponding to the hash code is inspected. If it is unused, a pointer is placed there to the file entry. File entries follow the hash table. If that slot is already in use, a linked list is constructed, headed at the table entry and threading through all entries with the same hash value.

Looking up a file follows the same procedure. The file name is hashed to select a hash-table entry. All the entries on the chain headed at that slot are checked to see if the file name is present. If the name is not on the chain, the file is not present in the directory.

Using a hash table has the advantage of much faster lookup, but the disadvantage of a much more complex administration. It is only really a serious candidate

in systems where it is expected that directories will routinely contain hundreds or thousands of files.

A different way to speed up searching large directories is to cache the results of searches. Before starting a search, a check is first made to see if the file name is in the cache. If so, it can be located immediately. Of course, caching only works if a relatively small number of files comprise the majority of the lookups.

4.3.4 Shared Files

When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Figure 4-18 shows the file system of Fig. 4-8 again, only with one of *C*'s files now present in one of *B*'s directories as well. The connection between *B*'s directory and the shared file is called a **link**. The file system itself is now a **DAG** (**Directed Acyclic Graph**), rather than a tree. Having the file system be a DAG complicates maintenance, but such is life.

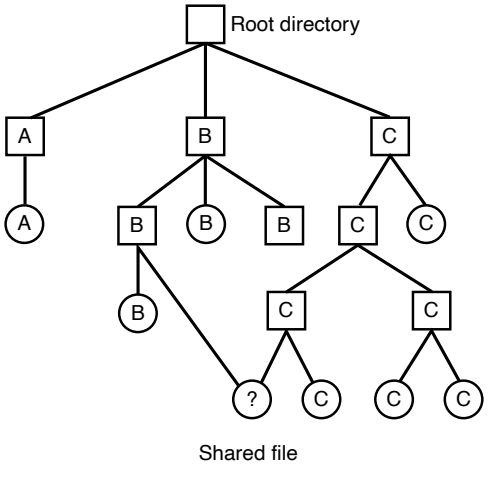

Figure 4-18. File system containing a shared file.

Sharing files is convenient, but it also introduces some problems. To start with, if directories really do contain disk addresses, then a copy of the disk addresses will have to be made in *B*'s directory when the file is linked. If either *B* or *C* subsequently appends to the file, the new blocks will be listed only in the directory of the user doing the append. The changes will not be visible to the other user, thus defeating the purpose of sharing.

This problem can be solved in two ways. In the first solution, disk blocks are not listed in directories, but in a little data structure associated with the file itself. The directories would then just have pointers to the little data structure. This is the approach used in UNIX (where the little data structure is the i-node).

In the second solution, *B* links to one of *C*'s files by having the system create a new file, of type LINK, and entering that file in *B*'s directory. The new file contains just the path name of the file to which it is linked. When *B* reads from the linked file, the operating system sees that the file being read from is of type LINK, looks up the name of the file, and reads that file. This approach is called **symbolic linking**, to contrast it with traditional (hard) linking, as discussed earlier.

Each of these methods has its drawbacks. In the first method, at the moment that *B* links to the shared file, the i-node records the file's owner as *C*. Creating a link does not change the ownership (see Fig. 4-19), but it does increase the link count in the i-node, so the system knows how many directory entries currently point to the file.

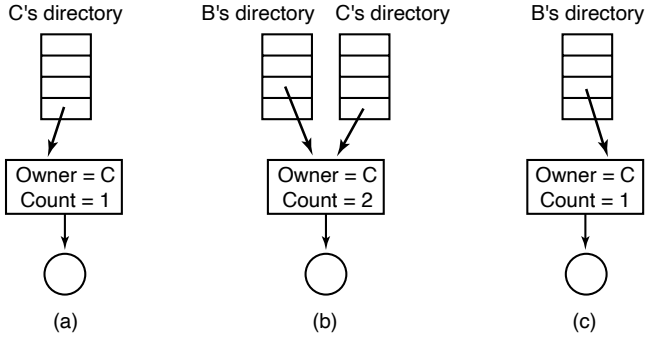

Figure 4-19. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

If *C* subsequently tries to remove the file, the system is faced with a problem. If it removes the file and clears the i-node, *B* will have a directory entry pointing to an invalid i-node. If the i-node is later reassigned to another file, *B*'s link will point to the wrong file. The system can see from the count in the i-node that the file is still in use, but there is no easy way for it to find all the directory entries for the file, in order to erase them. Pointers to the directories cannot be stored in the i-node because there can be an unlimited number of directories.

The only thing to do is remove *C*'s directory entry, but leave the i-node intact, with count set to 1, as shown in Fig. 4-19(c). We now have a situation in which *B* is the only user having a directory entry for a file owned by *C*. If the system does accounting or has quotas, *C* will continue to be billed for the file until *B* decides to remove it, if ever, at which time the count goes to 0 and the file is deleted.

With symbolic links this problem does not arise because only the true owner has a pointer to the i-node. Users who have linked to the file just have path names, not i-node pointers. When the *owner* removes the file, it is destroyed. Subsequent attempts to use the file via a symbolic link will fail when the system is unable to locate the file. Removing a symbolic link does not affect the file at all.

The problem with symbolic links is the extra overhead required. The file containing the path must be read, then the path must be parsed and followed, component by component, until the i-node is reached. All of this activity may require a considerable number of extra disk accesses. Furthermore, an extra i-node is needed for each symbolic link, as is an extra disk block to store the path, although if the path name is short, the system could store it in the i-node itself, as a kind of optimization. Symbolic links have the advantage that they can be used to link to files on machines anywhere in the world, by simply providing the network address of the machine where the file resides in addition to its path on that machine.

There is also another problem introduced by links, symbolic or otherwise. When links are allowed, files can have two or more paths. Programs that start at a given directory and find all the files in that directory and its subdirectories will locate a linked file multiple times. For example, a program that dumps all the files in a directory and its subdirectories onto a backup drive may make multiple copies of a linked file. Furthermore, if the backup drive is then read into another machine, unless the dump program is clever, the linked file may be copied twice onto the disk, instead of being linked.

4.3.5 Log-Structured File Systems

Changes in technology are putting pressure on current file systems. Let us consider computers with (magnetic) hard disks. In the next section, we will look at SSDs. In systems with hard disks, the CPUs keep getting faster, the disks are becoming much bigger and cheaper (but not much faster), and memories are growing exponentially in size. The one parameter that is not improving by leaps and bounds is disk seek time.

The combination of these factors led to a performance bottleneck in file systems. Research done at Berkeley attempted to alleviate this problem by designing a completely new kind of file system, LFS (the **Log-structured File System**). In this section, we will briefly describe how LFS works. For a more complete treatment, see the original paper on LFS (Rosenblum and Ousterhout, 1991).

The idea that drove the LFS design is that as CPUs get faster and RAM memories get larger, disk caches are also increasing rapidly. Consequently, it is now possible to satisfy a very substantial fraction of all read requests directly from the file-system cache, with no disk access needed. It follows from this observation that in the future, most disk accesses will be writes, so the read-ahead mechanism used in some file systems to fetch blocks before they are needed no longer gains much performance.

To make matters worse, in most file systems, writes are done in very small chunks. Small writes are highly inefficient, since a 50-μsec disk write is often preceded by a 10-msec seek and a 4-msec rotational delay. With these parameters, disk efficiency drops to a fraction of 1%.

To see where all the small writes come from, consider creating a new file on a UNIX system. To write this file, the i-node for the directory, the directory block, the i-node for the file, and the file itself must all be written. While these writes can be delayed, doing so exposes the file system to serious consistency problems if a crash occurs before the writes are done. For this reason, the i-node writes are generally done immediately.

From this reasoning, the LFS designers decided to reimplement the UNIX file system in such a way as to achieve the full bandwidth of the disk, even in the face of a workload consisting in large part of small random writes. The basic idea is to structure the entire disk as a great big log.

Periodically, and when there is a special need for it, all the pending writes being buffered in memory are collected into a single segment and written to the disk as a single contiguous segment at the end of the log. A single segment may thus contain i-nodes, directory blocks, and data blocks, all mixed together. At the start of each segment is a segment summary, telling what can be found in the segment. If the average segment can be made to be about 1 MB, almost the full bandwidth of the disk can be utilized.

In this design, i-nodes still exist and even have the same structure as in UNIX, but they are now scattered all over the log, instead of being at a fixed position on the disk. Nevertheless, when an i-node is located, locating the blocks is done in the usual way. Of course, finding an i-node is now much harder, since its address cannot simply be calculated from its i-number, as in UNIX. To make it possible to find i-nodes, an i-node map, indexed by i-number, is maintained. Entry *i* in this map points to i-node *i* on the disk. The map is kept on disk, but it is also cached, so the most heavily used parts will be in memory most of the time.

To summarize what we have said so far, all writes are initially buffered in memory, and periodically all the buffered writes are written to the disk in a single segment, at the end of the log. Opening a file now consists of using the map to locate the i-node for the file. Once the i-node has been located, the addresses of the blocks can be found from it. All of the blocks will themselves be in segments, somewhere in the log.

If disks were infinitely large, the above description would be the entire story. However, real disks are finite, so eventually the log will occupy the entire disk, at which time no new segments can be written to the log. Fortunately, many existing segments may have blocks that are no longer needed. For example, if a file is overwritten, its i-node will now point to the new blocks, but the old ones will still be occupying space in previously written segments.

To deal with this problem, LFS has a **cleaner** thread that spends its time scanning the log circularly to compact it. It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there. It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use. If not, that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment. The original segment is

then marked as free, so that the log can use it for new data. In this manner, the cleaner moves along the log, removing old segments from the back and putting any live data into memory for rewriting in the next segment. Consequently, the disk is a big circular buffer, with the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

The bookkeeping here is nontrivial, since when a file block is written back to a new segment, the i-node of the file (somewhere in the log) must be located, updated, and put into memory to be written out in the next segment. The i-node map must then be updated to point to the new copy. Nevertheless, it is possible to do the administration, and the performance results show that all this complexity is worthwhile. Measurements given in the papers cited above show that LFS outperforms UNIX by an order of magnitude on small writes, while having a performance that is as good as or better than UNIX for reads and large writes.

4.3.6 Journaling File Systems

Log-structured file systems are an interesting idea in general and one of the ideas inherent in them, robustness in the face of failure, can also be applied to more conventional file systems. The basic idea here is to keep a log of what the file system is going to do before it does it, so that if the system crashes before it can do its planned work, upon rebooting the system can look in the log to see what was going on at the time of the crash and finish the job. Such file systems, called **journaling file systems**, are very popular. Microsoft's NTFS file system and the Linux ext4 and ReiserFS file systems all use journaling. MacOS offers journaling file systems as an option. Journaling is the default and it is widely used. Below we will give a brief introduction to this topic.

To see the nature of the problem, consider a simple garden-variety operation that happens all the time: removing a file. This operation (in UNIX) requires three steps:

1. Remove the file from its directory.
2. Release the i-node to the pool of free i-nodes.
3. Return all the disk blocks to the pool of free disk blocks.

In Windows, analogous steps are required. In the absence of system crashes, the order in which these steps are taken does not matter; in the presence of crashes, it does. Suppose that the first step is completed and then the system crashes. The i-node and file blocks will not be accessible from any file, but will also not be available for reassignment; they are just off in limbo somewhere, decreasing the available resources. If the crash occurs after the second step, only the blocks are lost.

If the order of operations is changed and the i-node is released first, then after rebooting, the i-node may be reassigned, but the old directory entry will continue

to point to it, hence to the wrong file. If the blocks are released first, then a crash before the i-node is cleared will mean that a valid directory entry points to an i-node listing blocks now in the free storage pool and which are likely to be reused shortly, leading to two or more files randomly sharing the same blocks. None of these outcomes are good.

What the journaling file system does is first write a log entry listing the three actions to be completed. The log entry is then written to disk (and for good measure, possibly read back from the disk to verify that it was, in fact, written correctly). Only after the log entry has been written, do the various operations begin. After the operations complete successfully, the log entry is erased. If the system now crashes, upon recovery the file system can check the log to see if any operations were pending. If so, all of them can be rerun (multiple times in the event of repeated crashes) until the file is correctly removed.

To make journaling work, the logged operations must be **idempotent**, which means they can be repeated as often as needed without harm. Operations such as "Update the bitmap to mark i-node *k* or block *n* as free" can be repeated until the cows come home with no danger. Similarly, searching a directory and removing any entry called *foobar* is also idempotent. On the other hand, adding the newly freed blocks from i-node *K* to the end of the free list is not idempotent since they may already be there. The more-expensive operation "Search the list of free blocks and add block *n* to it if it is not already present" is idempotent. Journaling file systems have to arrange their data structures and loggable operations so they all are idempotent. Under these conditions, crash recovery can be made fast and secure.

For added reliability, a file system can introduce the database concept of an **atomic transaction**. When this concept is used, a group of actions can be bracketed by the begin transaction and end transaction operations. The file system then knows it must complete either all the bracketed operations or none of them, but not any other combinations.

NTFS has an extensive journaling system and its structure is rarely corrupted by system crashes. It has been in development since its first release with Windows NT in 1993. The first Linux file system to do journaling was ReiserFS, but its popularity was impeded by the fact that it was incompatible with the then-standard ext2 file system. In contrast, ext3 which was a less ambitious project than ReiserFS, also does journaling while maintaining compatibility with the previous ext2 system. Its successor, ext4 was similarly developed initially as a series of backward-compatible extensions to ext3.

4.3.7 Flash-based File Systems

SSDs uses flash memory and operates quite differently from hard disk drives. Generally, NAND-based flash is used (rather than NOR-based flash) in SSDs. Much of the difference is related to the physics that underpins the storage, which,

however fascinating, is beyond the scope of this chapter. Irrespective of the flash technology, here are important differences between hard disks and flash storage. There are no moving parts in flash storage and the problems of seek times and rotational delays that we mentioned in the previous section do not exist. This means that the access time (latency) is much better—on the order of several tens of microseconds instead of milliseconds. It also means that on SSDs there is not much of a gap in performance between random and sequential reads. As we shall see, random writes are still quite a bit more expensive—especially small ones.

Indeed, unlike magnetic disks, flash technology has asymmetric read and write performance: reads are much faster than writes. For instance, where a read takes a few tens of microseconds, a write can take hundreds. First, writes are slow because of how the flash cells that implement the bits are programmed—this is physics and not something we want to get into here. A second, and more impactful reason is that you can only write a unit of data after you have erased a suitable area on the device. In fact, flash memory distinguishes between a *unit of I/O* (often 4 KB), and a *unit of erase* (often 64-256 units of I/O, so up to several MB). Sadly, the industry takes great pleasure in confusing people and refers to a unit of I/O as a *page* and to the unit of erase as a *block*, except for those publications that refer to the unit of I/O as a block or even a sector and the unit of erase as a chunk. Of course, the meaning of a page is also quite different from that of a memory page in the previous chapter and the meaning of a block does not match that of a disk block either. To avoid confusion, we will use the terms *flash page* and *flash block* for the unit of I/O and the unit of erase, respectively.

To write a flash page, the SSD must first erase a flash block—an expensive operation taking hundreds of microseconds. Fortunately, after it has erased the block, there are many free flash pages in that space and the SSD can now write the flash pages in the flash block in order. In other words, it first writes flash page 0 in the block, then 1, then 2, etc. It cannot write flash page 0, followed by 2, and then 1. Also, the SSD cannot really overwrite a flash page that was written earlier. It first has to erase the entire flash block again (not just the page). Indeed, if you really wanted to overwrite some data in a file in-place, the SSD would need to save the other flash pages in the block somewhere else, erase the block in its entirety, and then rewrite the pages one by one—not a cheap operation at all! Instead, modifying data on an SSD simply makes the old flash page invalid and then rewrites the new content in another block. If there are no blocks with free pages available, this would require erasing a block first.

You do not want to keep writing the same flash pages all the time anyway, as flash memory suffers from wear. Repeatedly writing and erasing takes its toll and at some point the flash cells that hold the bits can no longer be used. A program/erase (P/E) cycle consists of erasing a cell and writing new content in it. Typical flash memory cells have a maximum endurance of a few thousand to a few hundred thousand P/E cycles before they kick the bucket. In other words, it is important to spread the wear across the flash memory cells as much as possible.

The device component that is responsible for handling such wear-leveling is known as the **FTL** (**Flash Translation Layer**). It has many other responsibilities also and it is sometimes referred to as the drive's secret sauce. The secret sauce typically runs on a simple processor with access to fast memory. It is shown on the left in Fig. 4-20. The data are stored in the flash packages (FPs) on the right. Each flash package consists of multiple dies and each die in turn contains a number of so-called planes: collections of flash blocks containing flash pages.

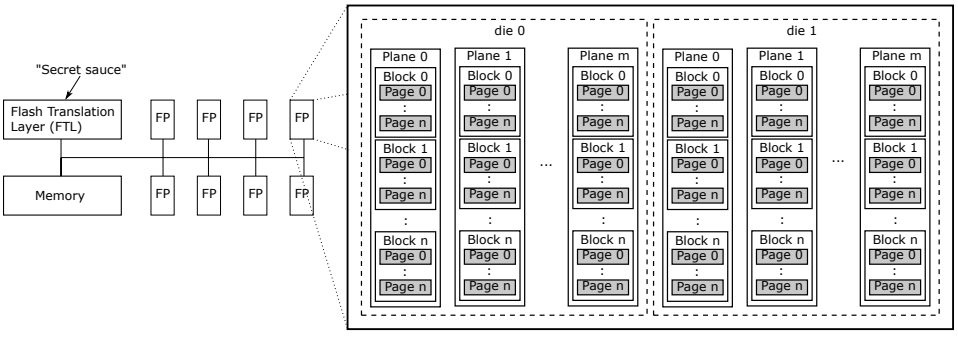

Figure 4-20. Components inside a typical flash SSD.

To access a specific flash page on the SSD, we need to address the corresponding die on the appropriate flash package, and on that die the right plane, block and page—a rather complicated, hierarchical address! Unfortunately, this is not how file systems work at all. The file system simply requests to read a disk block at a linear, logical disk address. How does the SSD translate between these logical addresses and the complex physical addresses on the device? Here is a hint: the Flash Translation Layer was not given its name for nothing.

Much like the paging mechanism in virtual memory, the FTL uses translation tables to indicate that logical block 54321 is really at die 0 of flash package 1, in plane 2 and block 5. Such translation tables are handy also for wear leveling, since the device is free to move a page to a different block (for instance, because it needs to be updated), as long as it adjusts the mapping in the translation table.

The FTL also takes care of managing blocks and pages that are no longer needed. Suppose that after deleting or moving data a few times, a flash block contains several invalid flash pages. Since only some of the pages are now valid, the device can free up space by copying the remaining valid pages to a block which still has free pages available and then erasing the original block. This is known as **garbage collection**. In reality, things are much more complicated. For instance, when do we do garbage collection? If we do it constantly and as early as possible, it may interfere with the user's I/O requests. If we do it too late, we may run out of free blocks. A reasonable compromise is to do it during idle periods, when the SSD is not busy otherwise.

In addition, the garbage collector needs to select both a victim block (the flash block to clean) and a target block (to which to write the live data still in the victim block). Should it simply pick these blocks in a random or round-robin fashion, or try to make a more informed decision? For instance, for the victim block, should it select the flash block with the least amount of valid data, or perhaps avoid the flash blocks that have a lot of wear already, or the blocks that contain a lot of "hot" data (i.e., data that is likely to be written again in the near future anyway)? Likewise, for the target block, should it pick based on the amount of available space, or the amount of accumulated wear on the flash block? Moreover, should it try to group hot and cold data to ensure that cold flash pages can mostly stay in the same block with no need for moving them around, while hot pages will perhaps be updated together close in time, so we can collect the updates in memory and then write them out to a new flash block in one blast? The answer is: yes. And if you're wondering which strategy is best, the answer is: it depends. Modern FTLs actually use a combination of these techniques.

Clearly, garbage collection is complex and a lot of work. It also leads to an interesting performance property. Suppose there are many flash blocks with invalid pages, but all of them have only a small number of such pages. In that case, the garbage collector will have to separate the valid from the invalid pages for many blocks, each time coalescing the valid data in new blocks and erasing the old blocks to free up space, at a significant cost in performance and wear. Can you now see why small random writes may be costlier for garbage collection than sequential ones?

In reality, small random writes are expensive regardless of garbage collection, if they overwrite an existing flash page in a full block. The problem has to do with the mappings in the translation tables. To save space, the FTL has two types of mappings: per page and per block. If everything was mapped per-page, we would need an enormous amount of memory to store the translation table. Where possible, therefore, the FTL tries to map a block of pages that belong together as a single entry. Unfortunately, that also means that modifying even a single byte in that block will invalidate the entire block and lead to lots of additional writes. The actual overheads of random writes depend on the garbage collection algorithm and the overall FTL implementation, both of which are typically as secret (and well-guarded) as the formula for Coca Cola.

The decoupling of logical disk block addresses and physical flash addresses creates an additional problem. With a hard disk drive, when the file system deletes a file, it knows exactly which blocks on the disk are now free for reuse and can reuse them as it sees fit. This is not the case with SSDs. The file system may decide to delete a file and mark the logical block addresses as free, but how is the SSD to know which of its flash pages have been deleted and can therefore be safely garbage collected? The answer is: it does not and needs to be told explicitly by the file system. For this, the file system may use the TRIM command which tells the SSD that certain flash pages are now free. Note that an SSD without the TRIM

command still works (and indeed some operating systems have worked without TRIM for years), but less efficiently. In this case, the SSD would only discover that the flash pages are invalid when the file system tries to overwrite them. We say that the TRIM command helps bridge the *semantic gap* between the FTL and the file system—the FTL does not have sufficient visibility to do its job efficiently without some help. It is a major difference between file systems for hard disks and file systems for SSD.

Let us recap what we have learned about SSDs so far. We saw that flash devices have excellent sequential read, but also very good random read performance, while random writes are slow (although still much faster than read or write accesses to a disk). Also, we know that frequent writes to the same flash cells rapidly reduces their lifetimes. Finally, we saw that doing complex things at the FTL is difficult due to the semantic gap.

The reason that we want a new file system for flash is not really the presence or absence of a TRIM command, but rather that the unique properties of flash make it a poor match for existing file systems such as NTFS or ext4. So what file system would be a good match? Since most reads can be served from the cache, we should look at the writes. We also know that we should avoid random writes and spread the writes evenly for wear-leveling. By now you may be thinking: "Wait, that sounds like a match for a log-structured file system," and you would be right. Log-structured file systems, with their immutable logs and sequential writes appear to be a perfect fit for flash-based storage.

Of course, a log-structured file system on flash does not automatically solve all problems. In particular, consider what happens when we update a large file. In terms of Fig. 4-15, a large file will use the disk block containing additional disk addresses that we see in the bottom right and that we will refer to as a (single) *indirect block*. Besides writing the updated flash page to a new block, the file system also needs to update the indirect block, since the logical (disk) address of the file data has changed. The update means that the flash page corresponding to the indirect block must be moved to another flash block. In addition, because the logical address of the indirect block has now changed, the file system should also update the i-node itself—leading to a new write to a new flash block. Finally, since the i-node is now in a new logical disk block, the file system must also update the i-node map, leading to another write on the SSD. In other words, a single file update leads to a cascade of writes of the corresponding meta-data. In real log-structured file systems, there may be more than one level of indirection (with double or even triple indirect blocks), so there will be even more writes. The phenomenon is known as the **recursive update problem** or **wandering tree problem**.

While recursive updates cannot be avoided altogether, it is possible to reduce the impact. For instance, instead of storing the actual disk address of the i-node or indirect block (in the i-node map and i-node, respectively), some file systems store the i-node / indirect block *number* and then maintain, at a fixed logical disk location, a global mapping of these (constant) numbers to logical block addresses on

disk. The advantage is that the file update in the example above only leads to an update of the indirect block and the global mapping, but not of any of the intermediate mappings. This solution was adopted in the popular Flash-Friendly File System (F2FS) supported by the Linux kernel.

In summary, while people may think of flash as a drop-in replacement for magnetic disks, it has led to many changes in the file system. This is nothing new. When magnetic disks started replacing magnetic tape, they led to many changes also. For instance, the *seek* operation was introduced and researchers started worrying about disk scheduling algorithms. In general, the introduction of new technology often leads to a flurry of activity and changes in the operating system to make optimal use of the new capabilities.

4.3.8 Virtual File Systems

Many different file systems are in use—often on the same computer—even for the same operating system. A Windows system may have a main NTFS file system, but also a legacy FAT-32 or FAT-16 drive or partition that contains old, but still needed, data, and from time to time, a flash drive with its own unique file system may be required as well. Windows handles these disparate file systems by identifying each one with a different drive letter, as in *C:*, *D:*, etc. When a process opens a file, the drive letter is explicitly or implicitly present so Windows knows which file system to pass the request to. There is no attempt to integrate heterogeneous file systems into a unified whole.

In contrast, all modern UNIX systems make a very serious attempt to integrate multiple file systems into a single structure. A Linux system could have ext4 as the root file system, with an ext3 partition mounted on */usr* and a second hard disk with a ReiserFS file system mounted on */home* as well as an F2FS flash file system temporarily mounted on */mnt*. From the user's point of view, there is a single file-system hierarchy. That it happens to encompass multiple (incompatible) file systems is not visible to users or processes.

However, the presence of multiple file systems is very definitely visible to the implementation, and since the pioneering work of Sun Microsystems (Kleiman, 1986), most UNIX systems have used the concept of a **VFS** (**Virtual File System**) to try to integrate multiple file systems into an orderly structure. The key idea is to abstract out that part of the file system that is common to all file systems and put that code in a separate layer that calls the underlying concrete file systems to actually manage the data. The overall structure is illustrated in Fig. 4-21. The discussion below is not specific to Linux or FreeBSD or any other version of UNIX, but gives the general flavor of how virtual file systems work in UNIX systems.

All system calls relating to files are directed to the virtual file system for initial processing. These calls, coming from user processes, are the standard POSIX calls, such as open, read, write, lseek, and so on. Thus the VFS has an "upper" interface to user processes and it is the well-known POSIX interface.

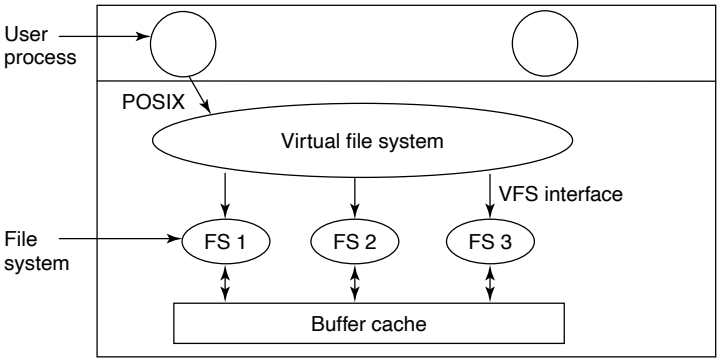

Figure 4-21. Position of the virtual file system.

The VFS also has a "lower" interface to the concrete file systems, which is labeled **VFS interface** in Fig. 4-21. This interface consists of several dozen function calls that the VFS can make to each file system to get work done. Thus to create a new file system that works with the VFS, the designers of the new file system must make sure that it supplies the function calls the VFS requires. An obvious example of such a function is one that reads a specific block from disk, puts it in the file system's buffer cache, and returns a pointer to it. Thus the VFS has two distinct interfaces: the upper one to the user processes and the lower one to the concrete file systems.

While most of the file systems under the VFS represent partitions on a local disk, this is not always the case. In fact, the original motivation for Sun to build the VFS was to support remote file systems using the **NFS** (**Network File System**) protocol. The VFS design is such that as long as the concrete file system supplies the functions the VFS requires, the VFS does not know or care where the data are stored or what the underlying file system is like. It requires is the proper interface to the underlying file systems.

Internally, most VFS implementations are essentially object oriented, even if they are written in C rather than C++. There are several key object types that are normally supported. These include the superblock (which describes a file system), the v-node (which describes a file), and the directory (which describes a file system directory). Each of these has associated operations (methods) that the concrete file systems must support. In addition, the VFS has some internal data structures for its own use, including the mount table and an array of file descriptors to keep track of all the open files in the user processes.

To understand how the VFS works, let us run through an example chronologically. When the system is booted, the root file system is registered with the VFS. In addition, when other file systems are mounted, either at boot time or during operation, they too must register with the VFS. When a file system registers, what

it basically does is provide a list of the addresses of the functions the VFS requires, either as one long call vector (table) or as several of them, one per VFS object, as the VFS demands. Thus once a file system has registered with the VFS, the VFS knows how to, say, read a block from it—it simply calls the fourth (or whatever) function in the vector supplied by the file system. Similarly, the VFS then also knows how to carry out every other function the concrete file system must supply: it just calls the function whose address was supplied when the file system registered.

After a file system has been mounted, it can be used. For example, if a file system has been mounted on */usr* and a process makes the call

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

while parsing the path, the VFS sees that a new file system has been mounted on */usr* and locates its superblock by searching the list of superblocks of mounted file systems. Having done this, it can find the root directory of the mounted file system and look up the path *include/unistd.h* there. The VFS then creates a v-node and makes a call to the concrete file system to return all the information in the file's i-node. This information is copied into the v-node (in RAM), along with other information, most importantly the pointer to the table of functions to call for operations on v-nodes, such as read, write, close, and so on.

After the v-node has been created, the VFS makes an entry in the file-descriptor table for the calling process and sets it to point to the new v-node. (For the purists, the file descriptor actually points to another data structure that contains the current file position and a pointer to the v-node, but this detail is not important for our purposes here.) Finally, the VFS returns the file descriptor to the caller so it can use it to read, write, and close the file.

Later when the process does a read using the file descriptor, the VFS locates the v-node from the process and file descriptor tables and follows the pointer to the table of functions, all of which are addresses within the concrete file system on which the requested file resides. The function that handles read is now called and code within the concrete file system goes and gets the requested block. The VFS has no idea whether the data are coming from the local disk, a remote file system over the network, a USB stick, or something different. The data structures involved are shown in Fig. 4-22. Starting with the caller's process number and the file descriptor, successively the v-node, read function pointer, and access function within the concrete file system are located.

In this manner, it becomes relatively straightforward to add new file systems. To make one, the designers first get a list of function calls the VFS expects and then write their file system to provide all of them. Alternatively, if the file system already exists and needs to be ported to the VFS, then they have to provide wrapper functions that do what the VFS needs, usually by making one or more native calls to the underlying concrete file system.

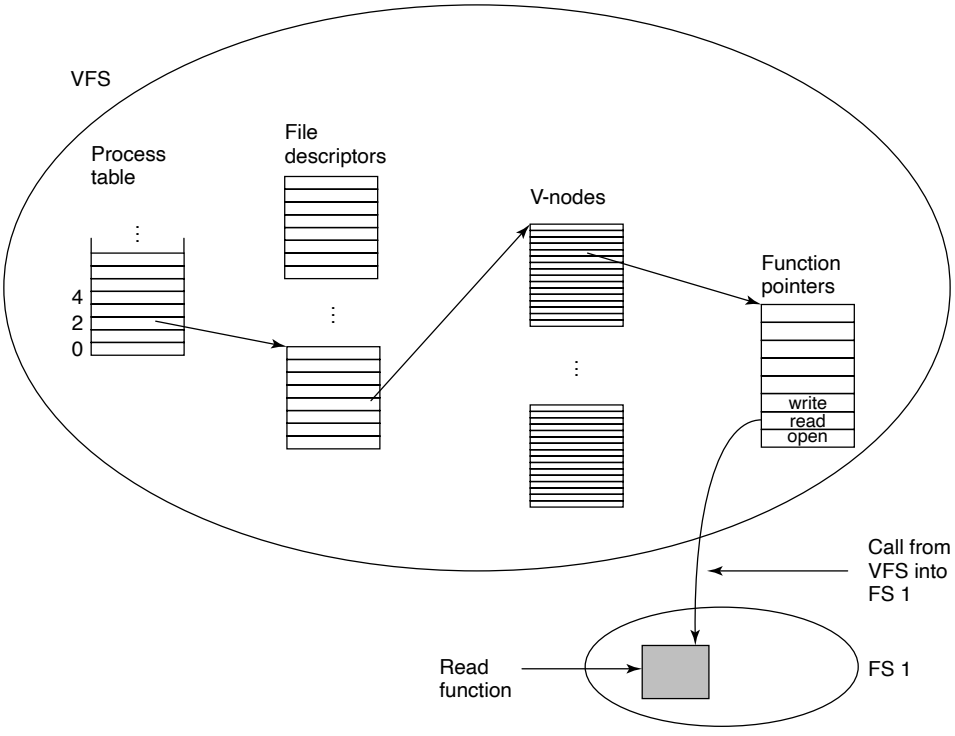

Figure 4-22. A simplified view of the data structures and code used by the VFS and concrete file system to do a read.

4.4 FILE-SYSTEM MANAGEMENT AND OPTIMIZATION

Making the file system work is one thing; making it work efficiently and robustly in real life is something quite different. In the following sections, we will look at some of the issues involved in managing disks.

4.4.1 Disk-Space Management

Files are normally stored on disk, so management of disk space is a major concern to file-system designers. Two general strategies are possible for storing an n byte file: n consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks[†]. The same trade-off is present in memory-management systems between pure segmentation and paging.

As we have seen, storing a file simply as a contiguous sequence of bytes has

[†]Disk blocks, not flash blocks. In general, "block" means *disk* block, unless explicitly stated otherwise.

the obvious problem that if a file grows, it may have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Block Size

Once it has been decided to store files in fixed-size blocks, the question arises how big the block should be. Given the way hard disks are organized, the sector, the track, and the cylinder are obvious candidates for the unit of allocation (although these are all device dependent, which is a minus). In flash-based systems, the flash page size is another candidate, while in a paging system, the memory page size is also a major contender.

Since magnetic disks have served as the storage work horse for years and led to many of the design choices, such as the common 4 KB block size still used today, let us consider them first. On a hard disk, having a large block size means that every file, even a 1-byte file, ties up an entire block. It also means that small files waste a large amount of disk space. On the other hand, a small block size means that most files will span multiple blocks and thus need multiple seeks and rotational delays to read them, reducing performance. Thus if the allocation unit is too large, we waste space; if it is too small, we waste time. The block size of 4 KB is considered a reasonable compromise for average users.

As an example, consider a disk with 1 MB per track, a rotation time of 8.33 msec, and an average seek time of 5 msec. The time in milliseconds to read a block of k bytes is then the sum of the seek, rotational delay, and transfer times:

$$5 + 4.165 + (k/1000000) \times 8.33$$

The dashed curve of Fig. 4-23 shows the data rate for such a disk as a function of block size. To compute the space efficiency, we need to make an assumption about the mean file size. For simplicity, let us assume that all files are 4 KB. While this is clearly not true in practice, it turns out that modern file systems are littered with files of a few kilobytes in size (e.g., icons, emojis, and emails) so this is not a crazy number either. The solid curve of Fig. 4-23 shows the space efficiency as a function of block size.

The two curves can be understood as follows. The access time for a block is completely dominated by the seek time and rotational delay, so given that it is going to cost 9 msec to access a block, the more data that are fetched, the better. Hence, the data rate goes up almost linearly with block size (until the transfers take so long that the transfer time begins to matter).

Now consider space efficiency. With 4-KB files and 1-KB, 2-KB, or 4-KB blocks, files use 4, 2, and 1 block, respectively, with no wastage. With an 8-KB block and 4-KB files, the space efficiency drops to 50%, and with a 16-KB block it

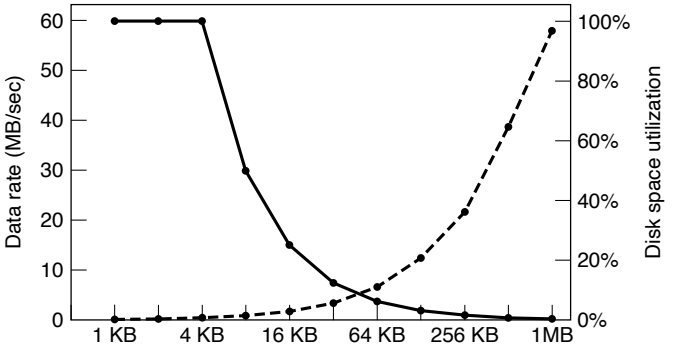

Figure 4-23. The dashed curve (left-hand scale) gives the data rate of a disk. The solid curve (right-hand scale) gives the disk-space efficiency. All files are 4 KB.

is down to 25%. In reality, few files are an exact multiple of the disk block size, so some space is always wasted in the last block of a file.

What the curves show, however, is that performance and space utilization are inherently in conflict. Small blocks are bad for performance but good for disk-space utilization. For these data, no reasonable compromise is available. The size closest to where the two curves cross is 64 KB, but the data rate is only 6.6 MB/sec and the space efficiency is about 7%, neither of which is very good. Historically, file systems have chosen sizes in the 1-KB to 4-KB range, but with disks now exceeding multiple TB, it might be better to increase the block size and accept the wasted disk space. Disk space is hardly in short supply any more.

So far we have looked at the optimal block size from the perspective of a hard disk and observed that if the allocation unit is too large, we waste space, while if it is too small, we waste time. With flash storage, we incur memory waste not just for large disk blocks, but also for smaller ones that do not fill up a flash page.

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used, as shown in Fig. 4-24. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is required for the pointer to the next block.) Consider a 1-TB disk, which has about 1 billion disk blocks. To store all these addresses at 255 per block requires about 4 million blocks. Generally, free blocks are used to hold the free list, so the storage is essentially free.

The other free-space management technique is the bitmap. A disk with n blocks requires a bitmap with n bits. Free blocks are represented by 1s in the map,

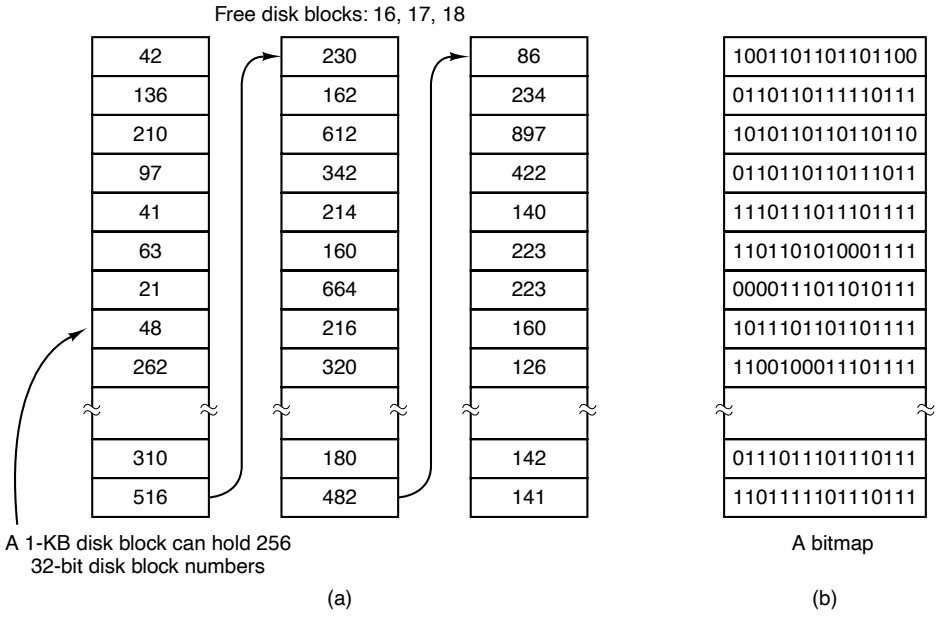

Figure 4-24. (a) Storing the free list on a linked list. (b) A bitmap.

allocated blocks by 0s (or vice versa). For our example 1-TB disk, we need 1 billion bits for the map, which requires around 130,000 1-KB blocks to store. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, vs. 32 bits in the linked-list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked-list scheme require fewer blocks than the bitmap.

If free blocks tend to come in long runs of consecutive blocks, the free-list system can be modified to keep track of runs of blocks rather than single blocks. An 8-, 16-, or 32-bit count could be associated with each block giving the number of consecutive free blocks. In the best case, a basically empty disk could be represented by two numbers: the address of the first free block followed by the count of free blocks. On the other hand, if the disk becomes severely fragmented, keeping track of runs is less efficient than keeping track of individual blocks because not only must the address be stored, but also the count.

This issue illustrates a problem operating system designers often have. There are multiple data structures and algorithms that can be used to solve a problem, but choosing the best one requires data that the designers do not have and will not have until the system is deployed and heavily used. And even then, the data may not be available. For instance, while we may measure the file size distribution and disk usage in one or two environments, we have little idea if these numbers are representative of home computers, corporate computers, government computers, not to mention tablets and smartphones, and others.

Getting back to the free list method for a moment, only one block of pointers need be kept in main memory. When a file is created, the needed blocks are taken from the block of pointers. When it runs out, a new block of pointers is read in from the disk. Similarly, when a file is deleted, its blocks are freed and added to the block of pointers in main memory. When this block fills up, it is written to disk.

Under certain circumstances, this method leads to unnecessary disk I/O. Consider the situation of Fig. 4-25(a), in which the block of pointers in memory has room for only two more entries. If a three-block file is freed, the pointer block overflows and has to be written to disk, leading to the situation of Fig. 4-25(b). If a three-block file is now written, the full block of pointers has to be read in again, taking us back to Fig. 4-25(a). If the three-block file just written was a temporary file, when it is freed, another disk write is needed to write the full block of pointers back to the disk. In short, when the block of pointers is almost empty, a series of short-lived temporary files can cause a lot of disk I/O.

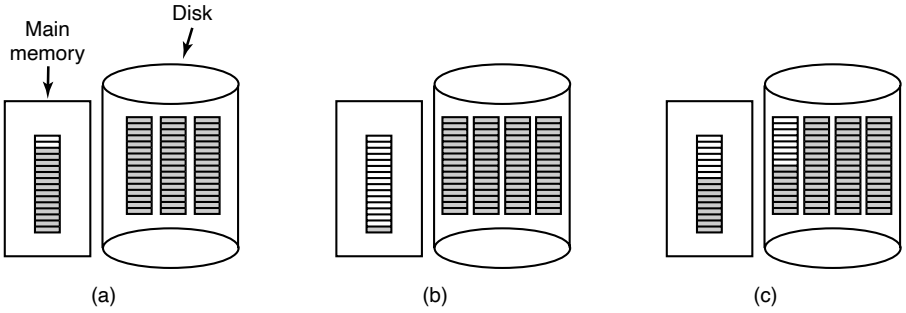

Figure 4-25. (a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

An alternative approach that avoids most of this disk I/O is to split the full block of pointers. Thus instead of going from Fig. 4-25(a) to Fig. 4-25(b), we go from Fig. 4-25(a) to Fig. 4-25(c) when three blocks are freed. Now the system can handle a series of temporary files without doing any disk I/O. If the block in memory fills up, it is written to the disk, and the half-full block from the disk is read in. The idea here is to keep most of the pointer blocks on disk full (to minimize disk usage), but keep the one in memory about half full, so it can handle both file creation and file removal without disk I/O on the free list.

With a bitmap, it is also possible to keep just one block in memory, going to disk for another only when it becomes completely full or empty. An additional benefit of this approach is that by doing all the allocation from a single block of the bitmap, the disk blocks will be close together, thus minimizing disk-arm motion.

Since the bitmap is a fixed-size data structure, if the kernel is (partially) paged, the bitmap can be put in virtual memory and have pages of it paged in as needed.

Disk Quotas

To prevent people from hogging too much disk space, multiuser operating systems often provide a mechanism for enforcing disk quotas. The idea is that the system administrator assigns each user a maximum allotment of files and blocks, and the operating system makes sure that the users do not exceed their quotas. A typical mechanism is described below.

When a user opens a file, the attributes and disk addresses are located and put into an open-file table in main memory. Among the attributes is an entry telling who the owner is. Any increases in the file's size will be charged to the owner's quota.

A second table contains the quota record for every user with a currently open file, even if the file was opened by someone else. This table is shown in Fig. 4-26. It is an extract from a quota file on disk for the users whose files are currently open. When all the files are closed, the record is written back to the quota file.

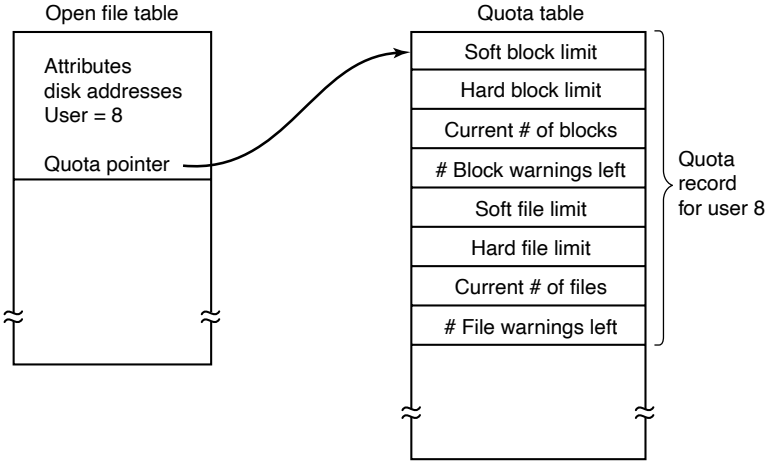

Figure 4-26. Quotas are kept track of on a per-user basis in a quota table.

When a new entry is made in the open-file table, a pointer to the owner's quota record is entered into it, to make it easy to find the various limits. Every time a block is added to a file, the total number of blocks charged to the owner is incremented, and a check is made against both the hard and soft limits. The soft limit may be exceeded, but the hard limit may not. An attempt to append to a file when the hard block limit has been reached will result in an error. Analogous checks also exist for the number of files to prevent a user from hogging all the i-nodes.

When a user attempts to log in, the system examines the quota file to see if the user has exceeded the soft limit for either number of files or number of disk blocks. If either limit has been violated, a warning is displayed, and the count of warnings remaining is reduced by one. If the count ever gets to zero, the user has ignored the warning one time too many, and is not permitted to log in. Getting permission to log in again will require some discussion with the system administrator.

This method has the property that users may go above their soft limits during a login session, provided they remove the excess before logging out. The hard limits may never be exceeded.

4.4.2 File-System Backups

Destruction of a file system is often a far greater disaster than destruction of a computer. If a computer is destroyed by fire, lightning surges, or a cup of coffee poured onto the keyboard, it is annoying and will cost money, but generally a replacement can be purchased with a minimum of fuss. Inexpensive personal computers can even be replaced within an hour by just going to a computer store (except at universities, where issuing a purchase order takes three committees, five signatures, and 90 days).

If a computer's file system is irrevocably lost, whether due to hardware or software failures, restoring all the information will be difficult, time consuming, and in many cases, impossible. For the people whose programs, documents, tax records, customer files, databases, marketing plans, or other data are gone forever, the consequences can be catastrophic. While the file system cannot offer any protection against physical destruction of the equipment and media, it can help protect the information. It is pretty straightforward: make backups. But that is not quite as simple as it sounds. Let us take a look.

Most people do not think making backups of their files is worth the time and effort—until one fine day their disk abruptly dies, at which time most of them undergo an instantaneous change of heart. Companies, however, (usually) well understand the value of their data and generally do a backup at least once a day, to a large disk or even good old-fashioned tape. Tape is still very cost efficient, costing less than $10/TB; no other medium comes close to that price. For companies with petabytes or exabytes of data, cost of the backup medium matters. Nevertheless, making backups is not quite as trivial as it sounds, so we will examine some of the related issues below.

Backups are generally made to handle one of two potential problems:

1. Recover from disaster
2. Recover from user mistakes

The first one covers getting the computer running again after a disk crash, fire, flood, or some other natural catastrophe. In practice, these things do not happen

very often, which is why many people do not bother with backups. These people also tend not to have fire insurance on their houses for the same reason.

The second reason is that users often accidentally remove files that they later need again. This problem occurs so often that when a file is "removed" in Windows, it is not deleted at all, but just moved to a special directory, the **recycle bin**, so it can be fished out and restored easily later. Backups take this principle further and allow files that were removed days, even weeks, ago to be restored from old backup tapes.

Making a backup takes a long time and occupies a large amount of space, so doing it efficiently and conveniently is important. These considerations raise the following issues. First, should the entire file system be backed up or only part of it? At many installations, the executable (binary) programs are kept in a limited part of the file-system tree. It is not necessary to back up these files if they can all be reinstalled from the manufacturer's Website. Also, most systems have a directory for temporary files. There is usually no reason to back it up either. In UNIX, all the special files (I/O devices) are kept in a directory */dev*. Not only is backing up this directory not necessary, it is downright dangerous because the backup program would hang forever if it tried to read each of these to completion. In short, it is usually desirable to back up only specific directories and everything in them rather than the entire file system.

Second, it is wasteful to back up files that have not changed since the previous backup, which leads to the idea of **incremental dumps**. The simplest form of incremental dumping is to make a complete dump (backup) periodically, say weekly or monthly, and to make a daily dump of only those files that have been modified since the last full dump. Even better is to dump only those files that have changed since they were last dumped. While this scheme minimizes dumping time, it makes recovery more complicated, because first the most recent full dump has to be restored, followed by all the incremental dumps in reverse order. To ease recovery, more sophisticated incremental dumping schemes are often used.

Third, since immense amounts of data are typically dumped, it may be desirable to compress the data before writing them to backup storage. However, with many compression algorithms, a single bad spot on the backup storage can foil the decompression algorithm and make an entire file or even an entire backup storage unreadable. Thus the decision to compress the backup stream must be carefully considered.

Fourth, it is difficult to perform a backup on an active file system. If files and directories are being added, deleted, and modified during the dumping process, the resulting dump may be inconsistent. However, since making a dump may take hours, it may be necessary to take the system offline for much of the night to make the backup, something that is not always acceptable. For this reason, algorithms have been devised for making rapid snapshots of the file-system state by copying critical data structures, and then requiring future changes to files and directories to copy the blocks instead of updating them in place (Hutchinson et al., 1999). In this

way, the file system is effectively frozen at the moment of the snapshot, so it can be backed up at leisure afterward.

Fifth and last, making backups introduces many nontechnical problems into an organization. The best online security system in the world may be useless if the system administrator keeps all the backup disks (or tapes) in his office and leaves it open and unguarded whenever he walks down the hall to get coffee. All a spy has to do is pop in for a second, put one tiny disk or tape in his pocket, and saunter off jauntily. Goodbye security. Also, making a daily backup has little use if the fire that burns down the computers also burns up all the backup media. For this reason, the backups should be kept off-site, but that introduces more security risks because now two sites must be secured. While these practical administration issues should be taken into account in any organization, below we will discuss only the technical issues involved in making file-system backups.

Two strategies can be used for dumping a disk to a backup medium: a physical dump or a logical dump. A **physical dump** starts at block 0 of the disk, writes all the disk blocks onto the output disk in order, and stops when it has copied the last one. Such a program is so simple that it can probably be made 100% bug free, something that can probably not be said about any other useful program.

Nevertheless, it is worth making several comments about physical dumping. For one thing, there is no value in backing up unused disk blocks. If the dumping program can obtain access to the free-block data structure, it can avoid dumping unused blocks. However, skipping unused blocks requires writing the number of each block in front of the block (or the equivalent), since it is no longer true that block k on the backup was block k on the disk.

A second concern is dumping bad blocks. It is nearly impossible to manufacture large disks without any defects. Some bad blocks are always present. Sometimes when a low-level format is done, the bad blocks are detected, marked as bad, and replaced by spare blocks reserved at the end of each track for just such emergencies. In many cases, the disk controller handles bad-block replacement transparently without the operating system even knowing about it.

However, sometimes blocks go bad after formatting, in which case the operating system will eventually detect them. Usually, it solves the problem by creating a "file" consisting of all the bad blocks—just to make sure they never appear in the free-block pool and are never assigned. Needless to say, this file is completely unreadable.

If all bad blocks are remapped by the disk controller and hidden from the operating system as just described, physical dumping works fine. On the other hand, if they are visible to the operating system and maintained in one or more bad-block files or bitmaps, it is absolutely essential that the physical dumping program get access to this information and avoid dumping them to prevent endless disk read errors while trying to back up the bad-block file.

Windows systems have paging and hibernation files that are not needed in the event of a restore and should not be backed up in the first place. Specific systems

may also have other internal files that should not be backed up, so the dumping program needs to be aware of them.

The main advantages of physical dumping are simplicity and great speed (basically, it can run at the speed of the disk). The main disadvantages are the inability to skip selected directories, make incremental dumps, and restore individual files upon request. For these reasons, most installations make logical dumps.

A **logical dump** starts at one or more specified directories and recursively dumps all files and directories found there that have changed since some given base date (e.g., the last backup for an incremental dump or system installation for a full dump). Thus, in a logical dump, the dump disk gets a series of carefully identified directories and files, which makes it easy to restore a specific file or directory upon request.

Since logical dumping is the most common form, let us examine a common algorithm in detail using the example of Fig. 4-27 to guide us. Most UNIX systems use this algorithm. In the figure, we see a file tree with directories (squares) and files (circles). The shaded items have been modified since the base date and thus need to be dumped. The unshaded ones do not need to be dumped.

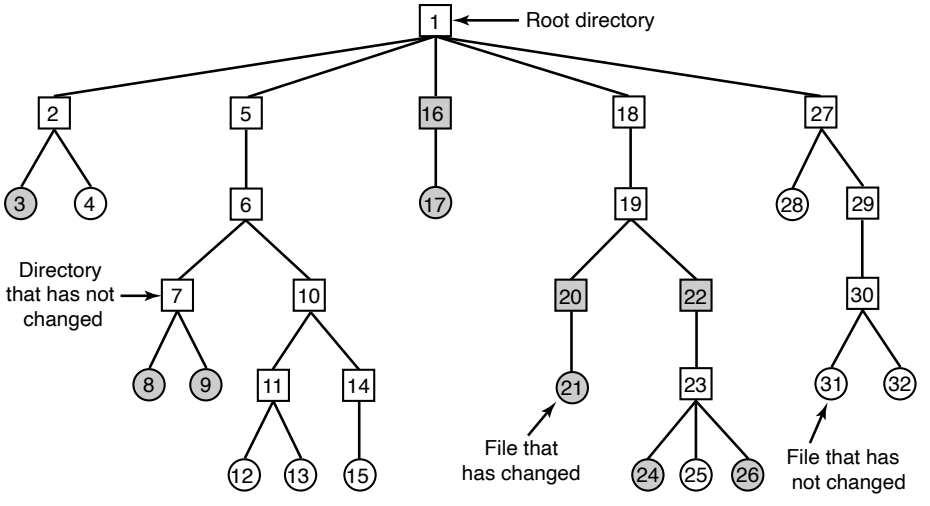

Figure 4-27. A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

This algorithm also dumps all directories (even unmodified ones) that lie on the path to a modified file or directory for two reasons. The first reason is to make it possible to restore the dumped files and directories to a fresh file system on a different computer. In this way, the dump and restore programs can be used to transport entire file systems between computers.

The second reason for dumping unmodified directories above modified files is to make it possible to incrementally restore a single file (possibly to handle recovery from user mistakes rather than system failure). Suppose that a full file-system dump is done Sunday evening and an incremental dump is done on Monday evening. On Tuesday, the directory */usr/jhs/proj/nr3* is removed, along with all the directories and files under it. On Wednesday morning bright and early, suppose the user wants to restore the file */usr/jhs/proj/nr3/plans/summary*. However, it is not possible to just restore the file *summary* because there is no place to put it. The directories *nr3* and *plans* must be restored first. To get their owners, modes, times, and whatever, correct, these directories must be present on the dump disk even though they themselves were not modified since the previous full dump.

The dump algorithm maintains a bitmap indexed by i-node number with several bits per i-node. Bits will be set and cleared in this map as the algorithm proceeds. The algorithm operates in four phases. Phase 1 begins at the starting directory (the root in this example) and examines all the entries in it. For each modified file, its i-node is marked in the bitmap. Each directory is also marked (whether or not it has been modified) and then recursively inspected.

At the end of phase 1, all modified files and all directories have been marked in the bitmap, as shown (by shading) in Fig. 4-28(a). Phase 2 conceptually recursively walks the tree again, unmarking any directories that have no modified files or directories in them or under them. This phase leaves the bitmap as shown in Fig. 4-28(b). Note that directories 10, 11, 14, 27, 29, and 30 are now unmarked because they contain nothing under them that has been modified. They will not be dumped. By way of contrast, directories 5 and 6 will be dumped even though they themselves have not been modified because they will be needed to restore today's changes to a fresh machine. For efficiency, phases 1 and 2 can be combined in one tree walk.

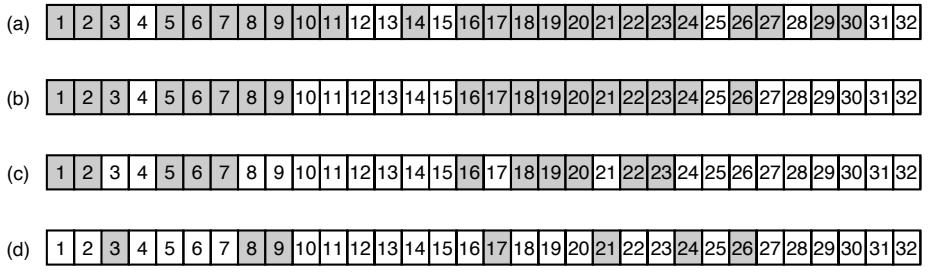

Figure 4-28. Bitmaps used by the logical dumping algorithm.

At this point, it is known which directories and files must be dumped. These are the ones that are marked in Fig. 4-28(b). Phase 3 then consists of scanning the i-nodes in numerical order and dumping all the directories that are marked for

dumping. These are shown in Fig. 4-28(c). Each directory is prefixed by the directory's attributes (owner, times, etc.) so that they can be restored. Finally, in phase 4, the files marked in Fig. 4-28(d) are also dumped, again prefixed by their attributes. This completes the dump.

Restoring a file system from the dump disk is straightforward. To start with, an empty file system is created on the disk. Then the most recent full dump is restored. Since the directories appear first on the dump disk, they are all restored first, giving a skeleton of the file system. Then the files themselves are restored. This process is then repeated with the first incremental dump made after the full dump, then the next one, and so on.

Although logical dumping is straightforward, there are a few tricky issues. For one, since the free block list is not a file, it is not dumped and hence it must be reconstructed from scratch after all the dumps have been restored. Doing so is always possible since the set of free blocks is just the complement of the set of blocks contained in all the files combined.

Another issue is links. If a file is linked to two or more directories, it is important that the file is restored only one time and that all the directories that are supposed to point to it do so.

Still another issue is the fact that UNIX files may contain holes. It is permitted to open a file, write a few bytes, then seek to a distant file offset and write a few more bytes. The blocks in between are not part of the file and should not be dumped and must not be restored. Core dump files often have a hole of hundreds of megabytes between the data segment and the stack. If not handled properly, each restored core file will fill this area with zeros and thus be the same size as the virtual address space (e.g., 2^{32} bytes, or worse yet, 2^{64} bytes).

Finally, special files, named pipes, and the like (anything that is not a real file) should never be dumped, no matter in which directory they may occur (they need not be confined to */dev*). For more information about file-system backups, see Zwicky (1991) and Chervenak et al., (1998).

4.4.3 File-System Consistency

Another area where reliability is an issue is file-system consistency. Many file systems read blocks, modify them, and write them out later. If the system crashes before all the modified blocks have been written out, the file system can be left in an inconsistent state. This problem is especially critical if some of the blocks that have not been written out are i-node blocks, directory blocks, or blocks containing the free list.

To deal with inconsistent file systems, most computers have a utility program that checks file-system consistency. For example, UNIX has *fsck*; Windows has *sfc* (and others). This utility can be run whenever the system is booted, especially after a crash. The description below tells how *fsck* works. *Sfc* is somewhat different because it works on a different file system, but the general principle of using

the file system's inherent redundancy to repair it is still a valid one. All file-system checkers verify each file system (disk partition) independently of the other ones. It is also important to note that some file systems, such the journaling file systems discussed earlier, are designed such that they do not require administrators to run a separate file system consistency checker after a crash, because they can handle most inconsistencies themselves.

Two kinds of consistency checks can be made: blocks and files. To check for block consistency, the program builds two tables, each one containing a counter for each block, initially set to 0. The counters in the first table keep track of how many times each block is present in a file; the counters in the second table record how often each block is present in the free list (or the bitmap of free blocks).

The program then reads all the i-nodes using a raw device, which ignores the file structure and just returns all the disk blocks starting at 0. Starting from an i-node, it is possible to build a list of all the block numbers used in the corresponding file. As each block number is read, its counter in the first table is incremented. The program then examines the free list or bitmap to find all the blocks that are not in use. Each occurrence of a block in the free list results in its counter in the second table being incremented.

If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 4-29(a). However, as a result of a crash, the tables might look like Fig. 4-29(b), in which block 2 does not occur in either table. It will be reported as being a **missing block**. While missing blocks do no real harm, they waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

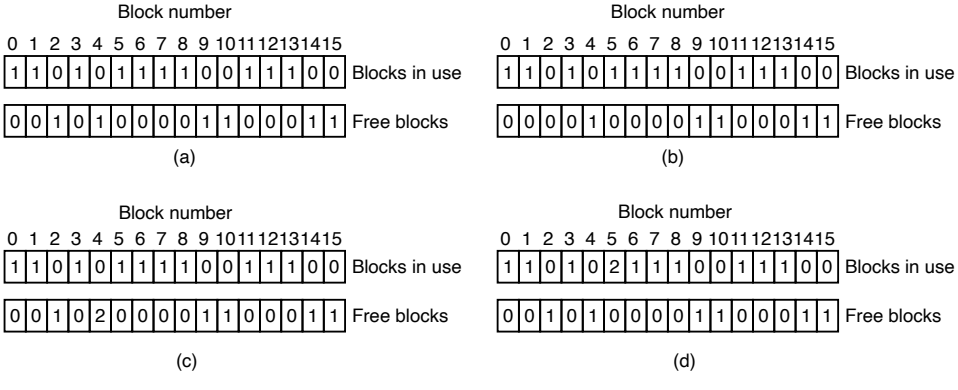

Figure 4-29. File-system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

Another situation that might occur is that of Fig. 4-29(c). Here we see a block, number 4, that occurs twice in the free list. (Duplicates can occur only if the free

list is really a list; with a bitmap it is impossible.) The solution here is also simple: rebuild the free list.

The worst thing that can happen is that the same data block is present in two or more files, as shown in Fig. 4-29(d) with block 5. If either of these files is removed, block 5 will be put on the free list, leading to a situation in which the same block is both in use and free at the same time. If both files are removed, the block will be put onto the free list twice.

The appropriate action for the file-system checker to take is to allocate a free block, copy the contents of block 5 into it, and insert the copy into one of the files. In this way, the information content of the files is unchanged (although almost assuredly one is garbled), but the file-system structure is at least made consistent. The error should be reported, to allow the user to inspect the damage.

In addition to checking to see that each block is properly accounted for, the file-system checker also checks the directory system. It, too, uses a table of counters, but these are per file, rather than per block. It starts at the root directory and recursively descends the tree, inspecting each directory in the file system. For every i-node in every directory, it increments a counter for that file's usage count. Remember that due to hard links, a file may appear in two or more directories. Symbolic links do not count and do not cause the counter for the target file to be incremented.

When the checker is all done, it has a list, indexed by i-node number, telling how many directories contain each file. It then compares these numbers with the link counts stored in the i-nodes themselves. These counts start at 1 when a file is created and are incremented each time a (hard) link is made to the file. In a consistent file system, both counts will agree. However, two kinds of errors can occur: the link count in the i-node can be too high or it can be too low.

If the link count is higher than the number of directory entries, then even if all the files are removed from the directories, the count will still be nonzero and the i-node will not be removed. This error is not serious, but it wastes space on the disk with files that are not in any directory. It should be fixed by setting the link count in the i-node to the correct value.

The other error is potentially catastrophic. If two directory entries are linked to a file, but the i-node says that there is only one, when either directory entry is removed, the i-node count will go to zero. When an i-node count goes to zero, the file system marks it as unused and releases all of its blocks. This action will result in one of the directories now pointing to an unused i-node, whose blocks may soon be assigned to other files. Again, the solution is just to force the link count in the i-node to the actual number of directory entries.

These two operations, checking blocks and checking directories, are often integrated for efficiency reasons (i.e., only one pass over the i-nodes is required). Other checks are also possible. For example, directories have a definite format, with i-node numbers and ASCII names. If an i-node number is larger than the number of i-nodes on the disk, the directory has been damaged.

Furthermore, each i-node has a mode, some of which are legal but strange, such as 007, which allows the owner and his group no access at all, but allows outsiders to read, write, and execute the file. It might be useful to at least report files that give outsiders more rights than the owner. Directories with more than, say, 1000 entries are also suspicious. Files located in user directories, but which are owned by the superuser and have the SETUID bit on, are potential security problems because such files acquire the powers of the superuser when executed by any user. With a little effort, one can put together a fairly long list of technically legal but still peculiar situations that might be worth reporting.

The previous paragraphs have discussed the problem of protecting the user against crashes. Some file systems also worry about protecting the user against himself. If the user intends to type

```
rm *.o
```

to remove all the files ending with *.o* (compiler-generated object files), but accidentally types

```
rm * .o
```

(note the space after the asterisk), *rm* will remove all the files in the current directory and then complain that it cannot find *.o*. This is a catastrophic error from which recovery is virtually impossible without heroic efforts and special software. In Windows, files that are removed are placed in the recycle bin (a special directory), from which they can later be retrieved if need be. Of course, no storage is reclaimed until they are actually deleted from this directory.

4.4.4 File-System Performance

Access to hard disk is much slower than access to flash storage and much slower still than access to memory. Reading a 32-bit memory word might take 10 nsec. Reading from a hard disk might proceed at 100 MB/sec, which is four times slower per 32-bit word, but to this must be added 5–10 msec to seek to the track and then wait for the desired sector to arrive under the read head. If only a single word is needed, the memory access is on the order of a million times as fast as disk access. As a result of this difference in access time, many file systems have been designed with various optimizations to improve performance. In this section, we will cover three of them.

Caching

The most common technique used to reduce disk accesses is the **block cache** or **buffer cache**. (Cache is pronounced "cash" and is derived from the French *cacher*, meaning to hide.) In this context, a cache is a collection of blocks that logically belong on the disk but are being kept in memory for performance reasons.

Various algorithms can be used to manage the cache, but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache.

Operation of the cache is illustrated in Fig. 4-30. Since there are many (often thousands of) blocks in the cache, some way is needed to determine quickly if a given block is present. The usual way is to hash the device and disk address and look up the result in a hash table. All the blocks with the same hash value are chained together on a linked list so that the collision chain can be followed.

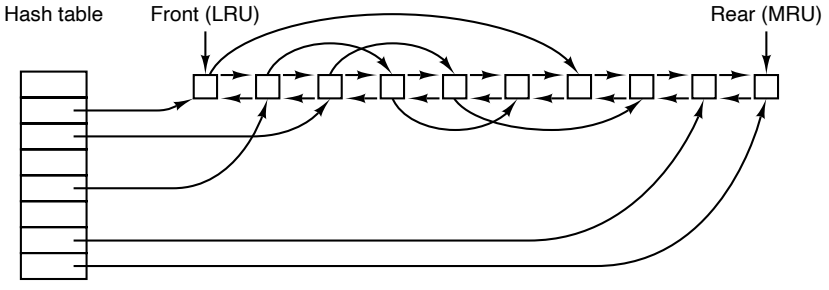

Figure 4-30. The buffer cache data structures.

When a block has to be loaded into a full cache, some block has to be removed (and rewritten to the disk if it has been modified since being brought in). This situation is very much like paging, and all the usual page-replacement algorithms described in Chap. 3, such as FIFO, second chance, and LRU, are applicable. One pleasant difference between paging and caching is that cache references are relatively infrequent, so that it is feasible to keep all the blocks in exact LRU order with linked lists.

In Fig. 4-30, we see that in addition to the collision chains starting at the hash table, there is also a bidirectional list running through all the blocks in the order of usage, with the least recently used block on the front of this list and the most recently used block at the end. When a block is referenced, it can be removed from its position on the bidirectional list and put at the end. In this way, exact LRU order can be maintained.

Unfortunately, there is a catch. Now that we have a situation in which exact LRU is possible, it turns out that LRU is undesirable. The problem has to do with the crashes and file-system consistency discussed in the previous section. If a critical block, such as an i-node block, is read into the cache and modified, but not rewritten to the disk, a crash will leave the file system in an inconsistent state. If the i-node block is put at the end of the LRU chain, it may be quite a while before it reaches the front and is rewritten to the disk.

Furthermore, some blocks, such as i-node blocks, are rarely referenced two times within a short interval. These considerations lead to a modified LRU scheme, taking two factors into account:

1. Is the block likely to be needed again soon?
2. Is the block essential to the consistency of the file system?

For both questions, blocks can be divided into categories such as i-node blocks, indirect blocks, directory blocks, full data blocks, and partially full data blocks. Blocks that will probably not be needed again soon go on the front, rather than the rear of the LRU list, so their buffers will be reused quickly. Blocks that might be needed again soon, such as a partly full block that is being written, go on the end of the list, so they will stay around for a long time.

The second question is independent of the first one. If the block is essential to the file-system consistency (basically, everything except data blocks), and it has been modified, it should be written to disk immediately, regardless of which end of the LRU list it is put on. By writing critical blocks quickly, we greatly reduce the probability that a crash will wreck the file system. While a user may be unhappy if one of his files is ruined in a crash, he is likely to be far more unhappy if the whole file system is lost.

Even with this measure to keep the file-system integrity intact, it is undesirable to keep data blocks in the cache too long before writing them out. Consider the plight of someone who is using a personal computer to write a book. Even if our writer periodically tells the editor to write the file being edited to the disk, there is a good chance that everything will still be in the cache and nothing on the disk. If the system crashes, the file-system structure will not be corrupted, but a whole day's work will be lost.

This situation need not happen often before we have a fairly unhappy user. Systems take two approaches to dealing with it. The UNIX way is to have a system call, sync, which forces all the modified blocks out onto the disk immediately. When the system is started up, a program, usually called *update*, is started up in the background to sit in an endless loop issuing sync calls, sleeping for 30 sec between calls. As a result, no more than 30 seconds of work is lost due to a crash.

Although Windows now has a system call equivalent to sync, called FlushFileBuffers, in the past it did not. Instead, it had a different strategy that was in some ways better than the UNIX approach (and in some ways worse). What it did was to write every modified block to disk as soon as it was written to the cache. Caches in which all modified blocks are written back to the disk immediately are called **write-through caches**. They require more disk I/O than nonwrite-through caches.

The difference between these two approaches can be seen when a program writes a 1-KB block full, one character at a time. UNIX will collect all the characters in the cache and write the block out once every 30 seconds, or whenever the block is removed from the cache. With a write-through cache, there is a disk access

for every character written. Of course, most programs do internal buffering, so they normally write not a character, but a line or a larger unit on each write system call.

A consequence of this difference in caching strategy is that just removing a disk from a UNIX system without doing a sync will almost always result in lost data, and frequently in a corrupted file system as well. With write-through caching, no problem arises. These differing strategies were chosen because UNIX was developed in an environment in which all disks were hard disks and not removable, whereas the first Windows file system was inherited from MS-DOS, which started out in the floppy-disk world. As hard disks became the norm, the UNIX approach, with its better efficiency (but worse reliability), became the norm, and it is also used now on Windows for hard disks. However, NTFS takes other measures (e.g., journaling) to improve reliability, as discussed earlier.

At this point, it is worth discussing the relationship between the buffer cache and the **page cache**. Conceptually they are different in that a page cache caches pages of files to optimize file I/O, while a buffer cache simply caches disk blocks. The buffer cache, which predates the page cache, really behaves like disk, except that the reads and writes access memory. The reason people added a page cache was that it seemed a good idea to move the cache higher up in the stack, so file requests could be served without going through the file system code and all its complexities. Phrased differently: files are in the page cache and disk blocks in the buffer cache. In addition, a cache at a higher level without need for the file system made it easier to integrate it with the memory management subsystem—as befitting a component called *page cache*. However, it has probably not escaped your notice that the files in the page cache are typically on disk also, so that their data are now in both of the caches.

Some operating systems therefore integrate the buffer cache with the page cache. This is especially attractive when memory-mapped files are supported. If a file is mapped onto memory, then some of its pages may be in memory because they were demand paged in. Such pages are hardly different from file blocks in the buffer cache. In this case, they can be treated the same way, with a single cache for both file blocks and pages. Even if the functions are still distinct, they point to the same data. For instance, as most data has both a file and a block representation, the buffer cache simply point into the page cache—leaving only one instance of the data cached in memory.

Block Read Ahead

A second technique for improving perceived file-system performance is to try to get blocks into the cache before they are needed to increase the hit rate. In particular, many files are read sequentially. When the file system is asked to produce block k in a file, it does that, but when it is finished, it makes a sneaky check in the cache to see if block $k + 1$ is already there. If it is not, it schedules a read for block

$k + 1$ in the hope that when it is needed, it will have already arrived in the cache. At the very least, it will be on the way.

Of course, this read-ahead strategy works only for files that are actually being read sequentially. If a file is being randomly accessed, read ahead does not help. In fact, it hurts by tying up disk bandwidth reading in useless blocks and removing potentially useful blocks from the cache (and possibly tying up more disk bandwidth writing them back to disk if they are dirty). To see whether read ahead is worth doing, the file system can keep track of the access patterns to each open file. For example, a bit associated with each file can keep track of whether the file is in "sequential-access mode" or "random-access mode." Initially, the file is given the benefit of the doubt and put in sequential-access mode. However, whenever a seek is done, the bit is cleared. If sequential reads start happening again, the bit is set once again. In this way, the file system can make a reasonable guess about whether it should read ahead or not. If it gets it wrong once in a while, it is not a disaster, just a little bit of wasted disk bandwidth.

Reducing Disk-Arm Motion

Caching and read ahead are not the only ways to increase file-system performance. Another important technique for hard disks is to reduce the amount of disk-arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder. When an output file is written, the file system has to allocate the blocks one at a time, on demand. If the free blocks are recorded in a bitmap, and the whole bitmap is in main memory, it is easy enough to choose a free block as close as possible to the previous block. With a free list, part of which is on disk, it is much harder to allocate blocks close together.

However, even with a free list, some block clustering can be done. The trick is to keep track of disk storage not in blocks, but in groups of consecutive blocks. If all sectors consist of 512 bytes, the system could use 1-KB blocks (2 sectors) but allocate disk storage in units of 2 blocks (4 sectors). This is not the same as having 2-KB disk blocks, since the cache would still use 1-KB blocks and disk transfers would still be 1 KB, but reading a file sequentially on an otherwise idle system would reduce the number of seeks by a factor of two, considerably improving performance. A variation on the same theme is to take account of rotational positioning. When allocating blocks, the system attempts to place consecutive blocks in a file in the same cylinder.

Another performance bottleneck in systems that use i-nodes or anything like them is that reading even a short file requires two disk accesses: one for the i-node and one for the block. In many file systems, the i-node placement is like the one shown in Fig. 4-31(a). Here all the i-nodes are near the start of the disk, so the average distance between an i-node and its blocks will be half the number of cylinders, requiring long seeks. This is clearly inefficient and needs to be improved.

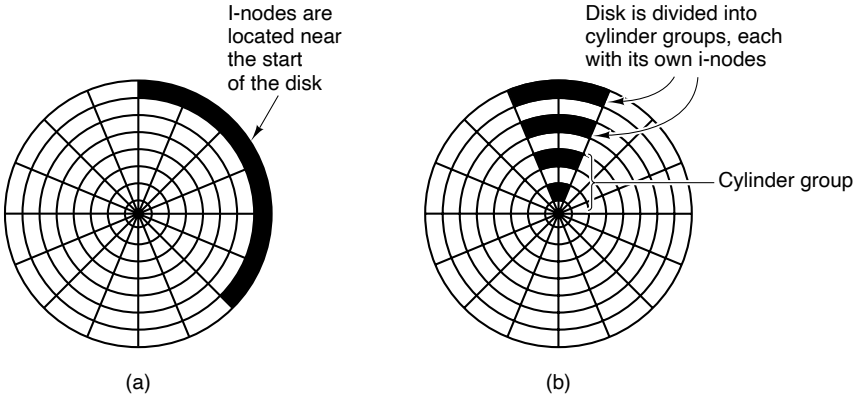

Figure 4-31. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

One easy performance improvement is to put the i-nodes in the middle of the disk, rather than at the start, thus reducing the average seek between the i-node and the first block by a factor of two. Another idea, shown in Fig. 4-31(b), is to divide the disk into cylinder groups, each with its own i-nodes, blocks, and free list (McKusick et al., 1984). When creating a new file, any i-node can be chosen, but an attempt is made to find a block in the same cylinder group as the i-node. If none is available, then a block in a nearby cylinder group is used.

Of course, disk-arm movement and rotation time are relevant only if the disk has them and are not relevant for SSDs, which have no moving parts whatsoever. For these drives, built on the same technology as flash cards, random (read) accesses are just as fast as sequential ones and many of the problems of traditional disks go away (only for new ones emerge).

4.4.5 Defragmenting Disks

When the operating system is initially installed, the programs and files it needs are installed consecutively starting at the beginning of the disk, each one directly following the previous one. All free disk space is in a single contiguous unit following the installed files. However, as time goes on, files are created and removed and typically the disk becomes badly fragmented, with files and holes all over the place. As a consequence, when a new file is created, the blocks used for it may be spread all over the disk, giving poor performance.

The performance can be restored by moving files around to make them contiguous and to put all (or at least most) of the free space in one or more large contiguous regions on the disk. Windows has a program, *defrag*, that does precisely this. Windows users should run it regularly, except on SSDs.

Defragmentation works better on file systems that have a lot of free space in a contiguous region at the end of the partition. This space allows the defragmentation program to select fragmented files near the start of the partition and copy all their blocks to the free space. Doing so frees up a contiguous block of space near the start of the partition into which the original or other files can be placed contiguously. The process can then be repeated with the next chunk of disk space, etc.

Some files cannot be moved, including the paging file, the hibernation file, and the journaling log, because the administration that would be required to do this is more trouble than it is worth. In some systems, these are fixed-size contiguous areas anyway, so they do not have to be defragmented. The one time when their lack of mobility is a problem is when they happen to be near the end of the partition and the user wants to reduce the partition size. The only way to solve this problem is to remove them altogether, resize the partition, and then recreate them afterward.

Linux file systems (especially ext3 and ext4) generally suffer less from defragmentation than Windows systems due to the way disk blocks are selected, so manual defragmentation is rarely required. Also, SSDs do not suffer from fragmentation at all. In fact, defragmenting an SSD is counterproductive. Not only is there no gain in performance, but SSDs wear out, so defragmenting them merely shortens their lifetimes.

4.4.6 Compression and Deduplication

In the "Age of Data," people tend to have, well, a lot of data. All these data must find a home on a storage device and often that home fills up quickly with cat pictures, cat videos, and other essential information. Of course, we can always buy a new and bigger SSD, but it would be nice if we could prevent it from filling up quite so quickly.

The simplest technique to use scarce storage space more efficiently is **compression**. Besides manually compressing files or folders, we can use a file system that compresses specific folders or even all data automatically. File systems such as NTFS (on Windows), Btrfs (Linux), and ZFS (on a variety of operating systems) all offer compression as an option. The compression algorithms commonly look for repeating sequences of data which they then encode efficiently. For instance, when writing file data they may discover that the 133 bytes at offset 1737 in the file are the same as the 133 bytes at offset 1500, so instead of writing the same bytes again, they insert a marker (237,133)—indicating that these 133 bytes can be found at a distance of 237 before the current offset.

Besides eliminating redundancy within a single file, several popular file systems also remove redundancy across files. On systems that store data from many users, for instance in a cloud or server environment, it is common to find files that contain the same data, as multiple users store the same documents, binaries, or videos. Such data duplication is even more pronounced in backup storage. If users

back up all their important files every week, each new backup probably contains (mostly) the same data.

Rather than storing the same data multiple times, several file systems implement **deduplication** to eliminate duplicate copies—exactly like the deduplication of pages in the memory subsystem that we discussed in the previous chapter. This is a very common phenomenon in operating systems: a technique (in this case deduplication) that is a good idea in one subsystem, is often a good idea in other subsystems also. Here we discuss deduplication in file systems, but the technique is also used in networking to prevent the same data from being sent over the network multiple times.

File system deduplication is possible at the granularity of files, portions of files, or even individual disk blocks. Nowadays, many file systems perform deduplication on fixed-size chunks of, say, 128 KB. When the deduplication procedure detects that two files contain chunks that are exactly the same, it will keep only a single physical copy that is shared by both files. Of course, as soon as the chunk in one of the files is overwritten, a unique copy must be made so that the changes do not affect the other file.

Deduplication can be done *inline* or *post-process*. With inline deduplication, the file system calculates a hash for every chunk that it is about to write and compares it to the hashes of existing chunks. If the chunk is already present, it will refrain from actually writing out the data and instead add a reference to the existing chunk. Of course, the additional calculations take time and slow down the write. In contrast, post-process deduplication always writes out the data and performs the hashing and comparisons in the background, without slowing down the process' file operations. Which method is better is debated almost as hotly as which editor is best, Emacs or Vi (even though the answer to that question is, of course, Emacs).

As the astute reader may have noticed, there is a problem with the use of hashes to determine chunk equivalence: even if it happens rarely, the pigeonhole principle says that chunks with different content *may* have the same hash. Some implementations of deduplication gloss over this little inconvenience and accept the (very low) probability of getting things wrong, but there also exist solutions that verify whether the chunks are truly equivalent before deduplicating them.

4.4.7 Secure File Deletion and Disk Encryption

However sophisticated the access restrictions at the level of the operating system, the physical bits on the hard disk or SSDs can always be read back by taking out the storage device and reading them back in another machine. This has many implications. For instance, the operating system may "delete" a file by removing it from the directories and freeing up the i-node for reuse, but that does not remove the content of the file on disk. Thus, an attacker can simply read the raw disk blocks to bypass all file system permissions, no matter how restrictive they are.

In fact, securely deleting data on disk is not easy. If the disk is old and not needed any more but the data must not fall into the wrong hands under any conditions, the best approach is to get a large flowerpot. Put in some thermite, put the disk in, and cover it with more thermite. Then light it and watch it burn nicely at 2500°C. Recovery will be impossible, even for a pro. If you are unfamiliar with the properties of thermite, it is strongly recommended that you do not try this at home.

However, if you want to reuse the disk, this technique is clearly not appropriate. Even if you overwrite the original content with zeros, it may not be enough. On some hard disks, data stored on the disk leave magnetic traces in areas close to the actual tracks. So even if the normal content in the tracks is zeroed out, a highly motivated and sophisticated attacker (such as a government intelligence agency) could still recover the original content by carefully inspecting the adjacent areas. In addition, there may be copies of the file in unexpected places on the disk (for instance, as a backup or in a cache), and these need to be wiped also. SSDs have even worse problems, as the file system has no control over what flash blocks are overwritten and when, since this is determined by the FTL. Usually by overwriting a disk with three to seven passes, alternating zeros and random numbers, will securely erase it though. There is software available to do this.

One way to make it impossible to recover data from disk, deleted or not, is by encrypting everything that is on the disk. Full disk encryption is available on all modern operating systems. As long as you do not write the password on a Post-It note stuck somewhere on your computer, full disk encryption with a powerful encryption algorithm will keep your data safe even if the disk falls in the hands of the baddies.

Full disk encryption is sometimes also provided by the storage devices themselves in the form of Self-Encrypting Drives (SEDs) with onboard cryptographic capabilities to do the encryption and decryption, leading to a performance boost as the cryptographic calculations are offloaded from the CPU. Unfortunately, researchers found that many SEDs have critical security weaknesses due to specification, design, and implementation issues (Meijer and Van Gastel, 2019).

As an example of full disk encryption, Windows makes use of the capabilities of such SEDs if they are present. If not, it takes care of the encryption itself, using a secret key, the *volume master key*, in a standard encryption algorithm called Advanced Encryption Standard (AES). Full disk encryption on Windows was designed to be as unobtrusive as possible and many users are blissfully unaware that their data are encrypted on disk. The volume master key used to encrypt or decrypt the data on regular (i.e., non SED) storage devices can be obtained by decrypting the (itself encrypted) key either with the user password or with the recovery key (that was automatically generated the first time the file system was encrypted), or by extracting the key from a special-purpose cryptoprocessor known as the Trusted Platform Module, or TPM. Either way, once it has the key, Windows can encrypt or decrypt the disk data as required.

4.5 EXAMPLE FILE SYSTEMS

In the following sections, we will discuss several example file systems, ranging from quite simple to more sophisticated. Since modern UNIX file systems and Windows's native file system are covered in the chapter on UNIX (Chap. 10) and the chapter on Windows (Chap. 11), we will not cover those systems here. We will, however, examine their predecessors below. As we have mentioned before, variants of the MS-DOS file system are still in use in digital cameras, portable music players, electronic picture frames, USB sticks, and other devices, so studying them is definitely still relevant.

4.5.1 The MS-DOS File System

The MS-DOS file system is the one the first IBM PCs came with. It was the main file system up through Windows 98 and Windows ME. It is still supported on Windows 10 and Windows 11. However, it and an extension of it (FAT-32) have become widely used for many embedded systems. Most digital cameras use it. Many MP3 players use it exclusively. Electronic picture frames use it. Some memory cards use it. Many other simple devices that store music, images, and so on still use it. It is still the preferred file system for disks and other devices that need to be read by both Windows and MacOS. Thus, the number of electronic devices using the MS-DOS file system is vastly larger now than at any time in the past, and certainly much larger than the number using the more modern NTFS file system. For that reason alone, it is worth looking at in some detail.

To read a file, an MS-DOS program must first make an open system call to get a handle for it. The open system call specifies a path, which may be either absolute or relative to the current working directory. The path is looked up component by component until the final directory is located and read into memory. It is then searched for the file to be opened.

Although MS-DOS directories are variable sized, they use a fixed-size 32-byte directory entry. The format of an MS-DOS directory entry is shown in Fig. 4-32. It contains the file name, attributes, creation date and time, starting block, and exact file size. File names shorter than 8 + 3 characters are left justified and padded with spaces on the right, in each field separately. The *Attributes* field is new and contains bits to indicate that a file is read-only, needs to be archived, is hidden, or is a system file. Read-only files cannot be written. This is to protect them from accidental damage. The archived bit has no actual operating system function (i.e., MS-DOS does not examine or set it). The intention is to allow user-level archive programs to clear it upon archiving a file and to have other programs set it when modifying a file. In this way, a backup program can just examine this attribute bit on every file to see which files to back up. The hidden bit can be set to prevent a file from appearing in directory listings. Its main use is to avoid confusing novice users with files they might not understand. Finally, the system bit also hides files. In

addition, system files cannot accidentally be deleted using the *del* command. The main components of MS-DOS have this bit set.

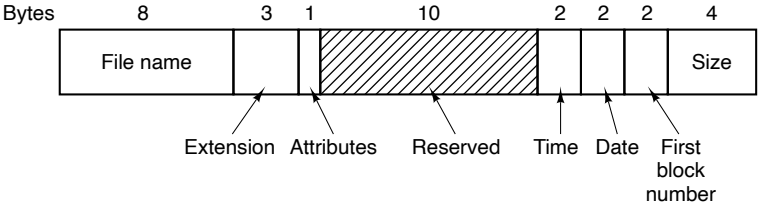

Figure 4-32. The MS-DOS directory entry.

The directory entry also contains the date and time the file was created or last modified. The time is accurate only to ±2 sec because it is stored in a 2-byte field, which can store only 65,536 unique values (a day contains 86,400 seconds). The time field is subdivided into seconds (5 bits), minutes (6 bits), and hours (5 bits). The date counts in days using three subfields: day (5 bits), month (4 bits), and year – 1980 (7 bits). With a 7-bit number for the year and time beginning in 1980, the highest expressible year is 2107. Thus. MS-DOS has a built-in Y2108 problem. To avoid catastrophe, MS-DOS users should begin with Y2108 compliance as early as possible. If MS-DOS had used the combined date and time fields as a 32-bit seconds counter, it could have represented every second exactly and delayed the catastrophe until 2116.

MS-DOS stores the file size as a 32-bit number, so in theory files can be as large as 4 GB. However, other limits (described below) restrict the maximum file size to 2 GB or less. A surprisingly large part of the entry (10 bytes) is unused.

MS-DOS keeps track of file blocks via a file allocation table in main memory. The directory entry contains the number of the first file block. This number is used as an index into a 64K entry FAT in main memory. By following the chain, all the blocks can be found. The operation of the FAT is illustrated in Fig. 4-14.

The FAT file system comes in three versions: FAT-12, FAT-16, and FAT-32, depending on how many bits a disk address contains. Actually, FAT-32 is something of a misnomer, since only the low-order 28 bits of the disk addresses are used. It should have been called FAT-28, but powers of two sound so much neater.

Another variant of the FAT file system is exFAT, which Microsoft introduced for large removable devices. Apple licensed exFAT, so that there is one modern file system that can be used to transfer files both ways between Windows and MacOS computers. Since exFAT is proprietary and Microsoft has not released the specification, we will not discuss it further here.

For all FATs, the disk block can be set to some multiple of 512 bytes (possibly different for each partition), with the set of allowed block sizes (called **cluster sizes** by Microsoft) being different for each variant. The first version of MS-DOS used FAT-12 with 512-byte blocks, giving a maximum partition size of $2^{12} \times 512$

bytes (actually only 4086 × 512 bytes because 10 of the disk addresses were used as special markers, such as end of file, bad block, etc.). With these parameters, the maximum disk partition size was about 2 MB and the size of the FAT table in memory was 4096 entries of 2 bytes each. Using a 12-bit table entry would have been too slow.

This system worked well for floppy disks, but when hard disks came out, it became a problem. Microsoft solved the problem by allowing additional block sizes of 1 KB, 2 KB, and 4 KB. This change preserved the structure and size of the FAT-12 table, but allowed disk partitions of up to 16 MB.

Since MS-DOS supported four disk partitions per disk drive, the new FAT-12 file system worked up to 64-MB disks. Beyond that, something had to give. What happened was the introduction of FAT-16, with 16-bit disk pointers. Additionally, block sizes of 8 KB, 16 KB, and 32 KB were permitted. (32,768 is the largest power of two that can be represented in 16 bits.) The FAT-16 table now occupied 128 KB of main memory all the time, but with the larger memories by then available, it was widely used and rapidly replaced the FAT-12 file system. The largest disk partition that can be supported by FAT-16 is 2 GB (64K entries of 32 KB each) and the largest disk, 8 GB, namely four partitions of 2 GB each. For quite a while, that was good enough.

But not forever. For business letters, this limit is not a problem, but for storing digital video using the DV standard, a 2-GB file holds just over 9 minutes of video. As a consequence of the fact that a PC disk can support only four partitions, the largest video that can be stored on a disk is about 38 minutes, no matter how large the disk is. This limit also means that the largest video that can be edited on line is less than 19 minutes, since both input and output files are needed.

Starting with the second release of Windows 95, the FAT-32 file system, with its 28-bit disk addresses, was introduced and the version of MS-DOS underlying Windows 95 was adapted to support FAT-32. In this system, partitions could theoretically be $2^{28} \times 2^{15}$ bytes, but they are actually limited to 2 TB (2048 GB) because internally the system keeps track of partition sizes in 512-byte sectors using a 32-bit number, and $2^9 \times 2^{32}$ is 2 TB. The maximum partition size for various block sizes and all three FAT types is shown in Fig. 4-33.

In addition to supporting larger disks, the FAT-32 file system has two other advantages over FAT-16. First, an 8-GB disk using FAT-32 can be a single partition. Using FAT-16 it has to be four partitions, which appears to the Windows user as the *C:*, *D:*, *E:*, and *F:* logical disk drives. It is up to the user to decide which file to place on which drive and keep track of what is where.

The other advantage of FAT-32 over FAT-16 is that for a given size disk partition, a smaller block size can be used. For example, for a 2-GB disk partition, FAT-16 must use 32-KB blocks; otherwise with only 64K available disk addresses, it cannot cover the whole partition. In contrast, FAT-32 can use, for example, 4-KB blocks for a 2-GB disk partition. The advantage of the smaller block size is that most files are much shorter than 32 KB. If the block size is 32 KB, a file of 10

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Figure 4-33. Maximum partition size for different block sizes. The empty boxes represent forbidden combinations.

bytes ties up 32 KB of disk space. If the average file is, say, 8 KB, then with a 32-KB block, three quarters of the disk will be wasted, not a terribly efficient way to use the disk. With an 8-KB file and a 4-KB block, there is no disk wastage, but the price paid is more RAM eaten up by the FAT. With a 4-KB block and a 2-GB disk partition, there are 512K blocks, so the FAT must have 512K entries in memory (occupying 2 MB of RAM).

MS-DOS uses the FAT to keep track of free disk blocks. Any block that is not currently allocated is marked with a special code. When MS-DOS needs a new disk block, it searches the FAT for an entry containing this code. Thus no bitmap or free list is required.

4.5.2 The UNIX V7 File System

Even early versions of UNIX had a fairly sophisticated multiuser file system since it was derived from MULTICS. Below we will discuss the V7 file system, the one for the PDP-11 that made UNIX famous. We will examine a modern UNIX file system in the context of Linux in Chap. 10.

The file system is in the form of a tree starting at the root directory, with the addition of links, forming a directed acyclic graph. File names can be up to 14 characters and can contain any ASCII characters except / (because that is the separator between components in a path) and NUL (because that is used to pad out names shorter than 14 characters). NUL has the numerical value of 0.

A UNIX directory entry contains one entry for each file in that directory. Each entry is extremely simple because UNIX uses the i-node scheme illustrated in Fig. 4-15. A directory entry contains only two fields: the file name (14 bytes) and the number of the i-node for that file (2 bytes), as shown in Fig. 4-34. These parameters limit the number of files per file system to 64K.

Like the i-node of Fig. 4-15, the UNIX i-node contains some attributes. The attributes contain the file size, three times (creation, last access, and last modification), owner, group, protection information, and a count of the number of directory

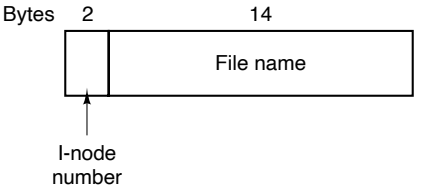

Figure 4-34. A UNIX V7 directory entry.

entries that point to the i-node. The latter field is needed due to links. Whenever a new link is made to an i-node, the count in the i-node is increased. When a link is removed, the count is decremented. When it gets to 0, the i-node is reclaimed and the disk blocks are put back in the free list.

Keeping track of disk blocks is done using a generalization of Fig. 4-15 in order to handle very large files. The first 10 disk addresses are stored in the i-node itself, so for small files, all the necessary information is right in the i-node, which is fetched from disk to main memory when the file is opened. For somewhat larger files, one of the addresses in the i-node is the address of a disk block called a **single indirect block**. This block contains additional disk addresses. If this still is not enough, another address in the i-node, called a **double indirect block**, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a **triple indirect block** can also be used. The complete picture is given in Fig. 4-35.

When a file is opened, the file system must take the file name supplied and locate its disk blocks. Let us consider how the path name */usr/ast/mbox* is looked up. We will use UNIX as an example, but the algorithm is basically the same for all hierarchical directory systems. First the file system locates the root directory. In UNIX its i-node is located at a fixed place on the disk. From this i-node, it locates the root directory, which can be anywhere on the disk, but say block 1.

After that it reads the root directory and looks up the first component of the path, *usr*, in the root directory to find the i-node number of the file */usr*. Locating an i-node from its number is straightforward, since each one has a fixed location on the disk. From this i-node, the system locates the directory for */usr* and looks up the next component, *ast*, in it. When it has found the entry for *ast*, it has the i-node for the directory */usr/ast*. From this i-node it can find the directory itself and look up *mbox*. The i-node for this file is then read into memory and kept there until the file is closed. The lookup process is illustrated in Fig. 4-36.

Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of from the root directory. Every directory has entries for . and .. which are put there when the directory is created. The entry . has the i-node number for the current directory, and the entry for .. has the i-node

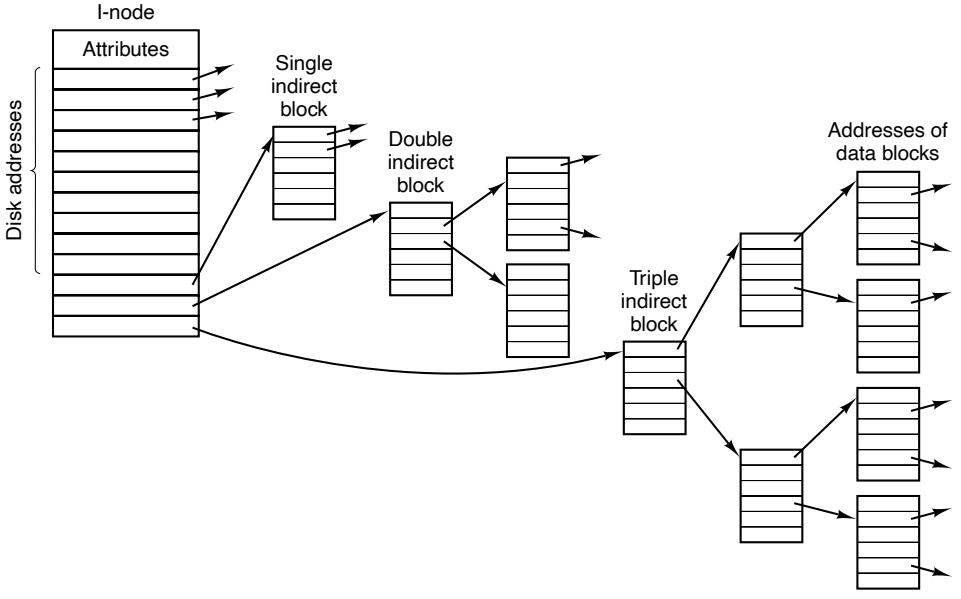

Figure 4-35. A UNIX i-node.

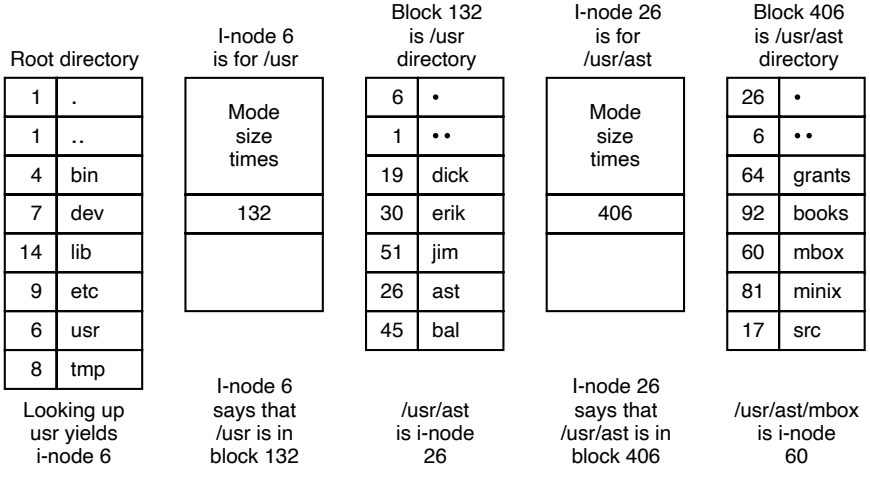

Figure 4-36. The steps in looking up */usr/ast/mbox*.

number for the parent directory. Thus, a procedure looking up *../chris/prog.c* simply looks up .. in the working directory, finds the i-node number for the parent directory, and searches that directory for *chris*. No special mechanism is needed to handle these names. As far as the directory system is concerned, they are just ordinary ASCII strings, just the same as any other names. The only bit of trickery here is that .. in the root directory points to itself.

4.6 RESEARCH ON FILE SYSTEMS

File systems have always attracted more research than other parts of the operating system and that is still the case. Entire conferences such as FAST, MSST, and NAS are devoted largely to file and storage systems.

A considerable amount of research addresses the reliability of storage and file systems. A powerful way to guarantee reliability is to formally *prove* the safety of your system even in the face of catastrophic events, such as crashes (Chen et al., 2017). Also, with the rising popularity of SSDs as the primary storage medium, it is interesting to look at how well they hold up in large enterprise storage systems (Maneas et al., 2020).

As we have seen in this chapter, file systems are complex beasts and developing new file systems is not easy. Many operating systems allow file systems to be developed in user space (e.g., the FUSE userspace filesystem framework on Linux), but the performance is generally much lower. With new storage devices such as low-level SSDs arriving on the market, the need for an agile storage stack is important and research is needed to develop high-performance file systems quickly (Miller et al., 2021). In fact, new advances in storage technology is driving much of the research on file systems. For instance, how do we build efficient file systems for new persistent memory (Chen et al., 2021; and Neal, 2021)? Or how can we speed up file system checking (Domingo, 2021)? Even fragmentation creates different issues on hard disks and SSDs and requires different approaches (Kesavan, 2019).

Storing increasing amounts of data on the same file system is challenging, especially in mobile devices, leading to the development of new methods to compress the data without slowing down the system too much, for instance by taking the access patterns of files into account (Ji et al., 2021). We have seen that as an alternative to per-file or per-block compression, some file systems today support deduplication across the entire system to prevent storing the same data twice. Unfortunately, deduplication tends to lead to poor data locality and trying to obtain good deduplication without performance loss due to lack of locality is difficult (Zou, 2021). Of course, with data deduplicated all over the place, it becomes much harder to estimate how much space if left or will be left when we delete a certain file (Harnik, 2019).

4.7 SUMMARY

When seen from the outside, a file system is a collection of files and directories, plus operations on them. Files can be read and written, directories can be created and destroyed, and files can be moved from directory to directory. Most modern file systems support a hierarchical directory system in which directories may have subdirectories and these may have subsubdirectories ad infinitum.

When seen from the inside, a file system looks quite different. The file system designers have to be concerned with how storage is allocated and how the system keeps track of which block goes with which file. Possibilities include contiguous files, linked lists, file-allocation tables, and i-nodes. Different systems have different directory structures. Attributes can go in the directories or somewhere else (e.g., an i-node). Disk space can be managed using free lists or bitmaps. File-system reliability is enhanced by making incremental dumps and by having a program that can repair sick file systems. File-system performance is important and can be enhanced in several ways, including caching, read ahead, and carefully placing the blocks of a file close to each other. Log-structured file systems also improve performance by doing writes in large units.

Examples of file systems include ISO 9660, MS-DOS, and UNIX. These differ in many ways, including how they keep track of which blocks go with which file, directory structure, and management of free disk space.

PROBLEMS

1. In Windows, when a user double clicks on a file listed by Windows Explorer, a program is run and given that file as a parameter. List two different ways the operating system could know which program to run.
2. In early UNIX systems, executable files (*a.out* files) began with a very specific magic number, not one chosen at random. These files began with a header, followed by the text and data segments. Why do you think a very specific number was chosen for executable files, whereas other file types had a more-or-less random magic number as the first word?
3. In Fig. 4-5, one of the attributes is the record length. Why does the operating system ever care about this?
4. Is the open system call in UNIX absolutely essential? What would the consequences be of not having it?
5. Systems that support sequential files always have an operation to rewind files. Do systems that support random-access files need this, too?
6. Some operating systems provide a system call rename to give a file a new name. Is there any difference at all between using this call to rename a file and just copying the file to a new file with the new name, followed by deleting the old one?

7. A simple operating system supports only a single directory but allows it to have arbitrarily many files with arbitrarily long file names. Can something approximating a hierarchical file system be simulated? How?

8. In UNIX and Windows, random access is done by having a special system call that moves the "current position" pointer associated with a file to a given byte in the file. Propose an alternative way to do random access without having this system call.

9. Consider the directory tree of Fig. 4-9. If */usr/jim* is the working directory, what is the absolute path name for the file whose relative path name is *../ast/x*?

10. Contiguous allocation of files leads to disk fragmentation, as mentioned in the text, because some space in the last disk block will be wasted in files whose length is not an integral number of blocks. Is this internal fragmentation or external fragmentation? Make an analogy with something discussed in the previous chapter.

11. Suppose a filesystem check reveals that a block has been allocated to two different files, */home/hjb/dadjokes.txt* and */etc/motd*. Both are text files. The filesystem check duplicates the block's data and re-assigns /etc/motd to use the new block. Answer the following questions. (i) In what realistic circumstance(s) could the data from both files still remain correct and consistent with their original content? (ii) How might the user investigate whether the files have been corrupted? (iii) If one or both of the files' data have been corrupted, what mechanisms might allow the user to recover the data?

12. One way to use contiguous allocation of the disk and not suffer from holes is to compact the disk every time a file is removed. Since all files are contiguous, copying a file requires a seek and rotational delay to read the file, followed by the transfer at full speed. Writing the file back requires the same work. Assuming a seek time of 5 msec, a rotational delay of 4 msec, a transfer rate of 8 MB/sec, and an average file size of 8 KB, how long does it take to read a file into main memory and then write it back to the disk at a new location? Using these numbers, how long would it take to compact half of a 16-GB disk?

13. MacOS has symbolic links and also aliases. An alias is similar to a symbolic link; however, unlike symbolic links, an alias stores additional metadata about the target file (such as its inode number and file size) so that, if the target file is moved within the same filesystem, accessing the alias will result in accessing the target file, as the filesystem will search for and find the original target. How could this behavior be beneficial compared to symbolic links? How could it cause problems?

14. Following on the previous question, in earlier MacOS versions, if the target file is moved and then another file is created with the original path of the target, the alias would still find and use the moved target file (not the new file with the same path/name). However, in versions of MacOS 10.2 or later, if the target file is moved and another is created in the old location, the alias will connect to the new file. Does this address the drawbacks from your answer to the previous question? Does it dampen the benefits you noted?

15. Some digital consumer devices need to store data, for example as files. Name a modern device that requires file storage and for which contiguous allocation would be a fine idea.

16. Consider the i-node shown in Fig. 4-15. If it contains 10 direct addresses of 4 bytes each and all disk blocks are 1024 KB, what is the largest possible file?

17. For a given class, the student records are stored in a file. The records are randomly accessed and updated. Assume that each student's record is of fixed size. Which of the three allocation schemes (contiguous, linked and table/indexed) will be most appropriate?

18. Consider a file whose size varies between 4 KB and 4 MB during its lifetime. Which of the three allocation schemes (contiguous, linked and table/indexed) will be most appropriate?

19. It has been suggested that efficiency could be improved and disk space saved by storing the data of a short file within the i-node. For the i-node of Fig. 4-15, how many bytes of data could be stored inside the i-node?

20. Two computer science students, Carolyn and Elinor, are having a discussion about i-nodes. Carolyn maintains that memories have gotten so large and so cheap that when a file is opened, it is simpler and faster just to fetch a new copy of the i-node into the i-node table, rather than search the entire table to see if it is already there. Elinor disagrees. Who is right?

21. Name one advantage of hard links over symbolic links and one advantage of symbolic links over hard links.

22. Explain how hard links and soft links differ with respective to i-node allocations.

23. Consider a 4-TB disk that uses 8-KB blocks and the free-list method. How many block addresses can be stored in one block?

24. Free disk space can be kept track of using a free list or a bitmap. Disk addresses require D bits. For a disk with B blocks, F of which are free, state the condition under which the free list uses less space than the bitmap. For D having the value 16 bits, express your answer as a percentage of the disk space that must be free.

25. The beginning of a free-space bitmap looks like this after the disk partition is first formatted: 1000 0000 0000 0000 (the first block is used by the root directory). The system always searches for free blocks starting at the lowest-numbered block, so after writing file *A*, which uses six blocks, the bitmap looks like this: 1111 1110 0000 0000. Show the bitmap after each of the following additional actions:

(a) File *B* is written, using five blocks.
(b) File *A* is deleted.
(c) File *C* is written, using eight blocks.
(d) File *B* is deleted.

26. What would happen if the bitmap or free list containing the information about free disk blocks was completely lost due to a crash? Is there any way to recover from this disaster, or is it bye-bye disk? Discuss your answers for UNIX and the FAT-16 file system separately.

27. Oliver Owl's night job at the university computing center is to change the tapes used for overnight data backups. While waiting for each tape to complete, he works on writing his thesis that proves Shakespeare's plays were written by extraterrestrial visitors.

His text processor runs on the system being backed up since that is the only one they have. Is there a problem with this arrangement?

28. We discussed making incremental dumps in some detail in the text. In Windows it is easy to tell when to dump a file because every file has an archive bit. This bit is missing in UNIX. How do UNIX backup programs know which files to dump?

29. Suppose that file 21 in Fig. 4-27 was not modified since the last dump. In what way would the four bitmaps of Fig. 4-28 be different?

30. It has been suggested that the first part of each UNIX file be kept in the same disk block as its i-node. What good would this do?

31. Consider Fig. 4-29. Is it possible that for some particular block number the counters in *both* lists have the value 2? How should this problem be corrected?

32. The performance of a file system depends upon the cache hit rate (fraction of blocks found in the cache). If it takes 1 msec to satisfy a request from the cache, but 40 msec to satisfy a request if a disk read is needed, give a formula for the mean time required to satisfy a request if the hit rate is h. Plot this function for values of h varying from 0 to 1.0.

33. For an external USB hard drive attached to a computer, which is more suitable: a write-through cache or a block cache?

34. Consider an application where students' records are stored in a file. The application takes a student ID as input and subsequently reads, updates, and writes the corresponding student record; this is repeated till the application quits. Would the "block read-ahead" technique be useful here?

35. Discuss the design issues involved in selecting the appropriate block size for a file system.

36. Consider a disk that has 10 data blocks starting from block 14 through 23. Let there be 2 files on the disk: f1 and f2. The directory structure lists that the first data blocks of f1 and f2 are, respectively, 22 and 16. Given the FAT table entries as below, what are the data blocks allotted to f1 and f2?

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

In the above notation, (x, y) indicates that the value stored in table entry x points to data block y.

37. In the text, we discussed two major ways to identify file type: file extensions and investigation of file content (e.g., by using headers and magic numbers). Many modern UNIX filesystems support extended attributes which can store additional metadata for a file, including file type. This data is stored as part of the file's attribute data (in the same way that file size and permissions are stored). How is the extended attribute approach for storing files better or worse than the file extension approach or identifying file type by content?

38. Consider the idea behind Fig. 4-23, but now for a disk with a mean seek time of 8 msec, a rotational rate of 15,000 rpm, and 262,144 bytes per track. What are the data rates for block sizes of 1 KB, 2 KB, and 4 KB, respectively?

39. In this chapter, we have seen that SSDs do their best to avoid writing the same memory cells frequently (because of the wear). However, many SSDs offer much more functionality than what we presented so far. For instance, many controllers implement compression. Explain why compression may help with reducing the wear.

40. Given a disk-block size of 4 KB and block-pointer address value of 4 bytes, what is the largest file size (in bytes) that can be accessed using 11 direct addresses and one indirect block?

41. The MS-DOS FAT-16 table contains 64K entries. Suppose that one of the bits had been needed for some other purpose and that the table contained exactly 32,768 entries instead. With no other changes, what would the largest MS-DOS file have been under this condition?

42. Files in MS-DOS have to compete for space in the FAT-16 table in memory. If one file uses k entries, that is k entries that are not available to any other file, what constraint does this place on the total length of all files combined?

43. How many disk operations are needed to fetch the i-node for a file with the path name */usr/ast/courses/os/handout.t*? Assume that the i-node for the root directory is in memory, but nothing else along the path is in memory. Also assume that all directories fit in one disk block.

44. In many UNIX systems, the i-nodes are kept at the start of the disk. An alternative design is to allocate an i-node when a file is created and put the i-node at the start of the first block of the file. Discuss the pros and cons of this alternative.

45. Write a program that reverses the bytes of a file, so that the last byte is now first and the first byte is now last. It must work with an arbitrarily long file, but try to make it reasonably efficient.

46. Write a program that starts at a given directory and descends the file tree from that point recording the sizes of all the files it finds. When it is all done, it should print a histogram of the file sizes using a bin width specified as a parameter (e.g., with 1024, file sizes of 0 to 1023 go in one bin, 1024 to 2047 go in the next bin, etc.).

47. Write a program that scans all directories in a UNIX file system and finds and locates all i-nodes with a hard link count of two or more. For each such file, it lists together all file names that point to the file.

48. Write a new version of the UNIX *ls* program. This version takes as an argument one or more directory names and for each directory lists all the files in that directory, one line per file. Each field should be formatted in a reasonable way given its type. List only the first disk address, if any.

49. Implement a program to measure the impact of application-level buffer sizes on read time. This involves writing to and reading from a large file (say, 2 GB). Vary the application buffer size (say, from 64 bytes to 4 KB). Use timing measurement routines (such as *gettimeofday* and *getitimer* on UNIX) to measure the time taken for different buffer sizes. Analyze the results and report your findings: does buffer size make a difference to the overall write time and per-write time?

50. Implement a simulated file system that will be fully contained in a single regular file stored on the disk. This disk file will contain directories, i-nodes, free-block information, file data blocks, etc. Choose appropriate algorithms for maintaining free-block information and for allocating data blocks (contiguous, indexed, linked). Your program will accept system commands from the user to perform file system operations, including at least one to create/delete directories, create/delete/open files, read/write from/to a selected file, and to list directory contents.