6

FILE SYSTEMS

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. However, the storage capacity is restricted to the size of the virtual address space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small.

A second problem with keeping information within a process' address space 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.

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. 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 concurrently.

The usual solution to all these problems is to store information on disks and other external media in units called **files**. Processes can then read them and write 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 only disappear when its owner explicitly removes it.

Files are managed by the operating system. How they are structured, named, accessed, used, protected, and implemented 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 users' standpoint, the most important aspect of a file system is how it appears to them, that is, 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 block are of less 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. Finally, we give some examples of real file systems.

6.1 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.

6.1.1 File Naming

Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield 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 one to eight 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. Many file systems support names as long as 255 characters.

Some file systems distinguish between upper and lower case letters, whereas others do not. UNIX falls in the first category; 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. Windows 95 and Windows 98 both use the MS-DOS file system, and thus inherit many of its properties, such as how file names are constructed. In addition, Windows NT and Windows 2000 support the MS-DOS file system and thus also inherit its properties. However, the latter two systems also have a native file system (NTFS) that has different properties (such as file names in Unicode). In this chapter, when we refer to the Windows file system, we mean the MS-DOS file system, which is the only file system supported by all versions of Windows. We will discuss the Windows 2000 native file system in Chap. 11.

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 are 1 to 8 characters, plus an optional extension of 1 to 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 *prog.c.Z*, where *.Z* is commonly used to indicate that the file (*prog.c*) has been compressed using the Ziv-Lempel compression algorithm. Some of the more common file extensions and their meanings are shown in Fig. 6-1.

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

Figure 6-1. Some typical file extensions.

In some systems (e.g., 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.

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.doc* starts Microsoft *Word* with *file.doc* as the initial file to edit.

6.1.2 File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 6-2. The file in Fig. 6-2(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.

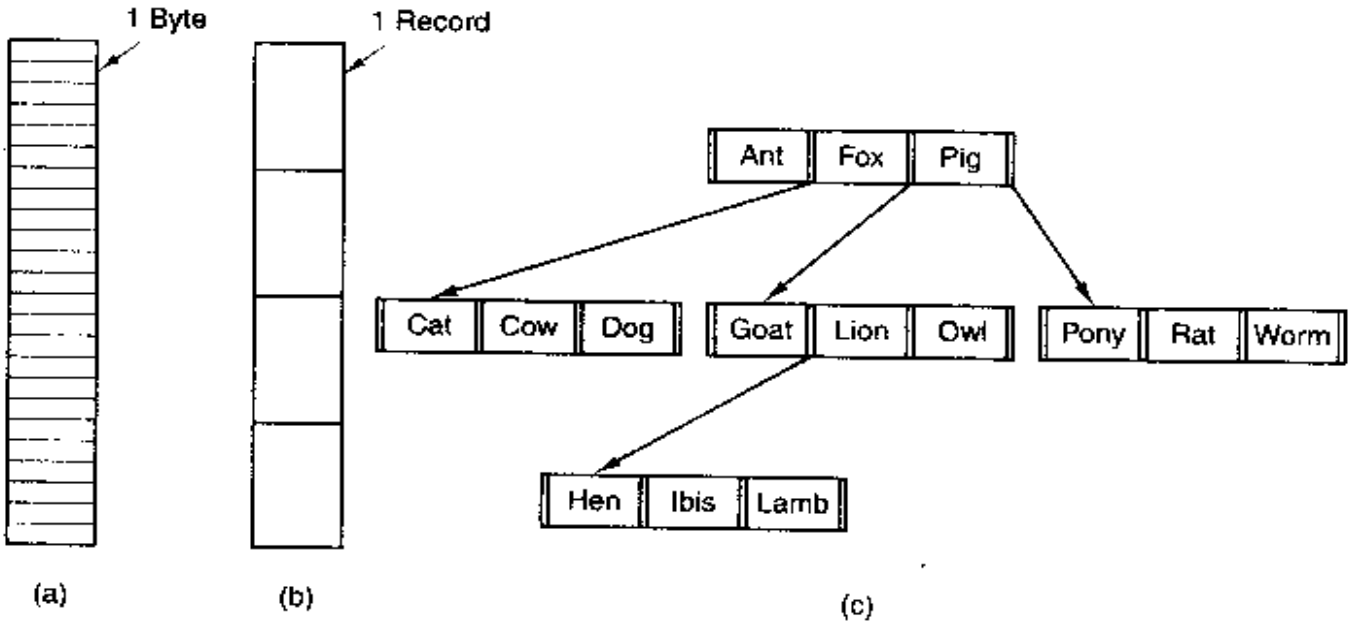

Figure 6-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Having the operating system regard files as nothing more than byte sequences provides the maximum flexibility. User programs can put anything they want in their files and name them any way that is 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.

The first step up in structure is shown in Fig. 6-2(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 king, 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 works this way.

The third kind of file structure is shown in Fig. 6-2(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. 6-2(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 but is widely used on the large mainframe computers still used in some commercial data processing.

6.1.3 File Types

Many operating systems support several types of files. UNIX 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. 6-2 are regular files. **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., MS-DOS) 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 files, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of what is apparently random junk. Usually, they have some internal structure known to programs that use them.

For example, in Fig. 6-3(a) we see a simple executable binary file taken from a version of UNIX. Although technically the file is just a sequence of bytes, the operating system will only execute a file if it has the proper format. It has five sections: header, text, data, relocation bits, and symbol table. The header starts with a so-called **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. Following 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 used 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, but 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 if 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 the operating system 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).

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.

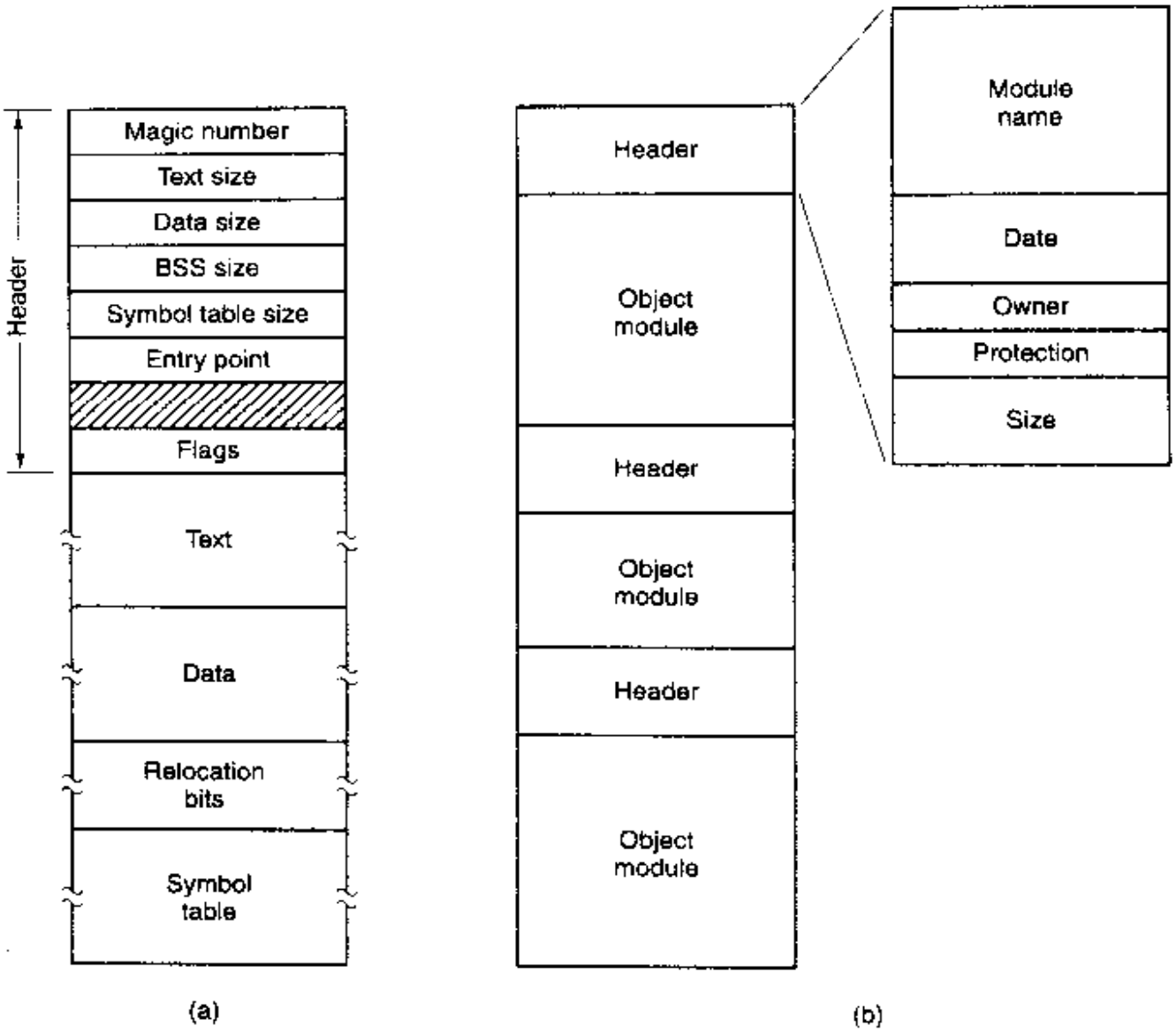

Figure 6-3. (a) An executable file. (b) An archive.

6.1.4 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 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 are 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.

In some older mainframe operating systems, files are classified as being either sequential or random access at the time they are created. This allows the system to use different storage techniques for the two classes. Modern operating systems do not make this distinction. All their files are automatically random access.

6.1.5 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 created and the file's size. We will call these extra items the file's **attributes**. The list of attributes varies considerably from system to system. The table of Fig. 6-4 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 is a bit that keeps track of whether the file has been backed up. 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 various 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 require the maximum size to be specified when the file is created, in order to let the operating system reserve the maximum amount of storage in ad-

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 has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 6-4. Some possible file attributes.

vance. Workstation and personal computer operating systems are clever enough to do without this feature.

6.1.6 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 much 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 only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.

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. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

6.1.7 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. 6-5. 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 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 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 and to use the names instead of the constants. Not only does this convention make programs easier to read, but it also makes them easier to maintain. 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.

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

```
/* 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 6-5. A simple program to copy a file.

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 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 end of file is 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. 6-5. We will examine the Windows 2000 calls in Chap. 11.

6.1.8 Memory-Mapped Files

Many programmers feel that accessing files as shown above is cumbersome and inconvenient, especially when compared to accessing ordinary memory. For this reason, some operating systems, starting with MULTICS, have provided a way to map files into the address space of a running process. Conceptually, we can imagine the existence of two new system calls, map and unmap. The former gives a file name and a virtual address, which causes the operating system to map the file into the address space at the virtual address.

For example, suppose that a file, *f*, of length 64 KB, is mapped into the virtual address starting at address 512K. Then any machine instruction that reads the contents of the byte at 512K gets byte 0 of the file, and so on. Similarly, a write to address 512K + 21000 modifies byte 21000 of the file. When the process

terminates, the modified file is left on the disk, just as though it had been changed by a combination of seek and write system calls.

What actually happens is that the system's internal tables are changed to make the file become the backing store for the memory region 512K to 576K. Thus a read from 512K causes a page fault, bringing in page 0 of the file. Similarly, a write to 512K + 1100 causes a page fault, bringing in the page containing that address, after which the write to memory can take place. If that page is ever evicted by the page replacement algorithm, it is written back to the appropriate place in the file. When the process finishes, all mapped, modified pages are written back to their files.

File mapping works best in a system that supports segmentation. In such a system, each file can be mapped onto its own segment so that byte *k* in the file is also byte *k* in the segment. In Fig. 6-6(a) we see a process that has two segments, text and data. Suppose that this process copies files, like the program of Fig. 6-5. First it maps the source file, say, *abc*, onto a segment. Then it creates an empty segment and maps it onto the destination file, *xyz* in our example. These operations give the situation shown in Fig. 6-6(b).

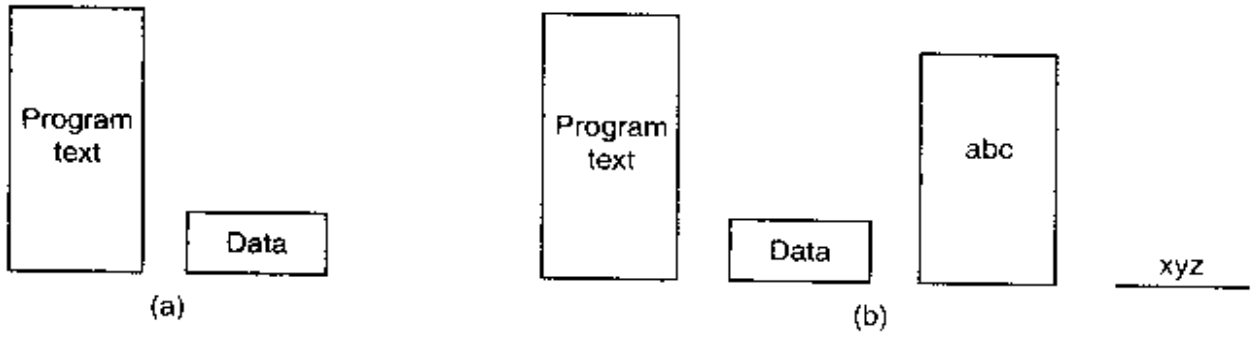

Figure 6-6. (a) A segmented process before mapping files into its address space. (b) The process after mapping an existing file *abc* into one segment and creating a new segment for file *xyz*.

At this point the process can copy the source segment into the destination segment using an ordinary copy loop. No read or write system calls are needed. When it is all done, it can execute the unmap system call to remove the files from the address space and then exit. The output file, *xyz*, will now exist, as though it had been created in the conventional way.

Although file mapping eliminates the need for I/O and thus makes programming easier, it introduces a few problems of its own. First, it is hard for the system to know the exact length of the output file, *xyz*, in our example. It can easily tell the number of the highest page written, but it has no way of knowing how many bytes in that page were written. Suppose that the program only uses page 0, and after execution all the bytes are still 0 (their initial value). Maybe *xyz* is a file consisting of 10 zeros. Maybe it is a file consisting of 100 zeros. Maybe it is a file consisting of 1000 zeros. Who knows? The operating system cannot tell. All it can do is create a file whose length is equal to the page size.

A second problem can (potentially) occur if a file is mapped in by one process and opened for conventional reading by another. If the first process modifies a page, that change will not be reflected in the file on disk until the page is evicted. The system has to take great care to make sure the two processes do not see inconsistent versions of the file.

A third problem with mapping is that a file may be larger than a segment, or even larger than the entire virtual address space. The only way out is to arrange the map system call to be able to map a portion of a file, rather than the entire file. Although this works, it is clearly less satisfactory than mapping the entire file.

6.2 DIRECTORIES

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

6.2.1 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 early 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. 6-7. Here the directory contains four files. The file *owners* are shown in the figure, not the file *names* (because the owners are important to the point we are about to make). The advantages of this scheme are its simplicity and the ability to locate files quickly—there is only one place to look, after all.

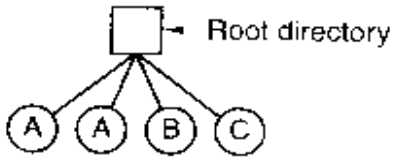

Figure 6-7. A single-level directory system containing four files, owned by three different people, *A*, *B*, and *C*.

The problem with having only one directory in a system with multiple users is that different users may accidentally use the same names for their files. For example, if user *A* creates a file called *mailbox*, and then later user *B* also creates a file called *mailbox*, *B*'s file will overwrite *A*'s file. Consequently, this scheme is

not used on multiuser systems any more, but could be used on a small embedded system, for example, a system in a car that was designed to store user profiles for a small number of drivers.

6.2.2 Two-level Directory Systems

To avoid conflicts caused by different users choosing the same file name for their own files, the next step up is giving each user a private directory. In that way, names chosen by one user do not interfere with names chosen by a different user and there is no problem caused by the same name occurring in two or more directories. This design leads to the system of Fig. 6-8. This design could be used, for example, on a multiuser computer or on a simple network of personal computers that shared a common file server over a local area network.

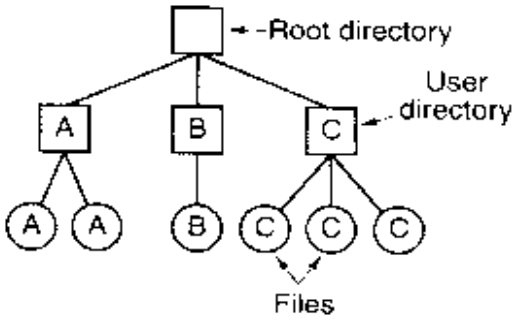

Figure 6-8. A two-level directory system. The letters indicate the owners of the directories and files.

Implicit in this design is that when a user tries to open a file, the system knows which user it is in order to know which directory to search. As a consequence, some kind of login procedure is needed, in which the user specifies a login name or identification, something not required with a single-level directory system.

When this system is implemented in its most basic form, users can only access files in their own directories. However, a slight extension is to allow users to access other users' files by providing some indication of whose file is to be opened. Thus, for example,

```
open("x")
```

might be the call to open a file called *x* in the user's directory, and

```
open("nancy/x")
```

might be the call to open a file *x* in the directory of another user, Nancy.

One situation in which users need to access files other than their own is to execute system binary programs. Having copies of all the utility programs present in each directory clearly is inefficient. At the very least, there is a need for a system directory with the executable binary programs.

6.2.3 Hierarchical Directory Systems

The two-level hierarchy eliminates name conflicts among users but is not satisfactory for users with a large number of files. Even on a single-user personal computer, it is inconvenient. It is quite common for users to want to group their files together in logical ways. A professor, for example, might have a collection of files that together form a book that he is writing for one course, a second collection of files containing student programs submitted for another course, a third group of files containing the code of an advanced compiler-writing system he is building, a fourth group of files containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on. Some way is needed to group these files together in flexible ways chosen by the user.

What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Fig. 6-9. 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.

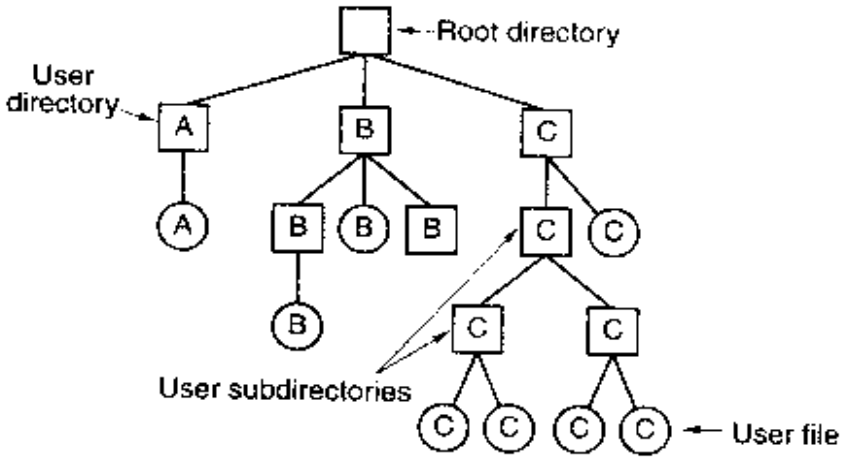

Figure 6-9. A hierarchical directory system.

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, nearly all modern file systems are organized in this manner.

6.2.4 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/ast*, then the file whose absolute path is */usr/ast/mailbox* can be referenced simply as *mailbox*. In other words, the UNIX command

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

and the command

```
cp mailbox mailbox.bak
```

do exactly the same thing if the working directory is */usr/ast*. 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, so when a process changes its working directory and later exits, no other processes are affected and no traces of the change are left behind in the file system. In this way it is always perfectly safe for a process to change its working directory whenever that 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. To see how these are used, consider the UNIX file tree of Fig. 6-10. A certain process has */usr/ast* as its working directory. It can use .. to go 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*.

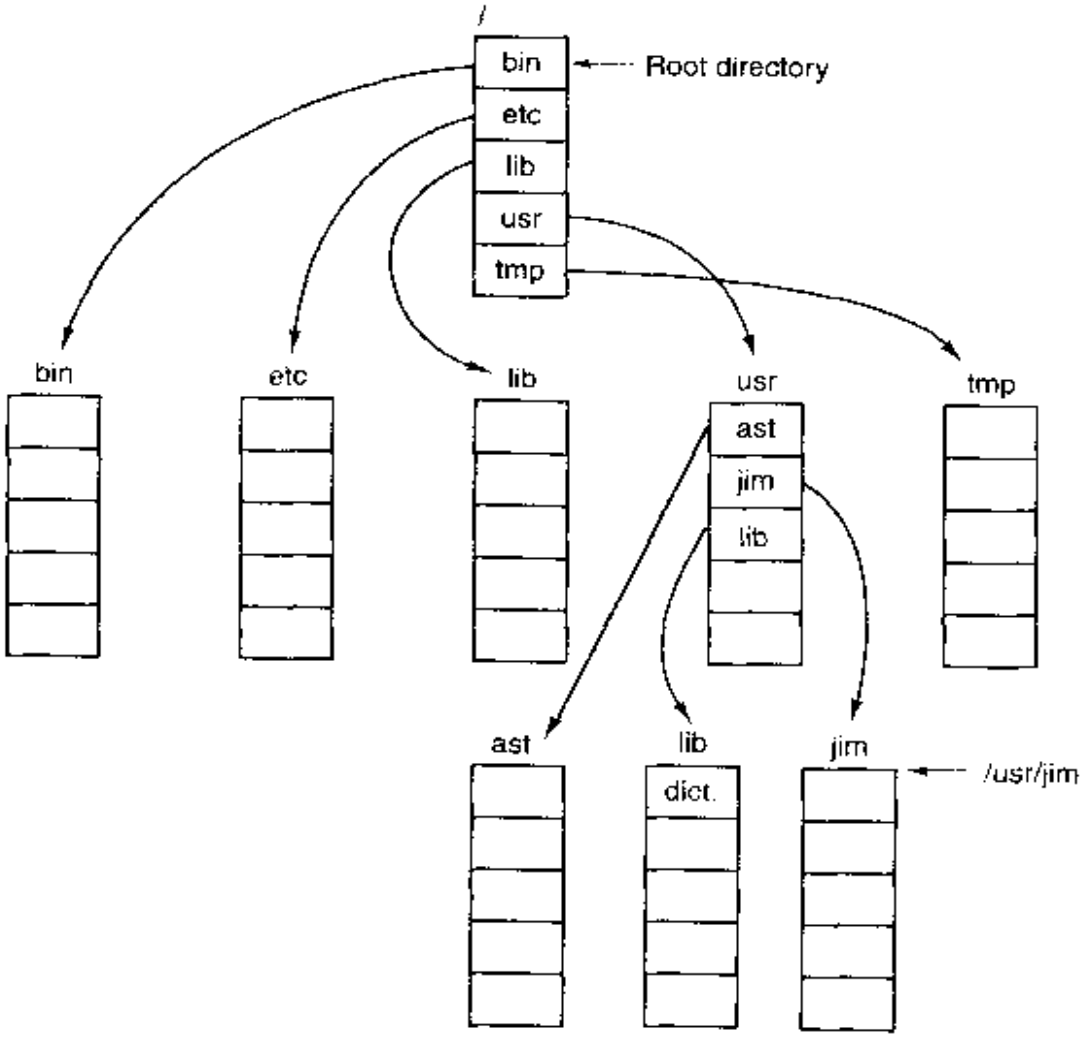

Figure 6-10. A UNIX directory tree.

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 there. Of course, a more normal way to do the copy would be to type

```
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.

6.2.5 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 system (or in a few cases, 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 usually 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.

6.3 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. Implementors 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.

6.3.1 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. 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, 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, so reserving a boot block is a good idea anyway.

Other than starting with a boot block, the layout of a disk partition varies strongly from file system to file system. Often the file system will contain some of the items shown in Fig. 6-11. 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.

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-

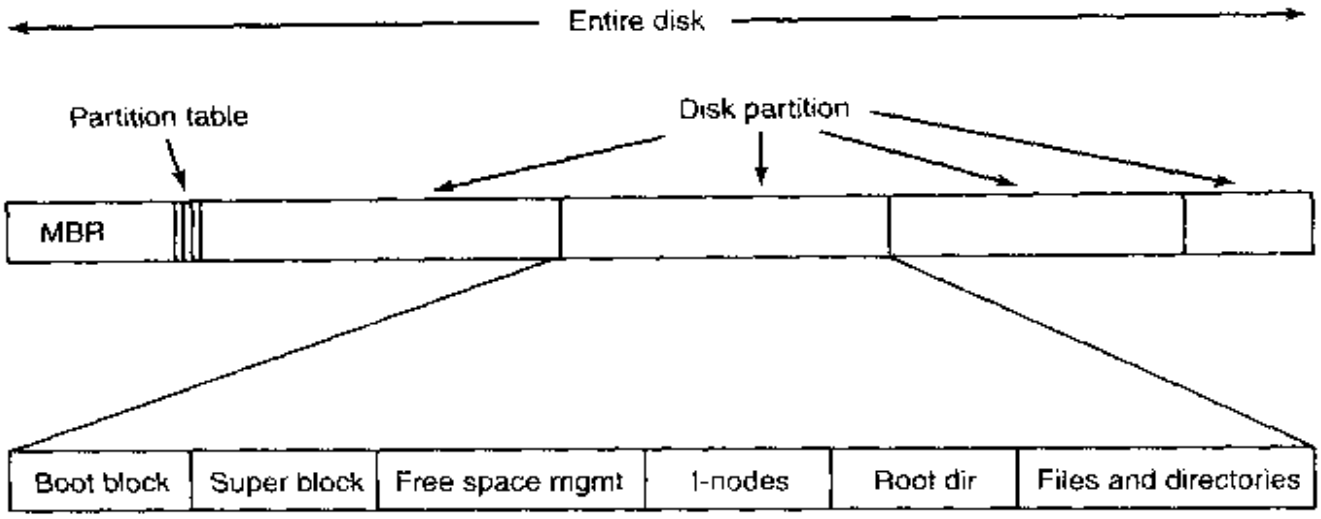

Figure 6-11. A possible file system layout.

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 typically contains all the other directories and files.

6.3.2 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. 6-12(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.

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

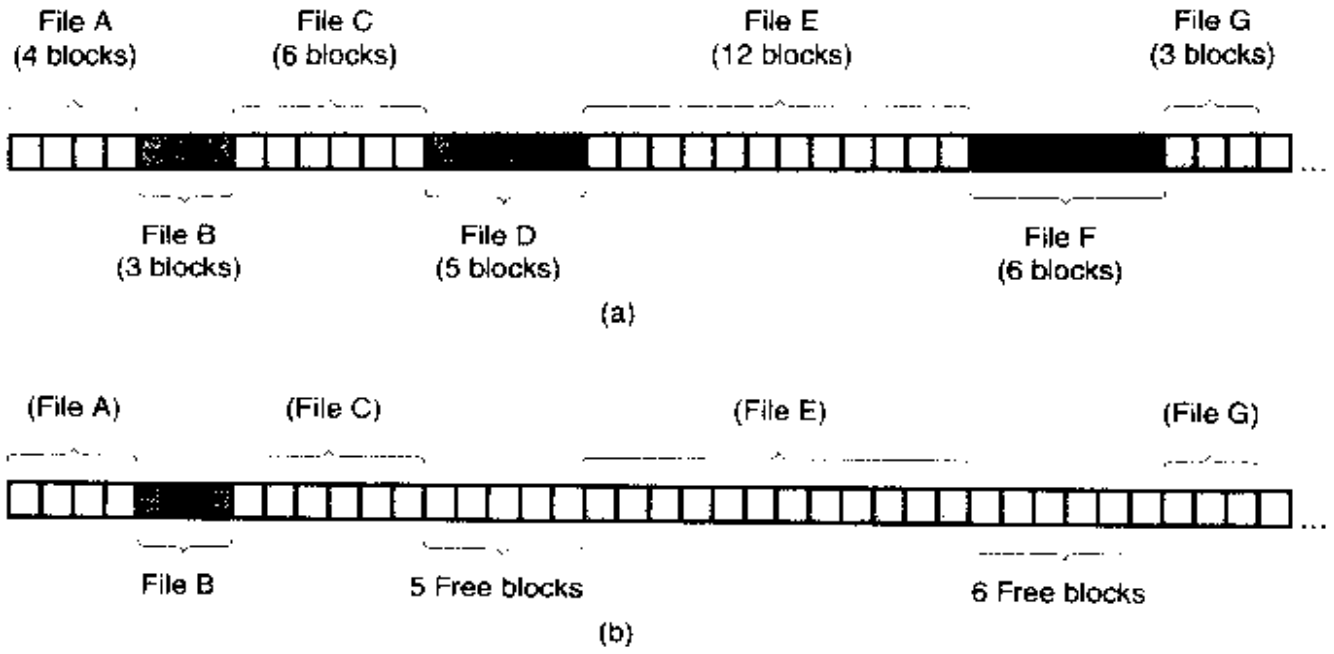

Figure 6-12. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

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.

Second, the read performance is excellent 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.

Unfortunately, contiguous allocation also has a significant drawback: in time, the disk becomes fragmented. To see how this comes about, examine Fig. 6-12(b). Here two files, *D* and F have been removed. When a file is removed, its blocks are 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. 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 of the correct size to place it in.

Imagine the consequences of such a design. The user starts a text editor or word processor in order to type a document. The first thing the program asks is

how many bytes the final document 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 MB, 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 and say 50 MB this time, and so on until a suitable hole was located. Still, this scheme is not likely to lead to happy users.

However, there is one situation in which contiguous allocation is feasible and, in fact, widely used: on CD-ROMs. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system. We will study the most common CD-ROM file system later in this chapter.

As we mentioned in Chap. 1, history often repeats itself in computer science as new generations of technology occur. Contiguous allocation was actually used on magnetic disk file systems years ago due to its simplicity and high performance (user friendliness did not count for much then). Then the idea was dropped due to the nuisance of having to specify final file size at file creation time. But with the advent of CD-ROMs, DVDs, and other write-once optical media, suddenly contiguous files are a good idea again. It is thus important to study old systems and ideas that were conceptually clean and simple because they may be applicable to future systems in surprising ways.

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. 6-13. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

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 to 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.

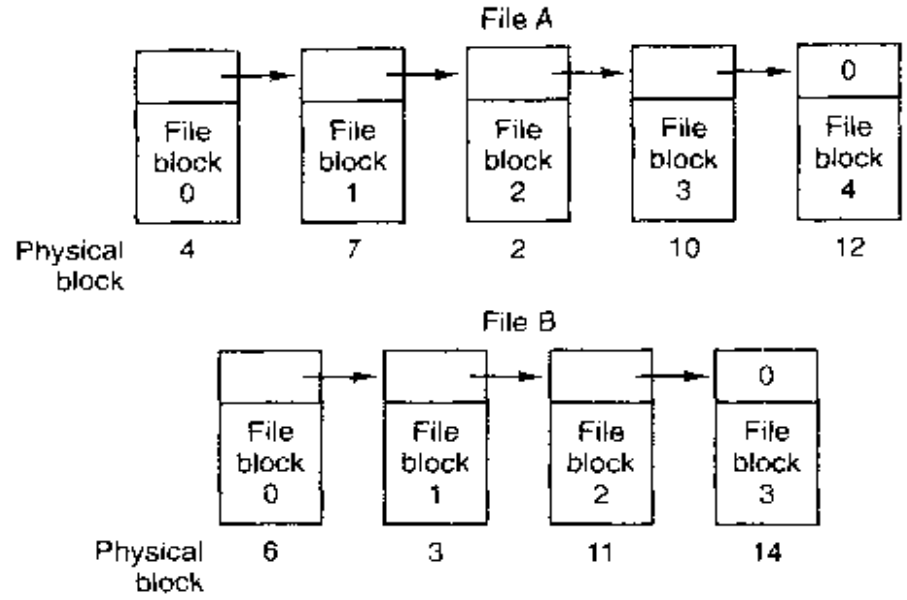

Figure 6-13. Storing a file as a linked list of disk blocks.

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 6-14 shows what the table looks like for the example of Fig. 6-13. 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. 6-14, 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)**.

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 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million 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 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time. Conceivably the table could be put in pageable memory, but it would still occupy a great deal of virtual memory and disk space as well as generating extra paging traffic.

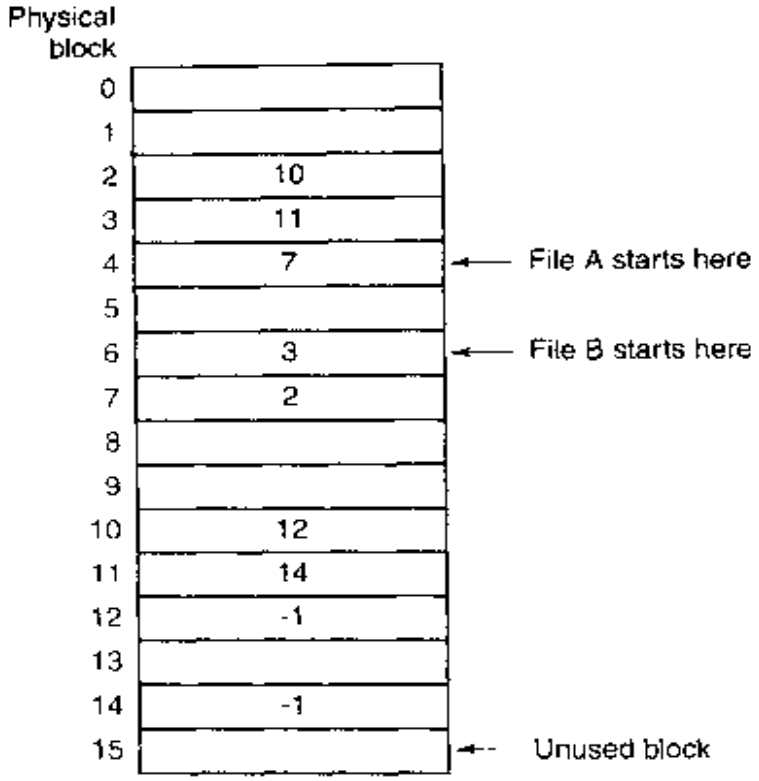

Figure 6-14. Linked list allocation using a file allocation table in main memory.

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. 6-15. 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 need only be in memory 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.

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 1 GB or 10 GB or 100 GB.

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

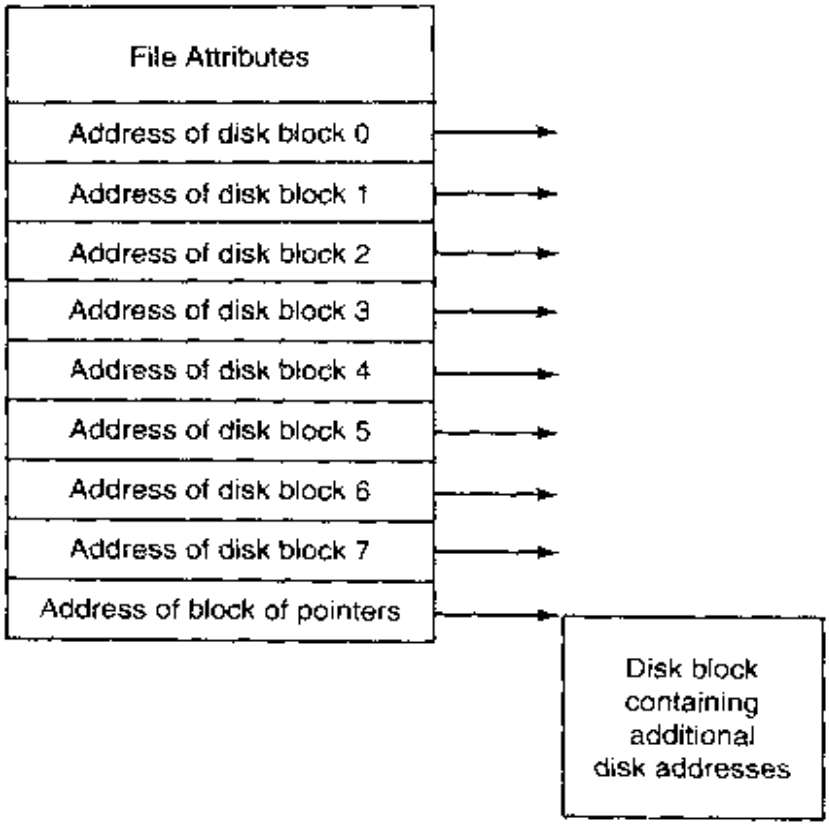

Figure 6-15. An example i-node.

of a block containing more disk block addresses, as shown in Fig. 6-15. 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 later.

6.3.3 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. 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 (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 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. Many systems do precisely that. This option is shown in Fig. 6-16(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.

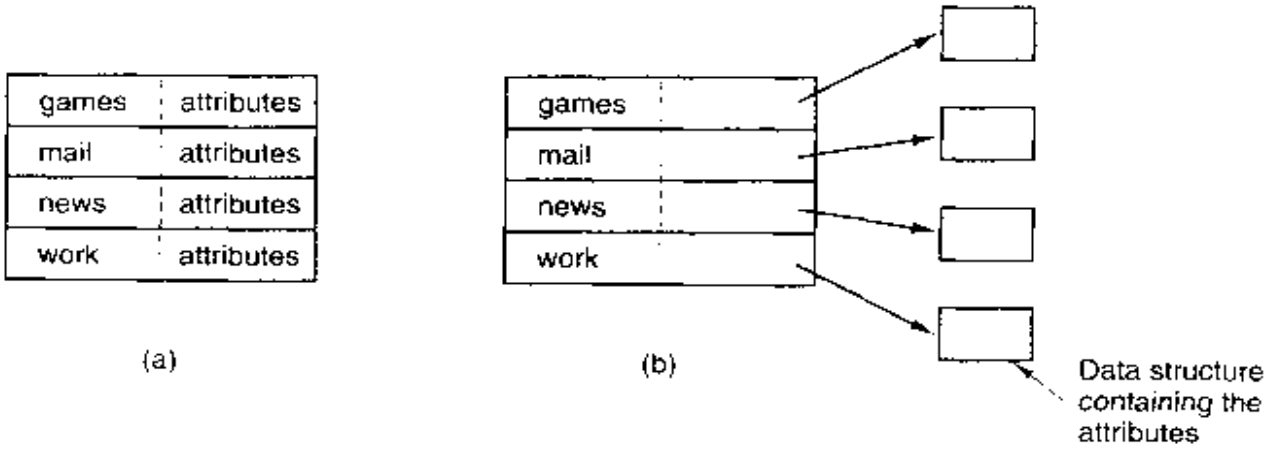

Figure 6-16. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

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. 6-16(b). As we shall see later, this method has certain advantages over putting them in the directory entry. The two approaches shown in Fig. 6-16 correspond to MS-DOS/Windows and UNIX, respectively, as we will see later in this chapter.

So far we have assumed 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?

The simplest approach is to set a limit on file name length, typically 255 characters, and then use one of the designs of Fig. 6-16 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. 6-17(a) in big-endian format (e.g., SPARC). 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.

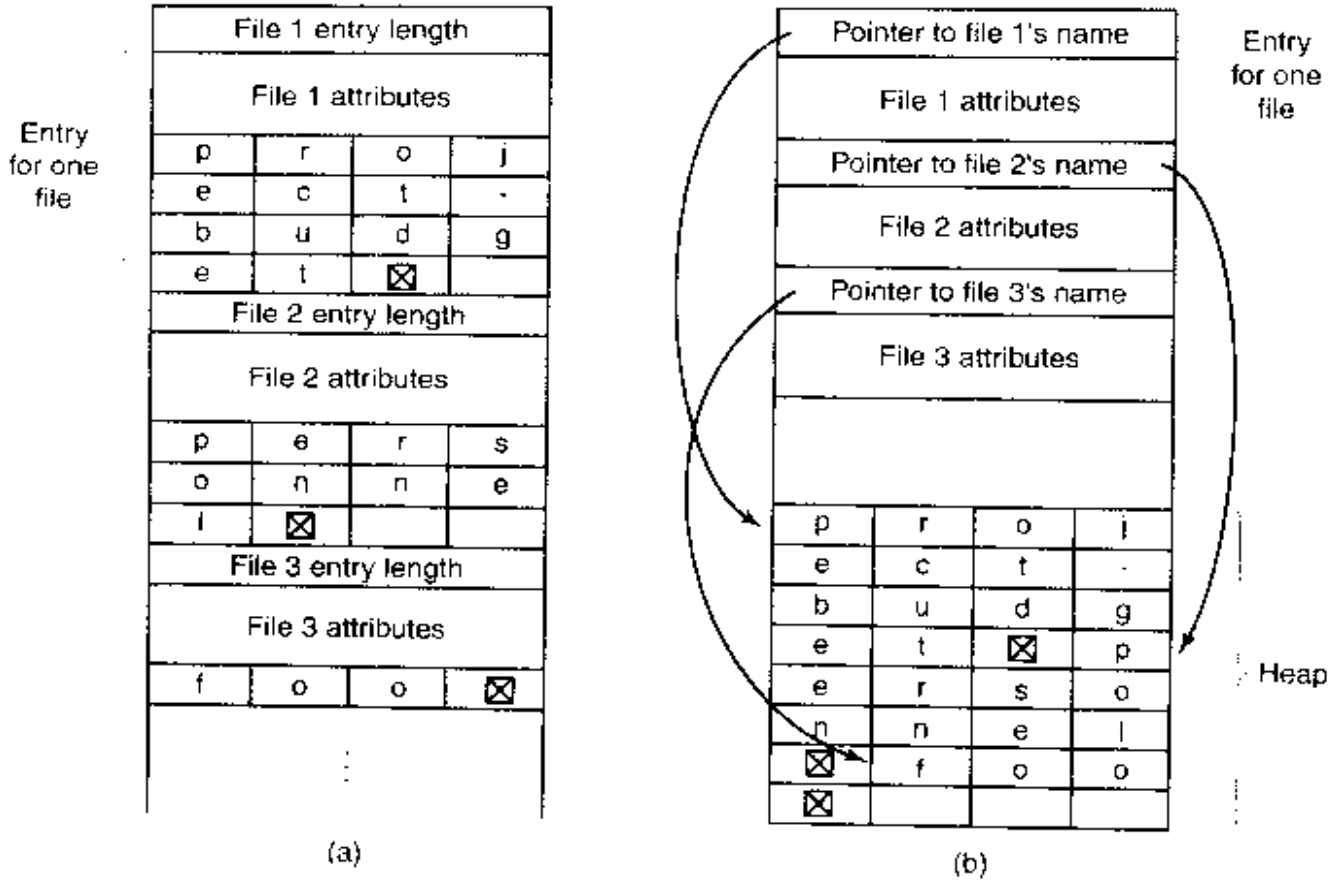

Figure 6-17. Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

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 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. 6-17(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 minor win here is that there is no longer any real need for file names to begin at word boundaries, so no filler characters are needed after file names in Fig. 6-17(b) and they are in Fig. 6-17(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 that 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 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 completely 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 quickly, avoiding a long lookup. Of course, caching only works if a relatively small number of files comprise the majority of the lookups.

6.3.4 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 6-18 shows the file system of Fig. 6-9 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 **Directed Acyclic Graph**, or **DAG**, rather than a tree.

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 point just to the little data structure. This is the approach used in UNIX (where the little data structure is the i-node).

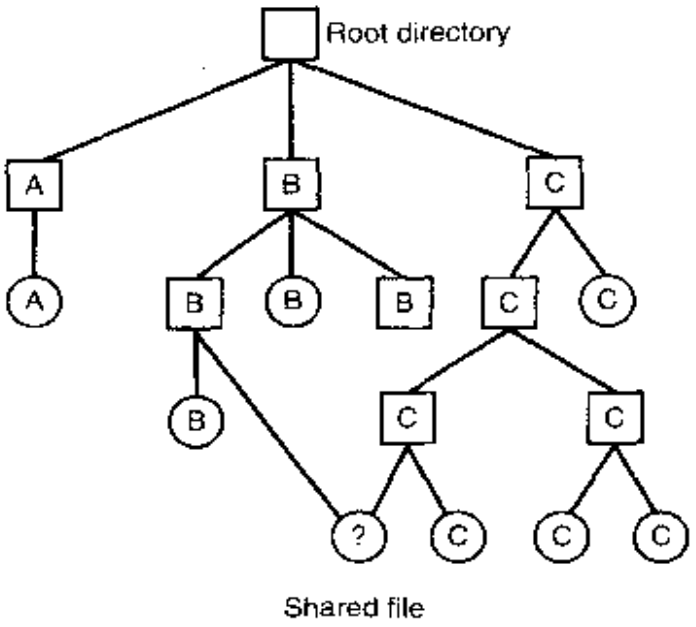

Figure 6-18. File system containing a shared file.

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**.

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. 6-19), but it does increase the link count in the i-node, so the system knows how many directory entries currently point to the file.

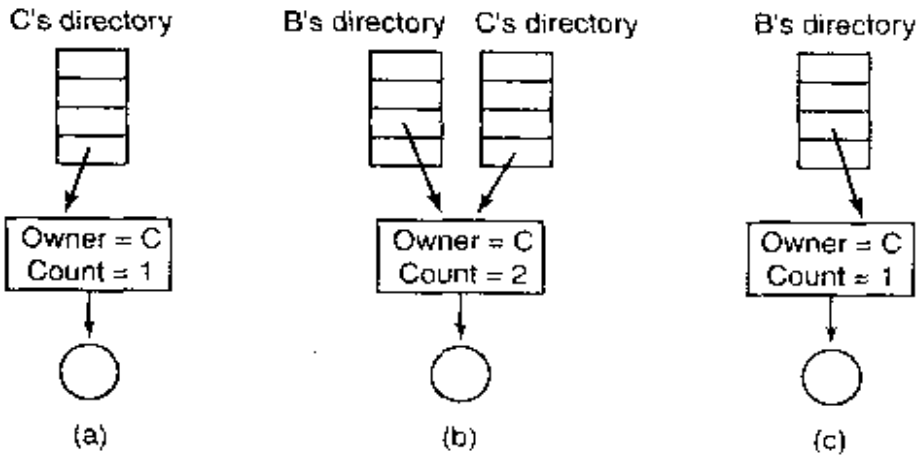

Figure 6-19. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

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 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. 6-19(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 an 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 tape may make multiple copies of a linked file. Furthermore, if the tape is then read into another machine, unless the dump program is clever, the linked file will be copied twice onto the disk, instead of being linked.

6.3.5 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 as a contiguous sequence of bytes has the obvious problem that if a file grows, it will probably 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 of how big the block should be. Given the way 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 a paging system, the page size is also a major contender.

Having a large allocation unit, such as a cylinder, means that every file, even a 1-byte file, ties up an entire cylinder. Studies (Mullender and Tanenbaum, 1984) have shown that the median file size in UNIX environments is about 1 KB, so allocating a 32-KB block for each file would waste 31/32 or 97 percent of the total disk space.

On the other hand, using a small allocation unit means that each file will consist of many blocks. Reading each block normally requires a seek and a rotational delay, so reading a file consisting of many small blocks will be slow.

As an example, consider a disk with 131,072 bytes per track, a rotation time of 8.33 msec, and an average seek time of 10 msec. The time in milliseconds to read a block of k bytes is then the sum of the seek, rotational delay, and transfer times:

$$10 + 4.165 + (k/131072) \times 8.33$$

The solid curve of Fig. 6-20 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. A recent measurement at the author's department, which has 1000 users and over 1 million UNIX disk files, gives a median size of 1680 bytes, meaning that half the files are smaller than 1680 bytes and half are larger. As an aside, the median is a better metric than the mean because a very small number of files can influence the mean enormously, but not the median. (The mean is in fact 10,845 bytes, due in part to a few 100 MB hardware manuals that happen to be online). For simplicity, let us assume all files are 2 KB, which leads to the dashed curve in Fig. 6-20 for the disk space efficiency.

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 14 msec to access a block, the more data that are fetched, the better. Hence the data rate goes up with block size (until the transfers take so long that

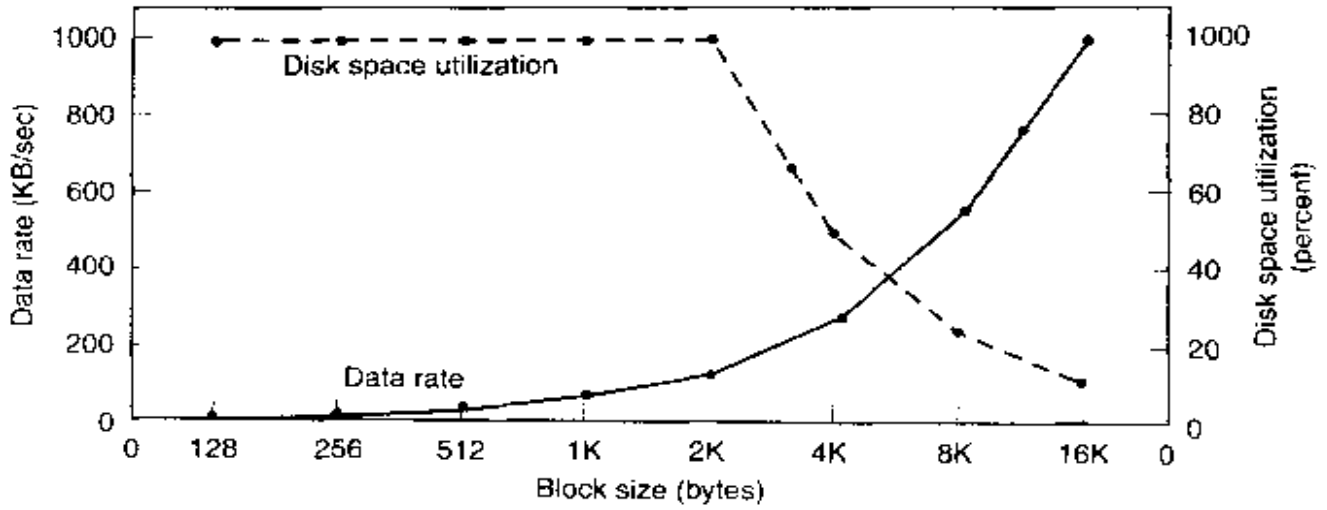

Figure 6-20. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All files are 2 KB.

the transfer time begins to dominate). With small blocks that are powers of two and 2-KB files, no space is wasted in a block. However, with 2-KB files and 4 KB or larger blocks, some disk space is wasted. In reality, few files are a 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. A compromise size is needed. For this data, 4 KB might be a good choice, but some operating systems made their choices a long time ago, when the disk parameters and file sizes were different. For UNIX, 1 KB is commonly used. For MS-DOS the block size can be any power of two from 512 bytes to 32 KB, but is determined by the disk size and for reasons unrelated to these arguments (the maximum number of blocks on a disk partition is 2^{16}, which forces large blocks on large disks).

In an experiment to see if Windows NT file usage was appreciably different from UNIX file usage, Vogels made measurements on files at Cornell University (Vogels, 1999). He observed that NT file usage is more complicated than on UNIX. He wrote:

> *When we type a few characters in the* notepad *text editor, saving this to a file will trigger 26 system calls, including 3 failed open attempts, 1 file overwrite and 4 additional open and close sequences.*

Nevertheless, he observed a median size (weighted by usage) of files just read at 1 KB, files just written as 2.3 KB and files read and written as 4.2 KB. Given the fact that Cornell has much more large-scale scientific computing than the author's institution and the difference in measurement technique (static versus dynamic), the results are reasonably consistent with a median file size of around 2 KB.

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. 6-21. 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 needed for the pointer to the next block). A 16-GB disk needs a free list of maximum 16,794 blocks to hold all 2^{24} disk block numbers. Often free blocks are used to hold the free list.

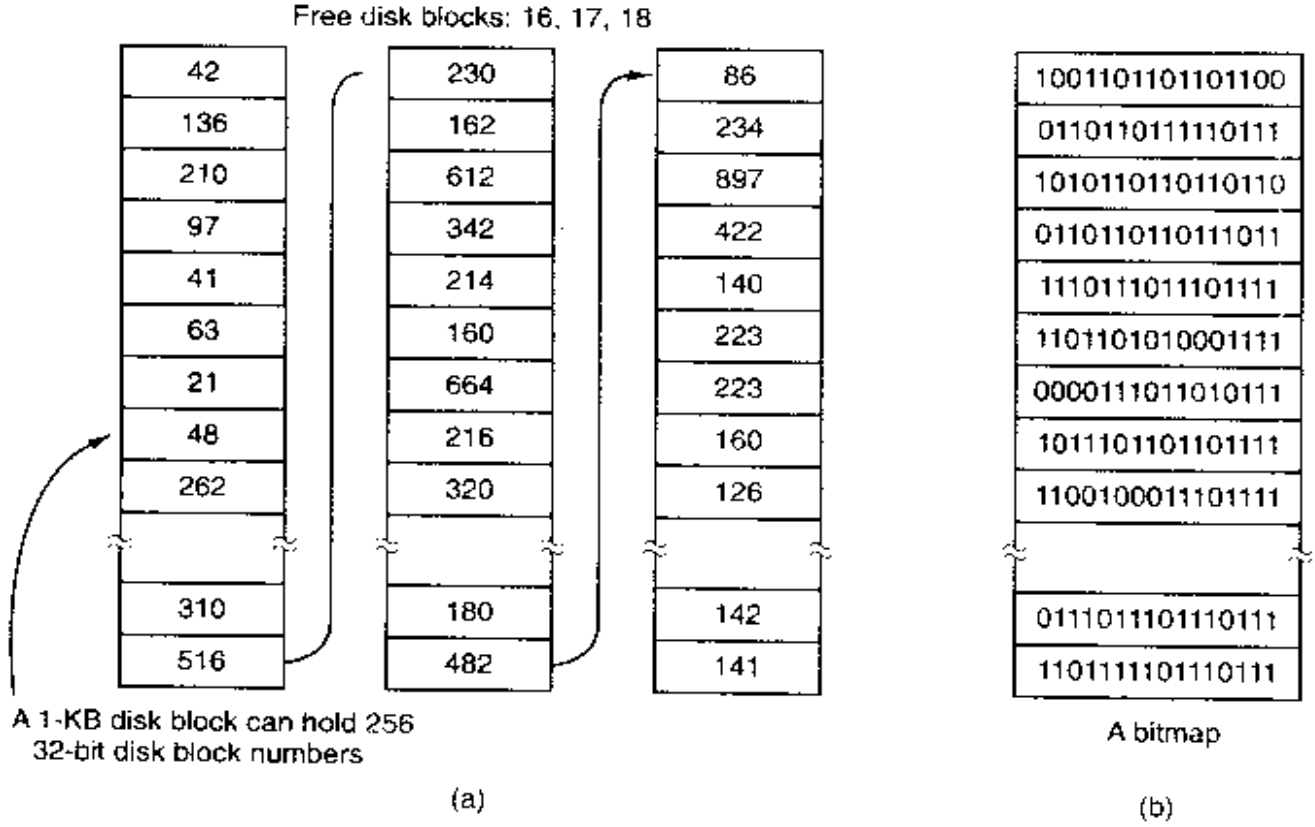

Figure 6-21. (a) Storing the free list on a linked list. (b) A bitmap.

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, allocated blocks by 0s (or vice versa). A 16-GB disk has 2^{24} 1-KB blocks and thus requires 2^{24} bits for the map, which requires 2048 blocks. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, versus 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. On the other hand, if there are many blocks free, some of them can be borrowed to hold the free list without any loss of disk capacity.

When the free list method is used, 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. 6-22(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. 6-22(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. 6-22(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.

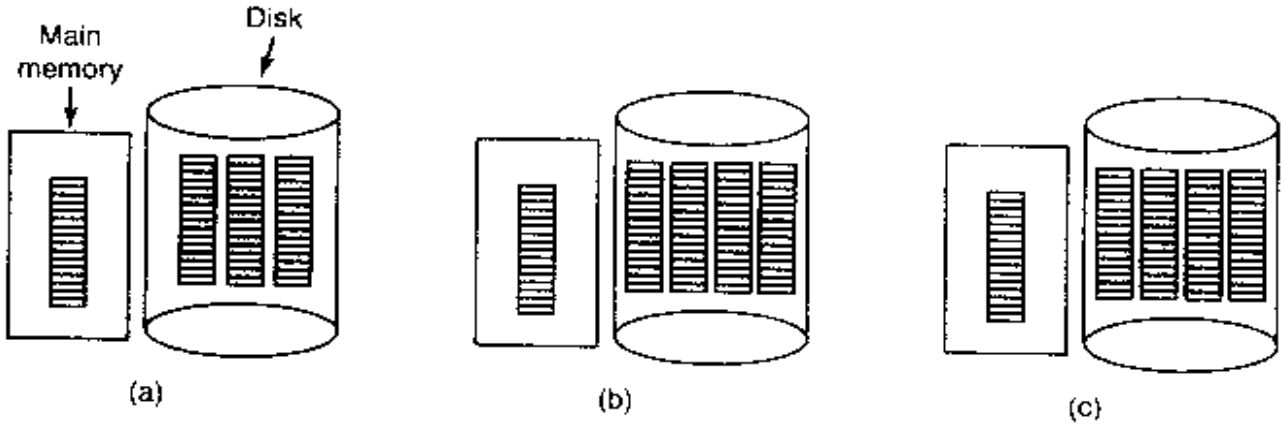

Figure 6-22. (a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

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. 6-22(a) to Fig. 6-22(b), we go from Fig. 6-22(a) to Fig. 6-22(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 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 one only when it becomes 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. 6-23. 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.

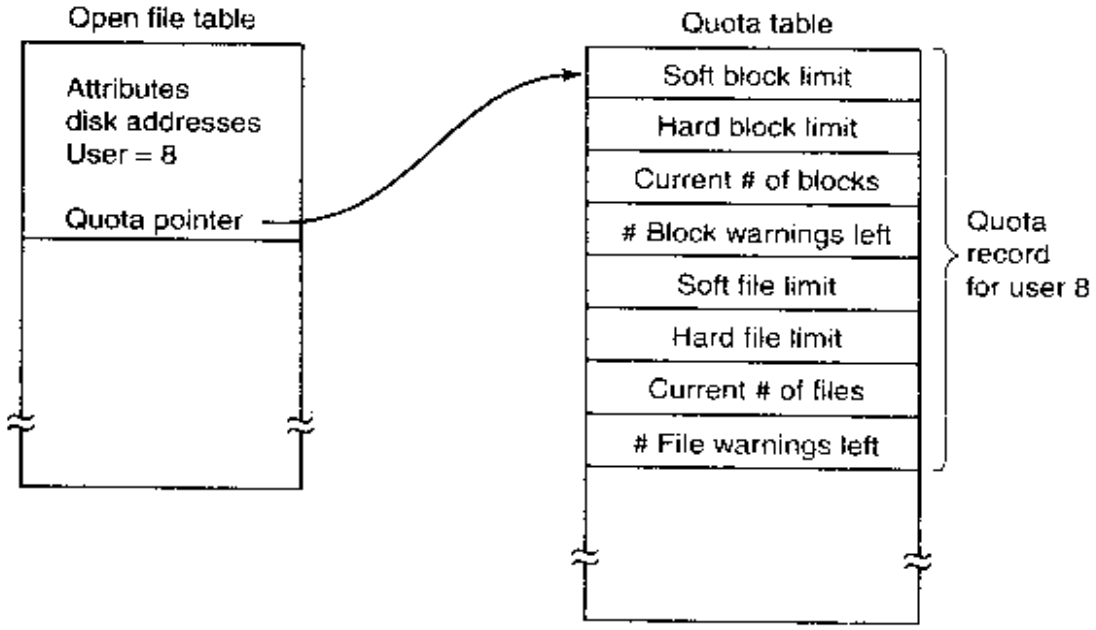

Figure 6-23. Quotas are kept track of on a per-user basis in a quota table.

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.

When a user attempts to log in, the system checks 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.

6.3.6 File System Reliability

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 the dealer (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, software, or rats gnawing on the floppy disks, restoring all the information will be difficult, time consuming, and in many cases, impossible. For the people whose programs, documents, customer files, tax records, 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. In this section we will look at some of the issues involved in safeguarding the file system.

Floppy disks are generally perfect when they leave the factory, but they can develop bad blocks during use. Hard disks frequently have bad blocks right from the start: it is just too expensive to manufacture them completely free of all defects. As we saw in Chap. 5, bad blocks are generally handled by the controller by replacing bad sectors with spares provided for that purpose. Despite this technique, there are other reliability issues which we will consider below.

Backups

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 a deathbed conversion. Companies, however, (usually) well understand the value of their data and generally do a backup at least once a day, usually to tape. Modern tapes hold tens or sometimes even hundreds of gigabytes and cost pennies per gigabyte. Nevertheless, Making backups is not quite as trivial as it sounds, so we will examine some of the related issues below.

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

1. Recover from disaster.
2. Recover from stupidity.

The first one covers getting the computer running again after a disk crash, fire, flood, or 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 manufacturers' CD-ROMs. 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 last 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 tape. However, with many compression algorithms, a single bad spot on the backup tape can foil the decompression algorithm and make an entire file or even an entire tape 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 tapes in his office and leaves it open and unguarded whenever he walks down the hall to get output from the printer. All a spy has to do is pop in for a second, put one tiny 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 tapes. For this reason, backup tapes should be kept off-site, but that introduces more security risks. For a thorough discussion of these and other practical administration issues, see (Nemeth et al., 2000). Below we will discuss only the technical issues involved in making file system backups.

Two strategies can be used for dumping a disk to tape: 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 tape 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 get 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 tape was block k on the disk.

A second concern is dumping bad blocks. If all bad blocks are remapped by the disk controller and hidden from the operating system as we described in Sec. 5.4.4, 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 during the dumping process.

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 tape 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. 6-24 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.

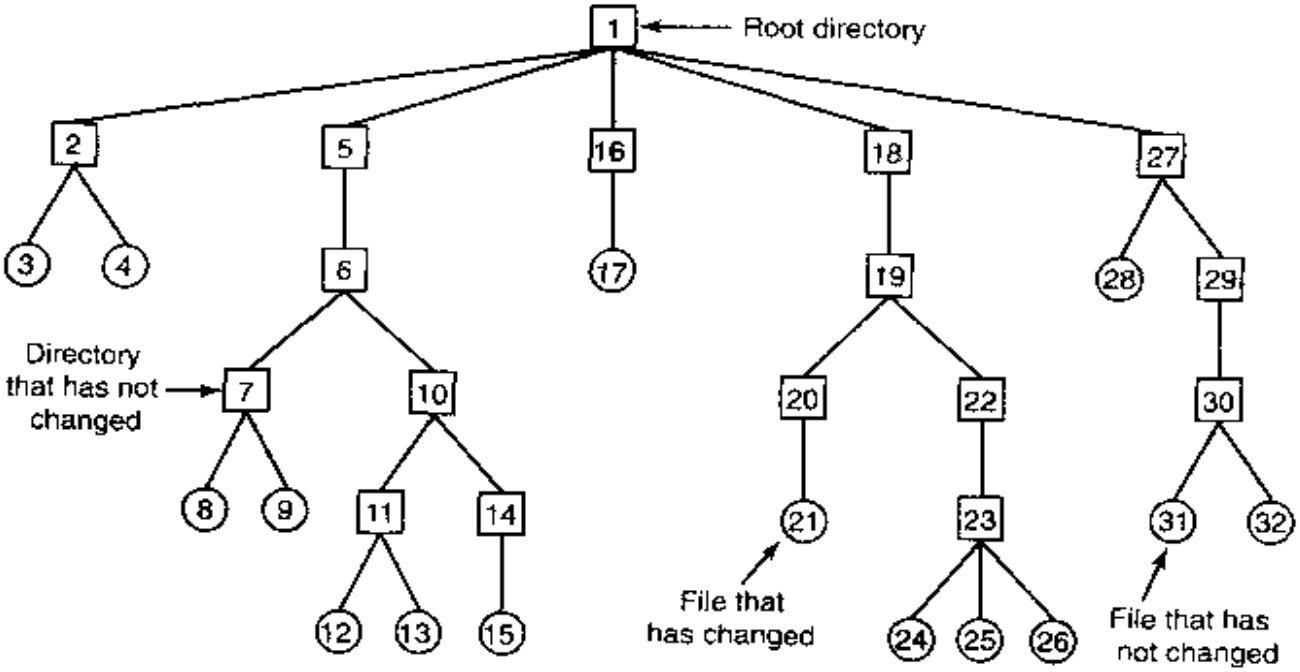

Figure 6-24. A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

This algorithm also dumps all directories (even unmodified ones) that lie on the path to a modified file or directory for two reasons. First, 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 stupidity). 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 the user wants to restore the file */usr/jhs/proj/nr3/plans/summary* However, 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, etc., correct, these directories must be present on the dump tape 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. 6-25(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. 6-25(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. In 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.

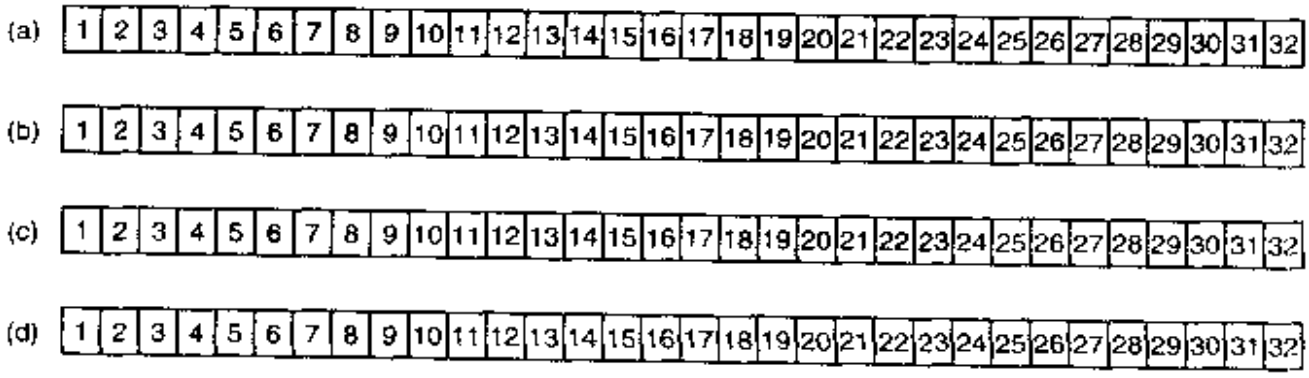

Figure 6-25. Bit maps used by the logical dumping algorithm.

At this point it is known which directories and files must be dumped. These are the ones marked in Fig. 6-25(b). Phase 3 consists of scanning the i-nodes in numerical order and dumping all the directories that are marked for dumping. These are shown in Fig. 6-25(c). Each directory is prefixed by the directory's attributes (owner, times, etc.) so they can be restored. Finally, in phase 4, the files marked in Fig. 6-25(d) are also dumped, again prefixed by their attributes. This completes the dump.

Restoring a file system from the dump tapes 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 tape, 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 legal 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 not be restored. Core files often have a large hole 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 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 (Chervenak et al., 1998; and Zwicky, 1991).

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 the problem of inconsistent file systems, most computers have a utility program that checks file system consistency. For example, UNIX has *fsck* and Windows has *scandisk*. This utility can be run whenever the system is booted, especially after a crash. The description below tells how *fsck* works. *Scandisk* 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 valid. All file system checkers verify each file system (disk partition) independently of the other ones.

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. 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. 6-26(a). However, as a result of a crash, the tables might look like Fig. 6-26(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 do 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.

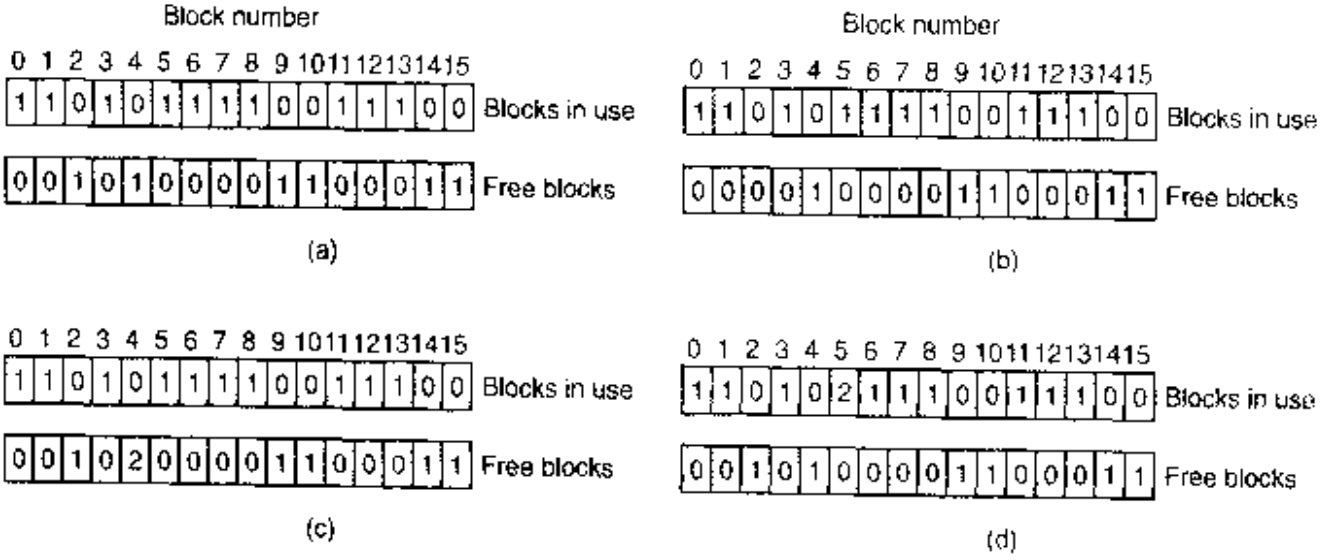

Figure 6-26. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

Another situation that might occur is that of Fig. 6-26(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. 6-26(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 file 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 it 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 0007, 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*. In MS-DOS and some other systems, when a file is removed, all that happens is that a bit is set in the directory or i-node marking the file as removed. No disk blocks are returned to the free list until they are actually needed. Thus, if the user discovers the error immediately, it is possible to run a special utility program that "unremoves" (i.e., restores) the removed files. In Windows, files that are removed are placed in the recycle bin, from which they can later be retrieved if need be. Of course, no storage is reclaimed until they are actually deleted from this directory.

6.3.7 File System Performance

Access to disk is much slower than access to memory. Reading a memory word might take 10 nsec. Reading from a hard disk might proceed at 10 MB/sec, which is forty 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. 6-27. 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 the collision chain can be followed.

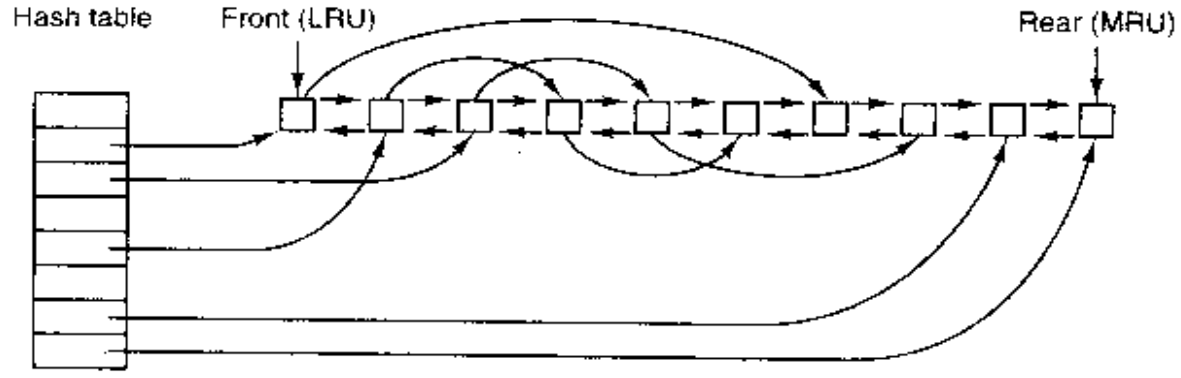

Figure 6-27. The buffer cache data structures.

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. 6-27, 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 of this list. 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 very 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.

The MS-DOS way is to write every modified block to disk as soon as it has been written. 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. MS-DOS will make 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 (floppy) 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 MS-DOS, 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 MS-DOS started out in the floppy disk world. As hard disks became the norm, the UNIX approach, with its better efficiency, became the norm, and is also used now on Windows for hard disks.

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 only works for files that are 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 it 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 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, as they are needed. 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 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 a 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 equivalent to i-nodes is that reading even a short file requires two disk accesses: one for the i-node and one for the block. The usual i-node placement is shown in Fig. 6-28(a). Here all the i-nodes are near the beginning of the disk, so the average distance between an i-node and its blocks will be about half the number of cylinders, requiring long seeks.

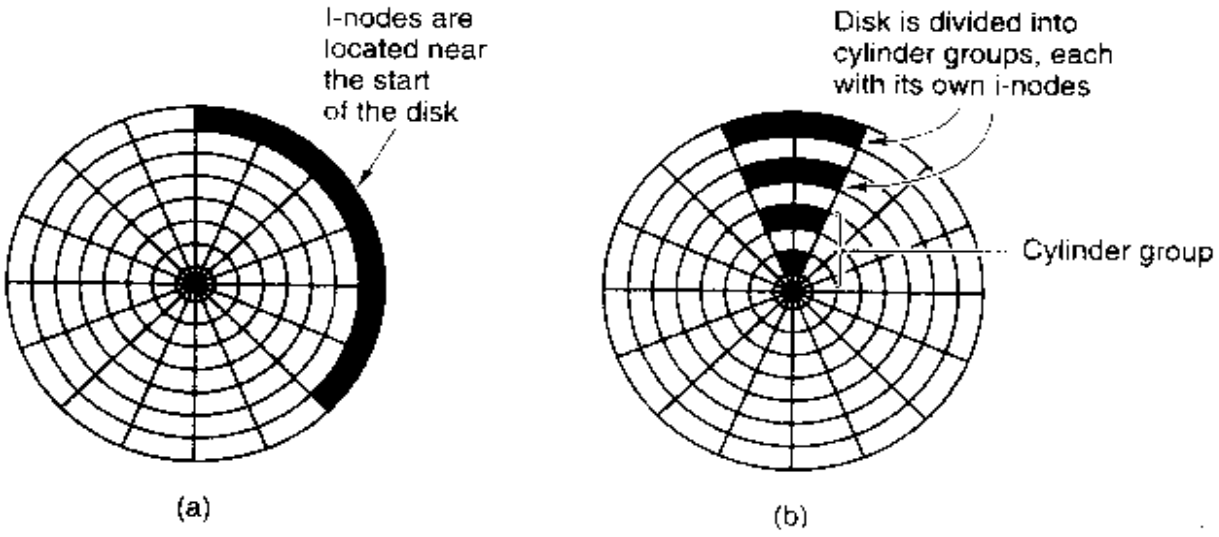

Figure 6-28. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

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. 6-28(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.

6.3.8 Log-Structured File Systems

Changes in technology are putting pressure on current file systems. In particular, CPUs keep getting faster, 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 means that a performance bottleneck is arising in many 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 (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 percent.

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 re-implement 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 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 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 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.

6.4 EXAMPLE FILE SYSTEMS

In the following sections we will discuss several example file systems, ranging from quite simple to highly sophisticated. Since modern UNIX file systems and Windows 2000's native file system are covered in the chapter on UNIX (Chap. 10) and the chapter on Windows 2000 (Chap. 11) we will not cover those systems here. We will, however, examine their predecessors below.

6.4.1 CD-ROM File Systems

As our first example of a file system, let us consider the file systems used on CD-ROMs. These systems are particularly simple because they were designed for write-once media. Among other things, for example, they have no provision for keeping track of free blocks because on a CD-ROM files cannot be freed or added after the disk has been manufactured. Below we will take a look at the main CD-ROM file system type and two extensions to it.

The ISO 9660 File System

The most common standard for CD-ROM file systems was adopted as an International Standard in 1988 under the name **ISO 9660**. Virtually every CD-ROM currently on the market is compatible with this standard, sometimes with

the extensions to be discussed below. One of the goals of this standard was to make every CD-ROM readable on every computer, independent of the byte ordering used and independent of the operating system used. As a consequence, some limitations were placed on the file system to make it possible for the weakest operating systems then in use (such as MS-DOS) to read it.

CD-ROMs do not have concentric cylinders the way magnetic disks do. Instead there is a single continuous spiral containing the bits in a linear sequence (although seeks across the spiral are possible). The bits along the spiral are divided into logical blocks (also called logical sectors) of 2352 bytes. Some of these are for preambles, error correction, and other overhead. The payload portion of each logical block is 2048 bytes. When used for music, CDs have leadins, leadouts, and intertrack gaps, but these are not used for data CD-ROMs. Often the position of a block along the spiral is quoted in minutes and seconds. It can be converted to a linear block number using the conversion factor of 1 sec = 75 blocks.

ISO 9660 supports CD-ROM sets with as many as $2^{16} - 1$ CDs in the set. The individual CD-ROMs may also be partitioned into logical volumes (partitions). However, below we will concentrate on ISO 9660 for a single unpartitioned CD-ROM.

Every CD-ROM begins with 16 blocks whose function is not defined by the ISO 9660 standard. A CD-ROM manufacturer could use this area for providing a bootstrap program to allow the computer to be booted from the CD-ROM, or for some other purpose. Next comes one block containing the **primary volume descriptor**, which contains some general information about the CD-ROM. Among this information are the system identifier (32 bytes), volume identifier (32 bytes), publisher identifier (128 bytes), and data preparer identifier (128 bytes). The manufacturer can fill in these fields in any desired way, except that only upper case letters, digits, and a very small number of punctuation marks may be used to ensure cross-platform compatibility.

The primary volume descriptor also contains the names of three files, which may contain the abstract, copyright notice, and bibliographic information, respectively. In addition, certain key numbers are also present, including the logical block size (normally 2048, but 4096, 8192, and larger powers of two are allowed in certain cases), the number of blocks on the CD-ROM, and the creation and expiration dates of the CD-ROM. Finally, the primary volume descriptor also contains a directory entry for the root directory, telling where to find it on the CD-ROM (i.e., which block it starts at). From this directory, the rest of the file system can be located.

In addition to the primary volume descriptor, a CD-ROM may contain a supplementary volume descriptor. It contains similar information to the primary, but that will not concern us here.

The root directory, and all other directories for that matter, consists of a variable number of entries, the last of which contains a bit marking it as the final one.

The directory entries themselves are also variable length. Each directory entry consists of 10 to 12 fields, some of which are in ASCII and others of which are numerical fields in binary. The binary fields are encoded twice, once in little-endian format (used on Pentiums, for example) and once in big-endian format (used on SPARCs, for example). Thus a 16-bit number uses 4 bytes and a 32-bit number uses 8 bytes. The use of this redundant coding was necessary to avoid hurting anyone's feelings when the standard was developed. If the standard had dictated little endian, then people from companies with big-endian products would have felt like second-class citizens and would not have accepted the standard. The emotional content of a CD-ROM can thus be quantified and measured exactly in kilobytes/hour of wasted space.

The format of an ISO 9660 directory entry is illustrated in Fig. 6-29. Since directory entries have variable lengths, the first field is a byte telling how long the entry is. This byte is defined to have the high-order bit on the left to avoid any ambiguity.

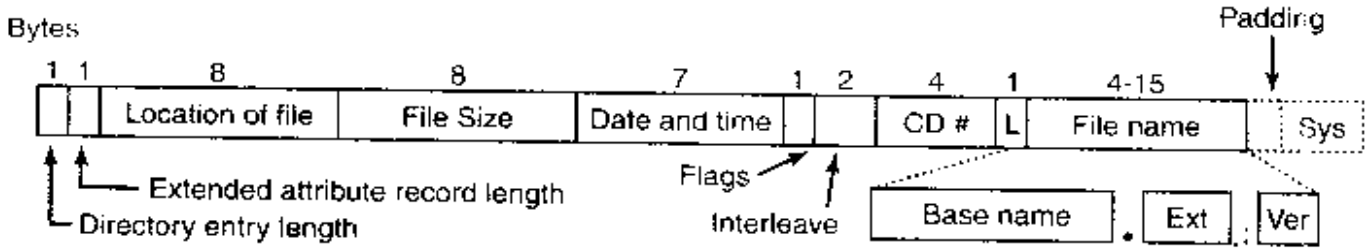

Figure 6-29. The ISO 9660 directory entry.

Directory entries may optionally have an extended attributes. If this feature is used for a directory entry, the second byte tells how long the extended attributes are.

Next comes the starting block of the file itself. Files are stored as contiguous runs of blocks, so a file's location is completely specified by the starting block and the size, which is contained in the next field.

The date and time that the CD-ROM was recorded is stored in the next field, with separate bytes for the year, month, day, hour, minute, second, and time zone. Years begin to count at 1900, which means that CD-ROMs will suffer from a Y2156 problem because the year following 2155 will be 1900. This problem could have been delayed by defining the origin of time to be 1988 (the year the standard was adopted). Had that been done, the problem would have been postponed until 2244. Every 88 extra years helps.

The *Flags* field contains a few miscellaneous bits, including one to hide the entry in listings (a feature copied from MS-DOS), one to distinguish an entry that is a file from an entry that is a directory, one to enable the use of the extended attributes, and one to mark the last entry in a directory. A few other bits are also

present in this field but they will not concern us here. The next field deals with interleaving pieces of files in a way that is not used in the simplest version of ISO 9660, so we will not consider it further.

The next field tells which CD-ROM the file is located on. It is permitted that a directory entry on one CD-ROM refers to a file located on another CD-ROM in the set. In this way it is possible to build a master directory on the first CD-ROM that lists all the files on all the CD-ROMs in the complete set.

The field marked *L* in Fig. 6-29 gives the size of the file name in bytes. It is followed by the file name itself. A file name consists of a base name, a dot, an extension, a semicolon, and a binary version number (1 or 2 bytes). The base name and extension may use upper case letters, the digits 0–9, and the underscore character. All other characters are forbidden to make sure that every computer can handle every file name. The base name can be up to eight characters; the extension can be up to three characters. These choices were dictated by the need to be MS-DOS compatible. A file name may be present in a directory multiple times, as long as each one has a different version number.

The last two fields are not always present. The *Padding* field is used to force every directory entry to be an even number of bytes, to align the numeric fields of subsequent entries on 2-byte boundaries. If padding is needed, a 0 byte is used. Finally, we have the *System use* field. Its function and size are undefined, except that it must be an even number of bytes. Different systems use it in different ways. The Macintosh keeps Finder flags here, for example.

Entries within a directory are listed in alphabetical order except for the first two entries. The first entry is for the directory itself. The second one is for its parent. In this respect, these entries are similar to the UNIX . and .. directory entries. The files themselves need not be in directory order.

There is no explicit limit to the number of entries in a directory. However, there is a limit to the depth of nesting. The maximum depth of directory nesting is eight.

ISO 9660 defines what are called three levels. Level 1 is the most restrictive and specifies that file names are limited to 8 + 3 characters as we have described, and also requires all files to be contiguous as we have described. Furthermore, it specifies that directory names be limited to eight characters with no extensions. Use of this level maximizes the chances that a CD-ROM can be read on every computer.

Level 2 relaxes the length restriction. It allows files and directories to have names of up to 31 characters, but still from the same set of characters.

Level 3 uses the same name limits as level 2, but partially relaxes the assumption that files have to be contiguous. With this level, a file may consist of several sections, each of which is a contiguous run of blocks. The same run may occur multiple times in a file and may also occur in two or more files. If large chunks of data are repeated in several files, level 3 provides some space optimization by not requiring the data to be present multiple times.

Rock Ridge Extensions

As we have seen, ISO 9660 is highly restrictive in several ways. Shortly after it came out, people in the UNIX community began working on an extension to make it possible to represent UNIX file systems on a CD-ROM. These extensions were named Rock Ridge, after a town in the Gene Wilder movie *Blazing Saddles*, probably because one of the committee members liked the film.

The extensions use the *System use* field in order to make Rock Ridge CD-ROMs readable on any computer. All the other fields retain their normal ISO 9660 meaning. Any system not aware of the Rock Ridge extensions just ignores them and sees a normal CD-ROM.

The extensions are divided up into the following fields:

1. PX - POSIX attributes.
2. PN - Major and minor device numbers.
3. SL - Symbolic link.
4. NM - Alternative name.
5. CL - Child location.
6. PL - Parent location.
7. RE - Relocation.
8. TF - Time stamps.

The *PX* field contains the standard UNIX *rwxrwxrwx* permission bits for the owner, group, and others. It also contains the other bits contained in the mode word, such as the SETUID and SETGID bits, and so on.

To allow raw devices to be represented on a CD-ROM, the *PN* field is present. It contains the major and minor device numbers associated with the file. In this way, the contents of the */dev* directory can be written to a CD-ROM and later reconstructed correctly on the target system.

The *SL* field is for symbolic links. It allows a file on one file system to refer to a file on a different file system.

Probably the most important field is *NM*. It allows a second name to be associated with the file. This name is not subject to the character set or length restrictions of ISO 9660, making it possible to express arbitrary UNIX file names on a CD-ROM.

The next three fields are used together to get around the ISO 9660 limit of directories that may only be nested eight deep. Using them it is possible to specify that a directory is to be relocated, and to tell where it goes in the hierarchy. It is effectively a way to work around the artificial depth limit.

Finally, the *TF* field contains the three timestamps included in each UNIX i-node, namely the time the file was created, the time it was last modified, and the time it was last accessed. Together, these extensions make it possible to copy a UNIX file system to a CD-ROM and then restore it correctly to a different system.

Joliet Extensions

The UNIX community was not the only group that wanted a way to extend ISO 9660. Microsoft also found it too restrictive (although it was Microsoft's own MS-DOS that caused most of the restrictions in the first place). Therefore Microsoft invented some extensions that were called **Joliet**. They were designed to allow Windows file systems to be copied to CD-ROM and then restored, in precisely the same way that Rock Ridge was designed for UNIX. Virtually all programs that run under Windows and use CD-ROMs support Joliet, including programs that burn CD-recordables. Usually, these programs offer a choice between the various ISO 9660 levels and Joliet.

The major extensions provided by Joliet are

1. Long file names.
2. Unicode character set.
3. Directory nesting deeper than eight levels.
4. Directory names with extensions

The first extension allows file names up to 64 characters. The second extension enables the use of the Unicode character set for file names. This extension is important for software intended for use in countries that do not use the Latin alphabet, such as Japan, Israel, and Greece. Since Unicode characters are two bytes, the maximum file name in Joliet occupies 128 bytes.

Like Rock Ridge, the limitation on directory nesting is removed by Joliet. Directories can be nested as deeply as needed. Finally, directory names can have extensions. It is not clear why this extension was included, since Windows directories virtually never use extensions, but maybe some day they will.

6.4.2 The CP/M File System

The first personal computers (then called microcomputers) came out in the early 1980s. A popular early type used the 8-bit Intel 8080 CPU and had 4 KB of RAM and a single 8-inch floppy disk with a capacity of 180 KB. Later versions used the slightly fancier (but still 8-bit) Zilog Z80 CPU, had up to 64 KB of RAM, and had a whopping 720-KB floppy disk as the mass storage device. Despite the slow speed and small amount of RAM, nearly all of these machines ran a surprisingly powerful disk-based operating system, called **CP/M** (**Control**

Program for Microcomputers) (Golden and Pechura, 1986). This system dominated its era as much as MS-DOS and later Windows dominated the IBM PC world. Two decades later, it has vanished without a trace (except for a small group of diehard fans), which gives reason to think that systems that now dominate the world may be essentially unknown when current babies become college students (Windows what?).

It is worth taking a look at CP/M for several reasons. First, it was a historically very important system and was the direct ancestor of MS-DOS. Second, current and future operating system designers who think that a computer needs 32 MB just to boot the operating system could probably learn a lot about simplicity from a system that ran quite well in 16 KB of RAM. Third, in the coming decades, embedded systems are going to be extremely widespread. Due to cost, space, weight, and power constraints, the operating systems used, for example, in watches, cameras, radios, and cellular telephones, are of necessity going to be lean and mean, not unlike CP/M. Of course, these systems do not have 8-inch floppy disks, but they may well have electronic disks using flash memory, and building a CP/M-like file system on such a device is straightforward.

The layout of CP/M in memory is shown in Fig. 6-30. At the top of main memory (in RAM) is the BIOS, which contains a basic library of 17 I/O calls used by CP/M (in this section we will describe CP/M 2.2, which was the standard version when CP/M was at the height of its popularity). These calls read and write the keyboard, screen, and floppy disk.

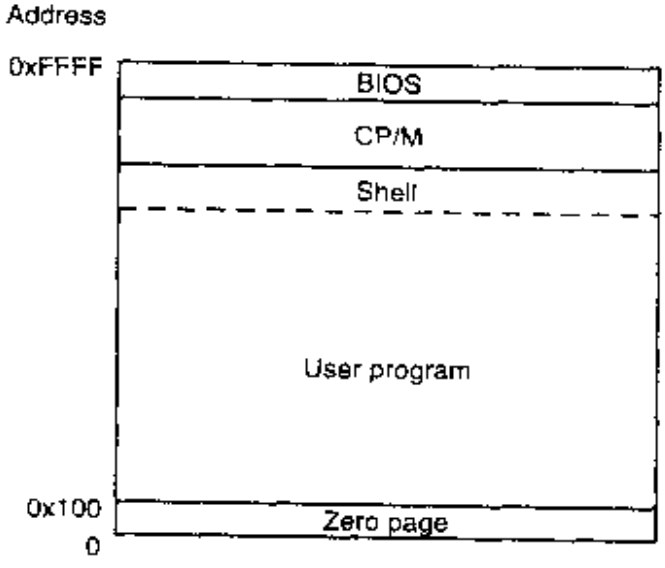

Figure 6-30. Memory layout of CP/M.

Just below the BIOS is the operating system proper. The size of the operating system in CP/M 2.2 is 3584 bytes. Amazing but true: a complete operating system in under 4 KB. Below the operating system comes the shell (command line processor), which chews up another 2 KB. The rest of memory is for user programs, except for the bottom 256 bytes, which are reserved for the hardware interrupt

vectors, a few variables, and a buffer for the current command line so user programs can get at it.

The reason for splitting the BIOS from CP/M itself (even though both are in RAM) is portability. CP/M interacts with the hardware only by making BIOS calls. To port CP/M to a new machine, all that is necessary is to port the BIOS there. Once that has been done, CP/M itself can be installed without modification.

A CP/M system has only one directory, which contains fixed-size (32-byte) entries. The directory size, although fixed for a given implementation, may be different in other implementations of CP/M All files in the system are listed in this directory. After CP/M boots, it reads in the directory and computes a bitmap containing the free disk blocks by seeing which blocks are not in any file. This bitmap, which is only 23 bytes for a 180-KB disk, is kept in memory during execution. At system shutdown time it is discarded, that is, not written back to the disk. This approach eliminates the need for a disk consistency checker (like *fsck*) and saves 1 block on the disk (percentually equivalent to saving 90 MB on a modern 16-GB disk).

When the user types a command, the shell first copies the command to a buffer in the bottom 256 bytes of memory. Then it looks up the program to be executed and reads it into memory at address 256 (above the interrupt vectors), and jumps to it. The program then begins running. It discovers its arguments by looking in the command line buffer. The program is allowed to overwrite the shell if it needs the memory. When the program finishes, it makes a system call to CP/M telling it to reload the shell (if it was overwritten) and execute it. In a nutshell, that is pretty much the whole CP/M story.

In addition to loading programs, CP/M provides 38 system calls, mostly file services, for user programs. The most important of these are reading and writing files. Before a file can be read, it must be opened. When CP/M gets an open system call, it has to read in and search the one and only directory. The directory is not kept in memory all the time to save precious RAM. When CP/M finds the entry, it immediately has the disk block numbers, since they are stored right in the directory entry, as are all the attributes. The format of a directory entry is given in Fig. 6-31.

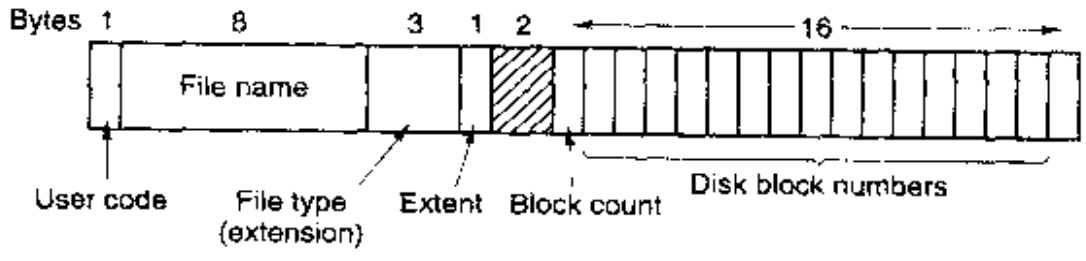

Figure 6-31. The CP/M directory entry format.

The fields in Fig. 6-31 have the following meanings. The *User code* field keeps track of which user owns the file. Although only one person can be logged

into a CP/M at any given moment, the system supports multiple users who take turns using the system. While searching for a file name, only those entries belonging to the currently logged-in user are checked. In effect, each user has a virtual directory without the overhead of managing multiple directories.

The next two fields give the name and extension of the file. The base name is up to eight characters; an optional extension of up to three characters may be present. Only upper case letters, digits, and a small number of special characters are allowed in file names. This 8 + 3 naming using upper case only was later taken over by MS-DOS.

The *Block count* field tells how many bytes this file entry contains, measured in units of 128 bytes (because I/O is actually done in 128-byte physical sectors). The last 1-KB block may not be full, so the system has no way to determine the exact size of a file. It is up to the user to put in some END-OF-FILE marker if desired. The final 16 fields contain the disk block numbers themselves. Each block is 1 KB, so that maximum file size is 16 KB. Note that physical I/O is done in units of 128-byte sectors and sizes are kept track of in sectors, but file blocks are allocated in units of 1 KB (8 sectors at a time) to avoid making the directory entry too large.

However, the CP/M designer realized that some files, even on a 180-KB floppy disk, might exceed 16 KB, so an escape hatch was built around the 16-KB limit. A file that is between 16 KB and 32 KB uses not one directory entry, but two. The first entry holds the first 16 blocks; the second entry holds the next 16 blocks. Beyond 32 KB, a third directory entry is used, and so on. The *Extent* field keeps track of the order of the directory entries so the system knows which 16 KB comes first, which comes second, and so on.

After an open call, the addresses of all the disk blocks are known, making read straightforward. The write call is also simple. It just requires allocating a free block from the bitmap in memory and then writing the block. Consecutive blocks on a file are not placed in consecutive blocks on the disk because the 8080 cannot process an interrupt and start reading the next block on time. Instead, interleaving is used to allow several blocks to be read on a single rotation.

CP/M is clearly not the last word in advanced file systems, but it is simple, fast, and can be implemented by a competent programmer in less than a week. For many embedded applications, it may be all that is needed.

6.4.3 The MS-DOS File System

To a first approximation, MS-DOS is a bigger and better version of CP/M. It runs only on Intel platforms, does not support multiprogramming, and runs only in the PC's real mode (which was originally the only mode). The shell has more features and there are more system calls, but the basic function of the operating system is still loading programs, handling the keyboard and screen, and managing the file system. It is the latter functionality that interests us here.

The MS-DOS file system was patterned closely on the CP/M file system, including the use of 8 + 3 (upper case) character file names. The first version (MS-DOS 1.0) was even limited to a single directory, like CP/M. However, starting with MS-DOS 2.0, the file system functionality was greatly expanded. The biggest improvement was the inclusion of a hierarchical file system in which directories could be nested to an arbitrary depth. This meant that the root directory (which still had a fixed maximum size) could contain subdirectories, and these could contain further subdirectories ad infinitem. Links in the style of UNIX were not permitted, so the file system formed a tree starting at the root directory.

It is common for different application programs to start out by creating a subdirectory in the root directory and put all their files there (or in subdirectories thereof), so that different applications do not conflict. Since directories are themselves just stored as files, there are no limits on the number of directories or files that may be created. Unlike CP/M, however, there is no concept of different users in MS-DOS. Consequently, the logged in user has access to all files.

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, like CP/M, they use a fixed-size 32-byte directory entry. The format of an MS-DOS directory entry is shown in Fig. 6-32. 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.

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 unique 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

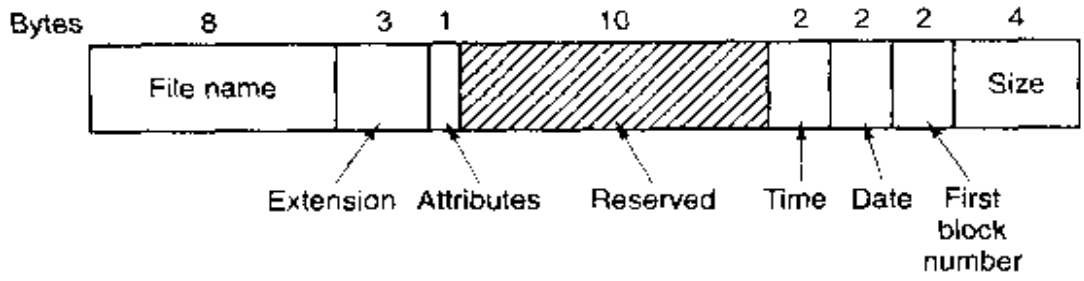

Figure 6-32. The MS-DOS directory entry.

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.

Unlike CP/M, which does not store the exact file size, MS-DOS does. Since a 32-bit number is used for the file size, 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 surprising large part of the entry (10 bytes) is unused.

Another way in which MS-DOS differs from CP/M is that it does not store a file's disk addresses in its directory entry, probably because the designers realized that large hard disks (by then common on minicomputers) would some day reach the MS-DOS world. Instead, 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. 6-14.

The FAT file system comes in three versions for MS-DOS: 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.

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 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. 6-33.

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 6-33. Maximum partition size for different block sizes. The empty boxes represent forbidden combinations.

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 bytes ties up 32 KB of disk space. If the average file is, say, 8 KB, then with a 32-KB block, ¾ 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.

6.4.4 The Windows 98 File System

The original release of Windows 95 used the MS-DOS file system, including the use of 8 + 3 character file names and the FAT-12 and FAT-16 file systems. Starting with the second release of Windows 95, file names longer than 8 + 3 characters were permitted. In addition, FAT-32 was introduced, primarily to allow larger disk partitions larger than 2 GB and disks larger than 8 GB, which were then available. Both the long file names and FAT-32 were used in Windows 98 in the same form as in the second release of Windows 95. Below we will describe these features of the Windows 98 file system, which have been carried forward into Windows Me as well.

Since long file names are more exciting for users than the FAT structure, let us look at them first. One way to introduce long file names would have been to just invent a new directory structure. The problem with this approach is that if Microsoft had done this, people who were still in the process of converting from Windows 3 to Windows 95 or Windows 98 could not have accessed their files from both systems. A political decision was made within Microsoft that files created using Windows 98 must be accessible from Windows 3 as well (for dual-boot machines). This constraint forced a design for handling long file names that was backward compatible with the old MS-DOS 8 + 3 naming system. Since such backward compatibility constraints are not unusual in the computer industry, it is worth looking in detail at how Microsoft accomplished this goal.

The effect of this decision to be backward compatible meant that the Windows 98 directory structure had to be compatible with the MS-DOS directory structure. As we saw, this structure is just a list of 32-byte entries as shown in

Fig. 6-32. This format came directly from CP/M (which was written for the 8080), which goes to show how long (obsolete) structures can live in the computer world.

However, it was possible to now allocate the 10 unused bytes in the entries of Fig. 6-32, and that was done, as shown in Fig. 6-34. This change has nothing to do with long names, but it is used in Windows 98, so it is worth understanding.

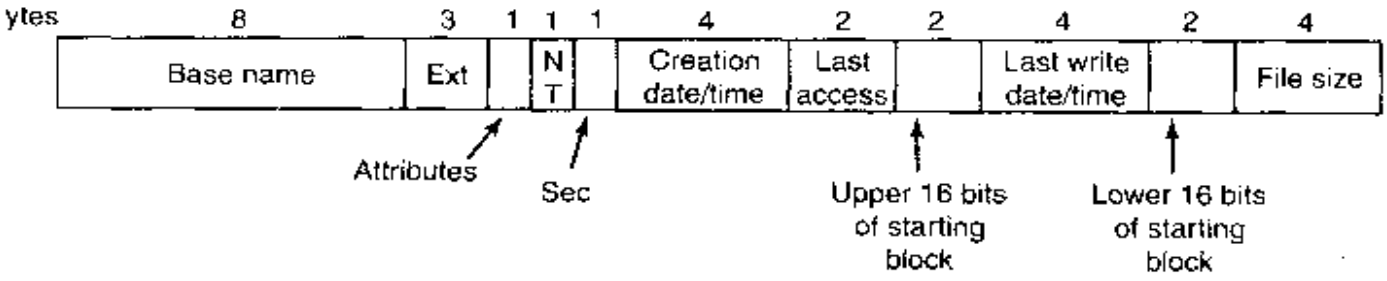

Figure 6-34. The extended MOS-DOS directory entry used in Windows 98.

The changes consist of the addition of five new fields where the 10 unused bytes used to be. The *NT* field is mostly there for some compatibility with Windows NT in terms of displaying file names in the correct case (in MS-DOS, all file names are upper case). The *Sec* field solves the problem that it is not possible to store the time of day in a 16-bit field. It provides additional bits so that the new *Creation time* field is accurate to 10 msec. Another new field is *Last access*, which stores the date (but not time) of the last access to the file. Finally, going to the FAT-32 file system means that block numbers are now 32 bits, so an additional 16-bit field is needed to store the upper 16 bits of the starting block number.

Now we come to the heart of the Windows 98 file system: how long file names are represented in a way that is backward compatible with MS-DOS. The solution chosen was to assign two names to each file: a (potentially) long file name (in Unicode, for compatibility with Windows NT), and an 8 + 3 name for compatibility with MS-DOS. Files can be accessed by either name. When a file is created whose name does not obey the MS-DOS naming rules (8 + 3 length, no Unicode, limited character set, no spaces, etc.), Windows 98 invents an MS-DOS name for it according to a certain algorithm. The basic idea is to take the first six characters of the name, convert them to upper case, if need be, and append ~1 to form the base name. If this name already exists, then the suffix ~2 is used, and so on. In addition, spaces and extra periods are deleted and certain special characters are converted to underscores. As an example, a file named *The time has come the walrus said* is assigned the MS-DOS name *THETIM~1*. If a subsequent file is created with the name *The time has come the rabbit said*, it is assigned the MS-DOS name *THETIM~2*, and so on.

Every file has an MS-DOS file name stored using the directory format of Fig. 6-34. If a file also has a long name, that name is stored in one or more directory entries directly preceding the MS-DOS file name. Each long-name entry holds up to 13 (Unicode) characters. The entries are stored in reverse order, with

the start of the file name just ahead of the MS-DOS entry and subsequent pieces before it. The format of each long-name entry is given in Fig. 6-35.

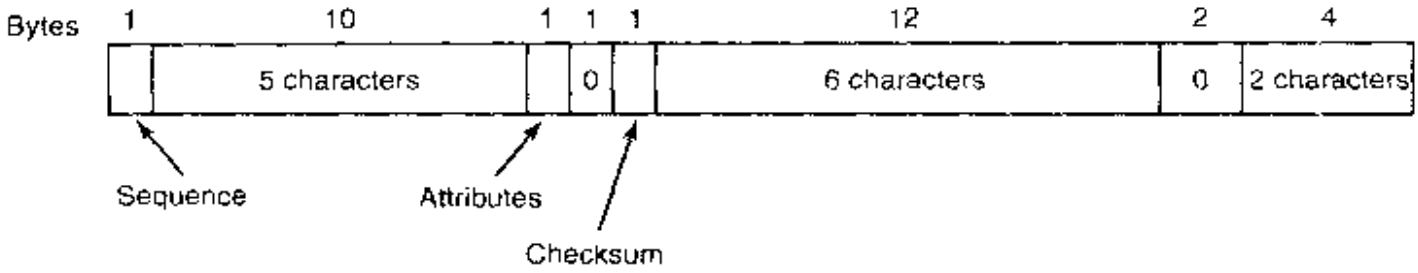

Figure 6-35. An entry for (part of) a long file name in Windows 98.

An obvious question is: "How does Windows 98 know whether a directory entry contains an MS-DOS file name or a (piece of a) long file name?" The answer lies in the *Attributes* field. For a long-name entry, this field has the value 0x0F, which represents an otherwise impossible combination of attributes. Old MS-DOS programs that read directories will just ignore it as invalid. Little do they know. The pieces of the name are sequenced using the first byte of the entry. The last part of the long name (the first entry in the sequence) is marked by adding 64 to the sequence number. Since only 6 bits are used for the sequence number, the theoretical maximum for file names is 63×13 or 819 characters. In fact they are limited to 260 characters for historical reasons.

Each long-name entry contains a *Checksum* field to avoid the following problem. First, a Windows 98 program creates a file with a long name. Second, the computer is rebooted to run MS-DOS or Windows 3. Third, an old program there then removes the MS-DOS file name from the directory but does not remove the long file name preceding it (because it does not know about it). Finally, some program creates a new file that reuses the newly-freed directory entry. At this point we have a valid sequence of long-name entries preceeding an MS-DOS file entry that has nothing to do with that file. The *Checksum* field allows Windows 98 to detect this situation by verifying that the MS-DOS file name following a long name does, in fact, belong to it. Of course, with only 1 byte being used, there is one chance in 256 that Windows 98 will not notice the file substitution.

To see an example of how long names work, consider the example of Fig. 6-36. Here we have a file called *The quick brown fox jumps over the lazy dog*. At 42-characters, it certainly qualifies as a long file name. The MS-DOS name constructed from it is *THEQUI~1* and is stored in the last entry.

Some redundancy is built into the directory structure to help detect problems in the event that an old Windows 3 program has made a mess of the directory. The sequence number byte at the start of each entry is not strictly needed since the 0x40 bit marks the first one, but it provides additional redundancy, for example. Also, the *Low* field of Fig. 6-36 (the lower half of the starting cluster) is 0 in all entries but the last one, again to avoid having old programs misinterpret it and

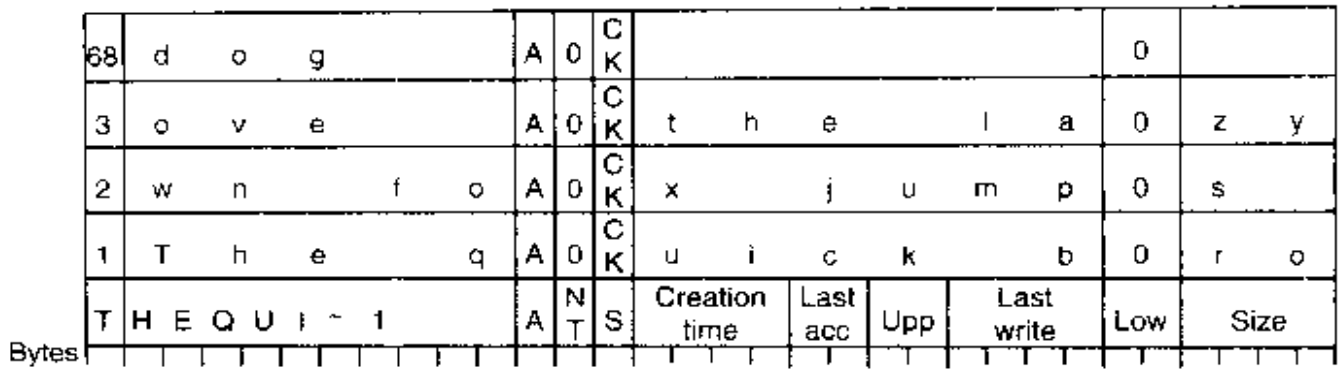

Figure 6-36. An example of how a long name is stored in Windows 98.

ruin the file system. The *NT* byte in Fig. 6-36 is used in NT and ignored in Windows 98. The *A* byte contains the attributes.

The implementation of the FAT-32 file system is conceptually similar to the implementation of the FAT-16 file system. However, instead of an array of 65,536 entries, there are as many entries as needed to cover the part of the disk with data on it. If the first million blocks are used, the table conceptually has 1 million entries. To avoid having all of them in memory at once, Windows 98 maintains a window into the table and keeps only in parts of it in memory at once.

6.4.5 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 modern versions 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 are 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. 6-15. 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. 6-37. These parameters limit the number of files per file system to 64K.

Like the i-node of Fig. 6-15, the UNIX i-nodes 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 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

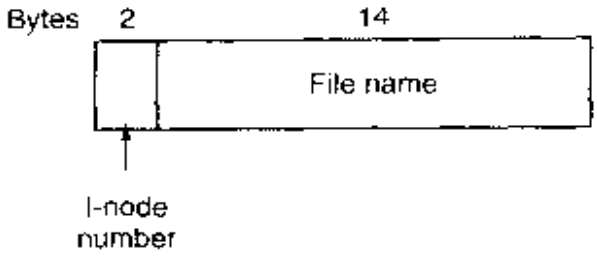

Figure 6-37. A UNIX V7 directory entry.

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. 6-15 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. 6-38.

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 in this case.

Then 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. 6-39.

Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of starting 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 number for the parent directory. Thus, a procedure looking up *../dick/prog.c* simply looks up .. in the working directory, finds the i-node number

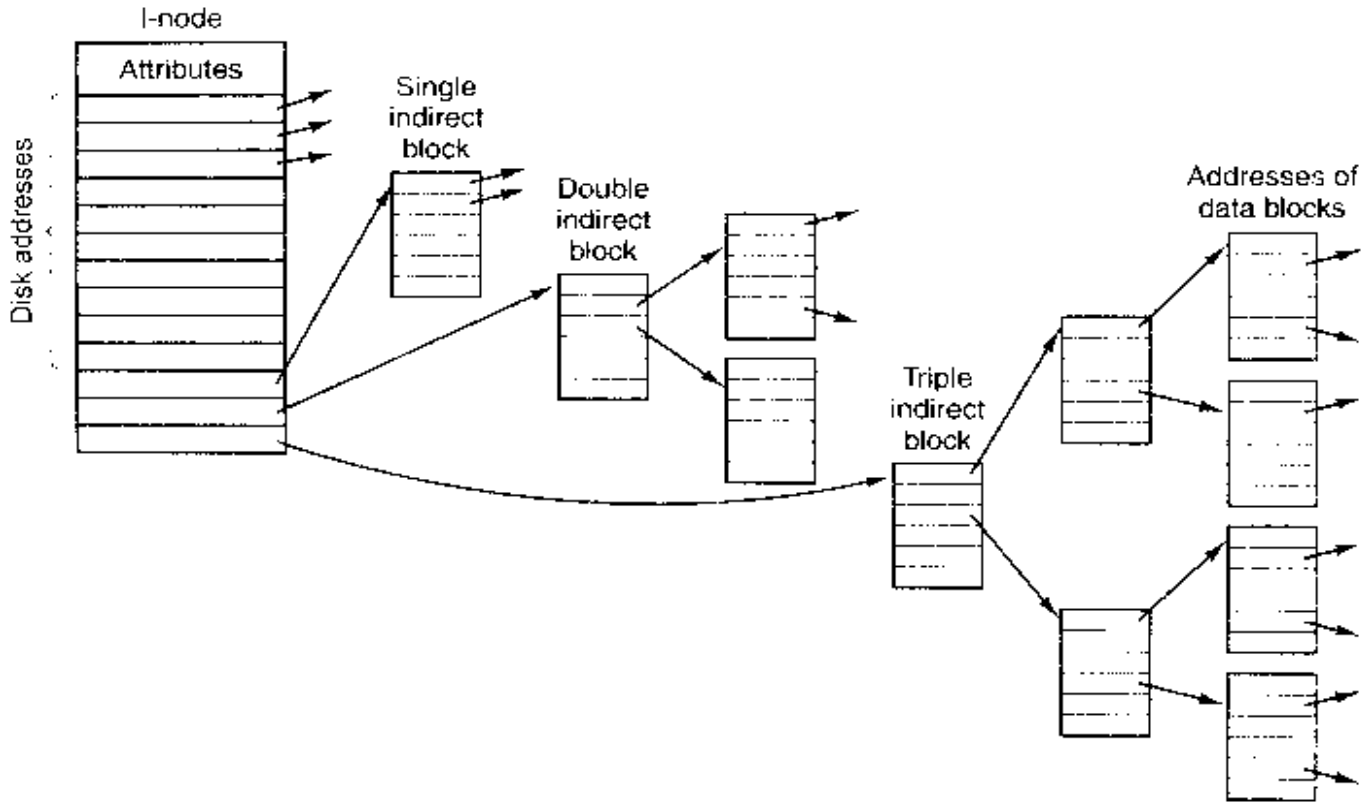

Figure 6-38. A UNIX i-node.

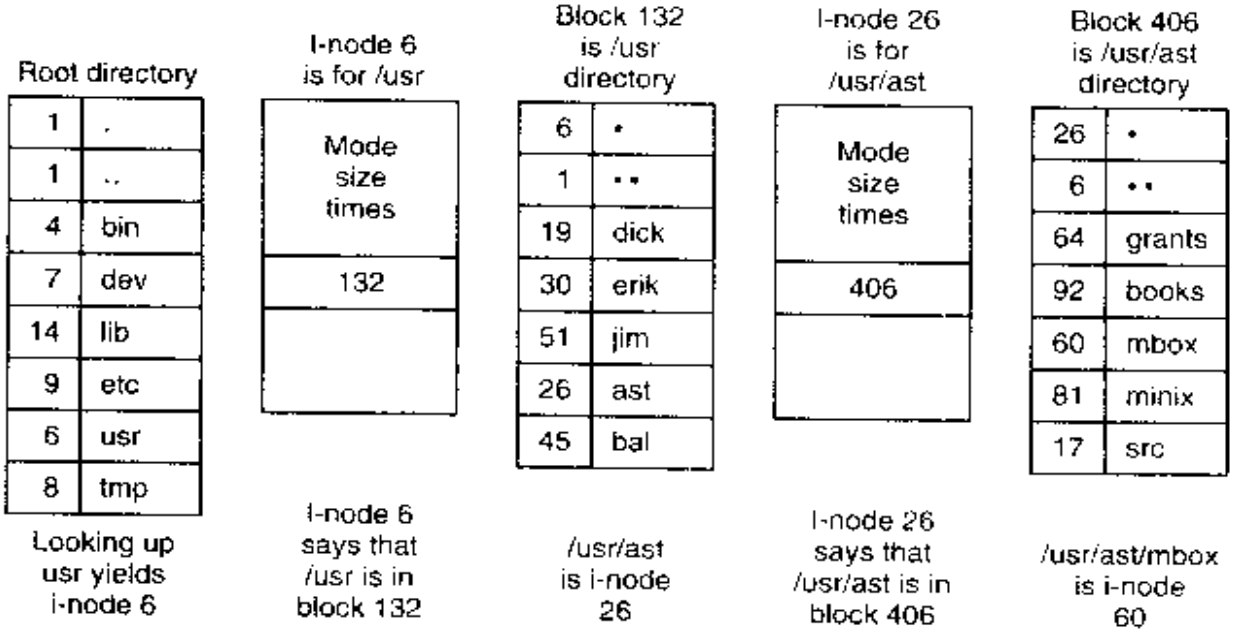

Figure 6-39. The steps in looking up */usr/ast/mbox*.

for the parent directory, and searches that directory for *dick*. 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.

6.5 RESEARCH ON FILE SYSTEMS

File systems have always attracted more research than other parts of the operating system and that is still the case. Some of this research relates to file system structure. Log-structured file systems and related topics are popular (Matthews et al., 1997; and Wang et al., 1999). The logical disk splits the file system into two distinct layers: the file system and the disk system (De Jonge et al., 1993). Building a file system from stackable layers is also a research topic (Heidemann and Popek, 1994).

Extensible file systems are analogous to the extensible kernels we saw in Chap 1. They allow new features to be added to the file system without having to redesign it from scratch (Karpovich et al., 1994; and Khalidi and Nelson, 1993).

Another research topic that has achieved a certain amount of popularity is making measurements of file system contents and use. People have measured what the file size distribution is, how long files live, whether all files are accessed equally, how reads compare to writes, and many other parameters (Douceur and Bolosky, 1999; Gill et al., 1994; Roselli and Lorch, 2000; and Vogels, 1999).

Other researchers have looked at file system performance and how to improve it using prefetching, caching, less copying, and other techniques. Usually, these researchers make measurements, find out where the bottlenecks are, eliminate at least one of them, and then make the measurements on the improved system to validate their results (Cao et al., 1995; Pai et al., 2000; and Patterson et al., 1995).

An area that many people rarely think about until disaster strikes is file system backup and recovery. Here, too, some new ideas have come up for better ways to do things (Chen et al., 1996; Devarakonda et al., 1996; and Hutchinson et al., 1999). Somewhat related to this is the question of what to do when a user removes a file—remove it or hide it? The Elephant file system, for example, never forgets (Santry et al., 1999a; and Santry et al., 1999b).

6.6 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 of 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, CP/M, MS-DOS, Windows 98, 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. Give 5 different path names for the file */etc/passwd*. *Hint*: Think about the directory entries "." and "..".
2. 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?
3. 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?
4. In Fig. 6-4, one of the attributes is the record length. Why does the operating system ever care about this?
5. Is the open system call in UNIX absolutely essential? What would the consequences be of not having it?
6. Systems that support sequential files always have an operation to rewind files. Do systems that support random access files need this too?
7. 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?
8. In some systems it is possible to map part of a file into memory. What restrictions must such systems impose? How is this partial mapping implemented?

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

10. 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.

11. Consider the directory tree of Fig. 6-10. If */usr/jim* is the working directory, what is the absolute path name for the file whose relative path name is *../ast/x*?

12. 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.

13. 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 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?

14. In light of the answer to the previous question, does compacting the disk ever make any sense?

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. How does MS-DOS implement random access to files?

17. Consider the i-node shown in Fig. 6-15. If it contains 10 direct addresses of 4 bytes each and all disk blocks are 1024 KB, what is the largest possible file?

18. 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. 6-15, how many bytes of data could be stored inside the i-node?

19. 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?

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

21. 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.

22. 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 6 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 5 blocks
(b) File *A* is deleted
(c) File *C* is written, using 8 blocks
(d) File *B* is deleted.

23. 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 answer for UNIX and the FAT-16 file system separately.

24. 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?

25. 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?

26. Suppose that file 21 in Fig. 6-24 was not modified since the last dump. In what way would the four bitmaps of Fig. 6-25 be different?

27. 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?

28. Consider Fig. 6-26. Is it possible that for some particular block number the counter in *both* lists have the value 2? How should this problem be corrected?

29. 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 from 0 to 1.0.

30. A floppy disk has 40 cylinders. A seek takes 6 msec per cylinder moved. If no attempt is made to put the blocks of a file close to each other, two blocks that are logically consecutive (i.e., follow one another in the file) will be about 13 cylinders apart, on the average. If, however, the operating system makes an attempt to cluster related blocks, the mean interblock distance can be reduced to 2 cylinders (for example). How long does it take to read a 100-block file in both cases, if the rotational latency is 100 msec and the transfer time is 25 msec per block?

31. Consider the idea behind Fig. 6-20, 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?

32. A certain file system uses 2-KB disk blocks. The median file size is 1 KB. If all files were exactly 1 KB, what fraction of the disk space would be wasted? Do you think the wastage for a real file system will be higher than this number or lower than it? Explain your answer.

33. CP/M was designed to run with a small floppy disk as the mass storage device. Suppose that it were to be ported to a modern computer with a large hard disk. What is the largest possible disk that could be supported without changing the size of the directory format shown in Fig. 6-31? The fields within the directory and other system parameters may be changed if necessary. What changes would you make?

34. 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 conditions?

35. 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?

36. A UNIX file system has 1-KB blocks and 4-byte disk addresses. What is the maximum file size if i-nodes contain 10 direct entries, and one single, double, and triple indirect entry each?

37. How many disk operations are needed to fetch the i-node for the file */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.

38. 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.

39. 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.

40. 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.).

41. 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.

42. 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.

43. Write CP/M in C or C++. Look on the Web to find information about it.