

- from a file. For a given input file, plot the number of page faults per 1000 memory references as a function of the number of page frames available.
42. Write a program that demonstrates the effect of TLB misses on the effective memory access time by measuring the per-access time it takes to stride through a large array.
- Explain the main concepts behind the program, and describe what you expect the output to show for some practical virtual memory architecture.
 - Run the program on some computer and explain how well the data fit your expectations.
 - Repeat part (b) but for an older computer with a different architecture and explain any major differences in the output.
43. Write a program that will demonstrate the difference between using a local page replacement policy and a global one for the simple case of two processes. You will need a routine that can generate a page reference string based on a statistical model. This model has N states numbered from 0 to $N-1$ representing each of the possible page references and a probability p_i associated with each state i representing the chance that the next reference is to the same page. Otherwise, the next page reference will be one of the other pages with equal probability.
- Demonstrate that the page reference string generation routine behaves properly for some small N .
 - Compute the page fault rate for a small example in which there is one process and a fixed number of page frames. Explain why the behavior is correct.
 - Repeat part (b) with two processes with independent page reference sequences and twice as many page frames as in Part (b).
 - Repeat part (c) but using a global policy instead of a local one. Also, contrast the per-process page fault rate with that of the local policy approach.

4

FILE SYSTEMS

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information 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:

- It must be possible to store a very large amount of information.
- The information must survive the termination of the process using it.
- Multiple processes must be able to access the information concurrently.

Magnetic disks have been used for years for this long-term storage. Tapes and optical disks are also used, but they have much lower performance. We will study

disks more in Chap. 5, but for the moment, it is sufficient to think of a disk as a linear sequence of fixed-size blocks and supporting two operations:

1. Read block k .
2. Write block k

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

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

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

and there are many more.

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

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

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

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

From the user's standpoint, the most important aspect of a file system is how it appears, 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 disk block are of no interest, although they are of great importance to the

designers of the file system. For this reason, we have structured the chapter as several sections. The first two are concerned with the user interface to files and directories, respectively. Then comes a detailed discussion of how the file system is implemented and managed. Finally, we give some examples of real file systems.

4.1 FILES

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

4.1.1 File Naming

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, called **FAT-16**, and thus inherit many of its properties, such as how file names are constructed. Windows 98 introduced some extensions to FAT-16, leading to **FAT-32**, but these two are quite similar. In addition, Windows NT, Windows 2000, Windows XP, and .WV support both FAT file systems, which are really obsolete now. These four NT-based operating systems have a native file system (NTFS) that has different properties (such as file names in Unicode). In this chapter, when we refer to the MS-DOS or FAT file systems, we mean FAT-16 and FAT-32 as used on Windows unless specified otherwise. We will discuss the FAT file systems later in this chapter and NTFS in Chap. 11, where we will examine Windows Vista in detail.

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 *homepage.html.zip*, where *.html* indicates a Web page in HTML and *.zip* indicates that the file (*homepage.html*) has been compressed using the *zip* program. Some of the more common file extensions and their meanings are shown in Fig. 4-1.

Extension	Meaning
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 4-1. 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.

4.1.2 File Structure

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

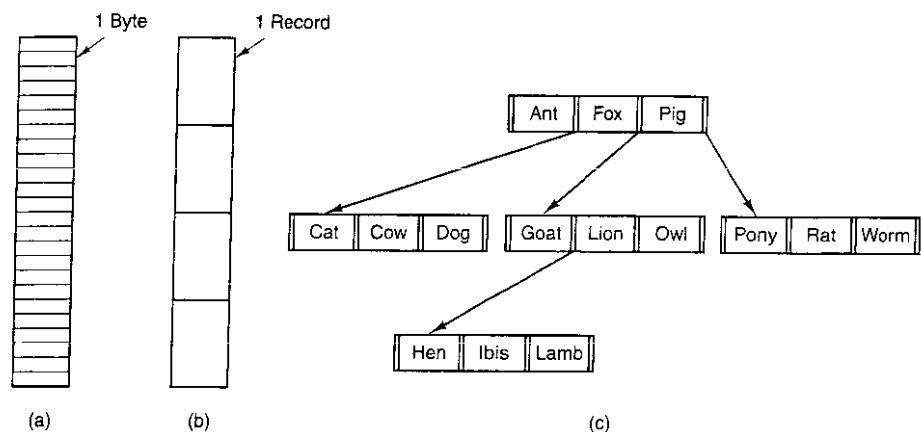


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Having the operating system regard files as nothing more than byte sequences provides the maximum 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. All versions of UNIX, MS-DOS, and Windows use this file model.

The first step up in structure is shown in Fig. 4-2(b). In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record. As a historical note, in decades gone by, when the 80-column punched card was 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 uses this model as its primary file system any more, but back in the days of 80-column punched cards and 132-character line printer paper this was a common model on mainframe computers.

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

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

4.1.3 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. 4-2 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, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of random junk. Usually, they have some internal structure known to programs that use them.

For example, in Fig. 4-3(a) we see a simple executable binary file taken from an early version of UNIX. Although technically the file is just a sequence of bytes, the operating system will 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.

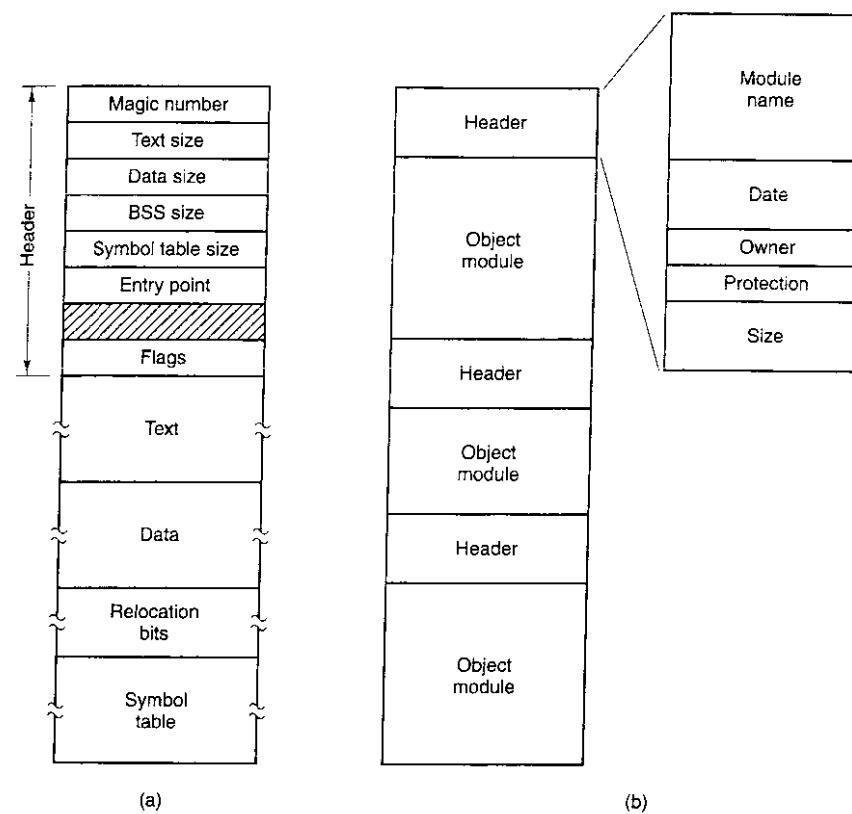


Figure 4-3. (a) An executable file. (b) An archive

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.

4.1.4 File Access

Early operating systems provided only one kind of file access: **sequential access**. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files could be rewound, however, so they could be read as often as needed. Sequential files were convenient when the storage medium was magnetic tape rather than disk.

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

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

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

4.1.5 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was last modified and the file's size. We will call these extra items the file's **attributes**. Some people call them **metadata**. The list of attributes varies considerably from system to system. The table of Fig. 4-4 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.

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

Figure 4-4. Some possible file attributes.

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

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

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

4.1.6 File Operations

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

1. **Create.** The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
2. **Delete.** When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. **Open.** Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
4. **Close.** When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.
5. **Read.** Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.

6. **Write.** Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
7. **Append.** This call is a restricted form of write. It can 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.

4.1.7 An Example Program Using File System Calls

In this section we will examine a simple UNIX program that copies one file from its source file to a destination file. It is listed in Fig. 4-5. 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,

```

/* 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) exit(0);                 /* no error on last read */
    else exit(5);                            /* error on last read */
}

```

Figure 4-5. A simple program to copy a file.

it will be created. The program must be called with exactly two arguments, both legal file names. The first is the source; the second is the output file.

The four `#include` statements near the top of the program cause a large number of definitions and function prototypes to be included in the program. These are needed to make the program conformant to the relevant international standards,

but will not concern us further. The next line is a function prototype for `main`, something required by ANSI C, but also not important for our purposes.

The first `#define` statement is a macro definition that defines the character string `BUF_SIZE` as a macro that expands into the number 4096. The program will read and write in chunks of 4096 bytes. It is considered good programming practice to give names to constants like this 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 status code is the only error reporting present in this program. A production version would normally print error messages as well.

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

Now comes the copy loop. It starts by trying to read in 4 KB of data to `buffer`. It does this by calling the library procedure `read`, which actually invokes the `read` system call. The first parameter identifies the file, the second gives the buffer, and the third tells how many bytes to read. The value assigned to `rd_count` gives the number of bytes actually read. Normally, this will be 4096, except if fewer bytes are remaining in the file. When the end of 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. 4-5. We will examine the Windows Vista calls in Chap. 11.

4.2 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.

4.2.1 Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory**, but since it is the only one, the name does not matter much. On 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. 4-6. Here the directory contains four files. The advantages of this scheme are its simplicity and the ability to locate files quickly—there is only one place to look, after all. It is often used on simple embedded devices such as telephones, digital cameras, and some portable music players.

4.2.2 Hierarchical Directory Systems

The single-level is adequate for simple dedicated applications (and was even used on the first personal computers), but for modern users with thousands of files, it would be impossible to find anything if all files were in a single directory.

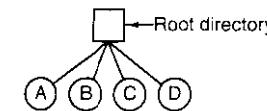


Figure 4-6. A single-level directory system containing four files.

Consequently, a way is needed to group related files together. A professor, for example, might have a collection of files that together form a book that 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.

What is needed is a hierarchy (i.e., a tree of directories). With this approach, there can be as many directories as are needed to group the files in natural ways. Furthermore, if multiple users share a common file server, as is the case on many company networks, each user can have a private root directory for his or her own hierarchy. This approach is shown in Fig. 4-7. Here, the directories *A*, *B*, and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

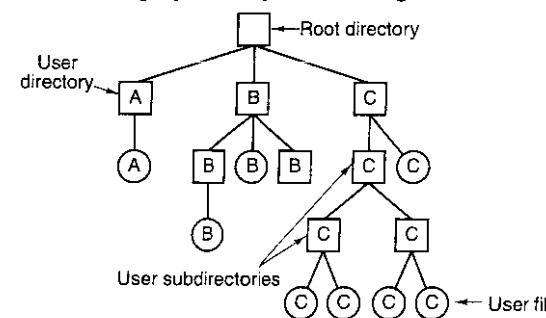


Figure 4-7. A hierarchical directory system.

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.

4.2.3 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an **absolute path name** consisting of the path from the

root directory to the file. As an example, the path `/usr/ast/mailbox` means that the root directory contains a subdirectory `usr`, which in turn contains a subdirectory `ast`, which contains the file `mailbox`. Absolute path names always start at the root directory and are unique. In UNIX the components of the path are separated by `/`. In Windows the separator is `\`. In MULTICS it was `>`. Thus the same path name would be written as follows in these three systems:

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

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 it 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 (except in the root directory, where it refers to itself). To see how these are used, consider the UNIX file tree of Fig. 4-8. A certain process has `/usr/ast` as its working directory. It can use .. to go higher up the tree. For example, it can copy the file `/usr/lib/dictionary` to its own directory using the command

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

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

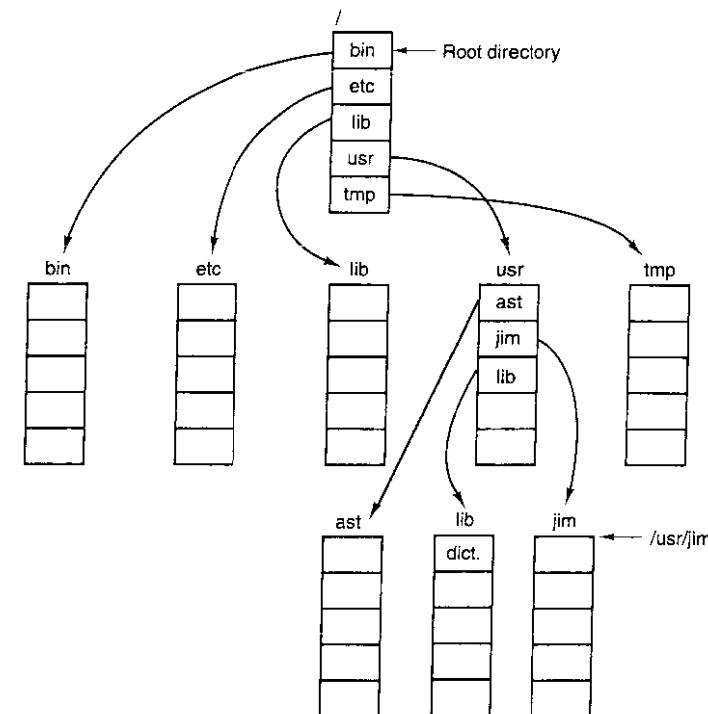


Figure 4-8. A UNIX directory tree.

The second argument (dot) names the current directory. When the `cp` command gets a directory name (including dot) as its last argument, it copies all the

files to that directory. Of course, a more normal way to do the copy would be to use the full absolute path name of the source file:

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

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

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

also works fine, as does

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

All of these do exactly the same thing.

4.2.4 Directory Operations

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

1. **Create.** A directory is created. It is empty except for dot and dotdot, which are put there automatically by the 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.

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

4.3 FILE SYSTEM IMPLEMENTATION

Now it is time to turn from the user's view of the file system to the implementor's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. 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.

4.3.1 File System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. 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.

Other than starting with a boot block, the layout of a disk partition varies a lot from file system to file system. Often the file system will contain some of the items shown in Fig. 4-9. The first one is the **superblock**. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file system type, the number of blocks in the file system, and other key administrative information.

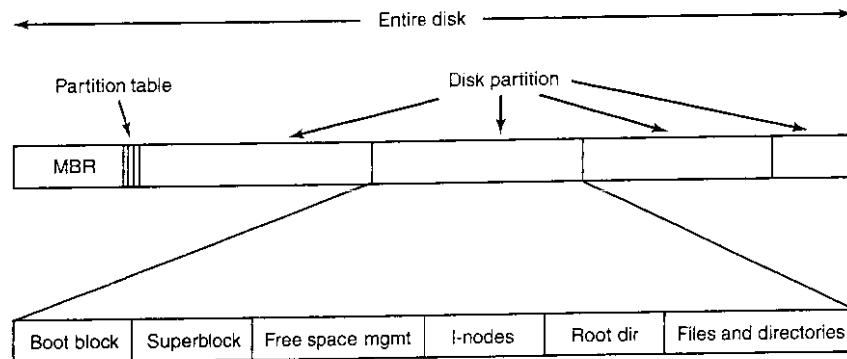


Figure 4-9. A possible file system layout.

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

4.3.2 Implementing Files

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

Contiguous Allocation

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

We see an example of contiguous storage allocation in Fig. 4-10(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 $3\frac{1}{2}$ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart. It has no actual significance in terms of storage.

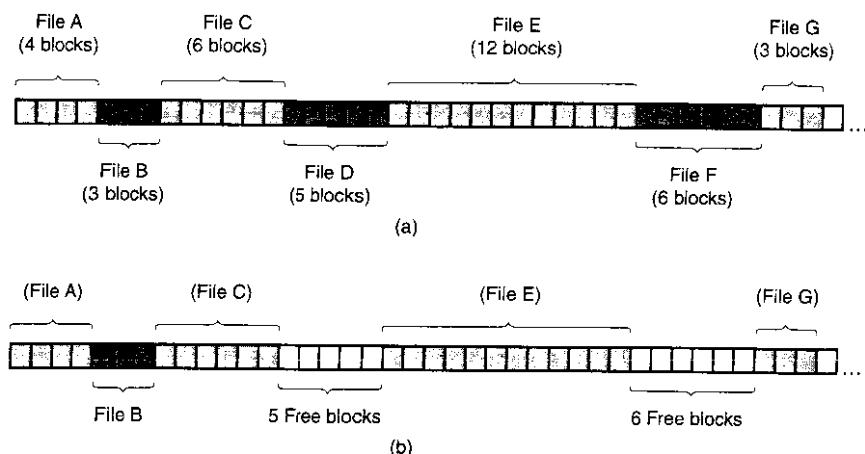


Figure 4-10. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

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

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 fairly significant drawback: over the course of time, the disk becomes fragmented. To see how this comes

about, examine Fig. 4-10(b). Here two files, *D* and *F*, have been removed. When a file is removed, its blocks are naturally freed, leaving a run of free blocks on the disk. The disk is not compacted on the spot to squeeze out the hole since, that would involve copying all the blocks following the hole, potentially millions of blocks. 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.

The situation with DVDs is a bit more complicated. In principle, a 90-min movie could be encoded as a single file of length about 4.5 GB, but the file system used, **UDF (Universal Disk Format)**, uses a 30-bit number to represent file length, which limits files to 1 GB. As a consequence, DVD movies are generally stored as three or four 1-GB files, each of which is contiguous. These physical pieces of the single logical file (the movie) are called **extents**.

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. 4-11. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

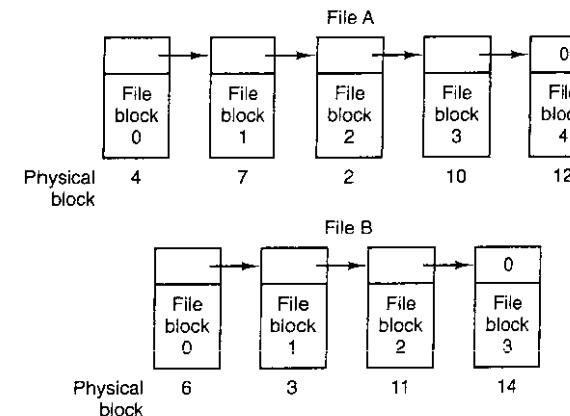


Figure 4-11. Storing a file as a linked list of disk blocks.

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

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

Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied 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.

Linked List Allocation Using a Table in Memory

Both disadvantages of the linked list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Figure 4-12 shows what the table looks like for the example of Fig. 4-11. In both figures,

we have two files. File A uses disk blocks 4, 7, 2, 10, and 12, in that order, and file B uses disk blocks 6, 3, 11, and 14, in that order. Using the table of Fig. 4-12, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., -1) that is not a valid block number. Such a table in main memory is called a **FAT** (File Allocation Table).

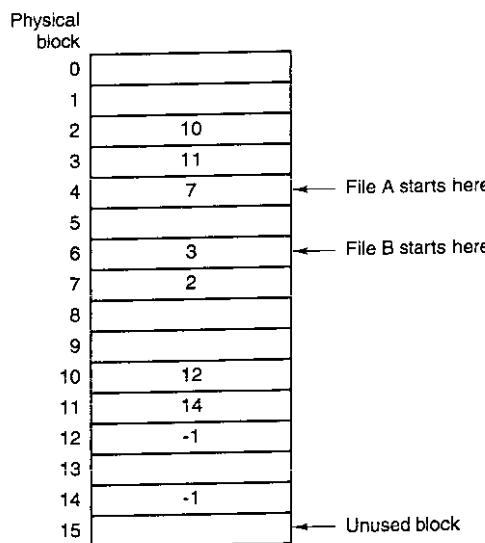


Figure 4-12. Linked list allocation using a file allocation table in main memory.

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

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 200-GB disk and a 1-KB block size, the table needs 200 million entries, one for each of the 200 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 600 MB or 800 MB of main memory all the time, depending on whether the system is optimized for space or time. Not wildly practical. Clearly the FAT idea does not scale well to large disks.

I-nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node** (**index-node**), which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Fig. 4-13. 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.

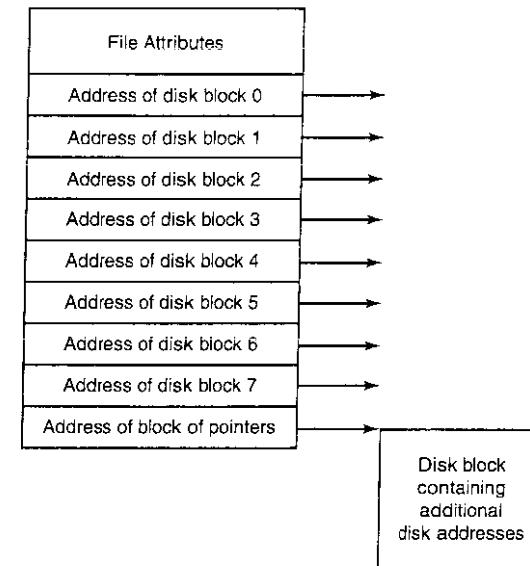


Figure 4-13. An example i-node.

This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the linked list of all disk blocks is proportional in size to the disk itself. If the disk has n blocks, the table needs n entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 10 GB or 100 GB or 1000 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 of a block containing more disk block addresses, as shown in Fig. 4-13. 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.

4.3.3 Implementing Directories

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

A closely related issue is where the attributes should be stored. Every file system maintains 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. 4-14(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.

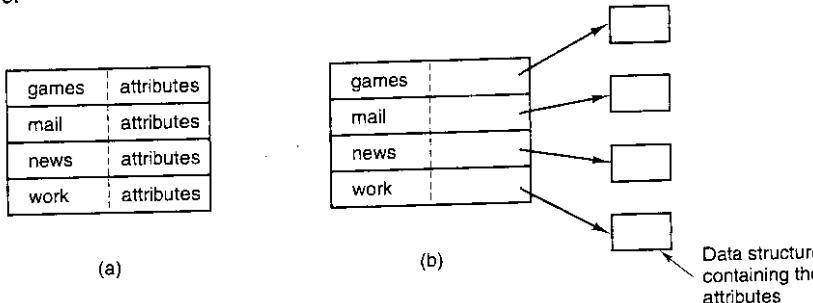


Figure 4-14. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

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. 4-14(b). As we shall see later, this method has some advantages over putting them in the directory entry. The two approaches shown in Fig. 4-14 correspond to Windows and UNIX, respectively, as we will see later.

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

The simplest approach is to set a limit on file name length, typically 255 characters, and then use one of the designs of Fig. 4-14 with 255 characters reserved for each file name. This approach is simple, but wastes a great deal of directory space, since few files have such long names. For efficiency reasons, a different structure is desirable.

One alternative is to give up the idea that all directory entries are the same size. With this method, each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes. This fixed-length header is followed by the actual file name, however long it may be, as shown in Fig. 4-15(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.

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. 4-15(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. 4-15(b) as they are in Fig. 4-15(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

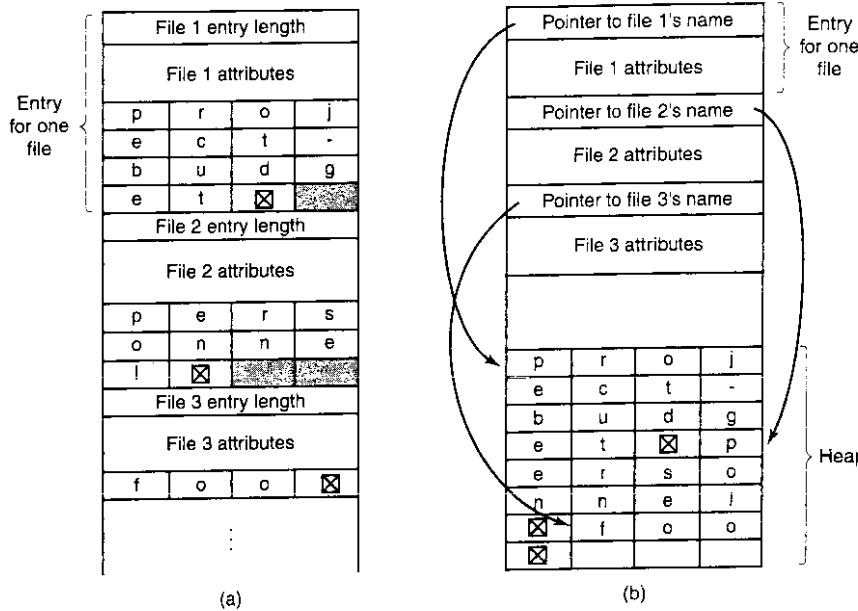


Figure 4-15. Two ways of handling long file names in a directory. (a) In-line
 (b) In a heap.

taking the remainder. Alternatively, the words comprising the file name can be added up and this quantity divided by n , or something similar.

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

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

Using a hash table has the advantage of much faster lookup, but the disadvantage of more complex administration. It is only really a serious candidate in systems where it is expected that directories will routinely contain hundreds or thousands of files.

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

4.3.4 Shared Files

When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Figure 4-16 shows the file system of Fig. 4-7 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.

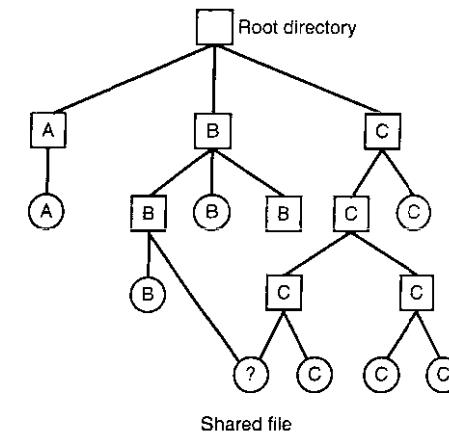


Figure 4-16. File system containing a shared file

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

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

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

Each of these methods has its drawbacks. In the first method, at the moment that *B* links to the shared file, the i-node records the file's owner as *C*. Creating a link does not change the ownership (see Fig. 4-17), but it does increase the link count in the i-node, so the system knows how many directory entries currently point to the file.

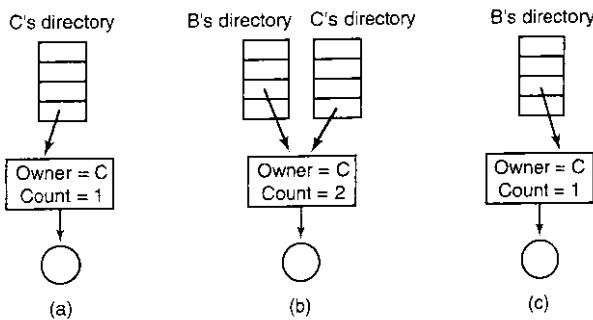


Figure 4-17. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

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

The only thing to do is remove *C*'s directory entry, but leave the i-node intact, with count set to 1, as shown in Fig. 4-17(c). We now have a situation in which *B* is the only user having a directory entry for a file owned by *C*. If the system does accounting or has quotas, *C* will continue to be billed for the file until *B* decides to remove it, if ever, at which time the count goes to 0 and the file is deleted.

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

The problem with symbolic links is the extra overhead required. The file containing the path must be read, then the path must be parsed and followed, component by component, until the i-node is reached. All of this activity may require a considerable number of extra disk accesses. Furthermore, an extra i-node is needed for each symbolic link, as is an extra disk block to store the path, although if the path name is short, the system could store it in the i-node itself, as a kind of

optimization. Symbolic links have the advantage that they can be used to link to files on machines anywhere in the world, by simply providing the network address of the machine where the file resides in addition to its path on that machine.

There is also another problem introduced by links, symbolic or otherwise. When links are allowed, files can have two or more paths. Programs that start at a given directory and find all the files in that directory and its subdirectories will locate a linked file multiple times. For example, a program that dumps all the files in a directory and its subdirectories onto a 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.

4.3.5 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- μ sec disk write is often preceded by a 10-msec seek and a 4-msec rotational delay. With these parameters, disk efficiency drops to a fraction of 1%.

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

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

The bookkeeping here is nontrivial, since when a file block is written back to a new segment, the i-node of the file (somewhere in the log) must be located, updated, and put into memory to be written out in the next segment. The i-node map must then be updated to point to the new copy. Nevertheless, it is possible to do the administration, and the performance results show that all this complexity is

worthy. Measurements given in the papers cited above show that LFS outperforms UNIX by an order of magnitude on small writes, while having a performance that is as good as or better than UNIX for reads and large writes.

4.3.6 Journaling File Systems

While log-structured file systems are an interesting idea, they are not widely used, in part due to their being highly incompatible with existing file systems. Nevertheless, one of the ideas inherent in them, robustness in the face of failure, can be easily applied to more conventional file systems. The basic idea here is to keep a log of what the file system is going to do before it does it, so that if the system crashes before it can do its planned work, upon rebooting the system can look in the log to see what was going on at the time of the crash and finish the job. Such file systems, called **journaling file systems**, are actually in use. Microsoft's NTFS file system and the Linux ext3 and ReiserFS file systems use journaling. Below we will give a brief introduction to this topic.

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

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

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

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

What the journaling file system does is first write a log entry listing the three actions to be completed. The log entry is then written to disk (and for good measure, possibly read back from the disk to verify its integrity). Only after the log entry has been written, do the various operations begin. After the operations

complete successfully, the log entry is erased. If the system now crashes, upon recovery the file system can check the log to see if any operations were pending. If so, all of them can be rerun (multiple times in the event of repeated crashes) until the file is correctly removed.

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

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

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

4.3.7 Virtual File Systems

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

In contrast, all modern UNIX systems make a very serious attempt to integrate multiple file systems into a single structure. A Linux system could have ext2 as the root file system, with an ext3 partition mounted on */usr* and a second hard disk with a ReiserFS file system mounted on */home* as well as an ISO 9660 CD-ROM

temporarily mounted on */mnt*. From the user’s point of view, there is a single file system hierarchy. That it happens to encompass multiple (incompatible) file systems is not visible to users or processes.

However, the presence of multiple file systems is very definitely visible to the implementation, and since the pioneering work of Sun Microsystems (Kleiman, 1986), most UNIX systems have used the concept of a **VFS (virtual file system)** to try to integrate multiple file systems into an orderly structure. The key idea is to abstract out that part of the file system that is common to all file systems and put that code in a separate layer that calls the underlying concrete file systems to actually manage the data. The overall structure is illustrated in Fig. 4-18. The discussion below is not specific to Linux or FreeBSD or any other version of UNIX, but gives the general flavor of how virtual file systems work in UNIX systems.

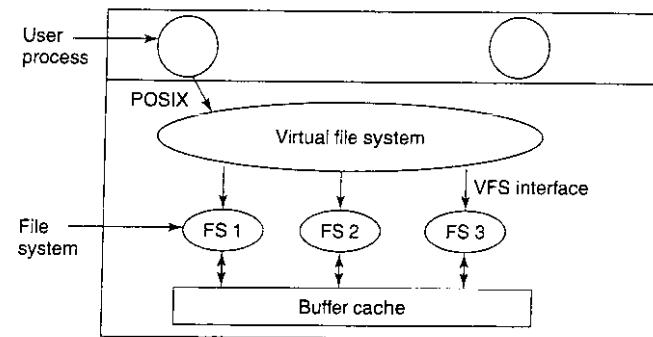


Figure 4-18. Position of the virtual file system.

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

The VFS also has a “lower” interface to the concrete file systems, which is labeled **VFS interface** in Fig. 4-18. This interface consists of several dozen function calls that the VFS can make to each file system to get work done. Thus to create a new file system that works with the VFS, the designers of the new file system must make sure that it supplies the function calls the VFS requires. An obvious example of such a function is one that reads a specific block from disk, puts it in the file system’s buffer cache, and returns a pointer to it. Thus the VFS has two distinct interfaces: the upper one to the user processes and the lower one to the concrete file systems.

While most of the file systems under the VFS represent partitions on a local disk, this is not always the case. In fact, the original motivation for Sun to build

the VFS was to support remote file systems using the **NFS** (**Network File System**) protocol. The VFS design is such that as long as the concrete file system supplies the functions the VFS requires, the VFS does not know or care where the data are stored or what the underlying file system is like.

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

To understand how the VFS works, let us run through an example chronologically. When the system is booted, the root file system is registered with the VFS. In addition, when other file systems are mounted, either at boot time or during operation, they, too must register with the VFS. When a file system registers, what it basically does is provide a list of the addresses of the functions the VFS requires, either as one long call vector (table) or as several of them, one per VFS object, as the VFS demands. Thus once a file system has registered with the VFS, the VFS knows how to, say, read a block from it—it simply calls the fourth (or whatever) function in the vector supplied by the file system. Similarly, the VFS then also knows how to carry out every other function the concrete file system must supply: it just calls the function whose address was supplied when the file system registered.

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

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

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

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

Later when the process does a `read` using the file descriptor, the VFS locates the v-node from the process and file descriptor tables and follows the pointer to the table of functions, all of which are addresses within the concrete file system on which the requested file resides. The function that handles `read` is now called and code within the concrete file system goes and gets the requested block. The VFS has no idea whether the data are coming from the local disk, a remote file system over the network, a CD-ROM, a USB stick, or something different. The data structures involved are shown in Fig. 4-19. Starting with the caller's process number and the file descriptor, successively the v-node, read function pointer, and access function within the concrete file system are located.

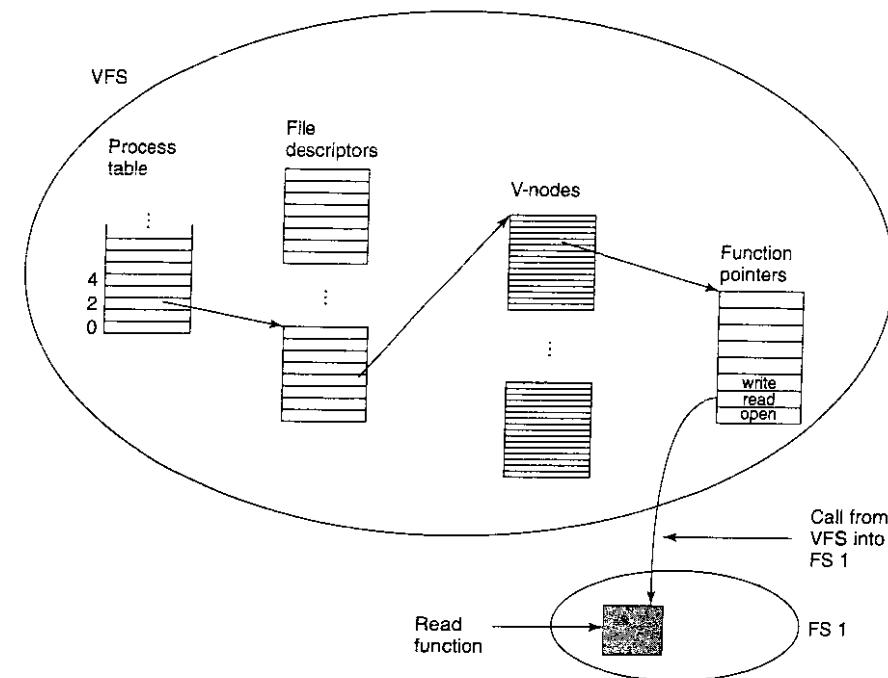


Figure 4-19. A simplified view of the data structures and code used by the VFS and concrete file system to do a `read`.

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

4.4 FILE SYSTEM MANAGEMENT AND OPTIMIZATION

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

4.4.1 Disk Space Management

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

As we have seen, storing a file 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 block size means that every file, even a 1-byte file, ties up an entire cylinder. It also means that small files waste a large amount of disk space. On the other hand, a small block size means that most files will span multiple blocks and thus need multiple seeks and rotational delays to read them, reducing performance. Thus if the allocation unit is too large, we waste space; if it is too small, we waste time.

Making a good choice requires having some information about the file size distribution. Tanenbaum et al. (2006) studied the file size distribution in the Computer Science Department of a large research university (the VU) in 1984 and then again in 2005, as well as on a commercial Web server hosting a political Website (www.electoral-vote.com). The results are shown in Fig. 4-20, where for each power-of-two file size, the percentage of all files smaller or equal to it is listed for each of the three data sets. For example, in 2005, 59.13% of all files at the VU were 4 KB or smaller and 90.84% of all files were 64 KB or smaller. The median file size was 2475 bytes. Some people may find this small size surprising.

Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

Length	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

Figure 4-20. Percentage of files smaller than a given size (in bytes).

What conclusions can we draw from these data? For one thing, with a block size of 1 KB, only about 30–50% of all files fit in a single block, whereas with a 4-KB block, the percentage of files that fit in a block goes up to the 60–70% range. Other data in the paper show that with a 4-KB block, 93% of the disk blocks are used by the 10% largest files. This means that wasting some space at the end of each small file hardly matters because the disk is filled up by a small number of large files (videos) and the total amount of space taken up by the small files hardly matters at all. Even doubling the space the smallest 90% of the files take up would be barely noticeable.

On the other hand, using a small block 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 1 MB per track, a rotation time of 8.33 msec, and an average seek time of 5 msec. The time in milliseconds to read a block of k bytes is then the sum of the seek, rotational delay, and transfer times:

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

The solid curve of Fig. 4-21 shows the data rate for such a disk as a function of block size. To compute the space efficiency, we need to make an assumption about the mean file size. For simplicity, let us assume that all files are 4 KB. Although this number is slightly larger than the data measured at the VU, students probably have more small files than would be present in a corporate data center,

so it might be a better guess on the whole. The dashed curve of Fig. 4-21 shows the space efficiency as a function of block size.

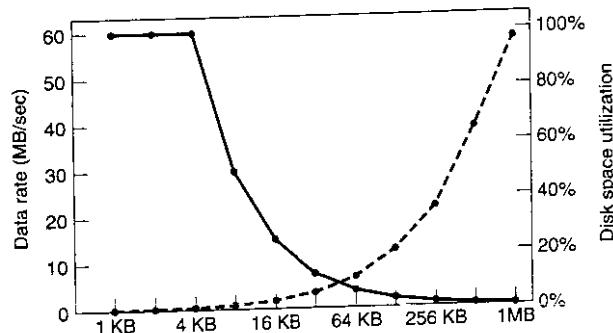


Figure 4-21. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All files are 4 KB.

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

Now consider space efficiency. With 4-KB files and 1-KB, 2-KB, or 4-KB blocks, files use 4, 2, and 1 block, respectively, with no wastage. With an 8-KB block and 4-KB files, the space efficiency drops to 50%, and with a 16-KB block it is down to 25%. In reality, few files are an exact multiple of the disk block size, so some space is always wasted in the last block of a file.

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

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 different data sets measurement techniques, and the year, these results are certainly compatible with the VU results.

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used, as shown in Fig. 4-22. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is required for the pointer to the next block.) Consider a 500-GB disk, which has about 488 million disk blocks. To store all these address at 255 per block requires about 1.9 million blocks. Generally, free blocks are used to hold the free list, so the storage is essentially free.

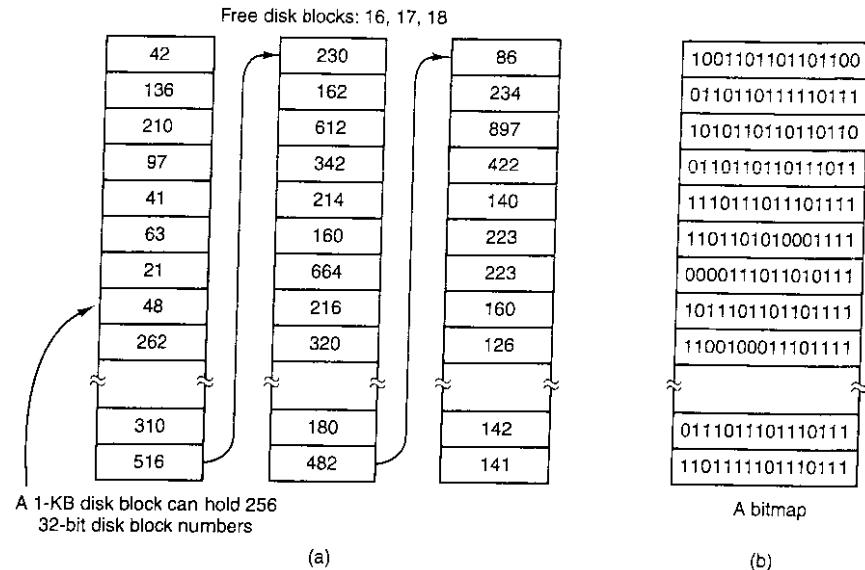


Figure 4-22. (a) Storing the free list on a linked list. (b) A bitmap.

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). For our example 500-GB disk, we need 488 million bits for the map, which requires just under 60,000 1-KB blocks to

store. 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.

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

This issue illustrates a problem operating system designers often have. There are multiple data structures and algorithms that can be used to solve a problem, but choosing the best one requires data that the designers do not have and will not have until the system is deployed and heavily used. And even then, the data may not be available. For example, our own measurements of file sizes at the VU in 1984 and 1995, the Website data, and the Cornell data are only four samples. While a lot better than nothing, we have little idea if they are also representative of home computers, corporate computers, government computers, and others. With some effort we might have been able to get a couple of samples from other kinds of computers, but even then it would be foolish to extrapolate to all computers of the kind measured.

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

Under certain circumstances, this method leads to unnecessary disk I/O. Consider the situation of Fig. 4-23(a), in which the block of pointers in memory has room for only two more entries. If a three-block file is freed, the pointer block overflows and has to be written to disk, leading to the situation of Fig. 4-23(b). If a three-block file is now written, the full block of pointers has to be read in again, taking us back to Fig. 4-23(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.

An alternative approach that avoids most of this disk I/O is to split the full block of pointers. Thus instead of going from Fig. 4-23(a) to Fig. 4-23(b), we go from Fig. 4-23(a) to Fig. 4-23(c) when three blocks are freed. Now the system can handle a series of temporary files without doing any disk I/O. If the block in memory fills up, it is written to the disk, and the half-full block from the disk is

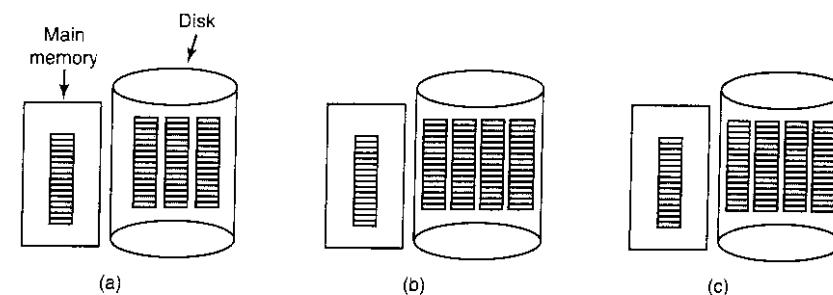


Figure 4-23. (a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

read in. The idea here is to keep most of the pointer blocks on disk full (to minimize disk usage), but keep the one in memory about half full, so it can handle both file creation and file removal without disk I/O on the free list.

With a bitmap, it is also possible to keep just one block in memory, going to disk for another only when it becomes full or empty. An additional benefit of this approach is that by doing all the allocation from a single block of the bitmap, the disk blocks will be close together, thus minimizing disk arm motion. Since the bitmap is a fixed-size data structure, if the kernel is (partially) paged, the bitmap can be put in virtual memory and have pages of it paged in as needed.

Disk Quotas

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

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

A second table contains the quota record for every user with a currently open file, even if the file was opened by someone else. This table is shown in Fig. 4-24. It is an extract from a quota file on disk for the users whose files are currently open. When all the files are closed, the record is written back to the quota file.

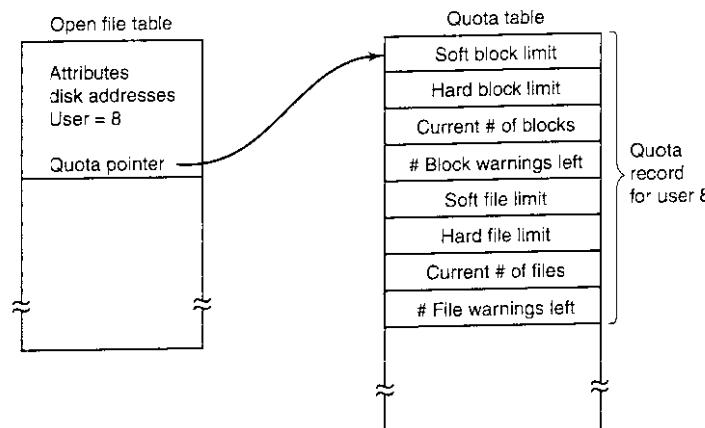


Figure 4-24. Quotas are kept track of on a per-user basis in a quota table.

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

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

4.4.2 File System Backups

Destruction of a file system is often a far greater disaster than destruction of a computer. If a computer is destroyed by fire, lightning surges, or a cup of coffee poured onto the keyboard, it is annoying and will cost money, but generally a replacement can be purchased with a minimum of fuss. Inexpensive personal computers can even be replaced within an hour by just going to a computer store

(except at universities, where issuing a purchase order takes three committees, five signatures, and 90 days).

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

Most people do not think making backups of their files is worth the time and effort—until one fine day their disk abruptly dies, at which time most of them undergo 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 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 manufacturer's 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 previous backup, which leads to the idea of **incremental dumps**. The simplest form of incremental dumping is to make a complete dump (backup) periodically, say weekly or monthly, and to make a daily dump of only those files that have been modified since the last full dump. Even better is to dump only those files that have changed since they were last dumped. While this scheme minimizes dumping time, it makes recovery more complicated, because first the most recent full dump has to be restored, followed by all the incremental dumps in reverse order. To ease recovery, more sophisticated incremental dumping schemes are often used.

Third, since immense amounts of data are typically dumped, it may be desirable to compress the data before writing them to 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 (because now two sites must be secured). 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 obtain access to the free block data structure, it can avoid dumping

unused blocks. However, skipping unused blocks requires writing the number of each block in front of the block (or the equivalent), since it is no longer true that block k on the tape was block k on the disk.

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

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

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

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. 4-25 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.

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

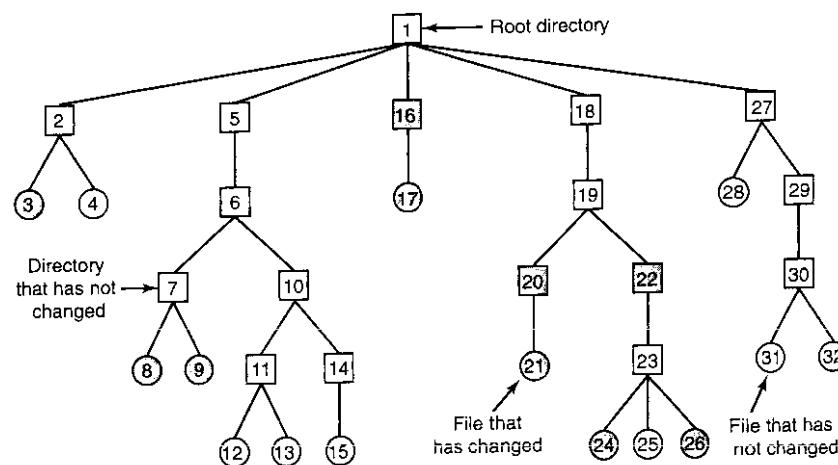


Figure 4-25. A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

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, and whatever., 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. 4-26(a). Phase 2 conceptually recursively walks the tree again, unmarking any directories that have no modified files or directories in them or under them. This phase leaves the bitmap as shown in Fig. 4-26(b). Note that directories 10, 11, 14, 27, 29, and 30 are now unmarked. They will not be dumped. By way of contrast, directories 5 and 6 will be dumped even though they

themselves have not been modified because they will be needed to restore today's changes to a fresh machine. For efficiency, phases 1 and 2 can be combined in one tree walk.

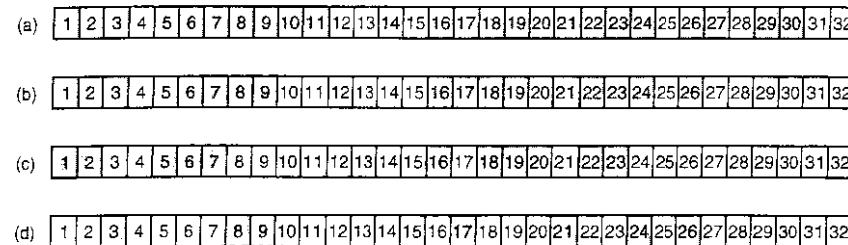


Figure 4-26. Bitmaps used by the logical dumping algorithm.

At this point it is known which directories and files must be dumped. These are the ones marked in Fig. 4-26(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. 4-26(c). Each directory is prefixed by the directory's attributes (owner, times, etc.) so that they can be restored. Finally, in phase 4, the files marked in Fig. 4-26(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 must not be restored. Core files often have a hole of hundreds of

megabytes between the data segment and the stack. If not handled properly, each restored core file will fill this area with zeros and thus be the same size as the virtual address space (e.g., 2^{32} bytes, or worse yet, 2^{64} bytes).

Finally, special files, named pipes, and the like 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).

Tape densities are not improving as fast as disk densities. This is gradually leading to a situation in which backing up a very large disk may require multiple tapes. While tape robots are available to change tapes automatically, if this trend continues, tapes will eventually become too small to use as a backup medium. In that case, the only way to back up a disk will be on another disk. While simply mirroring each disk with a spare is one possibility, more sophisticated schemes, called RAIDs, will be discussed in Chap. 5.

4.4.3 File System Consistency

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

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

If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 4-27(a). However, as a result of a crash, the tables might look like Fig. 4-27(b), in which block 2 does not occur in either table. It will be reported as being a **missing block**. While missing blocks do no real harm, they waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

Block number	Block number																														
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15																														
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	1	1	0	1	0	1	1	1	0	0	1	1	1	0	0	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> </table>	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0
1	1	0	1	0	1	1	1	0	0	1	1	1	0	0																	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0																	
0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1	0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 1 1																														
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1																	
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1																	
(a)	(b)																														
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15																														
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	1	1	0	1	0	1	1	1	0	0	1	1	1	0	0	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>2</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> </table>	1	1	0	1	0	2	1	1	1	0	0	1	1	1	0
1	1	0	1	0	1	1	1	0	0	1	1	1	0	0																	
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0																	
0 0 1 0 2 0 0 0 0 1 1 0 0 0 1 1	0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 1 1																														
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	1	0	1	0	0	0	0	0	1	1	0	0	0
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1																	
0	0	1	0	1	0	0	0	0	0	1	1	0	0	0																	
(c)	(d)																														

Figure 4-27. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

Another situation that might occur is that of Fig. 4-27(c). Here we see a block, number 4, that occurs twice in the free list. (Duplicates can occur only if the free list is really a list; with a bitmap it is impossible.) The solution here is also simple: rebuild the free list.

The worst thing that can happen is that the same data block is present in two or more files, as shown in Fig. 4-27(d) with block 5. If either of these files is removed, block 5 will be put on the free list, leading to a situation in which the same block is both in use and free at the same time. If both files are removed, the block will be put onto the free list twice.

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

In addition to checking to see that each block is properly accounted for, the file system checker also checks the directory system. It, too, uses a table of counters, but these are per file, rather than per block. It starts at the root directory and recursively descends the tree, inspecting each directory in the file system. For every i-node in every directory, it increments a counter for that file's usage count.

Remember that due to hard links, a file may appear in two or more directories. Symbolic links do not count and do not cause the counter for the target file to be incremented.

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

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

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

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

Furthermore, each i-node has a mode, some of which are legal but strange, such as 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 (a special directory), from which they can later be retrieved if need be. Of course, no storage is reclaimed until they are actually deleted from this directory.

4.4.4 File System Performance

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

Caching

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

Various algorithms can be used to manage the cache,¹ but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache.

Operation of the cache is illustrated in Fig. 4-28. Since there are many (often thousands of) blocks in the cache, some way is needed to determine quickly if a given block is present. The usual way is to hash the device and disk address and look up the result in a hash table. All the blocks with the same hash value are chained together on a linked list so that the collision chain can be followed.

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

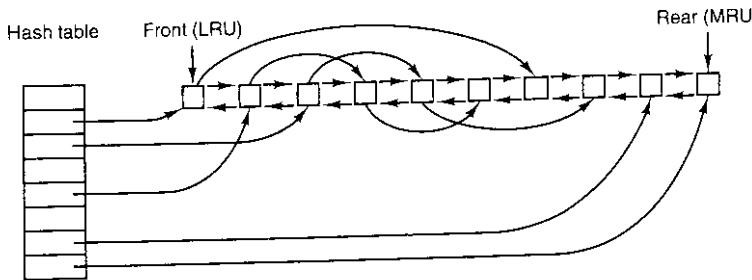


Figure 4-28. The buffer cache data structures.

are relatively infrequent, so that it is feasible to keep all the blocks in exact LRU order with linked lists.

In Fig. 4-28, 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.

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

The difference between these two approaches can be seen when a program writes a 1-KB block full, one character at a time. UNIX will collect all the characters in the cache and write the block out once every 30 seconds, or whenever the block is removed from the cache. With a write-through cache, there is a disk access for every character written. Of course, most programs do internal buffering, so they normally write not a character, but a line or a larger unit on each write system call.

A consequence of this difference in caching strategy is that just removing a (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 write-through caching no problem arises. These differing strategies were chosen because UNIX was developed in an environment in which all disks were hard disks and not removable, whereas the first Windows file system was inherited from MS-DOS, which started out in the floppy disk world. As hard disks became the norm, the UNIX approach, with its better efficiency (but worse reliability), became the norm, and is also used now on Windows for hard disks. However, NTFS takes other measures (journaling) to improve reliability, as discussed earlier.

Some operating systems integrate the buffer cache with the page cache. This is especially attractive when memory-mapped files are supported. If a file is mapped onto memory, then some of its pages may be in memory because they were demand paged in. Such pages are hardly different from file blocks in the buffer cache. In this case, they can be treated the same way, with a single cache for both file blocks and pages.

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 in a while, it is not a disaster, just a little bit of wasted disk bandwidth.

Reducing Disk Arm Motion

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

However, even with a free list, some block clustering can be done. The trick is to keep track of disk storage not in blocks, but in groups of consecutive blocks. If all sectors consist of 512 bytes, the system could use 1-KB blocks (2 sectors)

but allocate disk storage in units of 2 blocks (4 sectors). This is not the same as having 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 like them 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. 4-29(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.

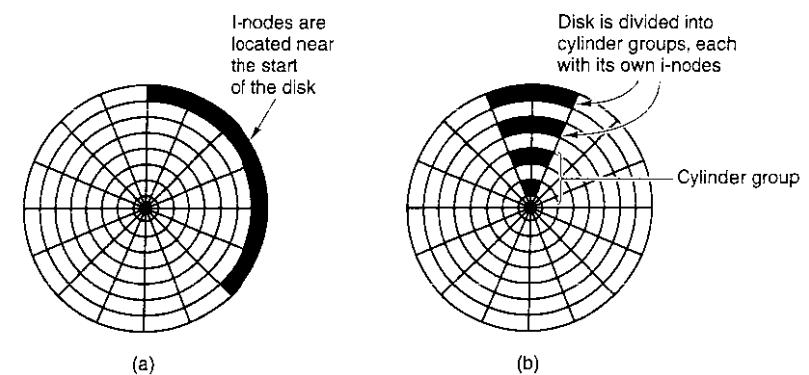


Figure 4-29. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

One easy performance improvement is to put the i-nodes in the middle of the disk, rather than at the start, thus reducing the average seek between the i-node and the first block by a factor of two. Another idea, shown in Fig. 4-29(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.

4.4.5 Defragmenting Disks

When the operating system is initially installed, the programs and files it needs are installed consecutively starting at the beginning of the disk, each one directly following the previous one. All free disk space is in a single contiguous unit

following the installed files. However, as time goes on, files are created and removed and typically the disk becomes badly fragmented, with files and holes all over the place. As a consequence, when a new file is created, the blocks used for it may be spread all over the disk, giving poor performance.

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

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

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

Linux file systems (especially ext2 and ext3) generally suffer less from defragmentation than Windows systems due to the way disk blocks are selected, so manual defragmentation is rarely required.

4.5 EXAMPLE FILE SYSTEMS

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

4.5.1 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.

Some years after the CD-ROM made its debut, the CD-R (CD Recordable) was introduced. Unlike the CD-ROM, it is possible to add files after the initial burning, but these are simply appended to the end of the CD-R. Files are never removed (although the directory can be updated to hide existing files). As a consequence of this "append-only" file system, the fundamental properties are not altered. In particular, all the free space is in one contiguous chunk at the end of the CD.

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. This information includes 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 SPARCcs, 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 whose products were big endian 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. 4-30. 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.

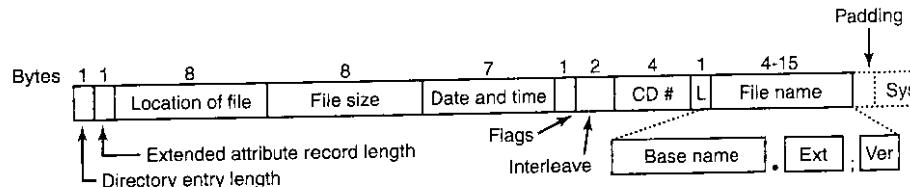


Figure 4-30. The ISO 9660 directory entry.

Directory entries may optionally have extended attributes. If this feature is used, 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. 4-30 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. This limit was arbitrarily set to make some implementations simpler.

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 (extents), 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 2 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.

4.5.2 The MS-DOS File System

The MS-DOS file system is the one the first IBM PCs came with. It was the main file system up through Windows 98 and Windows ME. It is still supported on Windows 2000, Windows XP, and Windows Vista, although it is no longer standard on new PCs now except for floppy disks. However, it and an extension of it (FAT-32) have become widely used for many embedded systems. Most digital cameras use it. Many MP3 players use it exclusively. The popular Apple iPod uses it as the default file system, although knowledgeable hackers can reformat the iPod and install a different file system. Thus the number of electronic devices using the MS-DOS file system is vastly larger now than at any time in the past, and certainly much larger than the number using the more modern NTFS file system. For that reason alone, it is worth looking at in some detail.

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

Although MS-DOS directories are variable sized, they use a fixed-size 32-byte directory entry. The format of an MS-DOS directory entry is shown in Fig. 4-31. 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 seconds). The

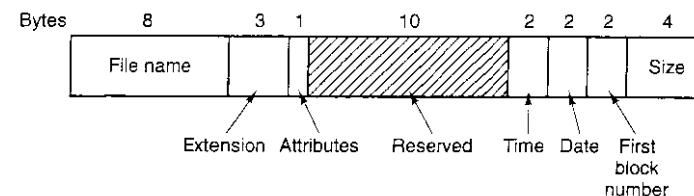


Figure 4-31. The MS-DOS directory entry.

time field is subdivided into seconds (5 bits), minutes (6 bits), and hours (5 bits). The date counts in days using three subfields: day (5 bits), month (4 bits), and year—1980 (7 bits). With a 7-bit number for the year and time beginning in 1980, the highest expressible year is 2107. Thus MS-DOS has a built-in Y2108 problem. To avoid catastrophe, MS-DOS users should begin with Y2108 compliance as early as possible. If MS-DOS had used the combined date and time fields as a 32-bit seconds counter, it could have represented every second exactly and delayed the catastrophe until 2116.

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

MS-DOS keeps track of file blocks via a file allocation table in main memory. The directory entry contains the number of the first file block. This number is used as an index into a 64K entry FAT in main memory. By following the chain, all the blocks can be found. The operation of the FAT is illustrated in Fig. 4-12.

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

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. 4-32.

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

Figure 4-32. 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, $\frac{1}{4}$ of the disk will be wasted, not a terribly efficient way to use the disk. With an 8-KB file and a 4-KB block, there is no disk wastage, but the price paid is more RAM eaten up by the FAT. With a 4-KB block and a 2-GB disk partition, there are 512K blocks, so the FAT must have 512K entries in memory (occupying 2 MB of RAM).

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

4.5.3 The UNIX V7 File System

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

The file system is in the form of a tree starting at the root directory, with the addition of links, forming a directed acyclic graph. File names 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. 4-13. A directory entry contains only two fields: the file name (14 bytes) and the number of the i-node for that file (2 bytes), as shown in Fig. 4-33. These parameters limit the number of files per file system to 64K.

Like the i-node of Fig. 4-13, the UNIX i-nodes contain 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 link is removed, the count is decremented. When it gets to 0, the i-node is reclaimed and the disk blocks are put back in the free list.

Keeping track of disk blocks is done using a generalization of Fig. 4-13 in order to handle very large files. The first 10 disk addresses are stored in the i-node

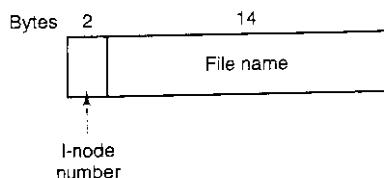


Figure 4-33. A UNIX V7 directory entry.

itself, so for small files, all the necessary information is right in the i-node, which is fetched from disk to main memory when the file is opened. For somewhat larger files, one of the addresses in the i-node is the address of a disk block called a **single indirect block**. This block contains additional disk addresses. If this still is not enough, another address in the i-node, called a **double indirect block**, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a **triple indirect block** can also be used. The complete picture is given in Fig. 4-34.

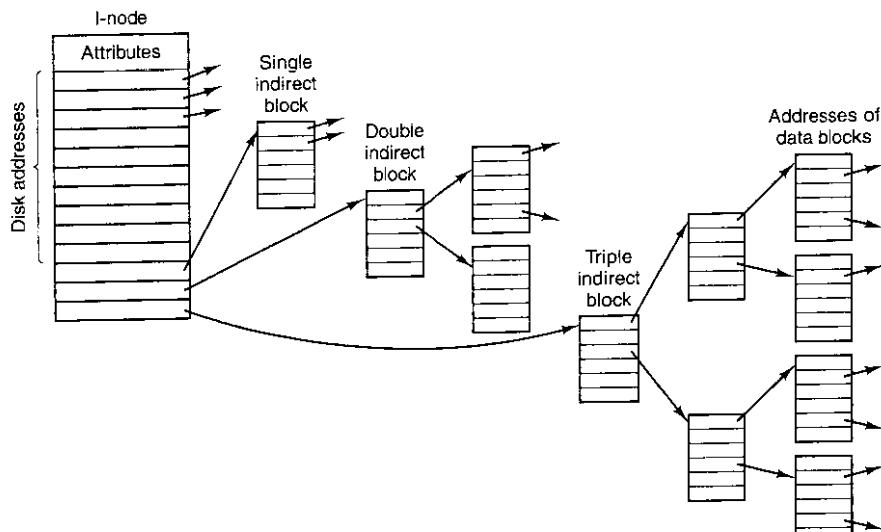
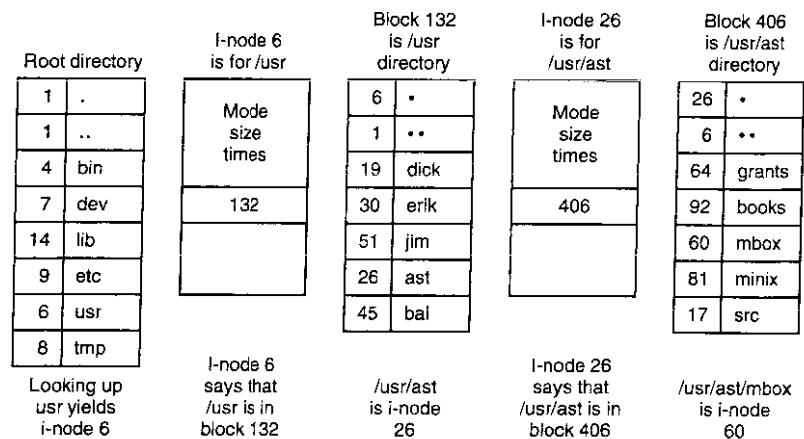


Figure 4-34. A UNIX i-node.

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.

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. 4-35.

Figure 4-35. The steps in looking up `/usr/ast/mbox`.

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 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. The only bit of trickery here is that `..` in the root directory points to itself.

4.6 RESEARCH ON FILE SYSTEMS

File systems have always attracted more research than other parts of the operating system and that is still the case. While standard file systems are fairly well understood, there is still quite a bit of research going on about optimizing buffer management (Burnett et al., 2002; Ding et al., 2007; Gnaid et al., 2004; Kroeger and Long, 2001; Pai et al., 2000; and Zhou et al., 2001). Work is going on about new kinds of file systems, such as user-level file systems (Mazières, 2001), flash file systems (Gal et al., 2005), journaling file systems (Prabhakaran et al., 2005; and Stein et al., 2001), versioning file systems (Cornell et al., 2004), peer-to-peer file systems (Muthitacharoen et al., 2002) and others. The Google file system is also unusual due to its great fault tolerance (Ghemawat et al., 2003). Different ways of finding things in file systems are also of interest (Padoleau and Ridoux, 2003).

Another area that has been getting attention is provenance—keeping track of the history of the data, including where they came from, who owns them, and how they have been transformed (Muniswamy-Reddy et al., 2006; and Shah et al., 2007). This information can be used in a variety of ways. Making backups is still getting some attention, too (Cox et al., 2002; and Rycroft, 2006), as is the related topic of recovery (Keeton et al., 2006). Related to backups is keeping data around and usable for decades (Baker et al., 2006; Maniatis et al., 2003). Reliability and security are also far from solved problems (Greenan and Miller, 2006; Wires and Feeley, 2007; Wright et al., 2007; and Yang et al., 2006). And finally performance has always been a research topic and still is (Caudill and Gavrikovska, 2006; Chiang and Huang, 2007; Stein, 2006; Wang et al., 2006a; and Zhang and Ghose, 2007).

4.7 SUMMARY

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

When seen from the inside, a file system looks quite different. The file system designers have to be concerned with how storage is allocated, and how the system keeps track of which block goes with which file. Possibilities include contiguous files, linked lists, file allocation tables, and i-nodes. Different systems have different directory structures. Attributes can go in the directories or somewhere else (e.g., an i-node). Disk space can be managed using free lists 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, MS-DOS, and UNIX. These differ in many ways, including how they keep track of which blocks go with which file, directory structure, and management of free disk space.

PROBLEMS

1. In 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?
2. In Fig. 4-4, one of the attributes is the record length. Why does the operating system ever care about this?
3. Is the open system call in UNIX absolutely essential? What would the consequences be of not having it?
4. Systems that support sequential files always have an operation to rewind files. Do systems that support random access files need this too?
5. 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?
6. 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?
7. 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?
8. In UNIX and Windows, random access is done by having a special system call that moves the “current position” pointer associated with a file to a given byte in the file. Propose an alternative way to do random access without having this system call.
9. Consider the directory tree of Fig. 4-8. If */usr/jim* is the working directory, what is the absolute path name for the file whose relative path name is *./ast/x*?
10. Contiguous allocation of files leads to disk fragmentation, as mentioned in the text, because some space in the last disk block will be wasted in files whose length is not an integral number of blocks. Is this internal fragmentation or external fragmentation? Make an analogy with something discussed in the previous chapter.
11. One way to use contiguous allocation of the disk and not suffer from holes is to compact the disk every time a file is removed. Since all files are contiguous, copying a file requires a seek and rotational delay to read the file, followed by the transfer at full

speed. Writing the file back requires the same work. Assuming a seek time of 5 msec, a rotational delay of 4 msec, a transfer rate of 8 MB/sec, and an average file size of 8 KB, how long does it take to read a file into main memory and then write it back to the disk at a new location? Using these numbers, how long would it take to compact half of a 16-GB disk?

12. In light of the answer to the previous question, does compacting the disk ever make any sense?
13. 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.
14. How does MS-DOS implement random access to files?
15. Consider the i-node shown in Fig. 4-13. If it contains 10 direct addresses of 4 bytes each and all disk blocks are 1024 KB, what is the largest possible file?
16. It has been suggested that efficiency could be improved and disk space saved by storing the data of a short file within the i-node. For the i-node of Fig. 4-13, how many bytes of data could be stored inside the i-node?
17. 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?
18. Name one advantage of hard links over symbolic links and one advantage of symbolic links over hard links.
19. 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.
20. The beginning of a free space bitmap looks like this after the disk partition is first formatted: 1000 0000 0000 0000 (the first block is used by the root directory). The system always searches for free blocks starting at the lowest-numbered block, so after writing file A, which uses six blocks, the bitmap looks like this: 1111 1110 0000 0000. Show the bitmap after each of the following additional actions:
 - (a) File B is written, using five blocks
 - (b) File A is deleted
 - (c) File C is written, using eight blocks
 - (d) File B is deleted.
21. What would happen if the bitmap or free list containing the information about free disk blocks was completely lost due to a crash? Is there any way to recover from this disaster, or is it bye-bye disk? Discuss your answers for UNIX and the FAT-16 file system separately.
22. 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?

23. 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?
24. Suppose that file 21 in Fig. 4-25 was not modified since the last dump. In what way would the four bitmaps of Fig. 4-26 be different?
25. 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?
26. Consider Fig. 4-27. Is it possible that for some particular block number the counters in *both* lists have the value 2? How should this problem be corrected?
27. The performance of a file system depends upon the cache hit rate (fraction of blocks found in the cache). If it takes 1 msec to satisfy a request from the cache, but 40 msec to satisfy a request if a disk read is needed, give a formula for the mean time required to satisfy a request if the hit rate is h . Plot this function for values of h varying from 0 to 1.0.
28. Consider the idea behind Fig. 4-21, 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?
29. 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.
30. The MS-DOS FAT-16 table contains 64K entries. Suppose that one of the bits had been needed for some other purpose and that the table contained exactly 32,768 entries instead. With no other changes, what would the largest MS-DOS file have been under this condition?
31. 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?
32. 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?
33. How many disk operations are needed to fetch the i-node for the file `/usr/ast/courses/os/handout.i`? 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.
34. 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.

35. 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.
36. 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.).
37. 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.
38. 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.

5

INPUT/OUTPUT

In addition to providing abstractions such as processes (and threads), address spaces, and files, an operating system also controls all the computer's I/O (Input/Output) devices. It must issue commands to the devices, catch interrupts, and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices (device independence). The I/O code represents a significant fraction of the total operating system. How the operating system manages I/O is the subject of this chapter.

This chapter is organized as follows. First we will look at some of the principles of I/O hardware, and then we will look at I/O software in general. I/O software can be structured in layers, with each layer having a well-defined task. We will look at these layers to see what they do and how they fit together.

Following that introduction, we will look at several I/O devices in detail: disks, clocks, keyboards, and displays. For each device we will look at its hardware and software. Finally, we will consider power management.

5.1 PRINCIPLES OF I/O HARDWARE

Different people look at I/O hardware in different ways. Electrical engineers look at it in terms of chips, wires, power supplies, motors, and all the other physical components that make up the hardware. Programmers look at the interface

presented to the software—the commands the hardware accepts, the functions it carries out, and the errors that can be reported back. In this book we are concerned with programming I/O devices, not designing, building, or maintaining them, so our interest will be restricted to how the hardware is programmed, not how it works inside. Nevertheless, the programming of many I/O devices is often intimately connected with their internal operation. In the next three sections we will provide a little general background on I/O hardware as it relates to programming. It may be regarded as a review and expansion of the introductory material in Sec. 1.4.

5.1.1 I/O Devices

I/O devices can be roughly divided into two categories: **block devices** and **character devices**. A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 bytes to 32,768 bytes. All transfers are in units of one or more entire (consecutive) blocks. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. Hard disks, CD-ROMs, and USB sticks are common block devices.

If you look closely, the boundary between devices that are block addressable and those that are not is not well defined. Everyone agrees that a disk is a block addressable device because no matter where the arm currently is, it is always possible to seek to another cylinder and then wait for the required block to rotate under the head. Now consider a tape drive used for making disk backups. Tapes contain a sequence of blocks. If the tape drive is given a command to read block N , it can always rewind the tape and go forward until it comes to block N . This operation is analogous to a disk doing a seek, except that it takes much longer. Also, it may or may not be possible to rewrite one block in the middle of a tape. Even if it were possible to use tapes as random access block devices, that is stretching the point somewhat: they are normally not used that way.

The other type of I/O device is the character device. A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, network interfaces, mice (for pointing), rats (for psychology lab experiments), and most other devices that are not disk-like can be seen as character devices.

This classification scheme is not perfect. Some devices just do not fit in. Clocks, for example, are not block addressable. Nor do they generate or accept character streams. All they do is cause interrupts at well-defined intervals. Memory-mapped screens do not fit the model well either. Still, the model of block and character devices is general enough that it can be used as a basis for making some of the operating system software dealing with I/O device independent. The file system, for example, deals just with abstract block devices and leaves the device-dependent part to lower-level software.

I/O devices cover a huge range in speeds, which puts considerable pressure on the software to perform well over many orders of magnitude in data rates. Fig. 5-1 shows the data rates of some common devices. Most of these devices tend to get faster as time goes on.

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner	400 KB/sec
Digital camcorder	3.5 MB/sec
802.11g Wireless	6.75 MB/sec
52x CD-ROM	7.8 MB/sec
Fast Ethernet	12.5 MB/sec
Compact flash card	40 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
SATA disk drive	300 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec

Figure 5-1. Some typical device, network, and bus data rates.

5.1.2 Device Controllers

I/O units typically consist of a mechanical component and an electronic component. It is often possible to separate the two portions to provide a more modular and general design. The electronic component is called the **device controller** or **adapter**. On personal computers, it often takes the form of a chip on the parentboard or a printed circuit card that can be inserted into a (PCI) expansion slot. The mechanical component is the device itself. This arrangement is shown in Fig. 1-6.

The controller card usually has a connector on it, into which a cable leading to the device itself can be plugged. Many controllers can handle two, four, or even eight identical devices. If the interface between the controller and device is a standard interface, either an official ANSI, IEEE, or ISO standard or a de facto

one, then companies can make controllers or devices that fit that interface. Many companies, for example, make disk drives that match the IDE, SATA, SCSI, USB, or FireWire (IEEE 1394) interface.

The interface between the controller and the device is often a very low-level interface. A disk, for example, might be formatted with 10,000 sectors of 512 bytes per track. What actually comes off the drive, however, is a serial bit stream, starting with a **preamble**, then the 4096 bits in a sector, and finally a checksum, also called an **Error-Correcting Code (ECC)**. The preamble is written when the disk is formatted and contains the cylinder and sector number, the sector size, and similar data, as well as synchronization information.

The controller's job is to convert the serial bit stream into a block of bytes and perform any error correction necessary. The block of bytes is typically first assembled, bit by bit, in a buffer inside the controller. After its checksum has been verified and the block has been declared to be error free, it can then be copied to main memory.

The controller for a monitor also works as a bit serial device at an equally low level. It reads bytes containing the characters to be displayed from memory and generates the signals used to modulate the CRT beam to cause it to write on the screen. The controller also generates the signals for making the CRT beam do a horizontal retrace after it has finished a scan line, as well as the signals for making it do a vertical retrace after the entire screen has been scanned. If it were not for the CRT controller, the operating system programmer would have to explicitly program the analog scanning of the tube. With the controller, the operating system initializes the controller with a few parameters, such as the number of characters or pixels per line and number of lines per screen, and lets the controller take care of actually driving the beam. Flat-screen TFT displays are different, but just as complicated.

5.1.3 Memory-Mapped I/O

Each controller has a few registers that are used for communicating with the CPU. By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action. By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command, and so on.

In addition to the control registers, many devices have a data buffer that the operating system can read and write. For example, a common way for computers to display pixels on the screen is to have a video RAM, which is basically just a data buffer, available for programs or the operating system to write into.

The issue thus arises of how the CPU communicates with the control registers and the device data buffers. Two alternatives exist. In the first approach, each control register is assigned an **I/O port** number, an 8- or 16-bit integer. The set of all the I/O ports form the **I/O port space** and is protected so that ordinary user

programs cannot access it (only the operating system can). Using a special I/O instruction such as

IN REG,PORT,

the CPU can read in control register PORT and store the result in CPU register REG. Similarly, using

OUT PORT,REG

the CPU can write the contents of REG to a control register. Most early computers, including nearly all mainframes, such as the IBM 360 and all of its successors, worked this way.

In this scheme, the address spaces for memory and I/O are different, as shown in Fig. 5-2(a). The instructions

IN R0,4

and

MOV R0,4

are completely different in this design. The former reads the contents of I/O port 4 and puts it in R0 whereas the latter reads the contents of memory word 4 and puts it in R0. The 4s in these examples refer to different and unrelated address spaces.

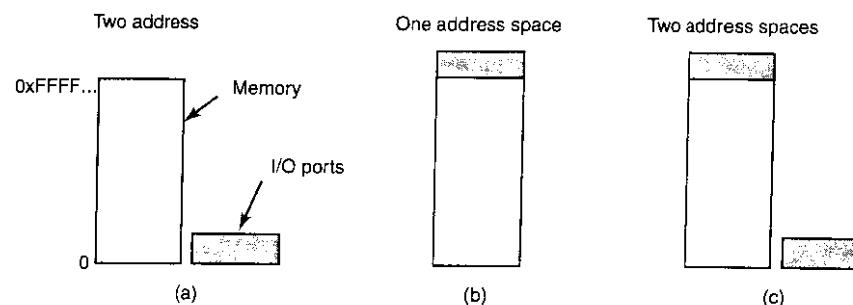


Figure 5-2. (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

The second approach, introduced with the PDP-11, is to map all the control registers into the memory space, as shown in Fig. 5-2(b). Each control register is assigned a unique memory address to which no memory is assigned. This system is called **memory-mapped I/O**. Usually, the assigned addresses are at the top of the address space. A hybrid scheme, with memory-mapped I/O data buffers and separate I/O ports for the control registers is shown in Fig. 5-2(c). The Pentium uses this architecture, with addresses 640K to 1M being reserved for device data buffers in IBM PC compatibles, in addition to I/O ports 0 through 64K.

How do these schemes work? In all cases, when the CPU wants to read a word, either from memory or from an I/O port, it puts the address it needs on the bus' address lines and then asserts a READ signal on a bus' control line. A second signal line is used to tell whether I/O space or memory space is needed. If it is memory space, the memory responds to the request. If it is I/O space, the I/O device responds to the request. If there is only memory space [as in Fig. 5-2(b)], every memory module and every I/O device compares the address lines to the range of addresses that it services. If the address falls in its range, it responds to the request. Since no address is ever assigned to both memory and an I/O device, there is no ambiguity and no conflict.

The two schemes for addressing the controllers have different strengths and weaknesses. Let us start with the advantages of memory-mapped I/O. First, if special I/O instructions are needed to read and write the device control registers, access to them requires the use of assembly code since there is no way to execute an IN or OUT instruction in C or C++. Calling such a procedure adds overhead to controlling I/O. In contrast, with memory-mapped I/O, device control registers are just variables in memory and can be addressed in C the same way as any other variables. Thus with memory-mapped I/O, a I/O device driver can be written entirely in C. Without memory-mapped I/O, some assembly code is needed.

Second, with memory-mapped I/O, no special protection mechanism is needed to keep user processes from performing I/O. All the operating system has to do is refrain from putting that portion of the address space containing the control registers in any user's virtual address space. Better yet, if each device has its control registers on a different page of the address space, the operating system can give user control over specific devices but not others by simply including the desired pages in its page table. Such a scheme can allow different device drivers to be placed in different address spaces, not only reducing kernel size but also keeping one driver from interfering with others.

Third, with memory-mapped I/O, every instruction that can reference memory can also reference control registers. For example, if there is an instruction, TEST, that tests a memory word for 0, it can also be used to test a control register for 0, which might be the signal that the device is idle and can accept a new command. The assembly language code might look like this:

```

LOOP: TEST PORT_4      // check if port 4 is 0
      BEQ READY        // if it is 0, go to ready
      BRANCH LOOP      // otherwise, continue testing

READY:

```

If memory-mapped I/O is not present, the control register must first be read into the CPU, then tested, requiring two instructions instead of one. In the case of the loop given above, a fourth instruction has to be added, slightly slowing down the responsiveness of detecting an idle device.

In computer design, practically everything involves trade-offs, and that is the case here too. Memory-mapped I/O also has its disadvantages. First, most computers nowadays have some form of caching of memory words. Caching a device control register would be disastrous. Consider the assembly code loop given above in the presence of caching. The first reference to PORT_4 would cause it to be cached. Subsequent references would just take the value from the cache and not even ask the device. Then when the device finally became ready, the software would have no way of finding out. Instead, the loop would go on forever.

To prevent this situation with memory-mapped I/O, the hardware has to be equipped with the ability to selectively disable caching, for example, on a per page basis. This feature adds extra complexity to both the hardware and the operating system, which has to manage the selective caching.

Second, if there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to. If the computer has a single bus, as in Fig. 5-3(a), having everyone look at every address is straightforward.

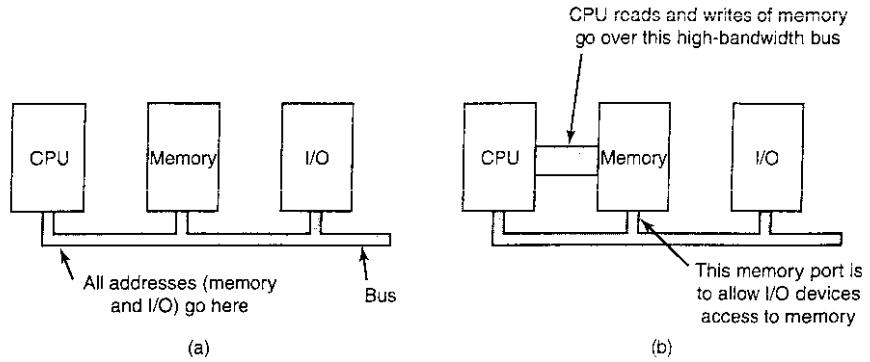


Figure 5-3. (a) A single-bus architecture. (b) A dual-bus memory architecture.

However, the trend in modern personal computers is to have a dedicated high-speed memory bus, as shown in Fig. 5-3(b), a property also found in mainframes, incidentally. This bus is tailored to optimize memory performance, with no compromises for the sake of slow I/O devices. Pentium systems can have multiple buses (memory, PCI, SCSI, USB, ISA), as shown in Fig. 1-12.

The trouble with having a separate memory bus on memory-mapped machines is that the I/O devices have no way of seeing memory addresses as they go by on the memory bus, so they have no way of responding to them. Again, special measures have to be taken to make memory-mapped I/O work on a system with multiple buses. One possibility is to first send all memory references to the memory. If the memory fails to respond, then the CPU tries the other buses. This design can be made to work but requires additional hardware complexity.

A second possible design is to put a snooping device on the memory bus to pass all addresses presented to potentially interested I/O devices. The problem here is that I/O devices may not be able to process requests at the speed the memory can.

A third possible design, which is the one used on the Pentium configuration of Fig. 1-12, is to filter addresses in the PCI bridge chip. This chip contains range registers that are preloaded at boot time. For example, 640K to 1M could be marked as a nonmemory range. Addresses that fall within one of the ranges marked as nonmemory are forwarded onto the PCI bus instead of to memory. The disadvantage of this scheme is the need for figuring out at boot time which memory addresses are not really memory addresses. Thus each scheme has arguments for and against it, so compromises and trade-offs are inevitable.

5.1.4 Direct Memory Access (DMA)

No matter whether a CPU does or does not have memory-mapped I/O, it needs to address the device controllers to exchange data with them. The CPU can request data from an I/O controller one byte at a time but doing so wastes the CPU's time, so a different scheme, called **DMA (Direct Memory Access)** is often used. The operating system can only use DMA if the hardware has a DMA controller, which most systems do. Sometimes this controller is integrated into disk controllers and other controllers, but such a design requires a separate DMA controller for each device. More commonly, a single DMA controller is available (e.g., on the parentboard) for regulating transfers to multiple devices, often concurrently.

No matter where it is physically located, the DMA controller has access to the system bus independent of the CPU, as shown in Fig. 5-4. It contains several registers that can be written and read by the CPU. These include a memory address register, a byte count register, and one or more control registers. The control registers specify the I/O port to use, the direction of the transfer (reading from the I/O device or writing to the I/O device), the transfer unit (byte at a time or word at a time), and the number of bytes to transfer in one burst.

To explain how DMA works, let us first look at how disk reads occur when DMA is not used. First the disk controller reads the block (one or more sectors) from the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it computes the checksum to verify that no read errors have occurred. Then the controller causes an interrupt. When the operating system starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in main memory.

When DMA is used, the procedure is different. First the CPU programs the DMA controller by setting its registers so it knows what to transfer where (step 1

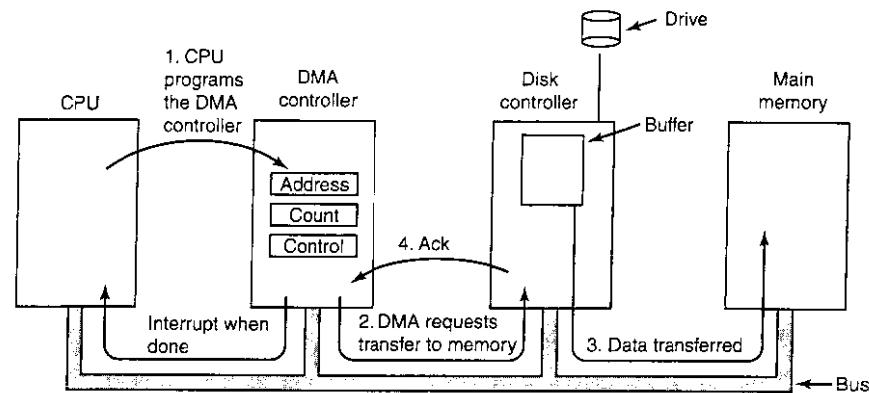


Figure 5-4. Operation of a DMA transfer.

in Fig. 5-4). It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin.

The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller (step 2). This read request looks like any other read request, and the disk controller does not know or care whether it came from the CPU or from a DMA controller. Typically, the memory address to write to is on the bus' address lines so when the disk controller fetches the next word from its internal buffer, it knows where to write it. The write to memory is another standard bus cycle (step 3). When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus (step 4). The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0. At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete. When the operating system starts up, it does not have to copy the disk block to memory; it is already there.

DMA controllers vary considerably in their sophistication. The simplest ones handle one transfer at a time, as described above. More complex ones can be programmed to handle multiple transfers at once. Such controllers have multiple sets of registers internally, one for each channel. The CPU starts by loading each set of registers with the relevant parameters for its transfer. Each transfer must use a different device controller. After each word is transferred (steps 2 through 4) in Fig. 5-4, the DMA controller decides which device to service next. It may be set up to use a round-robin algorithm, or it may have a priority scheme design to favor some devices over others. Multiple requests to different device controllers may be pending at the same time, provided that there is an unambiguous way to

tell the acknowledgements apart. Often a different acknowledgement line on the bus is used for each DMA channel for this reason.

Many buses can operate in two modes: word-at-a-time mode and block mode. Some DMA controllers can also operate in either mode. In the former mode, the operation is as described above: the DMA controller requests for the transfer of one word and gets it. If the CPU also wants the bus, it has to wait. The mechanism is called **cycle stealing** because the device controller sneaks in and steals an occasional bus cycle from the CPU once in a while, delaying it slightly. In block mode, the DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus. This form of operation is called **burst mode**. It is more efficient than cycle stealing because acquiring the bus takes time and multiple words can be transferred for the price of one bus acquisition. The down side to burst mode is that it can block the CPU and other devices for a substantial period of time if a long burst is being transferred.

In the model we have been discussing, sometimes called **fly-by mode**, the DMA controller tells the device controller to transfer the data directly to main memory. An alternative mode that some DMA controllers use is to have the device controller send the word to the DMA controller, which then issues a second bus request to write the word to wherever it is supposed to go. This scheme requires an extra bus cycle per word transferred, but is more flexible in that it can also perform device-to-device copies and even memory-to-memory copies (by first issuing a read to memory and then issuing a write to memory at a different address).

Most DMA controllers use physical memory addresses for their transfers. Using physical addresses requires the operating system to convert the virtual address of the intended memory buffer into a physical address and write this physical address into the DMA controller's address register. An alternative scheme used in a few DMA controllers is to write virtual addresses into the DMA controller instead. Then the DMA controller must use the MMU to have the virtual-to-physical translation done. Only in the case that the MMU is part of the memory (possible, but rare) rather than part of the CPU, can virtual addresses be put on the bus.

We mentioned earlier that the disk first reads data into its internal buffer before DMA can start. You may be wondering why the controller does not just store the bytes in main memory as soon as it gets them from the disk. In other words, why does it need an internal buffer? There are two reasons. First, by doing internal buffering, the disk controller can verify the checksum before starting a transfer. If the checksum is incorrect, an error is signaled and no transfer is done.

The second reason is that once a disk transfer has started, the bits keep arriving from the disk at a constant rate, whether the controller is ready for them or not. If the controller tried to write data directly to memory, it would have to go over the system bus for each word transferred. If the bus were busy due to some other device using it (e.g., in burst mode), the controller would have to wait. If

the next disk word arrived before the previous one had been stored, the controller would have to store it somewhere. If the bus were very busy, the controller might end up storing quite a few words and having a lot of administration to do as well. When the block is buffered internally, the bus is not needed until the DMA begins, so the design of the controller is much simpler because the DMA transfer to memory is not time critical. (Some older controllers did, in fact, go directly to memory with only a small amount of internal buffering, but when the bus was very busy, a transfer might have had to be terminated with an overrun error.)

Not all computers use DMA. The argument against it is that the main CPU is often far faster than the DMA controller and can do the job much faster (when the limiting factor is not the speed of the I/O device). If there is no other work for it to do, having the (fast) CPU wait for the (slow) DMA controller to finish is pointless. Also, getting rid of the DMA controller and having the CPU do all the work in software saves money, important on low-end (embedded) computers.

5.1.5 Interrupts Revisited

We briefly introduced interrupts in Sec. 1.4.5, but there is more to be said. In a typical personal computer system, the interrupt structure is as shown in Fig. 5-5. At the hardware level, interrupts work as follows. When an I/O device has finished the work given to it, it causes an interrupt (assuming that interrupts have been enabled by the operating system). It does this by asserting a signal on a bus line that it has been assigned. This signal is detected by the interrupt controller chip on the parentboard, which then decides what to do.

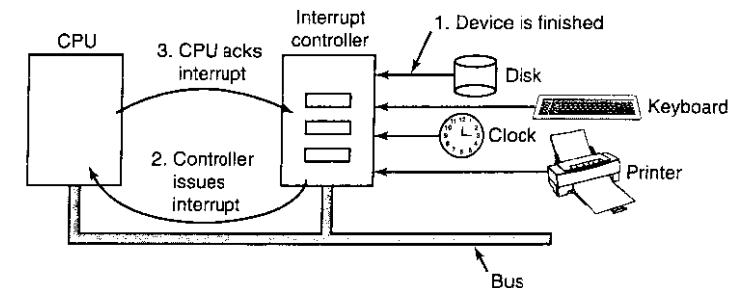


Figure 5-5. How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

If no other interrupts are pending, the interrupt controller processes the interrupt immediately. If another one is in progress, or another device has made a simultaneous request on a higher-priority interrupt request line on the bus, the

device is just ignored for the moment. In this case it continues to assert an interrupt signal on the bus until it is serviced by the CPU.

To handle the interrupt, the controller puts a number on the address lines specifying which device wants attention and asserts a signal to interrupt the CPU.

The interrupt signal causes the CPU to stop what it is doing and start doing something else. The number on the address lines is used as an index into a table called the **interrupt vector** to fetch a new program counter. This program counter points to the start of the corresponding interrupt service procedure. Typically traps and interrupts use the same mechanism from this point on, and frequently share the same interrupt vector. The location of the interrupt vector can be hardwired into the machine or it can be anywhere in memory, with a CPU register (loaded by the operating system) pointing to its origin.

Shortly after it starts running, the interrupt service procedure acknowledges the interrupt by writing a certain value to one of the interrupt controller's I/O ports. This acknowledgement tells the controller that it is free to issue another interrupt. By having the CPU delay this acknowledgement until it is ready to handle the next interrupt, race conditions involving multiple (almost simultaneous) interrupts can be avoided. As an aside, some (older) computers do not have a centralized interrupt controller, so each device controller requests its own interrupts.

The hardware always saves certain information before starting the service procedure. Which information is saved and where it is saved varies greatly from CPU to CPU. As a bare minimum, the program counter must be saved, so the interrupted process can be restarted. At the other extreme, all the visible registers and a large number of internal registers may be saved as well.

One issue is where to save this information. One option is to put it in internal registers that the operating system can read out as needed. A problem with this approach is that then the interrupt controller cannot be acknowledged until all potentially relevant information has been read out, lest a second interrupt over-write the internal registers saving the state. This strategy leads to long dead times when interrupts are disabled and possibly lost interrupts and lost data.

Consequently, most CPUs save the information on the stack. However, this approach, too, has problems. To start with: whose stack? If the current stack is used, it may well be a user process stack. The stack pointer may not even be legal, which would cause a fatal error when the hardware tried to write some words at the address pointed to. Also, it might point to the end of a page. After several memory writes, the page boundary might be exceeded and a page fault generated. Having a page fault occur during the hardware interrupt processing creates a bigger problem: where to save the state to handle the page fault?

If the kernel stack is used, there is a much better chance of the stack pointer being legal and pointing to a pinned page. However, switching into kernel mode may require changing MMU contexts and will probably invalidate most or all of the cache and TLB. Reloading all of these, statically or dynamically will increase the time to process an interrupt and thus waste CPU time.

Precise and Imprecise Interrupts

Another problem is caused by the fact that most modern CPUs are heavily pipelined and often superscalar (internally parallel). In older systems, after each instruction was finished executing, the microprogram or hardware checked to see if there was an interrupt pending. If so, the program counter and PSW were pushed onto the stack and the interrupt sequence began. After the interrupt handler ran, the reverse process took place, with the old PSW and program counter popped from the stack and the previous process continued.

This model makes the implicit assumption that if an interrupt occurs just after some instruction, all the instructions up to and including that instruction have been executed completely, and no instructions after it have executed at all. On older machines, this assumption was always valid. On modern ones it may not be.

For starters, consider the pipeline model of Fig. 1-6(a). What happens if an interrupt occurs while the pipeline is full (the usual case)? Many instructions are in various stages of execution. When the interrupt occurs, the value of the program counter may not reflect the correct boundary between executed instructions and nonexecuted instructions. In fact, many instructions may have been partially executed, with different instructions being more or less complete. In this situation, the program counter most likely reflects the address of the next instruction to be fetched and pushed into the pipeline rather than the address of the instruction that just was processed by the execution unit.

On a superscalar machine, such as that of Fig. 1-7(b), things are even worse. Instructions may be decomposed into micro-operations and the micro-operations may execute out of order, depending on the availability of internal resources such as functional units and registers. At the time of an interrupt, some instructions started long ago may not have started and others started more recently may be almost done. At the point when an interrupt is signaled, there may be many instructions in various states of completeness, with less relation between them and the program counter.

An interrupt that leaves the machine in a well-defined state is called a **precise interrupt** (Walker and Cragon, 1995). Such an interrupt has four properties:

1. The PC (Program Counter) is saved in a known place.
2. All instructions before the one pointed to by the PC have fully executed.
3. No instruction beyond the one pointed to by the PC has been executed.
4. The execution state of the instruction pointed to by the PC is known.

Note that there is no prohibition on instructions beyond the one pointed to by the PC from starting. It is just that any changes they make to registers or memory must be undone before the interrupt happens. It is permitted that the instruction pointed to has been executed. It is also permitted that it has not been executed.

However, it must be clear which case applies. Often, if the interrupt is an I/O interrupt, the instruction will not yet have started. However, if the interrupt is really a trap or page fault, then the PC generally points to the instruction that caused the fault so it can be restarted later. The situation of Fig. 5-6(a) illustrates a precise fault so it can be restarted later. All instructions up to the program counter (316) have completed and interrupted. All instructions up to the program counter (316) have completed and interrupted. None of those beyond it have started (or have been rolled back to undo their effects).

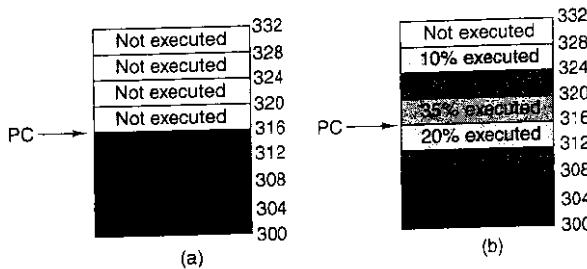


Figure 5-6. (a) A precise interrupt. (b) An imprecise interrupt.

An interrupt that does not meet these requirements is called an **imprecise interrupt** and makes life most unpleasant for the operating system writer, who now has to figure out what has happened and what still has to happen. Fig. 5-6(b) shows an imprecise interrupt, where different instructions near the program counter are in different stages of completion, with older ones not necessarily more complete than younger ones. Machines with imprecise interrupts usually vomit a large amount of internal state onto the stack to give the operating system the possibility of figuring out what was going on. The code necessary to restart the machine is typically extremely complicated. Also, saving a large amount of information to memory on every interrupt makes interrupts slow and recovery even worse. This leads to the ironic situation of having very fast superscalar CPUs sometimes being unsuitable for real-time work due to slow interrupts.

Some computers are designed so that some kinds of interrupts and traps are precise and others are not. For example, having I/O interrupts be precise but traps due to fatal programming errors be imprecise is not so bad since no attempt need be made to restart a running process after it has divided by zero. Some machines have a bit that can be set to force all interrupts to be precise. The downside of setting this bit is that it forces the CPU to carefully log everything it is doing and maintain shadow copies of registers so it can generate a precise interrupt at any instant. All this overhead has a major impact on performance.

Some superscalar machines, such as the Pentium series have precise interrupts to allow old software to work correctly. The price paid for precise interrupts is extremely complex interrupt logic within the CPU to make sure that when the interrupt controller signals that it wants to cause an interrupt, all instructions up to

some point are allowed to finish and none beyond that point are allowed to have any noticeable effect on the machine state. Here the price is paid not in time, but in chip area and in complexity of the design. If precise interrupts were not required for backward compatibility purposes, this chip area would be available for larger on-chip caches, making the CPU faster. On the other hand, imprecise interrupts make the operating system far more complicated and slower, so it is hard to tell which approach is really better.

5.2 PRINCIPLES OF I/O SOFTWARE

Let us now turn away from the I/O hardware and look at the I/O software. First we will look at the goals of the I/O software and then at the different ways I/O can be done from the point of view of the operating system.

5.2.1 Goals of the I/O Software

A key concept in the design of I/O software is known as **device independence**. What it means is that it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a hard disk, a CD-ROM, a DVD, or a USB stick without having to modify the program for each different device. Similarly, one should be able to type a command such as

sort <input>output

and have it work with input coming from any kind of disk or the keyboard and the output going to any kind of disk or the screen. It is up to the operating system to take care of the problems caused by the fact that these devices really are different and require very different command sequences to read or write.

Closely related to device independence is the goal of **uniform naming**. The name of a file or a device should simply be a string or an integer and not depend on the device in any way. In UNIX, all disks can be integrated in the file system hierarchy in arbitrary ways so the user need not be aware of which name corresponds to which device. For example, a USB stick can be **mounted** on top of the directory `/usr/ast/backup` so that copying a file to `/usr/ast/backup/monday` copies the file to the USB stick. In this way, all files and devices are addressed the same way: by a path name.

Another important issue for I/O software is **error handling**. In general, errors should be handled as close to the hardware as possible. If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again. Many errors are transient, such as read errors caused by specks of dust on the read head, and will frequently go away if the operation is repeated. Only if the

lower layers are not able to deal with the problem should the upper layers be told about it. In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.

Still another key issue is that of **synchronous** (blocking) versus **asynchronous** (interrupt-driven) transfers. Most physical I/O is asynchronous—the CPU starts the transfer and goes off to do something else until the interrupt arrives. User programs are much easier to write if the I/O operations are blocking—after a read system call the program is automatically suspended until the data are available in the buffer. It is up to the operating system to make operations that are actually interrupt-driven look blocking to the user programs.

Another issue for the I/O software is **buffering**. Often data that come off a device cannot be stored directly in its final destination. For example, when a packet comes in off the network, the operating system does not know where to put it until it has stored the packet somewhere and examined it. Also, some devices have severe real-time constraints (for example, digital audio devices), so the data must be put into an output buffer in advance to decouple the rate at which the buffer is filled from the rate at which it is emptied, in order to avoid buffer underruns. Buffering involves considerable copying and often has a major impact on I/O performance.

The final concept that we will mention here is sharable versus dedicated devices. Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as tape drives, have to be dedicated to a single user until that user is finished. Then another user can have the tape drive. Having two or more users writing blocks intermixed at random to the same tape will definitely not work. Introducing dedicated (unshared) devices also introduces a variety of problems, such as deadlocks. Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems.

5.2.2 Programmed I/O

There are three fundamentally different ways that I/O can be performed. In this section we will look at the first one (programmed I/O). In the next two sections we will examine the others (interrupt-driven I/O and I/O using DMA). The simplest form of I/O is to have the CPU do all the work. This method is called **programmed I/O**.

It is simplest to illustrate programmed I/O by means of an example. Consider a user process that wants to print the eight-character string “ABCDEFGHI” on the printer. It first assembles the string in a buffer in user space, as shown in Fig. 5-7(a).

The user process then acquires the printer for writing by making a system call to open it. If the printer is currently in use by another process, this call will fail

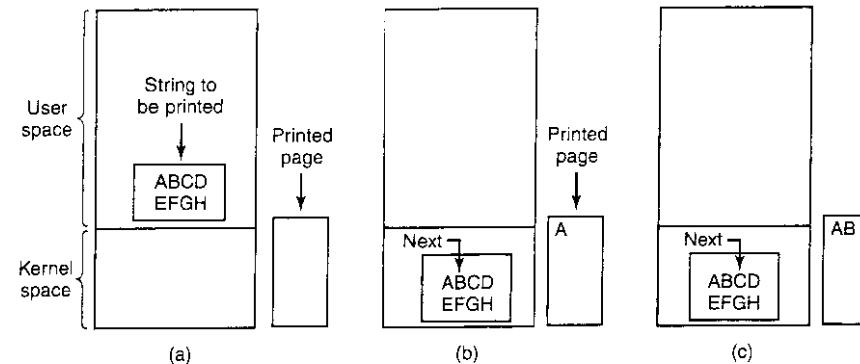


Figure 5-7. Steps in printing a string.

and return an error code or will block until the printer is available, depending on the operating system and the parameters of the call. Once it has the printer, the user process makes a system call telling the operating system to print the string on the printer.

The operating system then (usually) copies the buffer with the string to an array, say, p , in kernel space, where it is more easily accessed (because the kernel may have to change the memory map to get at user space). It then checks to see if the printer is currently available. If not, it waits until it is available. As soon as the printer is available, the operating system copies the first character to the printer's data register, in this example using memory-mapped I/O. This action activates the printer. The character may not appear yet because some printers buffer a line or a page before printing anything. In Fig. 5-7(b), however, we see that the first character has been printed and that the system has marked the “B” as the next character to be printed.

As soon as it has copied the first character to the printer, the operating system checks to see if the printer is ready to accept another one. Generally, the printer has a second register, which gives its status. The act of writing to the data register causes the status to become not ready. When the printer controller has processed the current character, it indicates its availability by setting some bit in its status register or putting some value in it.

At this point the operating system waits for the printer to become ready again. When that happens, it prints the next character, as shown in Fig. 5-7(c). This loop continues until the entire string has been printed. Then control returns to the user process.

The actions followed by the operating system are summarized in Fig. 5-8. First the data are copied to the kernel. Then the operating system enters a tight loop outputting the characters one at a time. The essential aspect of programmed

I/O, clearly illustrated in this figure, is that after outputting a character, the CPU continuously polls the device to see if it is ready to accept another one. This behavior is often called **polling** or **busy waiting**.

```
copy_from_user(buffer, p, count);
for (i = 0; i < count; i++) {
    /* p is the kernel buffer */
    /* loop on every character */
    while (*printer_status_reg != READY); /* loop until ready */
    /* output one character */
    *printer_data_register = p[i];
}
return_to_user();
```

Figure 5-8. Writing a string to the printer using programmed I/O.

Programmed I/O is simple but has the disadvantage of tying up the CPU full time until all the I/O is done. If the time to “print” a character is very short (because all the printer is doing is copying the new character to an internal buffer), then busy waiting is fine. Also, in an embedded system, where the CPU has nothing else to do, busy waiting is reasonable. However, in more complex systems, where the CPU has other work to do, busy waiting is inefficient. A better I/O method is needed.

5.2.3 Interrupt-Driven I/O

Now let us consider the case of printing on a printer that does not buffer characters but prints each one as it arrives. If the printer can print, say 100 characters/sec, each character takes 10 msec to print. This means that after every character is written to the printer’s data register, the CPU will sit in an idle loop for 10 msec waiting to be allowed to output the next character. This is more than enough time to do a context switch and run some other process for the 10 msec that would otherwise be wasted.

The way to allow the CPU to do something else while waiting for the printer to become ready is to use interrupts. When the system call to print the string is made, the buffer is copied to kernel space, as we showed earlier, and the first character is copied to the printer as soon as it is willing to accept a character. At that point the CPU calls the scheduler and some other process is run. The process that asked for the string to be printed is blocked until the entire string has printed.

The work done on the system call is shown in Fig. 5-9(a). When the printer has printed the character and is prepared to accept the next one, it generates an interrupt. This interrupt stops the current process and saves its state. Then the printer interrupt service procedure is run. A crude version of this code is shown in Fig. 5-9(b). If there are no more characters to print, the interrupt handler takes some action to unblock the user. Otherwise, it outputs the next character, acknowledges the interrupt, and returns to the process that was running just before the interrupt, which continues from where it left off.

```
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler();
if (count == 0) {
    unblock_user();
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

(a)

(b)

Figure 5-9. Writing a string to the printer using interrupt-driven I/O. (a) Code executed at the time the print system call is made. (b) Interrupt service procedure for the printer.

5.2.4 I/O Using DMA

An obvious disadvantage of interrupt-driven I/O is that an interrupt occurs on every character. Interrupts take time, so this scheme wastes a certain amount of CPU time. A solution is to use DMA. Here the idea is to let the DMA controller feed the characters to the printer one at time, without the CPU being bothered. In essence, DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU. This strategy requires special hardware (the DMA controller) but frees up the CPU during the I/O to do other work. An outline of the code is given in Fig. 5-10.

```
copy_from_user(buffer, p, count);
set_up_DMA_controller();
scheduler();
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

(a)

(b)

Figure 5-10. Printing a string using DMA. (a) Code executed when the print system call is made. (b) Interrupt service procedure.

The big win with DMA is reducing the number of interrupts from one per character to one per buffer printed. If there are many characters and interrupts are slow, this can be a major improvement. On the other hand, the DMA controller is usually much slower than the main CPU. If the DMA controller is not capable of driving the device at full speed, or the CPU usually has nothing to do anyway while waiting for the DMA interrupt, then interrupt-driven I/O or even programmed I/O may be better. Most of the time DMA is worth it though.

5.3 I/O SOFTWARE LAYERS

I/O software is typically organized in four layers, as shown in Fig. 5-11. Each layer has a well-defined function to perform and a well-defined interface to the adjacent layers. The functionality and interfaces differ from system to system, so the discussion that follows, which examines all the layers starting at the bottom, is not specific to one machine.

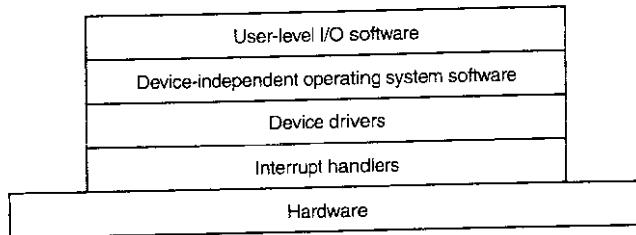


Figure 5-11. Layers of the I/O software system.

5.3.1 Interrupt Handlers

While programmed I/O is occasionally useful, for most I/O, interrupts are an unpleasant fact of life and cannot be avoided. They should be hidden away, deep in the bowels of the operating system, so that as little of the operating system as possible knows about them. The best way to hide them is to have the driver starting an I/O operation block until the I/O has completed and the interrupt occurs. The driver can block itself by doing a down on a semaphore, a wait on a condition variable, a receive on a message, or something similar, for example.

When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt. Then it can unblock the driver that started it. In some cases it will just complete up on a semaphore. In others it will do a signal on a condition variable in a monitor. In still others, it will send a message to the blocked driver. In all cases the net effect of the interrupt will be that a driver that was previously blocked will now be able to run. This model works best if drivers are structured as kernel processes, with their own states, stacks, and program counters.

Of course, reality is not quite so simple. Processing an interrupt is not just a matter of taking the interrupt, doing an up on some semaphore, and then executing an IRET instruction to return from the interrupt to the previous process. There is a great deal more work involved for the operating system. We will now give an outline of this work as a series of steps that must be performed in software after the hardware interrupt has completed. It should be noted that the details are very

system dependent, so some of the steps listed below may not be needed on a particular machine and steps not listed may be required. Also, the steps that do occur may be in a different order on some machines.

1. Save any registers (including the PSW) that have not already been saved by the interrupt hardware.
2. Set up a context for the interrupt service procedure. Doing this may involve setting up the TLB, MMU and a page table.
3. Set up a stack for the interrupt service procedure.
4. Acknowledge the interrupt controller. If there is no centralized interrupt controller, reenable interrupts.
5. Copy the registers from where they were saved (possibly some stack) to the process table.
6. Run the interrupt service procedure. It will extract information from the interrupting device controller's registers.
7. Choose which process to run next. If the interrupt has caused some high-priority process that was blocked to become ready, it may be chosen to run now.
8. Set up the MMU context for the process to run next. Some TLB setup may also be needed.
9. Load the new process' registers, including its PSW.
10. Start running the new process.

As can be seen, interrupt processing is far from trivial. It also takes a considerable number of CPU instructions, especially on machines in which virtual memory is present and page tables have to be set up or the state of the MMU stored (e.g., the *R* and *M* bits). On some machines the TLB and CPU cache may also have to be managed when switching between user and kernel modes, which takes additional machine cycles.

5.3.2 Device Drivers

Earlier in this chapter we looked at what device controllers do. We saw that each controller has some device registers used to give it commands or some device registers used to read out its status or both. The number of device registers and the nature of the commands vary radically from device to device. For example, a mouse driver has to accept information from the mouse telling how far it has moved and which buttons are currently depressed. In contrast, a disk driver may have to know all about sectors, tracks, cylinders, heads, arm motion, motor

drives, head settling times, and all the other mechanics of making the disk work properly. Obviously, these drivers will be very different.

As a consequence, each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the **device driver**, is generally written by the device's manufacturer and delivered along with the device. Since each operating system needs its own drivers, device manufacturers commonly supply drivers for several popular operating systems.

Each device driver normally handles one device type, or at most, one class of closely related devices. For example, a SCSI disk driver can usually handle multiple SCSI disks of different sizes and different speeds, and perhaps a SCSI CD-ROM as well. On the other hand, a mouse and joystick are so different that different drivers are usually required. However, there is no technical restriction on having one device driver control multiple unrelated devices. It is just not a good idea.

In order to access the device's hardware, meaning the controller's registers, the device driver normally has to be part of the operating system kernel, at least with current architectures. Actually, it is possible to construct drivers that run in user space, with system calls for reading and writing the device registers. This design isolates the kernel from the drivers and the drivers from each other, eliminating a major source of system crashes—buggy drivers that interfere with the kernel in one way or another. For building highly reliable systems, this is definitely the way to go. An example of a system in which the device drivers run as user processes is MINIX 3. However, since most other desktop operating systems expect drivers to run in the kernel, that is the model we will consider here.

Since the designers of every operating system know that pieces of code (drivers) written by outsiders will be installed in it, it needs to have an architecture that allows such installation. This means having a well-defined model of what a driver does and how it interacts with the rest of the operating system. Device drivers are normally positioned below the rest of the operating system, as is illustrated in Fig. 5-12.

Operating systems usually classify drivers into one of a small number of categories. The most common categories are the **block devices**, such as disks, which contain multiple data blocks that can be addressed independently, and the **character devices**, such as keyboards and printers, which generate or accept a stream of characters.

Most operating systems define a standard interface that all block drivers must support and a second standard interface that all character drivers must support. These interfaces consist of a number of procedures that the rest of the operating system can call to get the driver to do work for it. Typical procedures are those to read a block (block device) or write a character string (character device).

In some systems, the operating system is a single binary program that contains all of the drivers that it will need compiled into it. This scheme was the norm for years with UNIX systems because they were run by computer centers and I/O de-

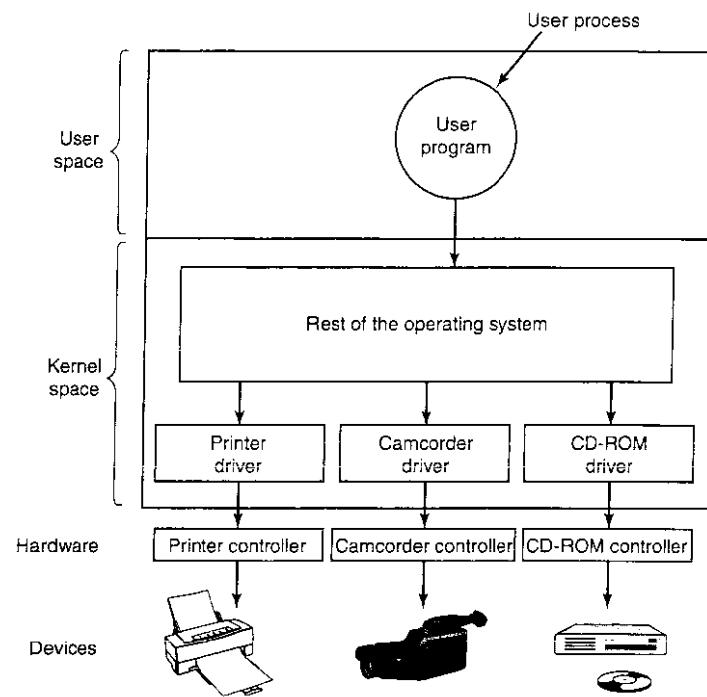


Figure 5-12. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

vices rarely changed. If a new device was added, the system administrator simply recompiled the kernel with the new driver to build a new binary.

With the advent of personal computers, with their myriad I/O devices, this model no longer worked. Few users are capable of recompiling or relinking the kernel, even if they have the source code or object modules, which is not always the case. Instead, operating systems, starting with MS-DOS, went over to a model in which drivers were dynamically loaded into the system during execution. Different systems handle loading drivers in different ways.

A device driver has several functions. The most obvious one is to accept abstract read and write requests from the device-independent software above it and see that they are carried out. But there are also a few other functions they must perform. For example, the driver must initialize the device, if needed. It may also need to manage its power requirements and log events.

Many device drivers have a similar general structure. A typical driver starts out by checking the input parameters to see if they are valid. If not, an error is

returned. If they are valid, a translation from abstract to concrete terms may be needed. For a disk driver, this may mean converting a linear block number into the head, track, sector, and cylinder numbers for the disk's geometry.

Next the driver may check if the device is currently in use. If it is, the request will be queued for later processing. If the device is idle, the hardware status will be examined to see if the request can be handled now. It may be necessary to switch the device on or start a motor before transfers can be begun. Once the device is on and ready to go, the actual control can begin.

Controlling the device means issuing a sequence of commands to it. The driver is the place where the command sequence is determined, depending on what has to be done. After the driver knows which commands it is going to issue, it starts writing them into the controller's device registers. After writing each command to the controller, it may be necessary to check to see if the controller accepted the command and is prepared to accept the next one. This sequence continues until all the commands have been issued. Some controllers can be given a linked list of commands (in memory) and told to read and process them all by itself without further help from the operating system.

After the commands have been issued, one of two situations will apply. In many cases the device driver must wait until the controller does some work for it, so it blocks itself until the interrupt comes in to unblock it. In other cases, however, the operation finishes without delay, so the driver need not block. As an example of the latter situation, scrolling the screen in character mode requires just writing a few bytes into the controller's registers. No mechanical motion is needed, so the entire operation can be completed in nanoseconds.

In the former case, the blocked driver will be awakened by the interrupt. In the latter case, it will never go to sleep. Either way, after the operation has been completed, the driver must check for errors. If everything is all right, the driver may have data to pass to the device-independent software (e.g., a block just read). Finally, it returns some status information for error reporting back to its caller. If any other requests are queued, one of them can now be selected and started. If nothing is queued, the driver blocks waiting for the next request.

This simple model is only a rough approximation to reality. Many factors make the code much more complicated. For one thing, an I/O device may complete while a driver is running, interrupting the driver. The interrupt may cause a device driver to run. In fact, it may cause the current driver to run. For example, while the network driver is processing an incoming packet, another packet may arrive. Consequently, drivers have to be **reentrant**, meaning that a running driver has to expect that it will be called a second time before the first call has completed.

In a hot pluggable system, devices can be added or removed while the computer is running. As a result, while a driver is busy reading from some device, the system may inform it that the user has suddenly removed that device from the system. Not only must the current I/O transfer be aborted without damaging any

kernel data structures, but any pending requests for the now-vanished device must also be gracefully removed from the system and their callers given the bad news. Furthermore, the unexpected addition of new devices may cause the kernel to juggle resources (e.g., interrupt request lines), taking old ones away from the driver and giving it new ones in their place.

Drivers are not allowed to make system calls, but they often need to interact with the rest of the kernel. Usually, calls to certain kernel procedures are permitted. For example, there are usually calls to allocate and deallocate hardwired pages of memory for use as buffers. Other useful calls are needed to manage the MMU, timers, the DMA controller, the interrupt controller, and so on.

5.3.3 Device-Independent I/O Software

Although some of the I/O software is device specific, other parts of it are device independent. The exact boundary between the drivers and the device-independent software is system (and device) dependent, because some functions that could be done in a device-independent way may actually be done in the drivers, for efficiency or other reasons. The functions shown in Fig. 5-13 are typically done in the device-independent software.

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Figure 5-13. Functions of the device-independent I/O software.

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Below we will look at the above issues in more detail.

Uniform Interfacing for Device Drivers

A major issue in an operating system is how to make all I/O devices and drivers look more or less the same. If disks, printers, keyboards, and so on, are all interfaced in different ways, every time a new device comes along, the operating system must be modified for the new device. Having to hack on the operating system for each new device is not a good idea.

One aspect of this issue is the interface between the device drivers and the rest of the operating system. In Fig. 5-14(a) we illustrate a situation in which each

device driver has a different interface to the operating system. What this means is that the driver functions available for the system to call differ from driver to driver. It might also mean that the kernel functions that the driver needs also differ from driver to driver. Taken together, it means that interfacing each new driver requires a lot of new programming effort.

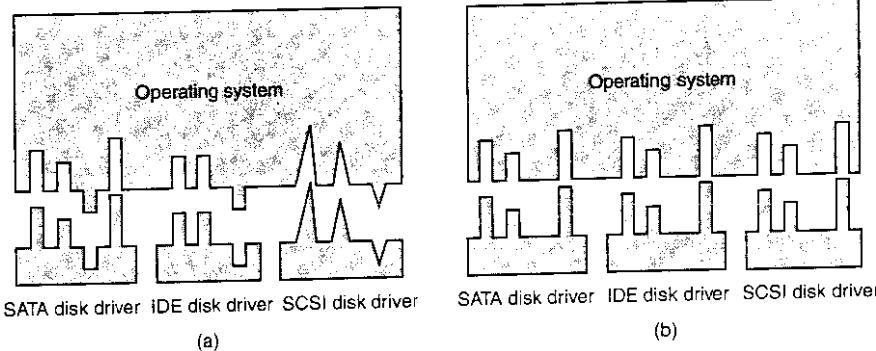


Figure 5-14. (a) Without a standard driver interface. (b) With a standard driver interface.

In contrast, in Fig. 5-14(b), we show a different design in which all drivers have the same interface. Now it becomes much easier to plug in a new driver, providing it conforms to the driver interface. It also means that driver writers know what is expected of them. In practice, not all devices are absolutely identical, but usually there are only a small number of device types and even these are generally almost the same.

The way this works is as follows. For each class of devices, such as disks or printers, the operating system defines a set of functions that the driver must supply. For a disk these would naturally include read and write, but also turning the power on and off, formatting, and other disk things. Often the driver contains a table with pointers into itself for these functions. When the driver is loaded, the operating system records the address of this table of function pointers, so when it needs to call one of the functions, it can make an indirect call via this table. This table of function pointers defines the interface between the driver and the rest of the operating system. All devices of a given class (disks, printers, etc.) must obey it.

Another aspect of having a uniform interface is how I/O devices are named. The device-independent software takes care of mapping symbolic device names onto the proper driver. For example, in UNIX a device name, such as `/dev/disk0`, uniquely specifies the i-node for a special file, and this i-node contains the **major device number**, which is used to locate the appropriate driver. The i-node also contains the **minor device number**, which is passed as a parameter to the driver

in order to specify the unit to be read or written. All devices have major and minor numbers, and all drivers are accessed by using the major device number to select the driver.

Closely related to naming is protection. How does the system prevent users from accessing devices that they are not entitled to access? In both UNIX and Windows, devices appear in the file system as named objects, which means that the usual protection rules for files also apply to I/O devices. The system administrator can then set the proper permissions for each device.

Buffering

Buffering is also an issue, both for block and character devices, for a variety of reasons. To see one of them, consider a process that wants to read data from a modem. One possible strategy for dealing with the incoming characters is to have the user process do a read system call and block waiting for one character. Each arriving character causes an interrupt. The interrupt service procedure hands the character to the user process and unblocks it. After putting the character somewhere, the process reads another character and blocks again. This model is indicated in Fig. 5-15(a).

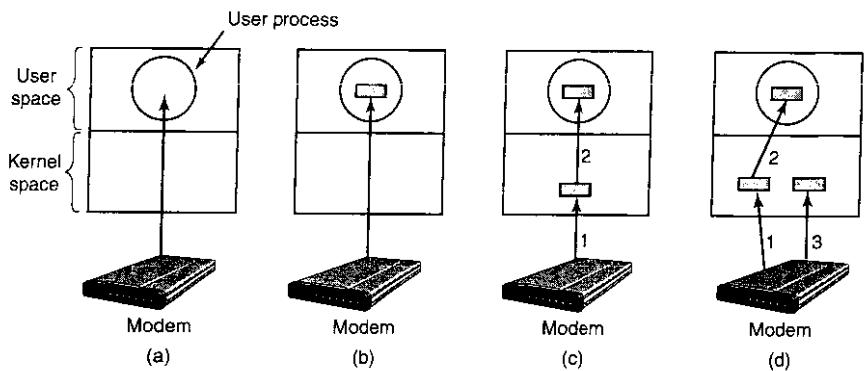


Figure 5-15. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

The trouble with this way of doing business is that the user process has to be started up for every incoming character. Allowing a process to run many times for short runs is inefficient, so this design is not a good one.

An improvement is shown in Fig. 5-15(b). Here the user process provides an n -character buffer in user space and does a read of n characters. The interrupt service procedure puts incoming characters in this buffer until it fills up. Then it wakes up the user process. This scheme is far more efficient than the previous

one, but it has a drawback: what happens if the buffer is paged out when a character arrives? The buffer could be locked in memory, but if many processes start locking pages in memory, the pool of available pages will shrink and performance will degrade.

Yet another approach is to create a buffer inside the kernel and have the interrupt handler put the characters there, as shown in Fig. 5-15(c). When this buffer is full, the page with the user buffer is brought in, if needed, and the buffer copied there in one operation. This scheme is far more efficient.

However, even this scheme suffers from a problem: What happens to characters that arrive while the page with the user buffer is being brought in from the disk? Since the buffer is full, there is no place to put them. A way out is to have a second kernel buffer. After the first buffer fills up, but before it has been emptied, the second one is used, as shown in Fig. 5-15(d). When the second buffer fills up, it is available to be copied to the user (assuming the user has asked for it). While the second buffer is being copied to user space, the first one can be used for new characters. In this way, the two buffers take turns: while one is being copied to user space, the other is accumulating new input. A buffering scheme like this is called **double buffering**.

Another form of buffering that is widely used is the **circular buffer**. It consists of a region of memory and two pointers. One pointer points to the next free word, where new data can be placed. The other pointer points to the first word of data in the buffer that has not been removed yet. In many situations, the hardware advances the first pointer as it adds new data (e.g., just arriving from the network) and the operating system advances the second pointer as it removes and processes data. Both pointers wrap around, going back to the bottom when they hit the top.

Buffering is also important on output. Consider, for example, how output is done to the modem without buffering using the model of Fig. 5-15(b). The user process executes a write system call to output n characters. The system has two choices at this point. It can block the user until all the characters have been written, but this could take a very long time over a slow telephone line. It could also release the user immediately and do the I/O while the user computes some more, but this leads to an even worse problem: how does the user process know that the output has been completed and it can reuse the buffer? The system could generate a signal or software interrupt, but that style of programming is difficult and prone to race conditions. A much better solution is for the kernel to copy the data to a kernel buffer, analogous in Fig. 5-15(c) (but the other way), and unblock the caller immediately. Now it does not matter when the actual I/O has been completed. The user is free to reuse the buffer the instant it is unblocked.

Buffering is a widely used technique, but it has a downside as well. If data get buffered too many times, performance suffers. Consider, for example, the network of Fig. 5-16. Here a user does a system call to write to the network. The kernel copies the packet to a kernel buffer to allow the user to proceed immediately (step 1). At this point the user program can reuse the buffer.

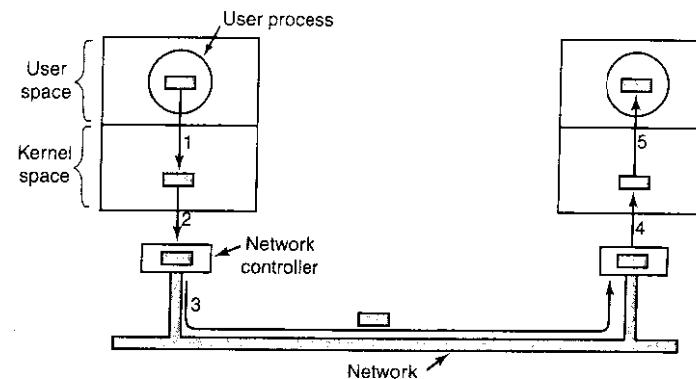


Figure 5-16. Networking may involve many copies of a packet.

When the driver is called, it copies the packet to the controller for output (step 2). The reason it does not output to the wire directly from kernel memory is that once a packet transmission has been started, it must continue at a uniform speed. The driver cannot guarantee that it can get to memory at a uniform speed because DMA channels and other I/O devices may be stealing many cycles. Failing to get a word on time would ruin the packet. By buffering the packet inside the controller, this problem is avoided.

After the packet has been copied to the controller's internal buffer, it is copied out onto the network (step 3). Bits arrive at the receiver shortly after being sent, so just after the last bit has been sent, that bit arrives at the receiver, where the packet has been buffered in the controller. Next the packet is copied to the receiver's kernel buffer (step 4). Finally, it is copied to the receiving process' buffer (step 5). Usually, the receiver then sends back an acknowledgement. When the sender gets the acknowledgement, it is free to send the next packet. However, it should be clear that all this copying is going to slow down the transmission rate considerably because all the steps must happen sequentially.

Error Reporting

Errors are far more common in the context of I/O than in other contexts. When they occur, the operating system must handle them as best it can. Many errors are device-specific and must be handled by the appropriate driver, but the framework for error handling is device independent.

One class of I/O errors is programming errors. These occur when a process asks for something impossible, such as writing to an input device (keyboard, scanner, mouse, etc.) or reading from an output device (printer, plotter, etc.). Other errors are providing an invalid buffer address or other parameter, and specifying

an invalid device (e.g., disk 3 when the system has only two disks), and so on. The action to take on these errors is straightforward: just report back an error code to the caller.

Another class of errors is the class of actual I/O errors, for example, trying to write a disk block that has been damaged or trying to read from a camcorder that has been switched off. In these circumstances, it is up to the driver to determine what to do. If the driver does not know what to do, it may pass the problem back up to device-independent software.

What this software does depends on the environment and the nature of the error. If it is a simple read error and there is an interactive user available, it may display a dialog box asking the user what to do. The options may include retrying a certain number of times, ignoring the error, or killing the calling process. If there is no user available, probably the only real option is to have the system call fail with an error code.

However, some errors cannot be handled this way. For example, a critical data structure, such as the root directory or free block list, may have been destroyed. In this case, the system may have to display an error message and terminate.

Allocating and Releasing Dedicated Devices

Some devices, such as CD-ROM recorders, can be used only by a single process at any given moment. It is up to the operating system to examine requests for device usage and accept or reject them, depending on whether the requested device is available or not. A simple way to handle these requests is to require processes to perform opens on the special files for devices directly. If the device is unavailable, the open fails. Closing such a dedicated device then releases it.

An alternative approach is to have special mechanisms for requesting and releasing dedicated devices. An attempt to acquire a device that is not available blocks the caller instead of failing. Blocked processes are put on a queue. Sooner or later, the requested device becomes available and the first process on the queue is allowed to acquire it and continue execution.

Device-Independent Block Size

Different disks may have different sector sizes. It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers, for example, by treating several sectors as a single logical block. In this way, the higher layers only deal with abstract devices that all use the same logical block size, independent of the physical sector size. Similarly, some character devices deliver their data one byte at a time (e.g., modems), while others deliver theirs in larger units (e.g., network interfaces). These differences may also be hidden.

5.3.4 User-Space I/O Software

Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel. System calls, including the I/O system calls, are normally made by library procedures. When a C program contains the call

```
count = write(fd, buffer, nbytes);
```

the library procedure *write* will be linked with the program and contained in the binary program present in memory at run time. The collection of all these library procedures is clearly part of the I/O system.

While these procedures do little more than put their parameters in the appropriate place for the system call, there are other I/O procedures that actually do real work. In particular, formatting of input and output is done by library procedures. One example from C is *printf*, which takes a format string and possibly some variables as input, builds an ASCII string, and then calls *write* to output the string. As an example of *printf*, consider the statement

```
printf("The square of %3d is %6d\n", i, i*i);
```

It formats a string consisting of the 14-character string "The square of " followed by the value *i* as a 3-character string, then the 4-character string " is ", then i^2 as six characters, and finally a line feed.

An example of a similar procedure for input is *scanf* which reads input and stores it into variables described in a format string using the same syntax as *printf*. The standard I/O library contains a number of procedures that involve I/O and all run as part of user programs.

Not all user-level I/O software consists of library procedures. Another important category is the spooling system. **Spooling** is a way of dealing with dedicated I/O devices in a multiprogramming system. Consider a typical spooled device: a printer. Although it would be technically easy to let any user process open the character special file for the printer, suppose a process opened it and then did nothing for hours. No other process could print anything.

Instead what is done is to create a special process, called a **daemon**, and a special directory, called a **spooling directory**. To print a file, a process first generates the entire file to be printed and puts it in the spooling directory. It is up to the daemon, which is the only process having permission to use the printer's special file, to print the files in the directory. By protecting the special file against direct use by users, the problem of having someone keeping it open unnecessarily long is eliminated.

Spooling is not only used for printers. It is also used in other I/O situations. For example, file transfer over a network often uses a network daemon. To send a file somewhere, a user puts it in a network spooling directory. Later on, the network daemon takes it out and transmits it. One particular use of spooled file

transmission is the USENET News system. This network consists of millions of machines around the world communicating using the Internet. Thousands of news groups exist on many topics. To post a news message, the user invokes a news program, which accepts the message to be posted and then deposits it in a spooling directory for transmission to other machines later. The entire news system runs outside the operating system.

Figure 5-17 summarizes the I/O system, showing all the layers and the principal functions of each layer. Starting at the bottom, the layers are the hardware, interrupt handlers, device drivers, device-independent software, and finally the user processes.

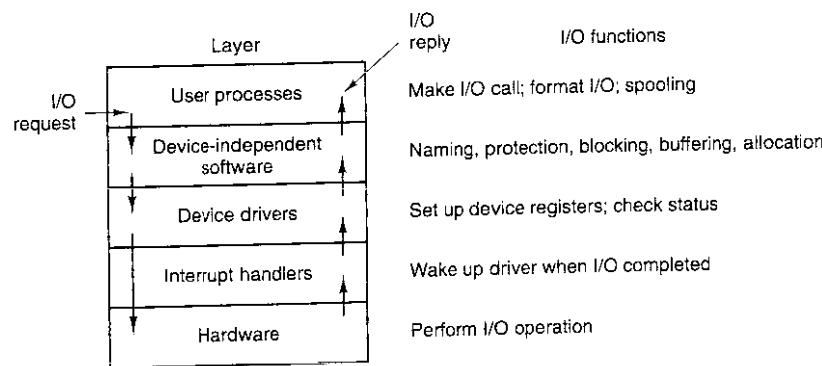


Figure 5-17. Layers of the I/O system and the main functions of each layer.

The arrows in Fig. 5-17 show the flow of control. When a user program tries to read a block from a file, for example, the operating system is invoked to carry out the call. The device-independent software looks for it in the buffer cache, for example. If the needed block is not there, it calls the device driver to issue the request to the hardware to go get it from the disk. The process is then blocked until the disk operation has been completed.

When the disk is finished, the hardware generates an interrupt. The interrupt handler is run to discover what has happened, that is, which device wants attention right now. It then extracts the status from the device and wakes up the sleeping process to finish off the I/O request and let the user process continue.

5.4 DISKS

Now we will begin studying some real I/O devices. We will begin with disks, which are conceptually simple, yet very important. After that we will examine clocks, keyboards, and displays.

5.4.1 Disk Hardware

Disk come in a variety of types. The most common ones are the magnetic disks (hard disks and floppy disks). They are characterized by the fact that reads and writes are equally fast, which makes them ideal as secondary memory (paging, file systems, etc.). Arrays of these disks are sometimes used to provide highly reliable storage. For distribution of programs, data, and movies, various kinds of optical disks (CD-ROMs, CD-Recordables, and DVDs) are also important. In the following sections we will first describe the hardware and then the software for these devices.

Magnetic Disks

Magnetic disks are organized into cylinders, each one containing as many tracks as there are heads stacked vertically. The tracks are divided into sectors, with the number of sectors around the circumference typically being 8 to 32 on floppy disks, and up to several hundred on hard disks. The number of heads varies from 1 to about 16.

Older disks have little electronics and just deliver a simple serial bit stream. On these disks, the controller does most of the work. On other disks, in particular, **IDE (Integrated Drive Electronics)** and **SATA (Serial ATA)** disks, the disk drive itself contains a microcontroller that does considerable work and allows the real controller to issue a set of higher-level commands. The controller often does track caching, bad block remapping, and much more.

A device feature that has important implications for the disk driver is the possibility of a controller doing seeks on two or more drives at the same time. These are known as **overlapped seeks**. While the controller and software are waiting for a seek to complete on one drive, the controller can initiate a seek on another drive. Many controllers can also read or write on one drive while seeking on one or more other drives, but a floppy disk controller cannot read or write on two drives at the same time. (Reading or writing requires the controller to move bits on a microsecond time scale, so one transfer uses up most of its computing power.) The situation is different for hard disks with integrated controllers, and in a system with more than one of these hard drives they can operate simultaneously, at least to the extent of transferring between the disk and the controller's buffer memory. Only one transfer between the controller and the main memory is possible at once, however. The ability to perform two or more operations at the same time can reduce the average access time considerably.

Figure 5-18 compares parameters of the standard storage medium for the original IBM PC with parameters of a disk made 20 years later to show how much disks changed in 20 years. It is interesting to note that not all parameters have improved as much. Average seek time is seven times better than it was, transfer rate is 1300 times better, while capacity is up by a factor of 50,000. This pattern

has to do with relatively gradual improvements in the moving parts, but much higher bit densities on the recording surfaces.

Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 μ sec

Figure 5-18. Disk parameters for the original IBM PC 360-KB floppy disk and a Western Digital WD 18300 hard disk.

One thing to be aware of in looking at the specifications of modern hard disks is that the geometry specified, and used by the driver software, is almost always different from the physical format. On old disks, the number of sectors per track was the same for all cylinders. Modern disks are divided into zones with more sectors on the outer zones than the inner ones. Fig. 5-19(a) illustrates a tiny disk with two zones. The outer zone has 32 sectors per track; the inner one has 16 sectors per track. A real disk, such as the WD 18300, typically has 16 or more zones, with the number of sectors increasing by about 4% per zone as one goes out from the innermost zone to the outermost zone.

To hide the details of how many sectors each track has, most modern disks have a virtual geometry that is presented to the operating system. The software is instructed to act as though there are x cylinders, y heads, and z sectors per track. The controller then remaps a request for (x, y, z) onto the real cylinder, head, and sector. A possible virtual geometry for the physical disk of Fig. 5-19(a) is shown in Fig. 5-19(b). In both cases the disk has 192 sectors, only the published arrangement is different than the real one.

For PCs, the maximum values for these three parameters are often (65535, 16, and 63), due to the need to be backward compatible with the limitations of the original IBM PC. On this machine, 16-, 4-, and 6-bit fields were used to specify these numbers, with cylinders and sectors numbered starting at 1 and heads numbered starting at 0. With these parameters and 512 bytes per sector, the largest possible disk is 31.5 GB. To get around this limit, all modern disks now support a

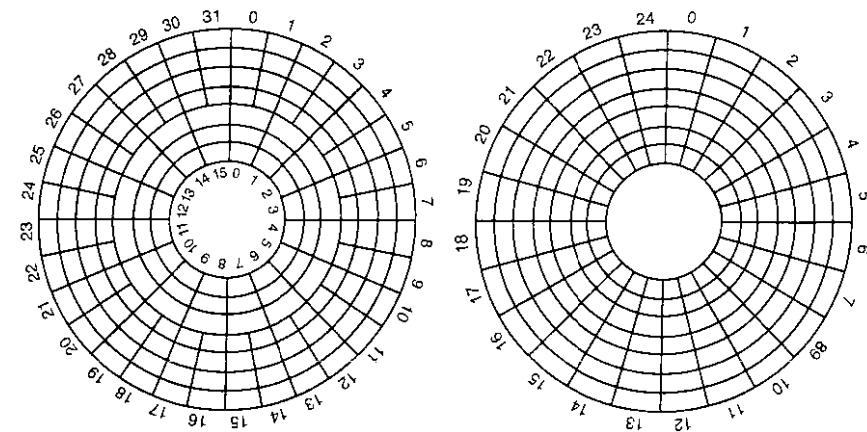


Figure 5-19. (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

system called **logical block addressing**, in which disk sectors are just numbered consecutively starting at 0, without regard to the disk geometry.

RAID

CPU performance has been increasing exponentially over the past decade, roughly doubling every 18 months. Not so with disk performance. In the 1970s, average seek times on minicomputer disks were 50 to 100 msec. Now seek times are slightly under 10 msec. In most technical industries (say, automobiles or aviation), a factor of 5 to 10 performance improvement in two decades would be major news (imagine 300 MPG cars), but in the computer industry it is an embarrassment. Thus the gap between CPU performance and disk performance has become much larger over time.

As we have seen, parallel processing is being used more and more to speed up CPU performance. It has occurred to various people over the years that parallel I/O might be a good idea too. In their 1988 paper, Patterson et al. suggested six specific disk organizations that could be used to improve disk performance, reliability, or both (Patterson et al., 1988). These ideas were quickly adopted by industry and have led to a new class of I/O device called a **RAID**. Patterson et al. defined RAID as **Redundant Array of Inexpensive Disks**, but industry redefined the I to be "Independent" rather than "Inexpensive" (maybe so they could charge more?). Since a villain was also needed (as in RISC versus CISC, also due to Patterson), the bad guy here was the **SLED** (Single Large Expensive Disk).

The basic idea behind a RAID is to install a box full of disks next to the computer, typically a large server, replace the disk controller card with a RAID controller, copy the data over to the RAID, and then continue normal operation. In other words, a RAID should look like a SLED to the operating system but have better performance and better reliability. Since SCSI disks have good performance, low price, and the ability to have up to seven drives on a single controller (15 for wide SCSI), it is natural that most RAIDs consist of a RAID SCSI controller plus a box of SCSI disks that appear to the operating system as a single large disk. In this way, no software changes are required to use the RAID, a big selling point for many system administrators.

In addition to appearing like a single disk to the software, all RAIDs have the property that the data are distributed over the drives, to allow parallel operation. Several different schemes for doing this were defined by Patterson et al., and they are now known as RAID level 0 through RAID level 5. In addition, there are a few other minor levels that we will not discuss. The term "level" is something of a misnomer since there is no hierarchy involved; there are simply six different organizations possible.

RAID level 0 is illustrated in Fig. 5-20(a). It consists of viewing the virtual single disk simulated by the RAID as being divided up into strips of k sectors each, with sectors 0 to $k - 1$ being strip 0, sectors k to $2k - 1$ as strip 1, and so on. For $k = 1$, each strip is a sector; for $k = 2$ a strip is two sectors, etc. The RAID level 0 organization writes consecutive strips over the drives in round-robin fashion, as depicted in Fig. 5-20(a) for a RAID with four disk drives.

Distributing data over multiple drives like this is called **striping**. For example, if the software issues a command to read a data block consisting of four consecutive strips starting at a strip boundary, the RAID controller will break this command up into four separate commands, one for each of the four disks, and have them operate in parallel. Thus we have parallel I/O without the software knowing about it.

RAID level 0 works best with large requests, the bigger the better. If a request is larger than the number of drives times the strip size, some drives will get multiple requests, so that when they finish the first request they start the second one. It is up to the controller to split the request up and feed the proper commands to the proper disks in the right sequence and then assemble the results in memory correctly. Performance is excellent and the implementation is straightforward.

RAID level 0 works worst with operating systems that habitually ask for data one sector at a time. The results will be correct, but there is no parallelism and hence no performance gain. Another disadvantage of this organization is that the reliability is potentially worse than having a SLED. If a RAID consists of four disks, each with a mean time to failure of 20,000 hours, about once every 5000 hours a drive will fail and all the data will be completely lost. A SLED with a mean time to failure of 20,000 hours would be four times more reliable. Because no redundancy is present in this design, it is not really a true RAID.

The next option, RAID level 1, shown in Fig. 5-20(b), is a true RAID. It duplicates all the disks, so there are four primary disks and four backup disks. On a write, every strip is written twice. On a read, either copy can be used, distributing the load over more drives. Consequently, write performance is no better than for a single drive, but read performance can be up to twice as good. Fault tolerance is excellent: if a drive crashes, the copy is simply used instead. Recovery consists of simply installing a new drive and copying the entire backup drive to it.

Unlike levels 0 and 1, which work with strips of sectors, RAID level 2 works on a word basis, possibly even a byte basis. Imagine splitting each byte of the single virtual disk into a pair of 4-bit nibbles, then adding a Hamming code to each one to form a 7-bit word, of which bits 1, 2, and 4 were parity bits. Further imagine that the seven drives of Fig. 5-20(c) were synchronized in terms of arm position and rotational position. Then it would be possible to write the 7-bit Hamming coded word over the seven drives, one bit per drive.

The Thinking Machines CM-2 computer used this scheme, taking 32-bit data words and adding 6 parity bits to form a 38-bit Hamming word, plus an extra bit for word parity, and spread each word over 39 disk drives. The total throughput was immense, because in one sector time it could write 32 sectors worth of data. Also, losing one drive did not cause problems, because loss of a drive amounted to losing 1 bit in each 39-bit word read, something the Hamming code could handle on the fly.

On the down side, this scheme requires all the drives to be rotationally synchronized, and it only makes sense with a substantial number of drives (even with 32 data drives and 6 parity drives, the overhead is 19%). It also asks a lot of the controller, since it must do a Hamming checksum every bit time.

RAID level 3 is a simplified version of RAID level 2. It is illustrated in Fig. 5-20(d). Here a single parity bit is computed for each data word and written to a parity drive. As in RAID level 2, the drives must be exactly synchronized, since individual data words are spread over multiple drives.

At first thought, it might appear that a single parity bit gives only error detection, not error correction. For the case of random undetected errors, this observation is true. However, for the case of a drive crashing, it provides full 1-bit error correction since the position of the bad bit is known. If a drive crashes, the controller just pretends that all its bits are 0s. If a word has a parity error, the bit from the dead drive must have been a 1, so it is corrected. Although both RAID levels 2 and 3 offer very high data rates, the number of separate I/O requests per second they can handle is no better than for a single drive.

RAID levels 4 and 5 work with strips again, not individual words with parity, and do not require synchronized drives. RAID level 4 [see Fig. 5-20(e)] is like RAID level 0, with a strip-for-strip parity written onto an extra drive. For example, if each strip is k bytes long, all the strips are EXCLUSIVE ORed together, resulting in a parity strip k bytes long. If a drive crashes, the lost bytes can be recomputed from the parity drive by reading the entire set of drives.

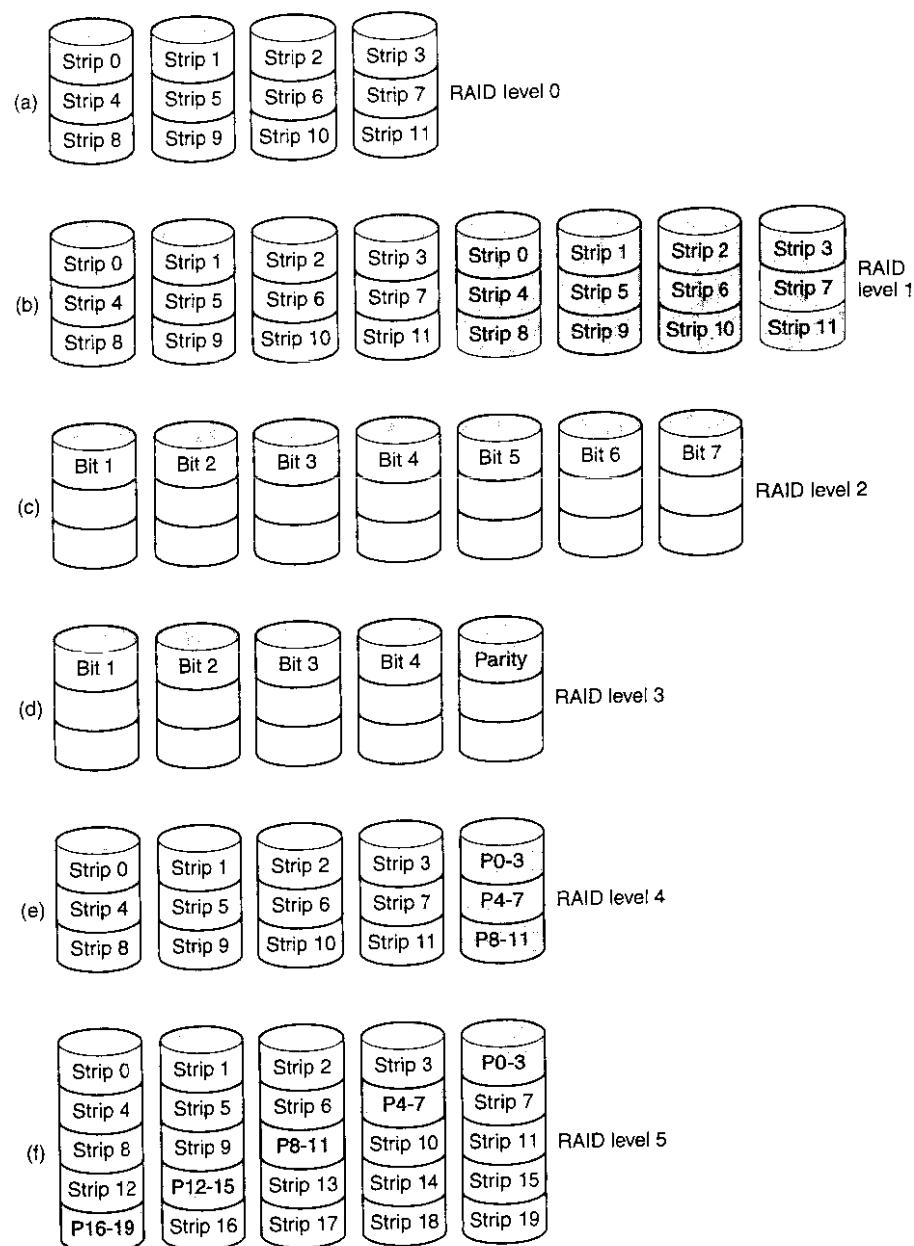


Figure 5-20. RAID levels 0 through 5. Backup and parity drives are shown shaded.

This design protects against the loss of a drive but performs poorly for small updates. If one sector is changed, it is necessary to read all the drives in order to recalculate the parity, which must then be rewritten. Alternatively, it can read the old user data and the old parity data and recompute the new parity from them. Even with this optimization, a small update requires two reads and two writes.

As a consequence of the heavy load on the parity drive, it may become a bottleneck. This bottleneck is eliminated in RAID level 5 by distributing the parity bits uniformly over all the drives, round robin fashion, as shown in Fig. 5-20(f). However, in the event of a drive crash, reconstructing the contents of the failed drive is a complex process.

CD-ROMs

In recent years, optical (as opposed to magnetic) disks have become available. They have much higher recording densities than conventional magnetic disks. Optical disks were originally developed for recording television programs, but they can be put to more esthetic use as computer storage devices. Due to their potentially enormous capacity, optical disks have been the subject of a great deal of research and have gone through an incredibly rapid evolution.

First-generation optical disks were invented by the Dutch electronics conglomerate Philips for holding movies. They were 30 cm across and marketed under the name LaserVision, but they did not catch on, except in Japan.

In 1980, Philips, together with Sony, developed the CD (Compact Disc), which rapidly replaced the 33 1/3-RPM vinyl record for music (except among connoisseurs, who still prefer vinyl). The precise technical details for the CD were published in an official International Standard (IS 10149), popularly called the **Red Book**, due to the color of its cover. (International Standards are issued by the International Organization for Standardization, which is the international counterpart of national standards groups like ANSI, DIN, etc. Each one has an IS number.) The point of publishing the disk and drive specifications as an International Standard is to allow CDs from different music publishers and players from different electronics manufacturers to work together. All CDs are 120 mm across and 1.2 mm thick, with a 15-mm hole in the middle. The audio CD was the first successful mass market digital storage medium. They are supposed to last 100 years. Please check back in 2080 for an update on how well the first batch did.

A CD is prepared in several steps. The step consists of using a high-power infrared laser to burn 0.8-micron diameter holes in a coated glass master disk. From this master, a mold is made, with bumps where the laser holes were. Into this mold, molten polycarbonate resin is injected to form a CD with the same pattern of holes as the glass master. Then a very thin layer of reflective aluminum is deposited on the polycarbonate, topped by a protective lacquer and finally a label. The depressions in the polycarbonate substrate are called **pits**; the unburned areas between the pits are called **lands**.

When played back, a low-power laser diode shines infrared light with a wavelength of 0.78 micron on the pits and lands as they stream by. The laser is on the polycarbonate side, so the pits stick out toward the laser as bumps in the otherwise flat surface. Because the pits have a height of one-quarter the wavelength of the flat surface. Because the pits have a height of one-quarter the wavelength of the flat surface. Because the pits have a height of one-quarter the wavelength of the flat surface. As a result, the two parts interfere destructively and return less light to the player's photodetector than light bouncing off a land. This is how the player tells a pit from a land. Although it might seem simpler to use a pit to record a 0 and a land to record a 1, it is more reliable to use a pit/land or land/pit transition for a 1 and its absence as a 0, so this scheme is used.

The pits and lands are written in a single continuous spiral starting near the center and working out a distance of 32 mm toward the edge. The spiral makes 22,188 revolutions around the disk (about 600 per mm). If unwound, it would be 5.6 km long. The spiral is illustrated in Fig. 5-21.

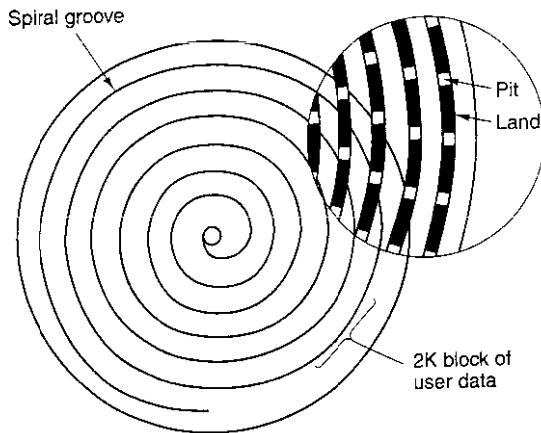


Figure 5-21. Recording structure of a compact disc or CD-ROM.

To make the music play at a uniform rate, it is necessary for the pits and lands to stream by at a constant linear velocity. Consequently, the rotation rate of the CD must be continuously reduced as the reading head moves from the inside of the CD to the outside. At the inside, the rotation rate is 530 RPM to achieve the desired streaming rate of 120 cm/sec; at the outside it has to drop to 200 RPM to give the same linear velocity at the head. A constant linear velocity drive is quite different than a magnetic disk drive, which operates at a constant angular velocity, independent of where the head is currently positioned. Also, 530 RPM is a far cry from the 3600 to 7200 RPM that most magnetic disks whirl at.

In 1984, Philips and Sony realized the potential for using CDs to store computer data, so they published the **Yellow Book** defining a precise standard for

what are now called **CD-ROMs (Compact Disc - Read Only Memory)**. To piggyback on the by-then already substantial audio CD market, CD-ROMs were to be the same physical size as audio CDs, mechanically and optically compatible with them, and produced using the same polycarbonate injection molding machines. The consequences of this decision were not only that slow variable-speed motors were required, but also that the manufacturing cost of a CD-ROM would be well under one dollar in moderate volume.

What the Yellow Book defined was the formatting of the computer data. It also improved the error-correcting abilities of the system, an essential step because although music lovers do not mind losing a bit here and there, computer lovers tend to be Very Picky about that. The basic format of a CD-ROM consists of encoding every byte in a 14-bit symbol, which is enough to Hamming encode an 8-bit byte with 2 bits left over. In fact, a more powerful encoding system is used. The 14-to-8 mapping for reading is done in hardware by table lookup.

At the next level up, a group of 42 consecutive symbols forms a 588-bit **frame**. Each frame holds 192 data bits (24 bytes). The remaining 396 bits are used for error correction and control. Of these, 252 are the error-correction bits in the 14-bit symbols and 144 are carried in the 8-bit symbol payloads. So far, this scheme is identical for audio CDs and CD-ROMs.

What the Yellow Book adds is the grouping of 98 frames into a **CD-ROM sector**, as shown in Fig. 5-22. Every CD-ROM sector begins with a 16-byte preamble, the first 12 of which are 00FFFFFFFFFFFF00 (hexadecimal), to allow the player to recognize the start of a CD-ROM sector. The next 3 bytes contain the sector number, needed because seeking on a CD-ROM with its single data spiral is much more difficult than on a magnetic disk with its uniform concentric tracks. To seek, the software in the drive calculates approximately where to go, moves the head there, and then starts hunting around for a preamble to see how good its guess was. The last byte of the preamble is the mode.

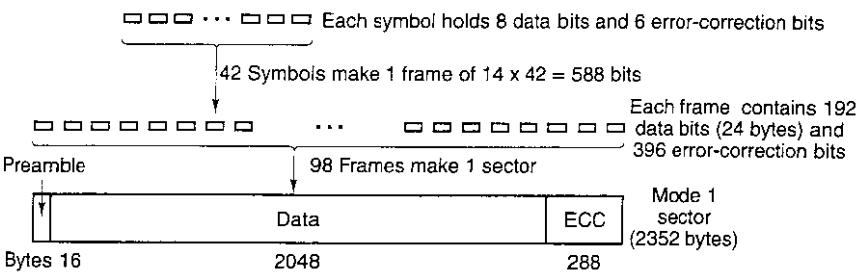


Figure 5-22. Logical data layout on a CD-ROM.

The Yellow Book defines two modes. Mode 1 uses the layout of Fig. 5-22, with a 16-byte preamble, 2048 data bytes, and a 288-byte error-correcting code (a

crossinterleaved Reed-Solomon code). Mode 2 combines the data and ECC fields into a 2336-byte data field for those applications that do not need (or cannot afford the time to perform) error correction, such as audio and video. Note that to provide excellent reliability, three separate error-correcting schemes are used: within a symbol, within a frame, and within a CD-ROM sector. Single-bit errors are corrected at the lowest level, short burst errors are corrected at the frame level, and any residual errors are caught at the sector level. The price paid for this reliability is that it takes 98 frames of 588 bits (7203 bytes) to carry a single 2048-byte payload, an efficiency of only 28%.

Single-speed CD-ROM drives operate at 75 sectors/sec, which gives a data rate of 153,600 bytes/sec in mode 1 and 175,200 bytes/sec in mode 2. Double-speed drives are twice as fast, and so on up to the highest speed. Thus a 40x drive can deliver data at a rate of $40 \times 153,600$ bytes/sec, assuming that the drive interface, bus, and operating system can all handle this data rate. A standard audio CD has room for 74 minutes of music, which, if used for mode 1 data, gives a capacity of 681,984,000 bytes. This figure is usually reported as 650 MB because 1 MB is 2^{20} bytes (1,048,576 bytes), not 1,000,000 bytes.

Note that even a 32x CD-ROM drive (4,915,200 bytes/sec) is no match for a fast SCSI-2 magnetic disk drive at 10 MB/sec, even though many CD-ROM drives use the SCSI interface (IDE CD-ROM drives also exist). When you realize that the seek time is usually several hundred milliseconds, it should be clear that CD-ROM drives are not in the same performance category as magnetic disk drives, despite their large capacity.

In 1986, Philips struck again with the **Green Book**, adding graphics and the ability to interleave audio, video, and data in the same sector, a feature essential for multimedia CD-ROMs.

The last piece of the CD-ROM puzzle is the file system. To make it possible to use the same CD-ROM on different computers, agreement was needed on CD-ROM file systems. To get this agreement, representatives of many computer companies met at Lake Tahoe in the High Sierras on the California-Nevada boundary and devised a file system that they called **High Sierra**. It later evolved into an International Standard (IS 9660). It has three levels. Level 1 uses file names of up to 8 characters optionally followed by an extension of up to 3 characters (the MS-DOS file naming convention). File names may contain only upper case letters, digits, and the underscore. Directories may be nested up to eight deep, but directory names may not contain extensions. Level 1 requires all files to be contiguous, which is not a problem on a medium written only once. Any CD-ROM conformant to IS 9660 level 1 can be read using MS-DOS, an Apple computer, a UNIX computer, or just about any other computer. CD-ROM publishers regard this property as being a big plus.

IS 9660 level 2 allows names up to 32 characters, and level 3 allows noncontiguous files. The Rock Ridge extensions (whimsically named after the town in the Gene Wilder film *Blazing Saddles*) allow very long names (for UNIX), UIDs,

GIDs, and symbolic links, but CD-ROMs not conforming to level 1 will not be readable on all computers.

CD-ROMs have become extremely popular for publishing games, movies, encyclopedias, atlases, and reference works of all kinds. Most commercial software now comes on CD-ROMs. Their combination of large capacity and low manufacturing cost makes them well suited to innumerable applications.

CD-Recordables

Initially, the equipment needed to produce a master CD-ROM (or audio CD, for that matter) was extremely expensive. But as usual in the computer industry, nothing stays expensive for long. By the mid 1990s, CD recorders no bigger than a CD player were a common peripheral available in most computer stores. These devices were still different from magnetic disks because once written, CD-ROMs could not be erased. Nevertheless, they quickly found a niche as a backup medium for large hard disks and also allowed individuals or startup companies to manufacture their own small-run CD-ROMs or make masters for delivery to high-volume commercial CD duplication plants. These drives are known as **CD-Rs (CD-Recordables)**.

Physically, CD-Rs start with 120-mm polycarbonate blanks that are like CD-ROMs, except that they contain a 0.6-mm wide groove to guide the laser for writing. The groove has a sinusoidal excursion of 0.3 mm at a frequency of exactly 22.05 kHz to provide continuous feedback so the rotation speed can be accurately monitored and adjusted if need be. CD-Rs look like regular CD-ROMs, except that they are gold colored on top instead of silver colored. The gold color comes from the use of real gold instead of aluminum for the reflective layer. Unlike silver CDs, which have physical depressions on them, on CD-Rs the differing reflectivity of pits and lands has to be simulated. This is done by adding a layer of dye between the polycarbonate and the reflective gold layer, as shown in Fig. 5-23. Two kinds of dye are used: cyanine, which is green, and pthalocyanine, which is a yellowish orange. Chemists can argue endlessly about which one is better. These dyes are similar to those used in photography, which explains why Eastman Kodak and Fuji are major manufacturers of blank CD-Rs.

In its initial state, the dye layer is transparent and lets the laser light pass through and reflect off the gold layer. To write, the CD-R laser is turned up to high power (8–16 mW). When the beam hits a spot of dye, it heats up, breaking a chemical bond. This change to the molecular structure creates a dark spot. When read back (at 0.5 mW), the photodetector sees a difference between the dark spots where the dye has been hit and transparent areas where it is intact. This difference is interpreted as the difference between pits and lands, even when read back on a regular CD-ROM reader or even on an audio CD player.

No new kind of CD could hold up its head with pride without a colored book, so CD-R has the **Orange Book**, published in 1989. This document defines CD-R

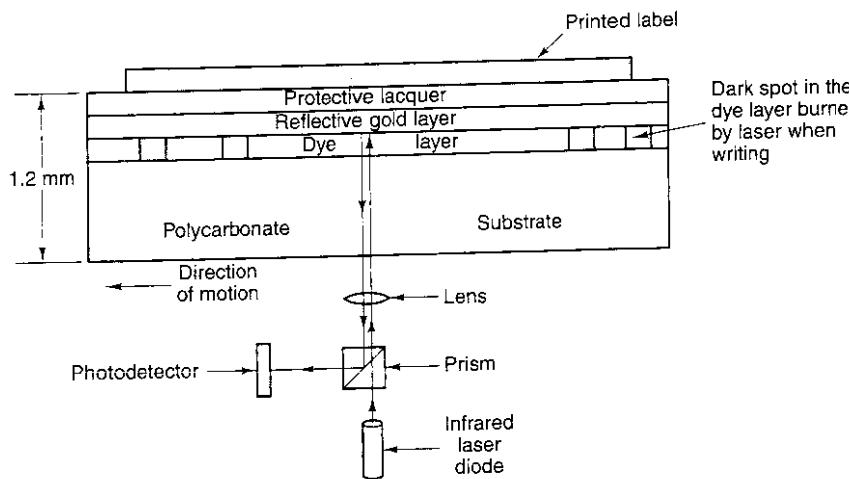


Figure 5-23. Cross section of a CD-R disk and laser (not to scale). A silver CD-ROM has a similar structure, except without the dye layer and with a pitted aluminum layer instead of a gold layer.

and also a new format, **CD-ROM XA**, which allows CD-Rs to be written incrementally, a few sectors today, a few tomorrow, and a few next month. A group of consecutive sectors written at once is called a **CD-ROM track**.

One of the first uses of CD-R was for the Kodak PhotoCD. In this system the customer brings a roll of exposed film and his old PhotoCD to the photo processor and gets back the same PhotoCD with the new pictures added after the old ones. The new batch, which is created by scanning in the negatives, is written onto the PhotoCD as a separate CD-ROM track. Incremental writing was needed because when this product was introduced, the CD-R blanks were too expensive to provide a new one for every film roll.

However, incremental writing creates a new problem. Prior to the Orange Book, all CD-ROMs had a single **VTOC (Volume Table of Contents)** at the start. That scheme does not work with incremental (i.e., multitrack) writes. The Orange Book's solution is to give each CD-ROM track its own VTOC. The files listed in the VTOC can include some or all of the files from previous tracks. After the CD-R is inserted into the drive, the operating system searches through all the CD-ROM tracks to locate the most recent VTOC, which gives the current status of the disk. By including some, but not all, of the files from previous tracks in the current VTOC, it is possible to give the illusion that files have been deleted. Tracks can be grouped into **sessions**, leading to **multisession** CD-ROMs. Standard audio CD players cannot handle multisession CDs since they expect a single VTOC at the start. Some computer applications can handle them, though.

CD-R makes it possible for individuals and companies to easily copy CD-ROMs (and audio CDs), generally in violation of the publisher's copyright. Several schemes have been devised to make such piracy harder and to make it difficult to read a CD-ROM using anything other than the publisher's software. One of them involves recording all the file lengths on the CD-ROM as multigigabyte, thwarting any attempts to copy the files to hard disk using standard copying software. The true lengths are embedded in the publisher's software or hidden (possibly encrypted) on the CD-ROM in an unexpected place. Another scheme uses intentionally incorrect ECCs in selected sectors, in the expectation that CD copying software will "fix" the errors. The application software checks the ECCs itself, refusing to work if they are correct. Using nonstandard gaps between the tracks and other physical "defects" are also possibilities.

CD-Rewritables

Although people are used to other write-once media such as paper and photographic film, there is a demand for a rewritable CD-ROM. One technology now available is **CD-RW (CD-ReWritable)**, which uses the same size media as CD-R. However, instead of cyanine or phthalocyanine dye, CR-RW uses an alloy of silver, indium, antimony, and tellurium for the recording layer. This alloy has two stable states: crystalline and amorphous, with different reflectivities.

CD-RW drives use lasers with three different powers. At high power, the laser melts the alloy, converting it from the high-reflectivity crystalline state to the low-reflectivity amorphous state to represent a pit. At medium power, the alloy melts and reforms in its natural crystalline state to become a land again. At low power, the state of the material is sensed (for reading), but no phase transition occurs.

The reason CD-RW has not replaced CD-R is that the CD-RW blanks are more expensive than the CR-R blanks. Also, for applications consisting of backing up hard disks, the fact that once written, a CD-R cannot be accidentally erased is a big plus.

DVD

The basic CD/CD-ROM format has been around since 1980. The technology has improved since then, so higher-capacity optical disks are now economically feasible and there is great demand for them. Hollywood would dearly love to eliminate analog video tapes in favor of digital disks, since disks have a higher quality, are cheaper to manufacture, last longer, take up less shelf space in video stores, and do not have to be rewound. The consumer electronics companies are always looking for a new blockbuster product, and many computer companies want to add multimedia features to their software.

This combination of technology and demand by three immensely rich and powerful industries led to **DVD**, originally an acronym for **Digital Video Disk**.

but now officially **Digital Versatile Disk**. DVDs use the same general design as CDs, with 120-mm injection-molded polycarbonate disks containing pits and lands that are illuminated by a laser diode and read by a photodetector. What is new is the use of

1. Smaller pits (0.4 microns versus 0.8 microns for CDs).
2. A tighter spiral (0.74 microns between tracks versus 1.6 microns for CDs).
3. A red laser (at 0.65 microns versus 0.78 microns for CDs).

Together, these improvements raise the capacity sevenfold, to 4.7 GB. A 1x DVD drive operates at 1.4 MB/sec (versus 150 KB/sec for CDs). Unfortunately, the switch to the red lasers used in supermarkets means that DVD players require a second laser or fancy conversion optics to be able to read existing CDs and CD-ROMs. But with the drop in price of lasers, most of them now have both of them so they can read both kinds of media.

Is 4.7 GB enough? Maybe. Using MPEG-2 compression (standardized in IS 13346), a 4.7 GB DVD disk can hold 133 minutes of full-screen, full-motion video at high resolution (720×480), as well as soundtracks in up to eight languages and subtitles in 32 more. About 92% of all the movies Hollywood has ever made are under 133 minutes. Nevertheless, some applications such as multimedia games or reference works may need more, and Hollywood would like to put multiple movies on the same disk, so four formats have been defined:

1. Single-sided, single-layer (4.7 GB).
2. Single-sided, dual-layer (8.5 GB).
3. Double-sided, single-layer (9.4 GB).
4. Double-sided, dual-layer (17 GB).

Why so many formats? In a word: politics. Philips and Sony wanted single-sided, dual-layer disks for the high capacity version, but Toshiba and Time Warner wanted double-sided, single-layer disks. Philips and Sony did not think people would be willing to turn the disks over, and Time Warner did not believe putting two layers on one side could be made to work. The compromise: all combinations, but the market will determine which ones survive.

The dual layering technology has a reflective layer at the bottom, topped with a semireflective layer. Depending on where the laser is focused, it bounces off one layer or the other. The lower layer needs slightly larger pits and lands to be read reliably, so its capacity is slightly smaller than the upper layer's.

Double-sided disks are made by taking two 0.6-mm single-sided disks and gluing them together back to back. To make the thicknesses of all versions the same, a single-sided disk consists of a 0.6-mm disk bonded to a blank substrate (or perhaps in the future, one consisting of 133 minutes of advertising, in the hope

that people will be curious as to what is down there). The structure of the double-sided, dual-layer disk is illustrated in Fig. 5-24.

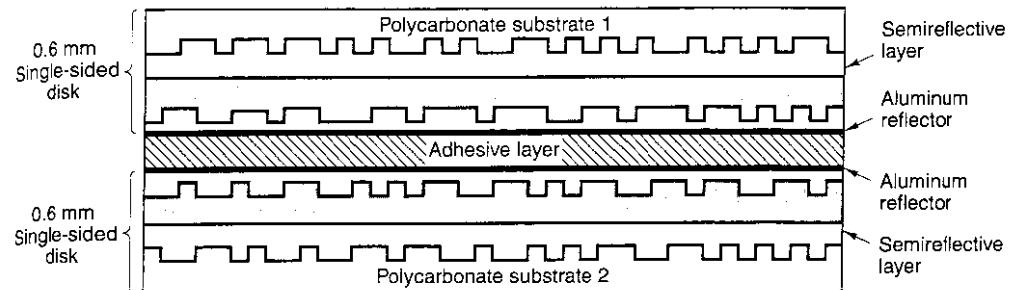


Figure 5-24. A double-sided, dual-layer DVD disk.

DVD was devised by a consortium of 10 consumer electronics companies, seven of them Japanese, in close cooperation with the major Hollywood studios (some of which are owned by the Japanese electronics companies in the consortium). The computer and telecommunications industries were not invited to the picnic, and the resulting focus was on using DVD for movie rental and sales shows. For example, standard features include real-time skipping of dirty scenes (to allow parents to turn a film rated NC17 into one safe for toddlers), six-channel sound, and support for Pan-and-Scan. The latter feature allows the DVD player to dynamically decide how to crop the left and right edges off movies (whose width:height ratio is 3:2) to fit on current television sets (whose aspect ratio is 4:3).

Another item the computer industry probably would not have thought of is an intentional incompatibility between disks intended for the United States and disks intended for Europe and yet other standards for other continents. Hollywood demanded this "feature" because new films are always released first in the United States and then shipped to Europe when the videos come out in the United States. The idea was to make sure European video stores could not buy videos in the U.S. too early, thereby reducing new movies' European theater sales. If Hollywood had been running the computer industry, we would have had 3.5-inch floppy disks in the United States and 9-cm floppy disks in Europe.

The folks who brought you single/double-sided DVDs and single/double-layer DVDs are at it again. The next generation also lacks a single standard due to political bickering by the industry players. One of the new devices is **Blu-ray**, which uses a 0.405 micron (blue) laser to pack 25 GB onto a single-layer disk and 50-GB onto a double-layer disk. The other one is **HD DVD**, which uses the same blue laser but has a capacity of only 15 GB (single layer) and 30 GB (double layer). This format war has split the movie studios, the computer manufacturers,

and the software companies. As a result of the lack of standardization, this generation is taking off rather slowly as consumers wait for the dust to settle to see which format will win. This stupidity on the part of the industry brings to mind George Santayana's famous remark: "Those who cannot learn from history are doomed to repeat it."

5.4.2 Disk Formatting

A hard disk consists of a stack of aluminum, alloy, or glass platters 5.25 inch or 3.5 inch in diameter (or even smaller on notebook computers). On each platter is deposited a thin magnetizable metal oxide. After manufacturing, there is no information whatsoever on the disk.

Before the disk can be used, each platter must receive a **low-level format** done by software. The format consists of a series of concentric tracks, each containing some number of sectors, with short gaps between the sectors. The format of a sector is shown in Fig. 5-25.

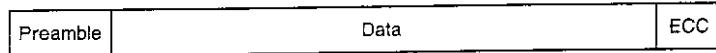


Figure 5-25. A disk sector.

The preamble starts with a certain bit pattern that allows the hardware to recognize the start of the sector. It also contains the cylinder and sector numbers and some other information. The size of the data portion is determined by the low-level formatting program. Most disks use 512-byte sectors. The ECC field contains redundant information that can be used to recover from read errors. The size and content of this field varies from manufacturer to manufacturer, depending on how much disk space the designer is willing to give up for higher reliability and how complex an ECC code the controller can handle. A 16-byte ECC field is not unusual. Furthermore, all hard disks have some number of spare sectors allocated to be used to replace sectors with a manufacturing defect.

The position of sector 0 on each track is offset from the previous track when the low-level format is laid down. This offset, called **cylinder skew**, is done to improve performance. The idea is to allow the disk to read multiple tracks in one continuous operation without losing data. The nature of the problem can be seen by looking at Fig. 5-19(a). Suppose that a request needs 18 sectors starting at sector 0 on the innermost track. Reading the first 16 sectors takes one disk rotation, but a seek is needed to move outward one track to get the 17th sector. By the time the head has moved one track, sector 0 has rotated past the head so an entire rotation is needed until it comes by again. That problem is eliminated by offsetting the sectors as shown in Fig. 5-26.

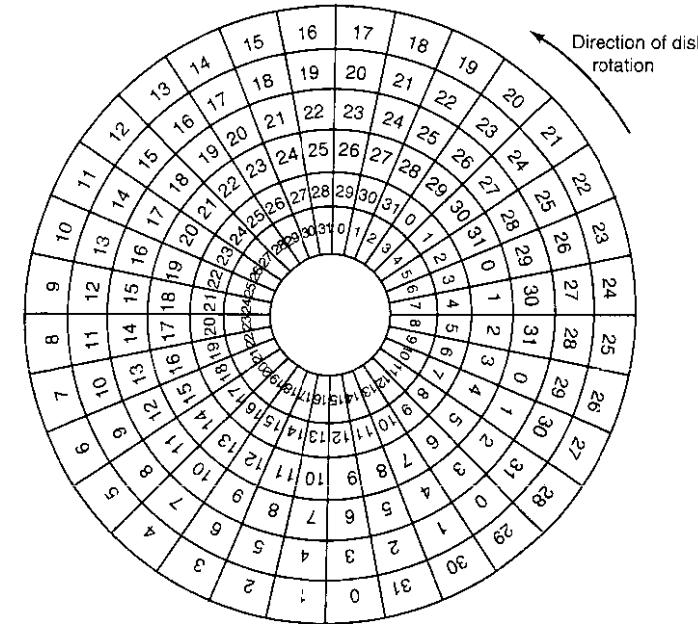


Figure 5-26. An illustration of cylinder skew.

The amount of cylinder skew depends on the drive geometry. For example, a 10,000-RPM drive rotates in 6 msec. If a track contains 300 sectors, a new sector passes under the head every 20 μ sec. If the track-to-track seek time is 800 μ sec, 40 sectors will pass by during the seek, so the cylinder skew should be 40 sectors, rather than the three sectors shown in Fig. 5-26. It is worth mentioning that switching between heads also takes a finite time, so there is **head skew** as well as cylinder skew, but head skew is not very large.

As a result of the low-level formatting, disk capacity is reduced, depending on the sizes of the preamble, intersector gap, and ECC, as well as the number of spare sectors reserved. Often the formatted capacity is 20% lower than the unformatted capacity. The spare sectors do not count toward the formatted capacity, so all disks of a given type have exactly the same capacity when shipped, independent of how many bad sectors they actually have (if the number of bad sectors exceeds the number of spares, the drive will be rejected and not shipped).

There is considerable confusion about disk capacity because some manufacturers advertised the unformatted capacity to make their drives look larger than they really are. For example, consider a drive whose unformatted capacity is 200×10^9 bytes. This might be sold as a 200-GB disk. However, after formatting, perhaps only 170×10^9 bytes are available for data. To add to the confusion, the

operating system will probably report this capacity as 158 GB, not 170 GB because software considers a memory of 1 GB to be 2^{30} (1,073,741,824) bytes, not 10^9 (1,000,000,000) bytes.

To make things worse, in the world of data communications, 1 Gbps means 1,000,000,000 bits/sec because the prefix *giga* really does mean 10^9 (a kilometer is 1000 meters, not 1024 meters, after all). Only with memory and disk sizes do kilo, mega, giga, and tera mean 2^{10} , 2^{20} , 2^{30} , and 2^{40} , respectively.

Formatting also affects performance. If a 10,000-RPM disk has 300 sectors per track of 512 bytes each, it takes 6 msec to read the 153,600 bytes on a track for a data rate of 25,600,000 bytes/sec or 24.4 MB/sec. It is not possible to go faster than this, no matter what kind of interface is present, even if it a SCSI interface at 80 MB/sec or 160 MB/sec.

Actually reading continuously at this rate requires a large buffer in the controller. Consider, for example, a controller with a one-sector buffer that has been given a command to read two consecutive sectors. After reading the first sector from the disk and doing the ECC calculation, the data must be transferred to main memory. While this transfer is taking place, the next sector will fly by the head. When the copy to memory is complete, the controller will have to wait almost an entire rotation time for the second sector to come around again.

This problem can be eliminated by numbering the sectors in an interleaved fashion when formatting the disk. In Fig. 5-27(a), we see the usual numbering pattern (ignoring cylinder skew here). In Fig. 5-27(b), we see **single interleaving**, which gives the controller some breathing space between consecutive sectors in order to copy the buffer to main memory.

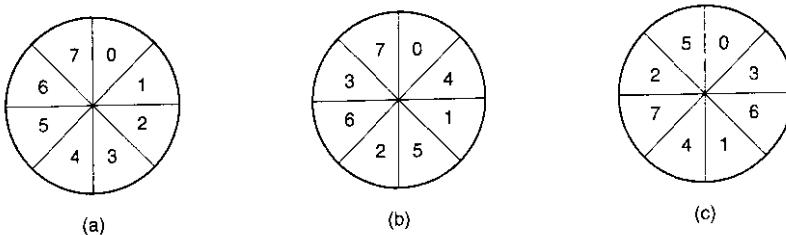


Figure 5-27. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

If the copying process is very slow, the **double interleaving** of Fig. 5-28(c) may be needed. If the controller has a buffer of only one sector, it does not matter whether the copying from the buffer to main memory is done by the controller, the main CPU, or a DMA chip; it still takes some time. To avoid the need for interleaving, the controller should be able to buffer an entire track. Many modern controllers can do this.

After low-level formatting is completed, the disk is partitioned. Logically, each partition is like a separate disk. Partitions are needed to allow multiple oper-

ating systems to coexist. Also, in some cases, a partition can be used for swapping. On the Pentium and most other computers, sector 0 contains the **master boot record**, which contains some boot code plus the partition table at the end. The partition table gives the starting sector and size of each partition. On the Pentium, the partition table has room for four partitions. If all of them are for Windows, they will be called C:, D:, E:, and F: and treated as separate drives. If three of them are for Windows and one is for UNIX, then Windows will call its partitions C:, D:, and E:. The first CD-ROM will then be F:. To be able to boot from the hard disk, one partition must be marked as active in the partition table.

The final step in preparing a disk for use is to perform a **high-level format** of each partition (separately). This operation lays down a boot block, the free storage administration (free list or bitmap), root directory, and an empty file system. It also puts a code in the partition table entry telling which file system is used in the partition because many operating systems support multiple incompatible file systems (for historical reasons). At this point the system can be booted.

When the power is turned on, the BIOS runs initially and then reads in the master boot record and jumps to it. This boot program then checks to see which partition is active. Then it reads in the boot sector from that partition and runs it. The boot sector contains a small program that general loads a larger bootstrap loader that searches the file system to find the operating system kernel. That program is loaded into memory and executed.

5.4.3 Disk Arm Scheduling Algorithms

In this section we will look at some issues related to disk drivers in general. First, consider how long it takes to read or write a disk block. The time required is determined by three factors:

1. Seek time (the time to move the arm to the proper cylinder).
2. Rotational delay (the time for the proper sector to rotate under the head).
3. Actual data transfer time.

For most disks, the seek time dominates the other two times, so reducing the mean seek time can improve system performance substantially.

If the disk driver accepts requests one at a time and carries them out in that order, that is, **First-Come, First-Served (FCFS)**, little can be done to optimize seek time. However, another strategy is possible when the disk is heavily loaded. It is likely that while the arm is seeking on behalf of one request, other disk requests may be generated by other processes. Many disk drivers maintain a table, indexed by cylinder number, with all the pending requests for each cylinder chained together in a linked list headed by the table entries.

Given this kind of data structure, we can improve upon the first-come, first-served scheduling algorithm. To see how, consider an imaginary disk with 40

cylinders. A request comes in to read a block on cylinder 11. While the seek to cylinder 11 is in progress, new requests come in for cylinders 1, 36, 16, 34, 9, and 12, in that order. They are entered into the table of pending requests, with a separate linked list for each cylinder. The requests are shown in Fig. 5-28.

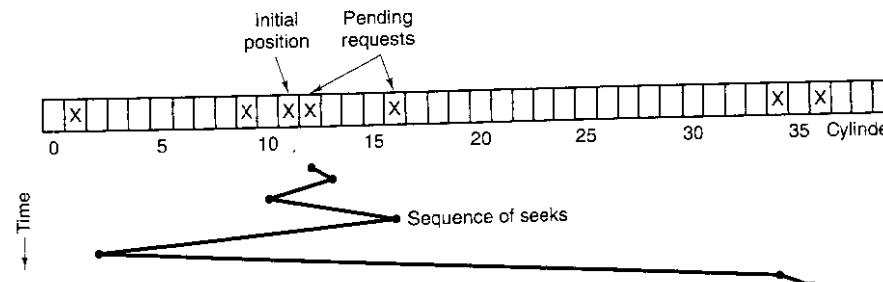


Figure 5-28. Shortest Seek First (SSF) disk scheduling algorithm.

When the current request (for cylinder 11) is finished, the disk driver has a choice of which request to handle next. Using FCFS, it would go next to cylinder 1, then to 36, and so on. This algorithm would require arm motions of 10, 35, 20, 18, 25, and 3, respectively, for a total of 111 cylinders.

Alternatively, it could always handle the closest request next, to minimize seek time. Given the requests of Fig. 5-28, the sequence is 12, 9, 16, 1, 34, and 36, shown as the jagged line at the bottom of Fig. 5-28. With this sequence, the arm motions are 1, 3, 7, 15, 33, and 2, for a total of 61 cylinders. This algorithm, **Shortest Seek First (SSF)**, cuts the total arm motion almost in half compared to FCFS.

Unfortunately, SSF has a problem. Suppose more requests keep coming in while the requests of Fig. 5-28 are being processed. For example, if, after going to cylinder 16, a new request for cylinder 8 is present, that request will have priority over cylinder 1. If a request for cylinder 13 then comes in, the arm will next go to 13, instead of 1. With a heavily loaded disk, the arm will tend to stay in the middle of the disk most of the time, so requests at either extreme will have to wait until a statistical fluctuation in the load causes there to be no requests near the middle. Requests far from the middle may get poor service. The goals of minimal response time and fairness are in conflict here.

Tall buildings also have to deal with this trade-off. The problem of scheduling an elevator in a tall building is similar to that of scheduling a disk arm. Requests come in continuously calling the elevator to floors (cylinders) at random. The computer running the elevator could easily keep track of the sequence in which customers pushed the call button and service them using FCFS or SSF.

However, most elevators use a different algorithm in order to reconcile the mutually conflicting goals of efficiency and fairness. They keep moving in the

same direction until there are no more outstanding requests in that direction, then they switch directions. This algorithm, known both in the disk world and the elevator world as the **elevator algorithm**, requires the software to maintain 1 bit: the current direction bit, *UP* or *DOWN*. When a request finishes, the disk or elevator driver checks the bit. If it is *UP*, the arm or cabin is moved to the next highest pending request. If no requests are pending at higher positions, the direction bit is reversed. When the bit is set to *DOWN*, the move is to the next lowest requested position, if any.

Figure 5-29 shows the elevator algorithm using the same seven requests as Fig. 5-28, assuming the direction bit was initially *UP*. The order in which the cylinders are serviced is 12, 16, 34, 36, 9, and 1, which yields arm motions of 1, 4, 18, 2, 27, and 8, for a total of 60 cylinders. In this case the elevator algorithm is slightly better than SSF, although it is usually worse. One nice property that the elevator algorithm has is that given any collection of requests, the upper bound on the total motion is fixed: it is just twice the number of cylinders.

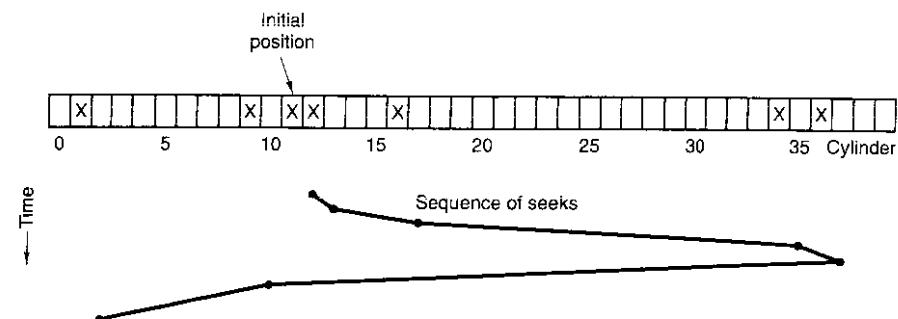


Figure 5-29. The elevator algorithm for scheduling disk requests.

A slight modification of this algorithm that has a smaller variance in response times (Teory, 1972) is to always scan in the same direction. When the highest numbered cylinder with a pending request has been serviced, the arm goes to the lowest-numbered cylinder with a pending request and then continues moving in an upward direction. In effect, the lowest-numbered cylinder is thought of as being just above the highest-numbered cylinder.

Some disk controllers provide a way for the software to inspect the current sector number under the head. With such a controller, another optimization is possible. If two or more requests for the same cylinder are pending, the driver can issue a request for the sector that will pass under the head next. Note that when multiple tracks are present in a cylinder, consecutive requests can be for different tracks with no penalty. The controller can select any of its heads almost instantaneously (head selection involves neither arm motion nor rotational delay).

If the disk has the property that seek time is much faster than the rotational delay, then a different optimization should be used. Pending requests should be sorted by sector number, and as soon as the next sector is about to pass under the head, the arm should be zipped over to the right track to read or write it.

With a modern hard disk, the seek and rotational delays so dominate performance that reading one or two sectors at a time is very inefficient. For this reason, many disk controllers always read and cache multiple sectors, even when only one is requested. Typically any request to read a sector will cause that sector and much or all the rest of the current track to be read, depending upon how much space is available in the controller's cache memory. The disk described in Fig. 5-18 has a 4-MB cache, for example. The use of the cache is determined dynamically by the controller. In its simplest mode, the cache is divided into two sections, one for reads and one for writes. If a subsequent read can be satisfied out of the controller's cache, it can return the requested data immediately.

It is worth noting that the disk controller's cache is completely independent of the operating system's cache. The controller's cache usually holds blocks that have not actually been requested, but which were convenient to read because they just happened to pass under the head as a side effect of some other read. In contrast, any cache maintained by the operating system will consist of blocks that were explicitly read and which the operating system thinks might be needed again in the near future (e.g., a disk block holding a directory block).

When several drives are present on the same controller, the operating system should maintain a pending request table for each drive separately. Whenever any drive is idle, a seek should be issued to move its arm to the cylinder where it will be needed next (assuming the controller allows overlapped seeks). When the current transfer finishes, a check can be made to see if any drives are positioned on the correct cylinder. If one or more are, the next transfer can be started on a drive that is already on the right cylinder. If none of the arms is in the right place, the driver should issue a new seek on the drive that just completed a transfer and wait until the next interrupt to see which arm gets to its destination first.

It is important to realize that all of the above disk scheduling algorithms tacitly assume that the real disk geometry is the same as the virtual geometry. If it is not, then scheduling disk requests makes no sense because the operating system cannot really tell whether cylinder 40 or cylinder 200 is closer to cylinder 39. On the other hand, if the disk controller can accept multiple outstanding requests, it can use these scheduling algorithms internally. In that case, the algorithms are still valid, but one level down, inside the controller.

5.4.4 Error Handling

Disk manufacturers are constantly pushing the limits of the technology by increasing linear bit densities. A track midway out on a 5.25-inch disk has a circumference of about 300 mm. If the track holds 300 sectors of 512 bytes, the

linear recording density may be about 5000 bits/mm taking into account the fact that some space is lost to preambles, ECCs, and intersector gaps. Recording 5000 bits/mm requires an extremely uniform substrate and a very fine oxide coating. Unfortunately, it is not possible to manufacture a disk to such specifications without defects. As soon as manufacturing technology has improved to the point where it is possible to operate flawlessly at such densities, disk designers will go to higher densities to increase the capacity. Doing so will probably reintroduce defects.

Manufacturing defects introduce bad sectors, that is, sectors that do not correctly read back the value just written to them. If the defect is very small, say, only a few bits, it is possible to use the bad sector and just let the ECC correct the errors every time. If the defect is bigger, the error cannot be masked.

There are two general approaches to bad blocks: deal with them in the controller or deal with them in the operating system. In the former approach, before the disk is shipped from the factory, it is tested and a list of bad sectors is written onto the disk. For each bad sector, one of the spares is substituted for it.

There are two ways to do this substitution. In Fig. 5-30(a), we see a single disk track with 30 data sectors and two spares. Sector 7 is defective. What the controller can do is remap one of the spares as sector 7 as shown in Fig. 5-30(b). The other way is to shift all the sectors up one, as shown in Fig. 5-30(c). In both cases the controller has to know which sector is which. It can keep track of this information through internal tables (one per track) or by rewriting the preambles to give the remapped sector numbers. If the preambles are rewritten, the method of Fig. 5-30(c) is more work (because 23 preambles must be rewritten) but ultimately gives better performance because an entire track can still be read in one rotation.

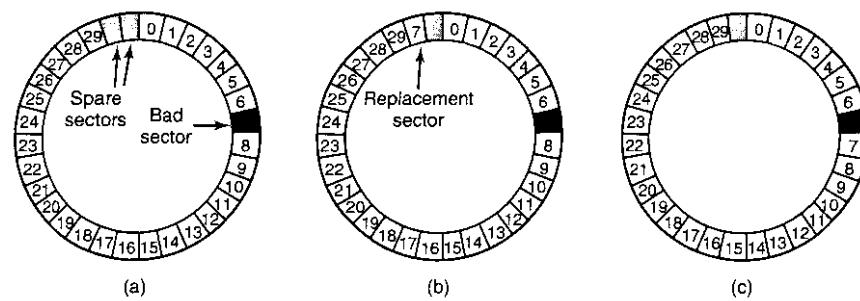


Figure 5-30. (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

Errors can also develop during normal operation after the drive has been installed. The first line of defense upon getting an error that the ECC cannot handle is to just try the read again. Some read errors are transient, that is, are caused by

specks of dust under the head and will go away on a second attempt. If the controller notices that it is getting repeated errors on a certain sector, it can switch to a spare before the sector has died completely. In this way, no data are lost and the operating system and user do not even notice the problem. Usually, the method of Fig. 5-30(b) has to be used since the other sectors might now contain data. Using the method of Fig. 5-30(c) would require not only rewriting the preambles, but copying all the data as well.

Earlier we said there were two general approaches to handling errors: handle them in the controller or in the operating system. If the controller does not have the capability to transparently remap sectors as we have discussed, the operating system must do the same thing in software. This means that it must first acquire a list of bad sectors, either by reading them from the disk, or simply testing the entire disk itself. Once it knows which sectors are bad, it can build remapping tables. If the operating system wants to use the approach of Fig. 5-30(c), it must shift the data in sectors 7 through 29 up one sector.

If the operating system is handling the remapping, it must make sure that bad sectors do not occur in any files and also do not occur in the free list or bitmap. One way to do this is to create a secret file consisting of all the bad sectors. If this file is not entered into the file system, users will not accidentally read it (or worse yet, free it).

However, there is still another problem: backups. If the disk is backed up file by file, it is important that the backup utility not try to copy the bad block file. To prevent this, the operating system has to hide the bad block file so well that even a backup utility cannot find it. If the disk is backed up sector by sector rather than file by file, it will be difficult, if not impossible, to prevent read errors during backup. The only hope is that the backup program has enough smarts to give up after 10 failed reads and continue with the next sector.

Bad sectors are not the only source of errors. Seek errors caused by mechanical problems in the arm also occur. The controller keeps track of the arm position internally. To perform a seek, it issues a series of pulses to the arm motor, one pulse per cylinder, to move the arm to the new cylinder. When the arm gets to its destination, the controller reads the actual cylinder number from the preamble of the next sector. If the arm is in the wrong place, a seek error has occurred.

Most hard disk controllers correct seek errors automatically, but most floppy controllers (including the Pentium's) just set an error bit and leave the rest to the driver. The driver handles this error by issuing a recalibrate command, to move the arm as far out as it will go and reset the controller's internal idea of the current cylinder to 0. Usually this solves the problem. If it does not, the drive must be repaired.

As we have seen, the controller is really a specialized little computer, complete with software, variables, buffers, and occasionally, bugs. Sometimes an unusual sequence of events, such as an interrupt on one drive occurring simultaneously with a recalibrate command for another drive will trigger a bug and cause

the controller to go into a loop or lose track of what it was doing. Controller designers usually plan for the worst and provide a pin on the chip which, when asserted, forces the controller to forget whatever it was doing and reset itself. If all else fails, the disk driver can set a bit to invoke this signal and reset the controller. If that does not help, all the driver can do is print a message and give up.

Recalibrating a disk makes a funny noise but otherwise normally is not disturbing. However, there is one situation where recalibration is a serious problem: systems with real-time constraints. When a video is being played off a hard disk, or files from a hard disk are being burned onto a CD-ROM, it is essential that the bits arrive from the hard disk at a uniform rate. Under these circumstances, recalibrations insert gaps into the bit stream and are therefore unacceptable. Special drives, called **AV disks (Audio Visual disks)**, which never recalibrate are available for such applications.

5.4.5 Stable Storage

As we have seen, disks sometimes make errors. Good sectors can suddenly become bad sectors. Whole drives can die unexpectedly. RAIDs protect against a few sectors going bad or even a drive falling out. However, they do not protect against write errors laying down bad data in the first place. They also do not protect against crashes during writes corrupting the original data without replacing them by newer data.

For some applications, it is essential that data never be lost or corrupted, even in the face of disk and CPU errors. Ideally, a disk should simply work all the time with no errors. Unfortunately, that is not achievable. What is achievable is a disk subsystem that has the following property: when a write is issued to it, the disk either correctly writes the data or it does nothing, leaving the existing data intact. Such a system is called **stable storage** and is implemented in software (Lampson and Sturgis, 1979). The goal is to keep the disk consistent at all costs. Below we will describe a slight variant of the original idea.

Before describing the algorithm, it is important to have a clear model of the possible errors. The model assumes that when a disk writes a block (one or more sectors), either the write is correct or it is incorrect and this error can be detected on a subsequent read by examining the values of the ECC fields. In principle, guaranteed error detection is never possible because with a, say, 16-byte ECC field guarding a 512-byte sector, there are 2^{4096} data values and only 2^{144} ECC values. Thus if a block is garbled during writing but the ECC is not, there are billions upon billions of incorrect combinations that yield the same ECC. If any of them occur, the error will not be detected. On the whole, the probability of random data having the proper 16-byte ECC is about 2^{-144} , which is small enough that we will call it zero, even though it is really not.

The model also assumes that a correctly written sector can spontaneously go bad and become unreadable. However, the assumption is that such events are so

rare that having the same sector go bad on a second (independent) drive during a reasonable time interval (e.g., 1 day) is small enough to ignore.

The model also assumes the CPU can fail, in which case it just stops. Any disk write in progress at the moment of failure also stops, leading to incorrect data in one sector and an incorrect ECC that can later be detected. Under all these conditions, stable storage can be made 100% reliable in the sense of writes either working correctly or leaving the old data in place. Of course, it does not protect against physical disasters, such as an earthquake happening and the computer falling 100 meters into a fissure and landing in a pool of boiling magma. It is tough to recover from this condition in software.

Stable storage uses a pair of identical disks with the corresponding blocks working together to form one error-free block. In the absence of errors, the corresponding blocks on both drives are the same. Either one can be read to get the same result. To achieve this goal, the following three operations are defined:

- 1. Stable writes.** A stable write consists of first writing the block on drive 1, then reading it back to verify that it was written correctly. If it was not written correctly, the write and reread are done again up to n times until they work. After n consecutive failures, the block is remapped onto a spare and the operation repeated until it succeeds, no matter how many spares have to be tried. After the write to drive 1 has succeeded, the corresponding block on drive 2 is written and reread, repeatedly if need be, until it, too, finally succeeds. In the absence of CPU crashes, when a stable write completes, the block has correctly been written onto both drives and verified on both of them.
- 2. Stable reads.** A stable read first reads the block from drive 1. If this yields an incorrect ECC, the read is tried again, up to n times. If all of these give bad ECCs, the corresponding block is read from drive 2. Given the fact that a successful stable write leaves two good copies of the block behind, and our assumption that the probability of the same block spontaneously going bad on both drives in a reasonable time interval is negligible, a stable read always succeeds.
- 3. Crash recovery.** After a crash, a recovery program scans both disks comparing corresponding blocks. If a pair of blocks are both good and the same, nothing is done. If one of them has an ECC error, the bad block is overwritten with the corresponding good block. If a pair of blocks are both good but different, the block from drive 1 is written onto drive 2.

In the absence of CPU crashes, this scheme always works because stable writes always write two valid copies of every block and spontaneous errors are assumed never to occur on both corresponding blocks at the same time. What about

in the presence of CPU crashes during stable writes? It depends on precisely when the crash occurs. There are five possibilities, as depicted in Fig. 5-31.

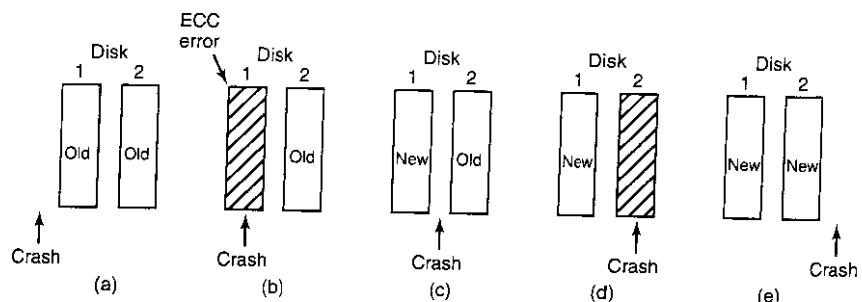


Figure 5-31. Analysis of the influence of crashes on stable writes.

In Fig. 5-31(a), the CPU crash happens before either copy of the block is written. During recovery, neither will be changed and the old value will continue to exist, which is allowed.

In Fig. 5-31(b), the CPU crashes during the write to drive 1, destroying the contents of the block. However the recovery program detects this error and restores the block on drive 1 from drive 2. Thus the effect of the crash is wiped out and the old state is fully restored.

In Fig. 5-31(c), the CPU crash happens after drive 1 is written but before drive 2 is written. The point of no return has been passed here: the recovery program copies the block from drive 1 to drive 2. The write succeeds.

Fig. 5-31(d) is like Fig. 5-31(b): during recovery, the good block overwrites the bad block. Again, the final value of both blocks is the new one.

Finally, in Fig. 5-31(e) the recovery program sees that both blocks are the same, so neither is changed and the write succeeds here too.

Various optimizations and improvements are possible to this scheme. For starters, comparing all the blocks pairwise after a crash is doable, but expensive. A huge improvement is to keep track of which block was being written during a stable write so that only one block has to be checked during recovery. Some computers have a small amount of **nonvolatile RAM**, which is a special CMOS memory powered by a lithium battery. Such batteries last for years, possibly even the whole life of the computer. Unlike main memory, which is lost after a crash, non-volatile RAM is not lost after a crash. The time of day is normally kept here (and incremented by a special circuit), which is why computers still know what time it is even after having been unplugged.

Suppose that a few bytes of nonvolatile RAM are available for operating system purposes. The stable write can put the number of the block it is about to update in nonvolatile RAM before starting the write. After successfully completing

the stable write, the block number in nonvolatile RAM is overwritten with an invalid block number, for example, -1. Under these conditions, after a crash the recovery program can check the nonvolatile RAM to see if a stable write happened to be in progress during the crash, and if so, which block was being written when the crash happened. The two copies of the block can then be checked for correctness and consistency.

If nonvolatile RAM is not available, it can be simulated as follows. At the start of a stable write, a fixed disk block on drive 1 is overwritten with the number of the block to be stably written. This block is then read back to verify it. After getting it correct, the corresponding block on drive 2 is written and verified. When the stable write completes correctly, both blocks are overwritten with an invalid block number and verified. Again here, after a crash it is easy to determine whether or not a stable write was in progress during the crash. Of course, this technique requires eight extra disk operations to write a stable block, so it should be used exceedingly sparingly.

One last point is worth making. We assumed that only one spontaneous decay of a good block to a bad block happens per block pair per day. If enough days go by, the other one might go bad too. Therefore, once a day a complete scan of both disks must be done repairing any damage. That way, every morning both disks are always identical. Even if both blocks in a pair go bad within a period of a few days, all errors are repaired correctly.

5.5 CLOCKS

Clocks (also called **timers**) are essential to the operation of any multiprogrammed system for a variety of reasons. They maintain the time of day and prevent one process from monopolizing the CPU, among other things. The clock software can take the form of a device driver, even though a clock is neither a block device, like a disk, nor a character device, like a mouse. Our examination of clocks will follow the same pattern as in the previous section: first a look at clock hardware and then a look at the clock software.

5.5.1 Clock Hardware

Two types of clocks are commonly used in computers, and both are quite different from the clocks and watches used by people. The simpler clocks are tied to the 110- or 220-volt power line and cause an interrupt on every voltage cycle, at 50 or 60 Hz. These clocks used to dominate, but are rare nowadays.

The other kind of clock is built out of three components: a crystal oscillator, a counter, and a holding register, as shown in Fig. 5-32. When a piece of quartz is properly cut and mounted under tension, it can be made to generate a periodic signal of very great accuracy, typically in the range of several hundred

megahertz, depending on the crystal chosen. Using electronics, this base signal can be multiplied by a small integer to get frequencies up to 1000 MHz or even more. At least one such circuit is usually found in any computer, providing a synchronizing signal to the computer's various circuits. This signal is fed into the counter to make it count down to zero. When the counter gets to zero, it causes a CPU interrupt.

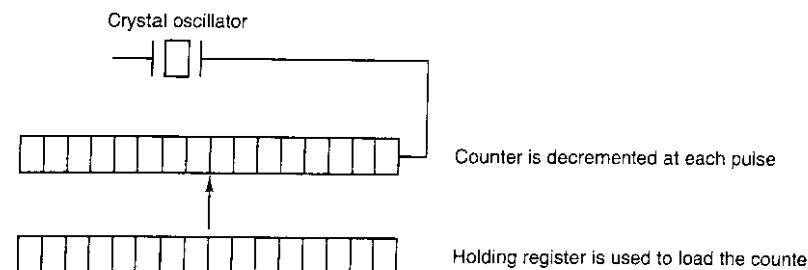


Figure 5-32. A programmable clock.

Programmable clocks typically have several modes of operation. In **one-shot mode**, when the clock is started, it copies the value of the holding register into the counter and then decrements the counter at each pulse from the crystal. When the counter gets to zero, it causes an interrupt and stops until it is explicitly started again by the software. In **square-wave mode**, after getting to zero and causing the interrupt, the holding register is automatically copied into the counter, and the whole process is repeated again indefinitely. These periodic interrupts are called **clock ticks**.

The advantage of the programmable clock is that its interrupt frequency can be controlled by software. If a 500-MHz crystal is used, then the counter is pulsed every 2 nsec. With (unsigned) 32-bit registers, interrupts can be programmed to occur at intervals from 2 nsec to 8.6 sec. Programmable clock chips usually contain two or three independently programmable clocks and have many other options as well (e.g., counting up instead of down, interrupts disabled, and more).

To prevent the current time from being lost when the computer's power is turned off, most computers have a battery-powered backup clock, implemented with the kind of low-power circuitry used in digital watches. The battery clock can be read at startup. If the backup clock is not present, the software may ask the user for the current date and time. There is also a standard way for a networked system to get the current time from a remote host. In any case the time is then translated into the number of clock ticks since 12 A.M. **UTC (Universal Coordinated Time)** (formerly known as Greenwich Mean Time) on Jan. 1, 1970, as UNIX does, or since some other benchmark moment. The origin of time for Windows is Jan. 1, 1980. At every clock tick, the real time is incremented by one