

5


FILE SYSTEMS

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

A second problem with keeping information within a process' address space is that when the process terminates, the information is lost. For many applications, (e.g., for databases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process.

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

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

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

The usual solution to all these problems is to store information on disks and other external media in units called **files**. Processes can then read them and write new ones if need be. Information stored in files must be **persistent**, that is, not be affected by process creation and termination. A file should only disappear when its owner explicitly removes it.

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

From the users' standpoint, the most important aspect of a file system is how it appears to them, that is, what constitutes a file, how files are named and protected, what operations are allowed on files, and so on. The details of whether linked lists or bitmaps are used to keep track of free storage and how many sectors there are in a logical block are of less interest, although they are of great importance to the designers of the file system. For this reason, we have structured the chapter as several sections. The first two are concerned with the user interface to files and directories, respectively. Then comes a discussion of alternative ways a file system can be implemented. Following a discussion of security and protection mechanisms, we conclude with a description of the MINIX 3 file system.

5.1 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.

5.1.1 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 (including all its variants) 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.

Windows falls in between these extremes. The Windows 95 and Windows 98 file systems are both based upon the MS-DOS file system, and thus inherit many of its properties, such as how file names are constructed. With each new version improvements were added but the features we will discuss are mostly common to MS-DOS and “classic” Windows versions. In addition, Windows NT, Windows 2000, and Windows XP support the MS-DOS file system. **However, the latter systems also have a native file system (NTFS) that has different properties (such as file names in Unicode).** This file system also has seen changes in successive versions. In this chapter, we will refer to the older systems as the Windows 98 file system. If a feature does not apply to the MS-DOS or Windows 95 versions we will say so. Likewise, we will refer to the newer system as either NTFS or the Windows XP file system, and we will point it out if an aspect under discussion does not also apply to the file systems of Windows NT or Windows 2000. When we say just Windows, we mean all Windows file systems since Windows 95.

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 this example that it is a C programming language source file. In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in *prog.c.bz2*, where *.bz2* is commonly used to indicate that the file (*prog.c*) has been compressed using the bzip2 compression algorithm. Some of the more common file extensions and their meanings are shown in Fig. 5-1.

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 files to compile and link together, some of them C files (e.g., *foo.c*), some of them assembly language files (e.g., *bar.s*), and some of them object files (e.g., *other.o*). The extension then becomes essential for the compiler to tell which are C files, which are assembly files, and which are object files.

In contrast, Windows is very much aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system and specify which program “owns” which one. When a user double clicks on a file name, the program assigned to its file extension is launched and given

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Graphical Interchange Format image
file.html	World Wide Web HyperText Markup Language document
file.iso	ISO image of a CD-ROM (for burning to CD)
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 5-1. Some typical file extensions.

the name of the file as parameter. For example, double clicking on *file.doc* starts Microsoft Word with *file.doc* as the initial file to edit.

Some might think it odd that Microsoft chose to make common extensions invisible by default since they are so important. Fortunately most of the “wrong by default” settings of Windows can be changed by a sophisticated user who knows where to look.

5.1.2 File Structure

Files can be structured in any one of several ways. **Three common possibilities are depicted in Fig. 5-2.** The file in Fig. 5-2(a) is just 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 98 use this approach.

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

The first step up in structure is shown in Fig. 5-2(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

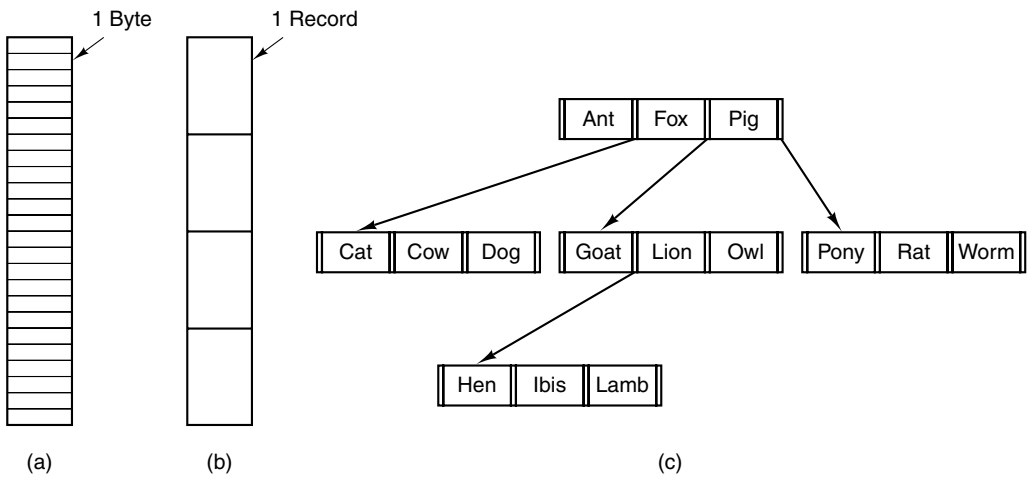


Figure 5-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

historical note, when the 80-column punched card was king many (mainframe) operating systems based their file systems on files consisting of 80-character records, in effect, card images. These systems also supported files of 132-character records, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of 80 characters and wrote it in units of 132 characters, although the final 52 could be spaces, of course. No current general-purpose system works this way.

The third kind of file structure is shown in Fig. 5-2(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. 5-2(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 98 but is widely used on the large mainframe computers still used in some commercial data processing.

5.1.3 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. Windows XP also uses **metadata** files, which we will mention later.

Regular files are the ones that contain user information. All the files of Fig. 5-2 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., Windows) use both. Lines need not all be of the same length.

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

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

For example, in Fig. 5-3(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.

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 operating systems 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.

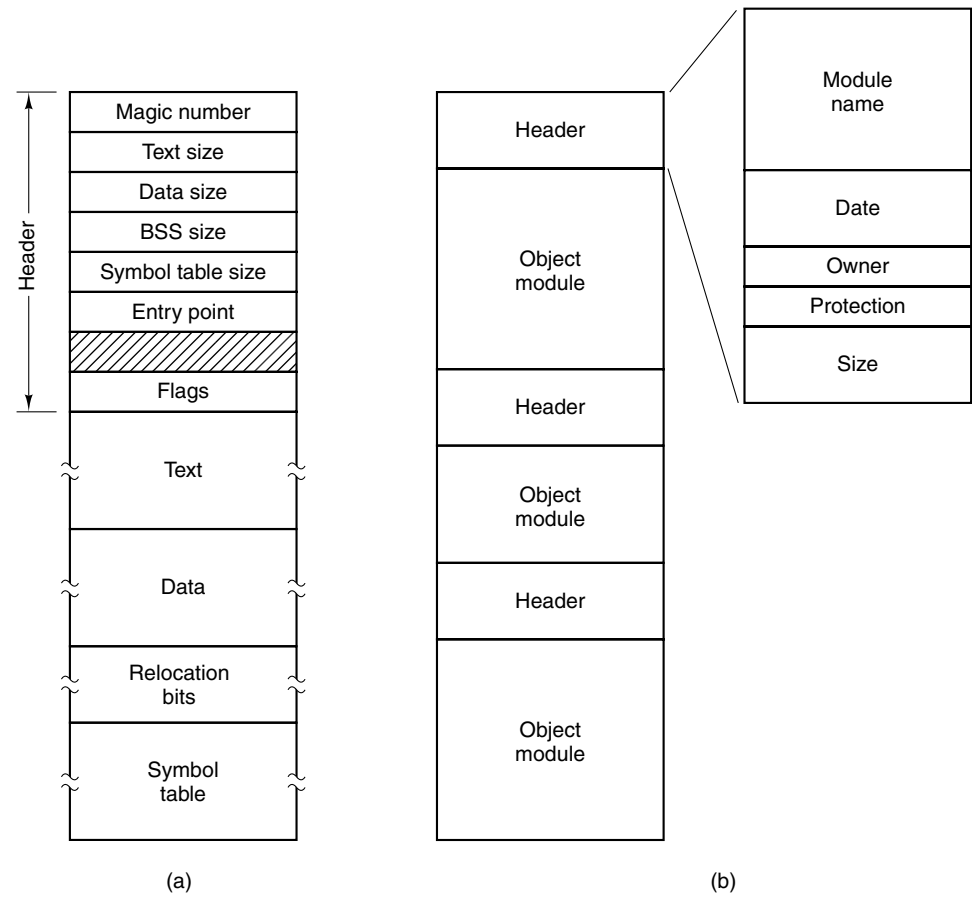


Figure 5-3. (a) An executable file. (b) An archive.

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.

5.1.4 File Access

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

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

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

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

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

5.1.5 File Attributes

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

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

The flags are bits or short fields that control or enable some specific property. Hidden files, for example, do not appear in listings of the files. The archive flag is a bit that keeps track of whether the file has been backed up. The backup program clears it, and the operating system sets it whenever a file is changed. In this

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

Figure 5-4. Some possible file attributes.

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. Modern operating systems are clever enough to do without this feature.

5.1.6 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. A system call for this purpose is always provided.
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 some internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.
5. **Read.** Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.
6. **Write.** Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
7. **Append.** This call is a restricted form of write. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.
8. **Seek.** For random access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.
9. **Get attributes.** Processes often need to read file attributes to do their work. For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When *make* is called, it examines the modification times of all

the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. **Set attributes.** Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.
11. **Rename.** It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.
12. **Lock.** Locking a file or a part of a file prevents multiple simultaneous access by different process. For an airline reservation system, for instance, locking the database while making a reservation prevents reservation of a seat for two different travelers.

5.2 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.

5.2.1 Simple Directories

A directory typically contains a number of entries, one per file. One possibility is shown in Fig. 5-5(a), in which each entry contains the file name, the file attributes, and the disk addresses where the data are stored. Another possibility is shown in Fig. 5-5(b). Here a directory entry holds the file name and a pointer to another data structure where the attributes and disk addresses are found. Both of these systems are commonly used.

When a file is opened, the operating system searches its directory until it finds the name of the file to be opened. It then extracts the attributes and disk addresses, either directly from the directory entry or from the data structure pointed to, and puts them in a table in main memory. All subsequent references to the file use the information in main memory.

The number of directories varies from system to system. The simplest form of directory system is a **single directory** containing all files for all users, as illustrated in Fig. 5-6(a). On early personal computers, this single-directory system was common, in part because there was only one user.

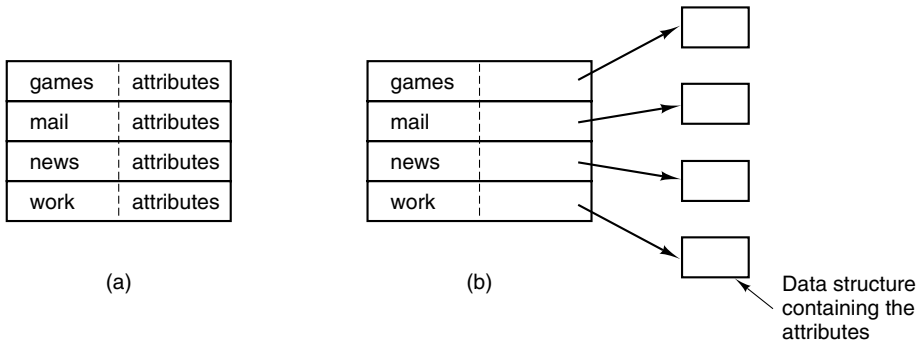


Figure 5-5. (a) Attributes in the directory entry. (b) Attributes elsewhere.

The problem with having only one directory in a system with multiple users is that different users may accidentally use the same names for their files. For example, if user *A* creates a file called *mailbox*, and then later user *B* also creates a file called *mailbox*, *B*'s file will overwrite *A*'s file. Consequently, this scheme is not used on multiuser systems any more, but could be used on a small embedded system, for example, a handheld personal digital assistant or a cellular telephone.

To avoid conflicts caused by different users choosing the same file name for their own files, the next step up is giving each user a private directory. In that way, names chosen by one user do not interfere with names chosen by a different user and there is no problem caused by the same name occurring in two or more directories. This design leads to the system of Fig. 5-6(b). This design could be used, for example, on a multiuser computer or on a simple network of personal computers that shared a common file server over a local area network.

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

When this system is implemented in its most basic form, users can only access files in their own directories.

5.2.2 Hierarchical Directory Systems

The two-level hierarchy eliminates file name conflicts between users. But another problem is that users with many files may want to group them in smaller subgroups, for instance a professor might want to separate handouts for a class from drafts of chapters of a new textbook. What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Fig. 5-6(c). Here, the directories *A*, *B*, and *C* contained in

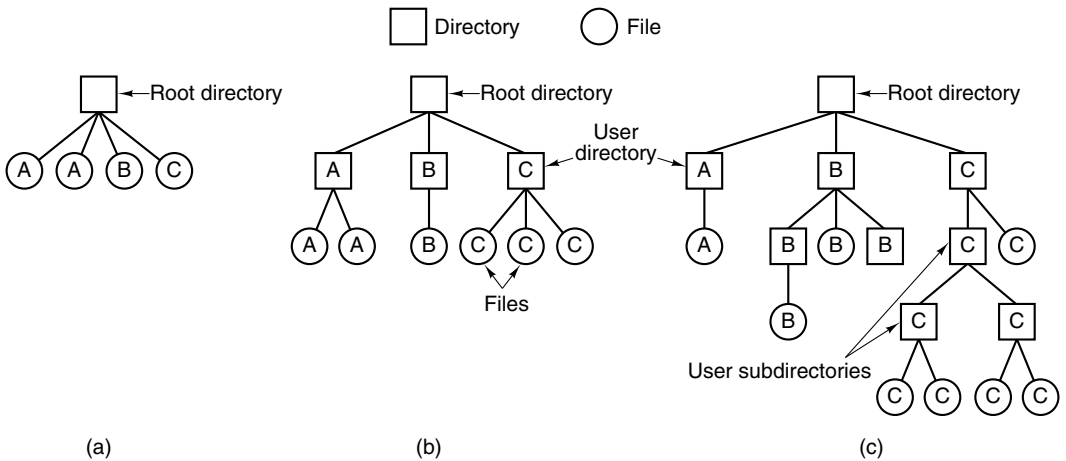


Figure 5-6. Three file system designs. (a) Single directory shared by all users. (b) One directory per user. (c) Arbitrary tree per user. The letters indicate the directory or file's owner.

the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

The ability to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work. For this reason nearly all modern PC and server file systems are organized this way.

However, as we have pointed out before, history often repeats itself with new technologies. Digital cameras have to record their images somewhere, usually on a flash memory card. The very first digital cameras had a single directory and named the files *DSC0001.JPG*, *DSC0002.JPG*, etc. However, it did not take very long for camera manufacturers to build file systems with multiple directories, as in Fig. 5-6(b). What difference does it make that none of the camera owners understand how to use multiple directories, and probably could not conceive of any use for this feature even if they did understand it? It is only (embedded) software, after all, and thus costs the camera manufacturer next to nothing to provide. Can digital cameras with full-blown hierarchical file systems, multiple login names, and 255-character file names be far behind?

5.2.3 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 **. Thus the same path name would be written as follows in these two systems:

Windows	<i>\usr\ast\mailbox</i>
UNIX	<i>/usr/ast/mailbox</i>

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

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

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

and the command

```
cp mailbox mailbox.bak
```

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

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

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

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

Most operating systems that support a hierarchical directory system have two special entries in every directory, “.” and “..”, generally pronounced “dot” and “dotdot.” **Dot refers to the current directory; dotdot refers to its parent.** To see how these are used, consider the UNIX file tree of Fig. 5-7. A certain process has */usr/ast/* as its working directory. It can use *..* to go up the tree. For example, it can copy the file */usr/lib/dictionary* to its own directory using the command

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

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

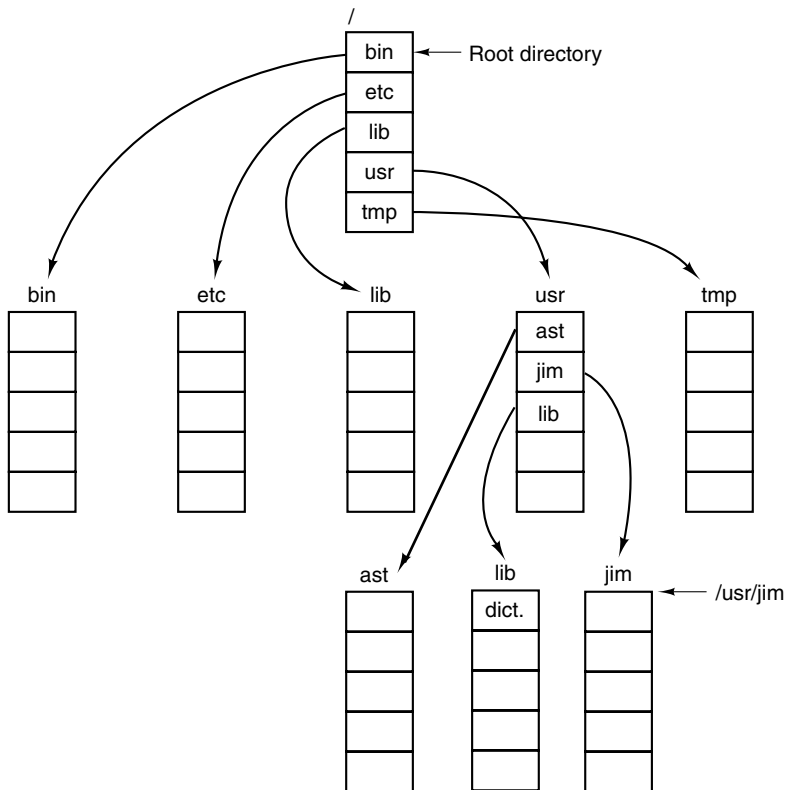


Figure 5-7. A UNIX directory tree.

The second argument (dot) names the current directory. When the *cp* command gets a directory name (including dot) as its last argument, it copies all the files there. Of course, a more normal way to do the copy would be to type

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

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

Nevertheless, typing

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

also works fine, as does

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

All of these do exactly the same thing.

5.2.4 Directory Operations

The 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.

5.3 FILE SYSTEM IMPLEMENTATION

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

5.3.1 File System Layout

File systems usually are stored on disks. We looked at basic disk layout in Chap. 2, in the section on bootstrapping MINIX 3. To review this material briefly, most disks can be divided up into 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 may be marked as active. When the computer is booted, the BIOS reads in and executes the code in 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 some time in the future, so reserving a boot block is a good idea anyway.

The above description must be true, regardless of the operating system in use, for any hardware platform on which the BIOS is to be able to start more than one operating system. The terminology may differ with different operating systems. For instance the master boot record may sometimes be called the **IPL (Initial Program Loader)**, **Volume Boot Code**, or simply **masterboot**. Some operating

systems do not require a partition to be marked active to be booted, and provide a menu for the user to choose a partition to boot, perhaps with a timeout after which a default choice is automatically taken. Once the BIOS has loaded an MBR or boot sector the actions may vary. For instance, more than one block of a partition may be used to contain the program that loads the operating system. The BIOS can be counted on only to load the first block, but that block may then load additional blocks if the implementer of the operating system writes the boot block that way. An implementer can also supply a custom MBR, but it must work with a standard partition table if multiple operating systems are to be supported.

On PC-compatible systems there can be no more than four **primary partitions** because there is only room for a four-element array of partition descriptors between the master boot record and the end of the first 512-byte sector. Some operating systems allow one entry in the partition table to be an **extended partition** which points to a linked list of **logical partitions**. This makes it possible to have any number of additional partitions. The BIOS cannot start an operating system from a logical partition, so initial startup from a primary partition is required to load code that can manage logical partitions.

An alternative to extended partitions is used by MINIX 3, which allows a partition to contain a **subpartition table**. An advantage of this is that the same code that manages a primary partition table can manage a subpartition table, which has the same structure. Potential uses for subpartitions are to have different ones for the root device, swapping, the system binaries, and the users' files. In this way, problems in one subpartition cannot propagate to another one, and a new version of the operating system can be easily installed by replacing the contents of some of the subpartitions but not all.

Not all disks are partitioned. Floppy disks usually start with a boot block in the first sector. The BIOS reads the first sector of a disk and looks for a magic number which identifies it as valid executable code, to prevent an attempt to execute the first sector of an unformatted or corrupted disk. A master boot record and a boot block use the same magic number, so the executable code may be either one. Also, what we say here is not limited to electromechanical disk devices. A device such as a camera or personal digital assistant that uses nonvolatile (e.g., flash) memory typically has part of the memory organized to simulate a disk.

Other than starting with a boot block, the layout of a disk partition varies considerably from file system to file system. A UNIX-like file system will contain some of the items shown in Fig. 5-8. 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.

Next might come information about free blocks in the file system. This might be followed by the i-nodes, an array of data structures, one per file, telling all about the file and where its blocks are located. After that might come the root directory, which contains the top of the file system tree. Finally, the remainder of the disk typically contains all the other directories and files.

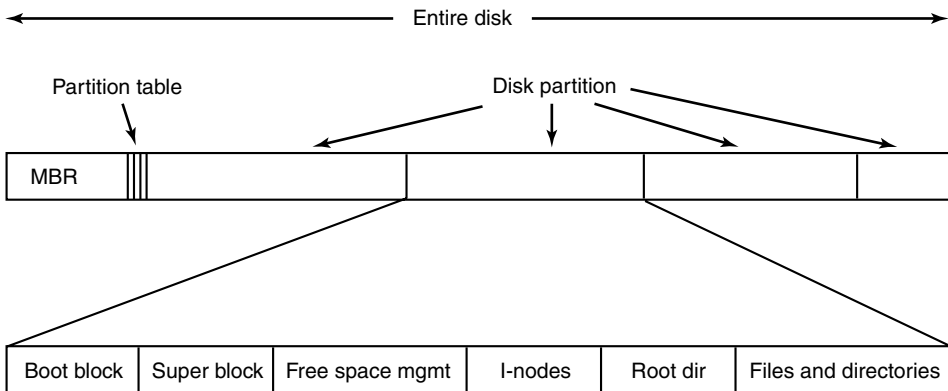


Figure 5-8. A possible file system layout.

5.3.2 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. 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 **major drawback**: in time, the **disk becomes fragmented, consisting of files and holes**. 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.

As we mentioned in Chap. 1, history may repeat 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. For such media, contiguous allocation is feasible and, in fact, widely used. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system. 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. 5-9. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

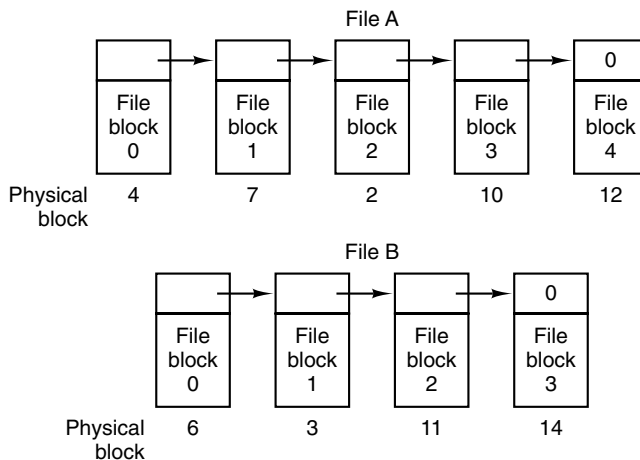


Figure 5-9. Storing a file as a linked list of disk blocks.

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 of each file). 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 5-10 shows what the table looks like for the example of Fig. 5-9. 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. 5-10, 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)**.

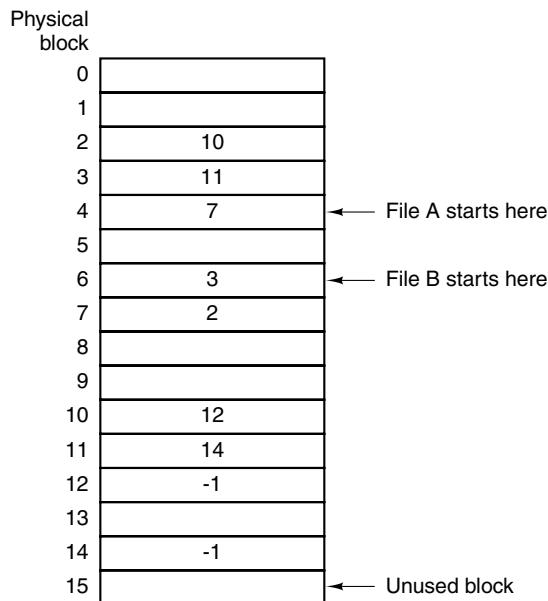


Figure 5-10. Linked list allocation using a file allocation table in main memory.

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. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time. Conceivably the table could be put in pageable memory, but it would still occupy a great deal of virtual memory and disk space as well as generating paging traffic. MS-DOS and Windows 98 use only FAT file systems and later versions of Windows also support it.

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. 5-11. Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only kn bytes. Only this much space need be reserved in advance.

This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the linked list of all disk blocks is proportional in size to the disk itself. If the disk has n blocks, the table needs n entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 1 GB or 10 GB or 100 GB.

One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of an **indirect block** containing more disk block addresses. This idea can be extended to use **double indirect blocks** and **triple indirect blocks**, as shown in Fig. 5-11.

5.3.3 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. Finding a directory entry means, of course, that the root directory must be

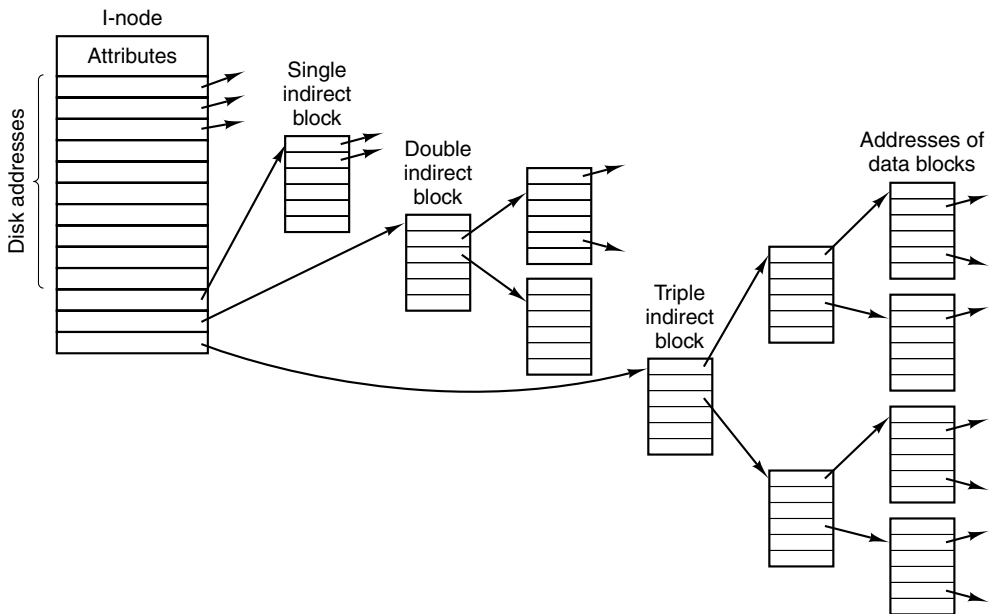


Figure 5-11. An i-node with three levels of indirect blocks.

located first. The root directory may be in a fixed location relative to the start of a partition. Alternatively, its position may be determined from other information, for instance, in a classic UNIX file system the superblock contains information about the size of the file system data structures that precede the data area. From the superblock the location of the i-nodes can be found. The first i-node will point to the root directory, which is created when a UNIX file system is made. In Windows XP, information in the boot sector (which is really much bigger than one sector) locates the MFT (Master File Table), which is used to locate other parts of the file system.

Once the root directory is located a search through the directory tree finds the desired directory entry. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (contiguous allocation), the number of the first block (both linked list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

A closely related issue is where the attributes should be stored. Every file system maintains file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. In its simplest form, 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, as we saw in Fig. 5-5(a).

For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries, as in Fig. 5-5(b). In this case, the directory entry can be shorter: just a file name and an i-node number.

Shared Files

In Chap. 1 we briefly mentioned **links** between files, which make it easy for several users working together on a project to share files. Figure 5-12 shows the file system of Fig. 5-6(c) again, only with one of C's files now present in one of B's directories as well.

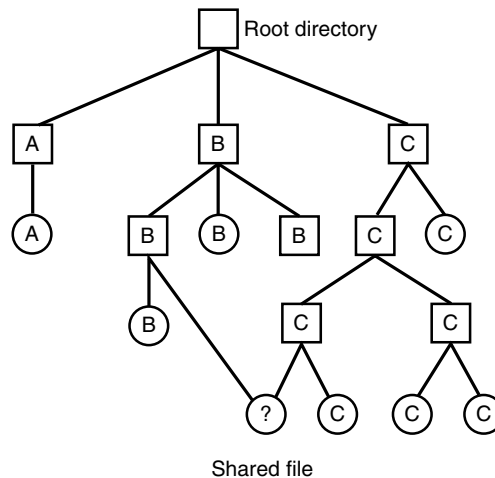


Figure 5-12. File system containing a shared file.

In UNIX the use of i-nodes for storing file attributes makes sharing easy; any number of directory entries can point to a single i-node. The i-node contains a field which is incremented when a new link is added, and which is decremented when a link is deleted. Only when the link count reaches zero are the actual data and the i-node itself deleted.

This kind of link is sometimes called a **hard link**. Sharing files using hard links is not always possible. A major limitation is that directories and i-nodes are data structures of a single file system (partition), so a directory in one file system cannot point to an i-node on another file system. Also, a file can have only one owner and one set of permissions. If the owner of a shared file deletes his own directory entry for that file, another user could be stuck with a file in his directory that he cannot delete if the permissions do not allow it.

An alternative way to share files is to create a new kind of **file whose data is the path to another file**. This kind of link will work across mounted file systems. In fact, if a means is provided for path names to include network addresses, such a link can refer to a file on a different computer. This second kind of link is called a **symbolic link in UNIX-like systems**, a **shortcut** in Windows, and an **alias** in Apple's Mac OS. Symbolic links can be used on systems where attributes are stored within directory entries. A little thought should convince you that multiple directory entries containing file attributes would be difficult to synchronize. Any change to a file would have to affect every directory entry for that file. But the extra directory entries for symbolic links do not contain the attributes of the file to which they point. A disadvantage of symbolic links is that when a file is deleted, or even just renamed, a link becomes an orphan.

Directories in Windows 98

The file system of the original release of Windows 95 was identical to the MS-DOS file system, but a second release added support for longer file names and bigger files. We will refer to this as the Windows 98 file system, even though it is found on some Windows 95 systems. **Two types of directory entry exist in Windows 98**. We will call the first one, shown in Fig. 5-13, a **base entry**.

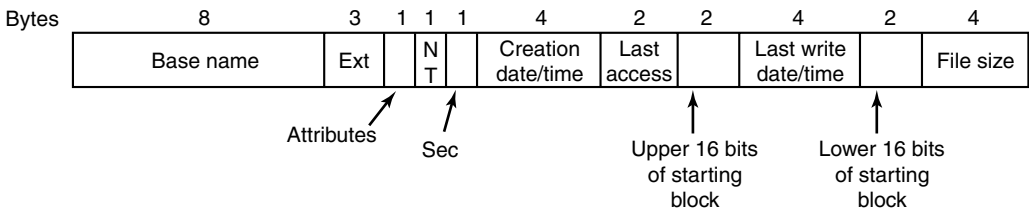


Figure 5-13. A Windows 98 base directory entry.

The base directory entry has all the information that was in the directory entries of older Windows versions, and more. The 10 bytes starting with the *NT* field are additions to the older Windows 95 structure, which fortunately (or more likely deliberately, with later improvement in mind) were not previously used. The most important upgrade is the field that increases the number of bits available for pointing to the starting block from 16 to 32. This increases the maximum potential size of the file system from 2^{16} blocks to 2^{32} blocks.

This structure provides only for the old-style 8 + 3 character filenames inherited from MS-DOS (and CP/M). How about long file names? The answer to the problem of providing long file names while retaining compatibility with the older systems was to use additional directory entries. Fig. 5-14 shows an **alternative form of directory entry** that can contain up to 13 characters of a long file name. For files with long names a shortened form of the name is generated automatically

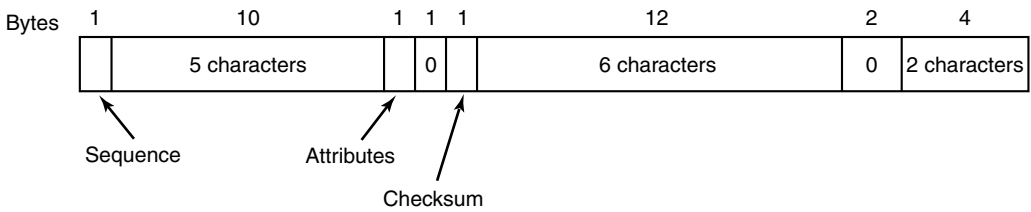


Figure 5-14. An entry for (part of) a long file name in Windows 98.

and placed in the *Base name* and *Ext* fields of an Fig. 5-13-style base directory entry. As many entries like that of Fig. 5-14 as are needed to contain the long file name are placed before the base entry, in reverse order. The *Attributes* field of each long name entry contains the value 0x0F, which is an impossible value for older (MS-DOS and Windows 95) files systems, so these entries will be ignored if the directory is read by an older system (on a floppy disk, for instance). A bit in the *Sequence* field tells the system which is the last entry.

If this seems rather complex, well, it is. Providing backward compatibility so an earlier simpler system can continue to function while providing additional features for a newer system is likely to be messy. A purist might decide not to go to so much trouble. However, a purist would probably not become rich selling new versions of operating systems.

Directories in UNIX

The traditional UNIX directory structure is extremely simple, as shown in Fig. 5-15. Each entry contains just a file name and its i-node number. All the information about the type, size, times, ownership, and disk blocks is contained in the i-node. Some UNIX systems have a different layout, but in all cases, a directory entry ultimately contains only an ASCII string and an i-node number.

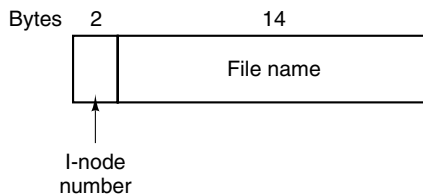


Figure 5-15. A Version 7 UNIX directory entry.

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 system locates the root directory. The

i-nodes form a simple array which is located using information in the superblock. The first entry in this array is the i-node of the root directory.

The file system 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 relative to the first one. 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. 5-16.

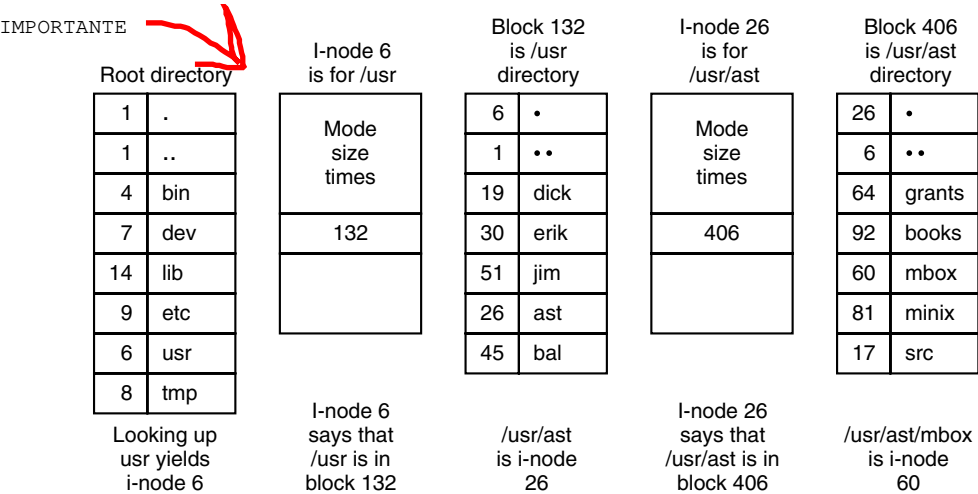


Figure 5-16. The steps in looking up */usr/ast/mbox*.

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.

Directories in NTFS

Microsoft's NTFS (New Technology File System) is the default file system. We do not have space for a detailed description of NTFS, but will just briefly look at some of the problems NTFS deals with and the solutions used.

One problem is long file and path names. NTFS allows long file names (up to 255 characters) and path names (up to 32,767 characters). But since older versions of Windows cannot read NTFS file systems, a complicated backward-compatible directory structure is not needed, and filename fields are variable length. Provision is made to have a second 8 + 3 character name so an older system can access NTFS files over a network.

NTFS provides for multiple character sets by using Unicode for filenames. Unicode uses 16 bits for each character, enough to represent multiple languages with very large symbol sets (e.g., Japanese). But using multiple languages raises problems in addition to representation of different character sets. Even among Latin-derived languages there are subtleties. For instance, in Spanish some combinations of two characters count as single characters when sorting. Words beginning with “ch” or “ll” should appear in sorted lists after words that begin with “cz” or “lz”, respectively. The problem of case mapping is more complex. If the default is to make filenames case sensitive, there may still be a need to do case-insensitive searches. For Latin-based languages it is obvious how to do that, at least to native users of these languages. In general, if only one language is in use, users will probably know the rules. However, Unicode allows a mixture of languages: Greek, Russian, and Japanese filenames could all appear in a single directory at an international organization. **The NTFS solution is an attribute for each file that defines the case conventions for the language of the filename.**

More attributes is the NTFS solution to many problems. In UNIX, a file is a sequence of bytes. In NTFS a file is a collection of attributes, and each attribute is a stream of bytes. The basic NTFS data structure is the **MFT (Master File Table)** that provides for 16 attributes, each of which can have a length of up to 1 KB within the MFT. If that is not enough, an attribute within the MFT can be a header that points to an additional file with an extension of the attribute values. This is known as a **nonresident attribute**. The MFT itself is a file, and it has an entry for every file and directory in the file system. Since it can grow very large, when an NTFS file system is created about 12.5% of the space on the partition is reserved for growth of the MFT. Thus it can grow without becoming fragmented, at least until the initial reserved space is used, after which another large chunk of space will be reserved. So if the MFT becomes fragmented it will consist of a small number of very large fragments.

What about data in NTFS? Data is just another attribute. In fact an NTFS file may have more than one data stream. This feature was originally provided to allow Windows servers to serve files to Apple Macintosh clients. In the original Macintosh operating system (through Mac OS 9) all files had two data streams, called the resource fork and the data fork. Multiple data streams have other uses, for instance a large graphic image may have a smaller thumbnail image associated with it. A stream can contain up to 2^{64} bytes. **At the other extreme, NTFS can handle small files by putting a few hundred bytes in the attribute header. This is called an immediate file** (Mullender and Tanenbaum, 1984).

We have only touched upon a few ways that NTFS deals with issues not addressed by older and simpler file systems. NTFS also provides features such as a sophisticated protection system, encryption, and data compression. Describing all these features and their implementation would require much more space than we can spare here. For a more thorough look at NTFS see Tanenbaum (2001) or look on the World Wide Web for more information.

5.3.4 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 blocks 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. However, having a large allocation unit, such as a cylinder, means that every file, even a 1-byte file, ties up an entire cylinder.

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

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

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

The solid curve of Fig. 5-17 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. An early study showed that the mean file size in UNIX environments is about 1 KB (Mullender and Tanenbaum, 1984). A measurement made in

2005 at the department of one of the authors (AST), which has 1000 users and over 1 million UNIX disk files, gives a median size of 2475 bytes, meaning that half the files are smaller than 2475 bytes and half are larger. As an aside, the median is a better metric than the mean because a very small number of files can influence the mean enormously, but not the median. A few 100-MB hardware manuals or a promotional videos or to can greatly skew the mean but have little effect on the median.

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

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

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

For simplicity, let us assume all files are 2 KB, which leads to the dashed curve in Fig. 5-17 for the disk space efficiency.

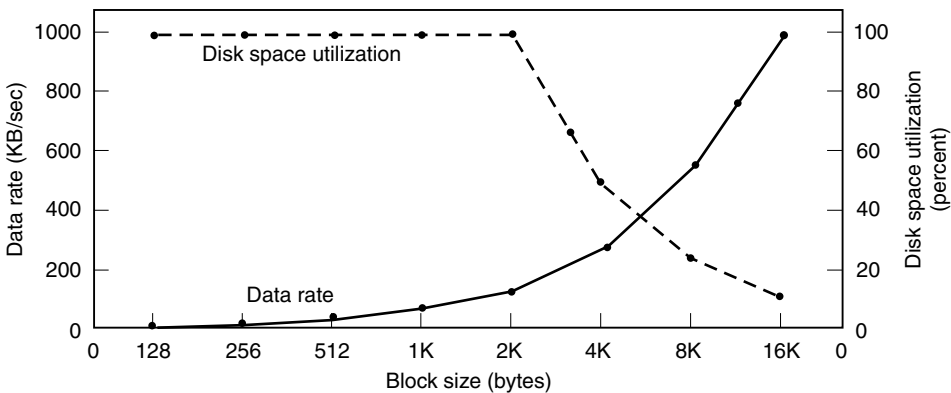


Figure 5-17. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All files are 2 KB.

The two curves can be understood as follows. The access time for a block is completely dominated by the seek time and rotational delay, so given that it is going to cost 14 msec to access a block, the more data that are fetched, the better. Hence the data rate goes up with block size (until the transfers take so long that the transfer time begins to dominate). With small blocks that are powers of two

and 2-KB files, **no space is wasted in a block**. However, with 2-KB files and 4 KB or larger blocks, some disk space is wasted. In reality, few files are a multiple of the disk block size, **so some space is always wasted in the last block of a file**.

What the curves show, however, is that **performance and space utilization are inherently in conflict**. Small blocks are bad for performance but good for disk space utilization. A compromise size is needed. For this data, 4 KB might be a good choice, but some operating systems made their choices a long time ago, when the disk parameters and file sizes were different. **For UNIX, 1 KB is commonly used**. For MS-DOS the block size can be any power of two from 512 bytes to 32 KB, but is determined by the disk size and for reasons unrelated to these arguments (the maximum number of blocks on a disk partition is 2^{16} , which forces large blocks on large disks).

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. 5-18. 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 256 free blocks. (One slot is needed for the pointer to the next block). A 256-GB disk needs a free list of maximum 1,052,689 blocks to hold all 2^{28} disk block numbers. **Often free blocks are used to hold the free list**.

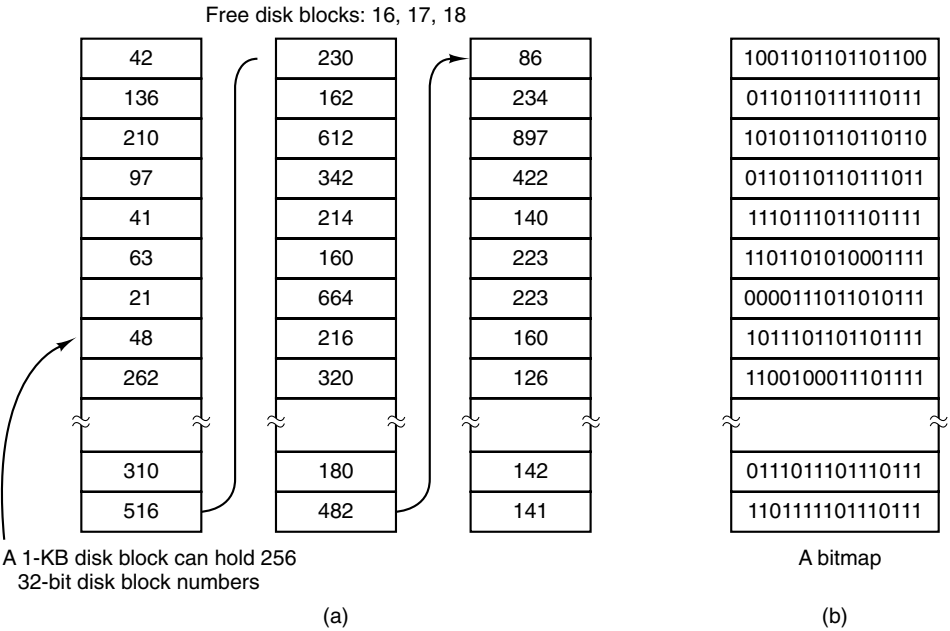


Figure 5-18. (a) Storing the free list on a linked list. (b) A bitmap.

The other free space management technique is the bitmap. A disk with n blocks requires a bitmap with n bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa). A 256-GB disk has 2^{28} 1-KB blocks and thus requires 2^{28} bits for the map, which requires 32,768 blocks. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked list scheme require fewer blocks than the bitmap. On the other hand, if there are many blocks free, some of them can be borrowed to hold the free list without any loss of disk capacity.

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

5.3.5 File System Reliability

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

If a computer's file system is irrevocably lost, whether due to hardware, software, or rats gnawing on the backup tapes, restoring all the information will be difficult and time consuming at best, and in many cases will be impossible. For the people whose programs, documents, customer files, tax records, databases, marketing plans, or other data are gone forever, the consequences can be catastrophic. While the file system cannot offer any protection against physical destruction of the equipment and media, it can help protect the information. In this section we will look at some of the issues involved in safeguarding the file system.

Floppy disks are generally perfect when they leave the factory, but they can develop bad blocks during use. It is arguable that this is more likely now than it was in the days when floppy disks were more widely used. Networks and large capacity removable devices such as writeable CDs have led to floppy disks being used infrequently. Cooling fans draw air and airborne dust in through floppy disk drives, and a drive that has not been used for a long time may be so dirty that it ruins the next disk that is inserted. A floppy drive that is used frequently is less likely to damage a disk.

Hard disks frequently have bad blocks right from the start: it is just too expensive to manufacture them completely free of all defects. As we saw in Chap. 3, bad blocks on hard disks are generally handled by the controller by replacing bad


sectors with spares provided for that purpose. On these disks, tracks are at least one sector bigger than needed, so that at least one bad spot can be skipped by leaving it in a gap between two consecutive sectors. A few spare sectors are provided on each cylinder so the controller can do automatic sector remapping if it notices that a sector needs more than a certain number of retries to be read or written. Thus the user is usually unaware of bad blocks or their management. Nevertheless, when a modern IDE or SCSI disk fails, it will usually fail horribly, because it has run out of spare sectors. SCSI disks provide a “recovered error” when they remap a block. If the driver notes this and displays a message on the monitor the user will know it is time to buy a new disk when these messages begin to appear frequently.

A simple software solution to the bad block problem exists, suitable for use on older disks. This approach requires the user or file system to carefully construct a file containing all the bad blocks. This technique removes them from the free list, so they will never occur in data files. As long as the bad block file is never read or written, no problems will arise. Care has to be taken during disk backups to avoid reading this file and trying to back it up.

Backups

Most people do not think making backups of their files is worth the time and effort—until one fine day their disk abruptly dies, at which time most of them undergo a deathbed conversion. Companies, however, (usually) well understand the value of their data and generally do a backup at least once a day, usually to tape. Modern tapes hold tens or sometimes even hundreds of gigabytes and cost pennies per gigabyte. Nevertheless, making backups is not quite as trivial as it sounds, so we will examine some of the related issues below.

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

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

The first one covers getting the computer running again after a disk crash, fire, flood, or other natural catastrophe. In practice, these things do not happen very often, which is why many people do not bother with backups. These people also tend not to have fire insurance on their houses for the same reason.

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

Making a backup takes a long time and occupies a large amount of space, so doing it efficiently and conveniently is important. These considerations raise the

following issues. First, should the entire file system be backed up or only part of it? At many installations, the executable (binary) programs are kept in a limited part of the file system tree. It is not necessary to back up these files if they can all be reinstalled from the manufacturers' CD-ROMs. Also, most systems have a directory for temporary files. There is usually no reason to back it up either. In UNIX, all the special files (I/O devices) are kept in a directory `/dev/`. Not only is backing up this directory not necessary, it is downright dangerous because the backup program would hang forever if it tried to read each of these to completion. In short, it is usually desirable to back up only specific directories and everything in them rather than the entire file system.

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

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

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

Fifth and last, making backups introduces many **nontechnical problems** into an organization. The best online security system in the world may be useless if the system administrator keeps all the backup tapes in his office and leaves it open and unguarded whenever he walks down the hall to get output from the printer. All a spy has to do is pop in for a second, put one tiny tape in his pocket, and saunter off jauntily. Goodbye security. Also, making a daily backup has little use if the fire that burns down the computers also burns up all the backup tapes. For this reason, backup tapes should be kept off-site, but that introduces more security

risks. For a thorough discussion of these and other practical administration issues, see Nemeth et al. (2001). Below we will discuss only the technical issues involved in making file system backups.

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

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

A second concern is dumping bad blocks. If all bad blocks are remapped by the disk controller and hidden from the operating system as we described in Sec. 5.4.4, physical dumping works fine. On the other hand, if they are visible to the operating system and maintained in one or more “bad block files” or bitmaps, it is absolutely essential that the physical dumping program get access to this information and avoid dumping them to prevent endless disk read errors during the dumping process.

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

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

In order to be able to properly restore even a single file correctly, all information needed to recreate the path to that file must be saved to the backup medium. Thus the first step in doing a logical dump is doing an analysis of the directory tree. Obviously, we need to save any file or directory that has been modified. But for proper restoration, all directories, even unmodified ones, that lie on the path to a modified file or directory must be saved. This means saving not just the data (file names and pointers to i-nodes), all the attributes of the directories must be saved, so they can be restored with the original permissions. The directories and their attributes are written to the tape first, and then modified files (with their attributes) are saved. This makes 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.

A 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 accidental deletion). Suppose that a full file system dump is done Sunday evening and an incremental dump is done on Monday evening. On Tuesday the directory `/usr/jhs/proj/nr3/` is removed, along with all the directories and files under it. On Wednesday morning bright and early, a user wants to restore the file `/usr/jhs/proj/nr3/plans/summary`. However, it is not possible to just restore the file `summary` because there is no place to put it. The directories `nr3/` and `plans/` must be restored first. To get their owners, modes, times, etc., correct, these directories must be present on the dump tape even though they themselves were not modified since the previous full dump.

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

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

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

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

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

File System Consistency

Another area where reliability is an issue is file system consistency. Many file systems read blocks, modify them, and write them out later. If the system crashes before all the modified blocks have been written out, the file system can

be left in an **inconsistent state**. This problem is especially critical if some of the blocks that have not been written out are **i-node blocks, directory blocks, or blocks containing the free list**.

To deal with the problem of inconsistent file systems, **most computers have a utility program that checks file system consistency**. For example, UNIX has *fsck* and Windows has *chkdsk* (or *scandisk* in earlier versions). This utility can be run whenever the system is booted, especially after a crash. The description below tells how *fsck* works. *Chkdsk* is somewhat different because it works on a different file system, but the general principle of using the file system's inherent redundancy to repair it is still valid. All file system checkers verify each file system (disk partition) independently of the other ones.

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

The program then reads all the i-nodes. Starting from an i-node, it is possible to build a list of all the block numbers used in the corresponding file. As each block number is read, its counter in the first table is incremented. The program then examines the free list or bitmap, to find all the blocks that are not in use. **Each occurrence of a block in the free list results in its counter in the second table being incremented.**

If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 5-19(a). However, as a result of a crash, the tables might look like Fig. 5-19(b), in which block 2 does not occur in either table. **It will be reported as being a missing block.** While missing blocks do no real harm, they do waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: **the file system checker just adds them to the free list.**

Another situation that might occur is that of Fig. 5-19(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. 5-19(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.

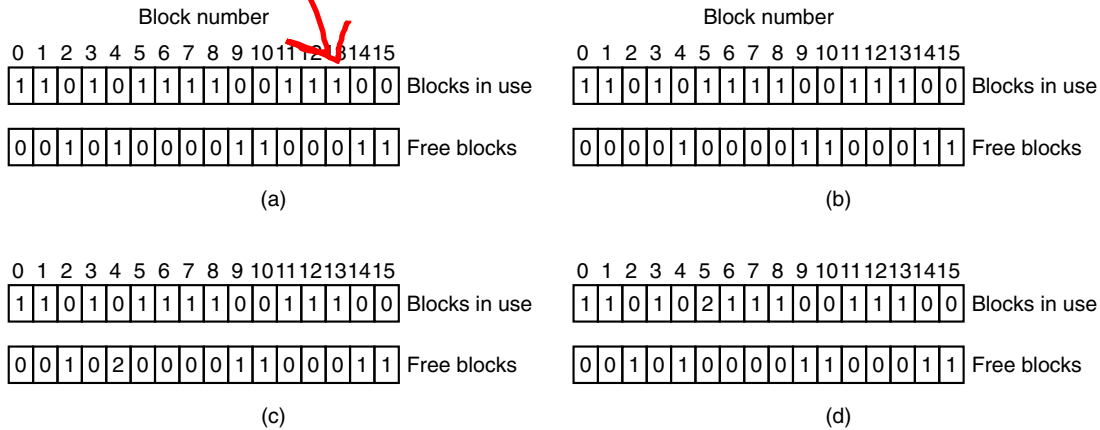


Figure 5-19. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

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

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

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

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

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

Furthermore, each i-node has a mode, some of which are legal but strange, such as 0007, which allows the owner and his group no access at all, but allows outsiders to read, write, and execute the file. It might be useful to at least report files that give outsiders more rights than the owner. Directories with more than, say, 1000 entries are also suspicious. Files located in user directories, but which are owned by the superuser and have the SETUID bit on, are potential security problems because such files acquire the powers of the superuser when executed by any user. With a little effort, one can put together a fairly long list of technically legal but still peculiar situations that might be worth reporting.

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

```
rm *.o
```

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

```
rm * .o
```

(note the space after the asterisk), *rm* will remove all the files in the current directory and then complain that it cannot find `.o`. In some systems, when a file is removed, all that happens is that a bit is set in the directory or i-node marking the file as removed. No disk blocks are returned to the free list until they are actually needed. Thus, if the user discovers the error immediately, it is possible to run a special utility program that “unremoves” (i.e., restores) the removed files. In Windows, files that are removed are placed in the recycle bin, from which they can later be retrieved if need be. Of course, no storage is reclaimed until they are actually deleted from this directory.

Mechanisms like this are insecure. **A secure system would actually overwrite the data blocks with zeros or random bits when a disk is deleted, so another user could not retrieve it.** Many users are unaware how long data can live. Confidential or sensitive data can often be recovered from disks that have been discarded (Garfinkel and Shelat, 2003).

5.3.6 File System Performance

Access to disk is much slower than access to memory. Reading a memory word might take **10 nsec**. Reading from a hard disk might proceed at **10 MB/sec**, which is forty times slower per 32-bit word, and 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 *ca-cher*, meaning to hide.) In this context, a cache is a collection of blocks that logically belong on the disk but are being kept in memory for performance reasons.

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

Operation of the cache is illustrated in Fig. 5-20. Since there are many (often thousands of) blocks in the cache, some way is needed to determine quickly if a given block is present. The usual way is to hash the device and disk address and look up the result in a hash table. All the blocks with the same hash value are chained together on a linked list so the collision chain can be followed.

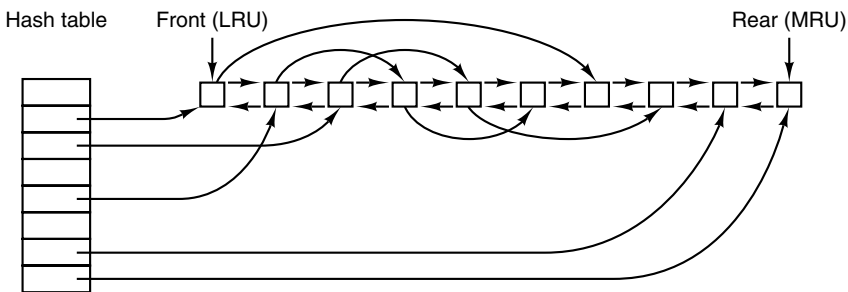


Figure 5-20. The buffer cache data structures.

When a block has to be loaded into a full cache, some block has to be removed (and rewritten to the disk if it has been modified since being brought in). This situation is very much like paging, and all the usual page replacement algorithms described in Chap. 4, such as FIFO, second chance, and LRU, are applicable. One pleasant difference between paging and caching is that cache references are relatively infrequent, so that it is feasible to keep all the blocks in exact LRU order with linked lists.

In Fig. 5-20, 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 twice 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 system crash, a comforting thought for many people.

The Windows way is to write every modified block to disk as soon as it has been written. Caches in which all modified blocks are written back to the disk immediately are called **write-through caches**. They require more disk I/O than nonwrite-through caches. The difference between these two approaches can be seen when a program writes a 1-KB block full, one character at a time. UNIX will collect all the characters in the cache and write the block out once every 30 seconds, or whenever the block is removed from the cache. Windows will make a disk access for every character written. Of course, most programs do internal buffering, so they normally write not a character, but a line or a larger unit on each write system call.

A consequence of this difference in caching strategy is that just removing a (floppy) disk from a UNIX system without doing a sync will almost always result in lost data, and frequently in a corrupted file system as well. With Windows, 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 Windows started out in the floppy disk world. As hard disks became the norm, the UNIX approach, with its better efficiency, became the norm, and is also used now on Windows for hard disks.

Block Read Ahead

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

Of course, this read ahead strategy only works for files that are being read sequentially. If a file is being randomly accessed, read ahead does not help. In fact, it hurts by tying up disk bandwidth reading in useless blocks and removing potentially useful blocks from the cache (and possibly tying up more disk bandwidth writing them back to disk if they are dirty). To see whether read ahead is worth doing, the file system can keep track of the access patterns to each open file. For example, a bit associated with each file can keep track of whether the file is in “sequential access mode” or “random access mode.” Initially, the file is given the benefit of the doubt and put in sequential access mode. However, whenever a seek is done, the bit is cleared. If sequential reads start happening again, the bit is set once again. In this way, the file system can make a reasonable guess about whether it should read ahead or not. If it gets it wrong once 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, as they are needed. If the free blocks are recorded in a bitmap, and the whole bitmap is in main memory, it is easy enough to choose a free block as close as possible to the previous block. With a free list, part of which is on disk, it is much harder to allocate blocks close together.

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

Another performance bottleneck in systems that use i-nodes or anything equivalent to i-nodes is that reading even a short file requires two disk accesses: one for the i-node and one for the block. The usual i-node placement is shown in Fig. 5-21(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.

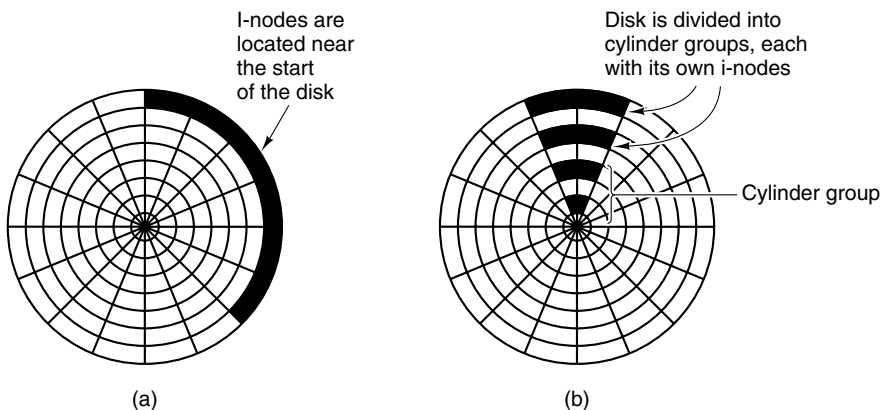


Figure 5-21. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

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. 5-21(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.

5.3.7 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 percent.

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

From this reasoning, the LFS designers decided to re-implement the UNIX file system in such a way as to achieve the full bandwidth of the disk, even in the face of a workload consisting in large part of small random writes. The basic idea is to structure the entire disk as a log. Periodically, and also 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, data blocks, and other kinds of 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-node number, as in UNIX. To make it possible to find i-nodes, an i-node map, indexed by i-node 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 in order to improve performance.

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

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

To deal with this problem, LFS has a **cleaner** thread that spends its time scanning the log circularly to compact it. It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there. It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use. If not, that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment. The original segment is then marked as free, so the log can use it for new data. In this manner, the cleaner moves along the log, removing old segments from the back and putting any live data into memory for rewriting in the next segment. Consequently, the disk is a big circular buffer, with the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

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

5.4 SECURITY

File systems generally contain information that is highly valuable to their users. Protecting this information against unauthorized usage is therefore a major concern of all file systems. In the following sections we will look at a variety of issues concerned with security and protection. These issues apply equally well to timesharing systems as to networks of personal computers connected to shared servers via local area networks.

5.4.1 The Security Environment

People frequently use the terms “security” and “protection” interchangeably. Nevertheless, it is frequently useful to make a distinction between the general problems involved in making sure that files are not read or modified by unauthorized persons, which include technical, administrative, legal, and political issues on the one hand, and the specific operating system mechanisms used to provide security, on the other. To avoid confusion, we will use the term **security** to refer to the overall problem, and the term **protection mechanisms** to refer to the specific operating system mechanisms used to safeguard information in the computer. The boundary between them is not well defined, however. First we will look at security to see what the nature of the problem is. Later on in the chapter we will look at the protection mechanisms and models available to help achieve security.

Security has many facets. Three of the more important ones are the nature of the threats, the nature of intruders, and accidental data loss. We will now look at these in turn.

Threats

From a security perspective, computer systems have three general goals, with corresponding threats to them, as listed in Fig. 5-22. The first one, **data confidentiality**, is concerned with having secret data remain secret. More specifically, if the owner of some data has decided that these data are only to be made available to certain people and no others, the system should guarantee that release of the data to unauthorized people does not occur. As a bare minimum, the owner should be able to specify who can see what, and the system should enforce these specifications.

The second goal, **data integrity**, means that unauthorized users should not be able to modify any data without the owner’s permission. Data modification in this context includes not only changing the data, but also removing data and adding false data as well. If a system cannot guarantee that data deposited in it remain unchanged until the owner decides to change them, it is not worth much as an information system. Integrity is usually more important than confidentiality.

Goal	Threat
Data confidentiality	Exposure of data
Data integrity	Tampering with data
System availability	Denial of service

Figure 5-22. Security goals and threats.

The third goal, **system availability**, means that nobody can disturb the system to make it unusable. Such **denial of service** attacks are increasingly common. For example, if a computer is an Internet server, sending a flood of requests to it may cripple it by eating up all of its CPU time just examining and discarding incoming requests. If it takes, say, 100 μ sec to process an incoming request to read a Web page, then anyone who manages to send 10,000 requests/sec can wipe it out. Reasonable models and technology for dealing with attacks on confidentiality and integrity are available; foiling denial-of-services attacks is much harder.

Another aspect of the security problem is **privacy**: protecting individuals from misuse of information about them. This quickly gets into many legal and moral issues. Should the government compile dossiers on everyone in order to catch *X*-cheaters, where *X* is “welfare” or “tax,” depending on your politics? Should the police be able to look up anything on anyone in order to stop organized crime? Do employers and insurance companies have rights? What happens when these rights conflict with individual rights? All of these issues are extremely important but are beyond the scope of this book.

Intruders

Most people are pretty nice and obey the law, so why worry about security? Because there are unfortunately a few people around who are not so nice and want to cause trouble (possibly for their own commercial gain). In the security literature, people who are nosing around places where they have no business being are called **intruders** or sometimes **adversaries**. Intruders act in two different ways. Passive intruders just want to read files they are not authorized to read. Active intruders are more malicious; they want to make unauthorized changes. When designing a system to be secure against intruders, it is important to keep in mind the kind of intruder one is trying to protect against. Some common categories are

1. Casual prying by nontechnical users. Many people have personal computers on their desks that are connected to a shared file server, and human nature being what it is, some of them will read other people’s electronic mail and other files if no barriers are placed in the way. Most UNIX systems, for example, have the default that all newly created files are publicly readable.

2. Snooping by insiders. Students, system programmers, operators, and other technical personnel often consider it to be a personal challenge to break the security of the local computer system. They often are highly skilled and are willing to devote a substantial amount of time to the effort.
3. Determined attempts to make money. Some bank programmers have attempted to steal from the bank they were working for. Schemes have varied from changing the software to truncate rather than round interest, keeping the fraction of a cent for themselves, to siphoning off accounts not used in years, to blackmail (“Pay me or I will destroy all the bank’s records.”).
4. Commercial or military espionage. Espionage refers to a serious and well-funded attempt by a competitor or a foreign country to steal programs, trade secrets, patentable ideas, technology, circuit designs, business plans, and so forth. Often this attempt will involve wiretapping or even erecting antennas directed at the computer to pick up its electromagnetic radiation.

It should be clear that trying to keep a hostile foreign government from stealing military secrets is quite a different matter from trying to keep students from inserting a funny message-of-the-day into the system. The amount of effort needed for security and protection clearly depends on who the enemy is thought to be.

Malicious Programs

Another category of security pest is malicious programs, sometimes called **malware**. In a sense, a writer of malware is also an intruder, often with high technical skills. The difference between a conventional intruder and malware is that the former refers to a person who is personally trying to break into a system to cause damage whereas the latter is a program written by such a person and then released into the world. Some malware seems to have been written just to cause damage, but some is targeted more specifically. It is becoming a huge problem and a great deal has been written about it (Aycock and Barker, 2005; Cerf, 2005; Ledin, 2005; McHugh and Deek, 2005; Treese, 2004; and Weiss, 2005)

The most well known kind of malware is the **virus**. Basically a virus is a piece of code that can reproduce itself by attaching a copy of itself to another program, analogous to how biological viruses reproduce. The virus can do other things in addition to reproducing itself. For example, it can type a message, display an image on the screen, play music, or something else harmless. Unfortunately, it can also modify, destroy, or steal files (by e-mailing them somewhere).

Another thing a virus can do is to render the computer unusable as long as the virus is running. This is called a **DOS (Denial Of Service)** attack. The usual ap-

proach is to consume resources wildly, such as the CPU, or filling up the disk with junk. Viruses (and the other forms of malware to be described) can also be used to cause a **DDOS (Distributed Denial Of Service)** attack. In this case the virus does not do anything immediately upon infecting a computer. At a predetermined date and time thousands of copies of the virus on computers all over the world start requesting web pages or other network services from their target, for instance the Web site of a political party or a corporation. This can overload the targeted server and the networks that service it.

Malware is frequently created for profit. Much (if not most) unwanted junk e-mail (“spam”) is relayed to its final destinations by networks of computers that have been infected by viruses or other forms of malware. A computer infected by such a rogue program becomes a slave, and reports its status to its master, somewhere on the Internet. The master then sends spam to be relayed to all the e-mail addresses that can be gleaned from e-mail address books and other files on the slave. Another kind of malware for profit scheme installs a **key logger** on an infected computer. A key logger records everything typed at the keyboard. It is not too difficult to filter this data and extract information such as username—password combinations or credit card numbers and expiration dates. This information is then sent back to a master where it can be used or sold for criminal use.

Related to the virus is the **worm**. Whereas a virus is spread by attaching itself to another program, and is executed when its host program is executed, a worm is a free-standing program. Worms spread by using networks to transmit copies of themselves to other computers. Windows systems always have a *Startup* directory for each user; any program in that folder will be executed when the user logs in. So all the worm has to do is arrange to put itself (or a shortcut to itself) in the *Startup* directory on a remote system. Other ways exist, some much more difficult to detect, to cause a remote computer to execute a program file that has been copied to its file system. The effects of a worm can be the same as those of a virus. Indeed, the distinction between a virus and a worm is not always clear; some malware uses both methods to spread.

Another category of malware is the **Trojan horse**. This is a program that apparently performs a valid function—perhaps it is a game or a supposedly “improved” version of a useful utility. But when the Trojan horse is executed some other function is performed, perhaps launching a worm or virus or performing one of the nasty things that malware does. The effects of a Trojan horse are likely to be subtle and stealthy. Unlike worms and viruses, Trojan horses are voluntarily downloaded by users, and as soon as they are recognized for what they are and the word gets out, a Trojan horse will be deleted from reputable download sites.

Another kind of malware is the **logic bomb**. This device is a piece of code written by one of a company’s (currently employed) programmers and secretly inserted into the production operating system. As long as the programmer feeds it its daily password, it does nothing. However, if the programmer is suddenly fired

and physically removed from the premises without warning, the next day the logic bomb does not get its password, so it goes off.

Going off might involve clearing the disk, erasing files at random, carefully making hard-to-detect changes to key programs, or encrypting essential files. In the latter case, the company has a tough choice about whether to call the police (which may or may not result in a conviction many months later) or to give in to this blackmail and to rehire the ex-programmer as a “consultant” for an astronomical sum to fix the problem (and hope that he does not plant new logic bombs while doing so).

Yet another form of malware is **spyware**. This is usually obtained by visiting a Web site. In its simplest form spyware may be nothing more than a **cookie**. Cookies are small files exchanged between web browsers and web servers. They have a legitimate purpose. A cookie contains some information that will allow the Web site to identify you. It is like the ticket you get when you leave a bicycle to be repaired. When you return to the shop, your half of the ticket gets matched with your bicycle (and its repair bill). Web connections are not persistent, so, for example, if you indicate an interest in buying this book when visiting an online bookstore, the bookstore asks your browser to accept a cookie. When you have finished browsing and perhaps have selected other books to buy, you click on the page where your order is finalized. At that point the web server asks your browser to return the cookies it has stored from the current session. It can use the information in these to generate the list of items you have said you want to buy.

Normally, cookies used for a purpose like this expire quickly. They are quite useful, and e-commerce depends upon them. But some Web sites use cookies for purposes that are not so benign. For instance, advertisements on Web sites are often furnished by companies other than the information provider. Advertisers pay Web site owners for this privilege. If a cookie is placed when you visit a page with information about, say, bicycle equipment, and you then go to another Web site that sells clothing, the same advertising company may provide ads on this page, and may collect cookies you obtained elsewhere. Thus you may suddenly find yourself viewing ads for special gloves or jackets especially made for cyclists. Advertisers can collect a lot of information about your interests this way; you may not want to share so much information about yourself.

What is worse, there are various ways a Web site may be able to download executable program code to your computer. Most browsers accept **plug-ins** to add additional function, such as displaying new kinds of files. Users often accept offers for new plugins without knowing much about what the plugin does. Or a user may willingly accept an offer to be provided with a new cursor for the desktop that looks like a dancing kitten. And a bug in a web browser may allow a remote site to install an unwanted program, perhaps after luring the user to a page that has been carefully constructed to take advantage of the vulnerability. Any time a program is accepted from another source, voluntarily or not, there is a risk it could contain code that does you harm.

Accidental Data Loss

In addition to threats caused by malicious intruders, valuable data can be lost by accident. Some of the common causes of accidental data loss are

1. Acts of God: fires, floods, earthquakes, wars, riots, or rats gnawing tapes or floppy disks.
2. Hardware or software errors: CPU malfunctions, unreadable disks or tapes, telecommunication errors, program bugs.
3. Human errors: incorrect data entry, wrong tape or disk mounted, wrong program run, lost disk or tape, or some other mistake.

Most of these can be dealt with by maintaining adequate backups, preferably far away from the original data. While protecting data against accidental loss may seem mundane compared to protecting against clever intruders, in practice, probably more damage is caused by the former than the latter.

5.4.2 Generic Security Attacks

Finding security flaws is not easy. The usual way to test a system's security is to hire a group of experts, known as **tiger teams** or **penetration teams**, to see if they can break in. Hebbard et al. (1980) tried the same thing with graduate students. In the course of the years, these penetration teams have discovered a number of areas in which systems are likely to be weak. Below we have listed some of the more common attacks that are often successful. When designing a system, be sure it can withstand attacks like these.

1. Request memory pages, disk space, or tapes and just read them. Many systems do not erase them before allocating them, and they may be full of interesting information written by the previous owner.
2. Try illegal system calls, or legal system calls with illegal parameters, or even legal system calls with legal but unreasonable parameters. Many systems can easily be confused.
3. Start logging in and then hit DEL, RUBOUT or BREAK halfway through the login sequence. In some systems, the password checking program will be killed and the login considered successful.
4. Try modifying complex operating system structures kept in user space (if any). In some systems (especially on mainframes), to open a file, the program builds a large data structure containing the file name and many other parameters and passes it to the system. As the file is read and written, the system sometimes updates the structure itself. Changing these fields can wreak havoc with the security.

5. Spoof the user by writing a program that types “login:” on the screen and go away. Many users will walk up to the terminal and willingly tell it their login name and password, which the program carefully records for its evil master.
6. Look for manuals that say “Do not do X.” Try as many variations of *X* as possible.
7. Convince a system programmer to change the system to skip certain vital security checks for any user with your login name. This attack is known as a **trapdoor**.
8. All else failing, the penetrator might find the computer center director’s secretary and offer a large bribe. The secretary probably has easy access to all kinds of wonderful information, and is usually poorly paid. Do not underestimate problems caused by personnel.

These and other attacks are discussed by Linde (1975). Many other sources of information on security and testing security can be found, especially on the Web. A recent Windows-oriented work is Johansson and Riley (2005).

5.4.3 Design Principles for Security

Saltzer and Schroeder (1975) have identified several general principles that can be used as a guide to designing secure systems. A brief summary of their ideas (based on experience with MULTICS) is given below.

First, the system design should be public. Assuming that the intruder will not know how the system works serves only to delude the designers.

Second, the default should be no access. Errors in which legitimate access is refused will be reported much faster than errors in which unauthorized access is allowed.

Third, check for current authority. The system should not check for permission, determine that access is permitted, and then squirrel away this information for subsequent use. Many systems check for permission when a file is opened, and not afterward. This means that a user who opens a file, and keeps it open for weeks, will continue to have access, even if the owner has long since changed the file protection.

Fourth, give each process the least privilege possible. If an editor has only the authority to access the file to be edited (specified when the editor is invoked), editors with Trojan horses will not be able to do much damage. This principle implies a fine-grained protection scheme. We will discuss such schemes later in this chapter.

Fifth, the protection mechanism should be simple, uniform, and built into the lowest layers of the system. Trying to retrofit security to an existing insecure system is nearly impossible. Security, like correctness, is not an add-on feature.

Sixth, the scheme chosen must be psychologically acceptable. If users feel that protecting their files is too much work, they just will not do it. Nevertheless, they will complain loudly if something goes wrong. Replies of the form “It is your own fault” will generally not be well received.

5.4.4 User Authentication

Many protection schemes are based on the assumption that the system knows the identity of each user. The problem of identifying users when they log in is called **user authentication**. Most authentication methods are based on identifying something the user knows, something the user has, or something the user is.

Passwords

The most widely used form of authentication is to require the user to type a password. Password protection is easy to understand and easy to implement. In UNIX it works like this: The login program asks the user to type his name and password. The password is immediately encrypted. The login program then reads the password file, which is a series of ASCII lines, one per user, until it finds the line containing the user’s login name. If the (encrypted) password contained in this line matches the encrypted password just computed, the login is permitted, otherwise it is refused.

Password authentication is easy to defeat. One frequently reads about groups of high school, or even junior high school students who, with the aid of their trusty home computers, have broken into some top secret system owned by a large corporation or government agency. Virtually all the time the break-in consists of guessing a user name and password combination.

Although more recent studies have been made (e.g., Klein, 1990) the classic work on password security remains the one done by Morris and Thompson (1979) on UNIX systems. They compiled a list of likely passwords: first and last names, street names, city names, words from a moderate-sized dictionary (also words spelled backward), license plate numbers, and short strings of random characters.

They then encrypted each of these using the known password encryption algorithm and checked to see if any of the encrypted passwords matched entries in their list. Over 86 percent of all passwords turned up in their list.

If all passwords consisted of 7 characters chosen at random from the 95 printable ASCII characters, the search space becomes 95^7 , which is about 7×10^{13} . At 1000 encryptions per second, it would take 2000 years to build the list to check the password file against. Furthermore, the list would fill 20 million magnetic tapes. Even requiring passwords to contain at least one lowercase character, one uppercase character, and one special character, and be at least seven characters long would be a major improvement over unrestricted user-chosen passwords.

Even if it is considered politically impossible to require users to pick reasonable passwords, Morris and Thompson have described a technique that renders their own attack (encrypting a large number of passwords in advance) almost useless. Their idea is to associate an n -bit random number with each password. The random number is changed whenever the password is changed. The random number is stored in the password file in unencrypted form, so that everyone can read it. Instead of just storing the encrypted password in the password file, the password and the random number are first concatenated and then encrypted together. This encrypted result is stored in the password file.

Now consider the implications for an intruder who wants to build up a list of likely passwords, encrypt them, and save the results in a sorted file, f , so that any encrypted password can be looked up easily. If an intruder suspects that *Marilyn* might be a password, it is no longer sufficient just to encrypt *Marilyn* and put the result in f . He has to encrypt 2^n strings, such as *Marilyn0000*, *Marilyn0001*, *Marilyn0002*, and so forth and enter all of them in f . This technique increases the size of f by 2^n . UNIX uses this method with $n = 12$. It is known as **salting** the password file. Some versions of UNIX make the password file itself unreadable but provide a program to look up entries upon request, adding just enough delay to greatly slow down any attacker.

Although this method offers protection against intruders who try to precompute a large list of encrypted passwords, it does little to protect a user *David* whose password is also *David*. One way to encourage people to pick better passwords is to have the computer offer advice. Some computers have a program that generates random easy-to-pronounce nonsense words, such as *fotally*, *garbungy*, or *bipitty* that can be used as passwords (preferably with some upper case and special characters thrown in).

Other computers require users to change their passwords regularly, to limit the damage done if a password leaks out. The most extreme form of this approach is the **one-time password**. When one-time passwords are used, the user gets a book containing a list of passwords. Each login uses the next password in the list. If an intruder ever discovers a password, it will not do him any good, since next time a different password must be used. It is suggested that the user try to avoid losing the password book.

It goes almost without saying that while a password is being typed in, the computer should not display the typed characters, to keep them from prying eyes near the terminal. What is less obvious is that passwords should never be stored in the computer in unencrypted form. Furthermore, not even the computer center management should have unencrypted copies. Keeping unencrypted passwords anywhere is looking for trouble.

A variation on the password idea is to have each new user provide a long list of questions and answers that are then stored in the computer in encrypted form. The questions should be chosen so that the user does not need to write them down. In other words, they should be things no one forgets. Typical questions are:

1. Who is Marjolein's sister?
2. On what street was your elementary school?
3. What did Mrs. Woroboff teach?

At login, the computer asks one of them at random and checks the answer.

Another variation is **challenge-response**. When this is used, the user picks an algorithm when signing up as a user, for example x^2 . When the user logs in, the computer types an argument, say 7, in which case the user types 49. The algorithm can be different in the morning and afternoon, on different days of the week, from different terminals, and so on.

Physical Identification

A completely different approach to authorization is to check to see if the user has some item, normally a plastic card with a magnetic stripe on it. The card is inserted into the terminal, which then checks to see whose card it is. This method can be combined with a password, so a user can only log in if he (1) has the card and (2) knows the password. Automated cash-dispensing machines usually work this way.

Yet another approach is to measure physical characteristics that are hard to forge. For example, a fingerprint or a voiceprint reader in the terminal could verify the user's identity. (It makes the search go faster if the user tells the computer who he is, rather than making the computer compare the given fingerprint to the entire data base.) Direct visual recognition is not yet feasible but may be one day.

Another technique is signature analysis. The user signs his name with a special pen connected to the terminal, and the computer compares it to a known specimen stored on line. Even better is not to compare the signature, but compare the pen motions made while writing it. A good forger may be able to copy the signature, but will not have a clue as to the exact order in which the strokes were made.

Finger length analysis is surprisingly practical. When this is used, each terminal has a device like the one of Fig. 5-23. The user inserts his hand into it, and the length of each of his fingers is measured and checked against the data base.

We could go on and on with more examples, but two more will help make an important point. Cats and other animals mark off their territory by urinating around its perimeter. Apparently cats can identify each other this way. Suppose that someone comes up with a tiny device capable of doing an instant urinalysis, thereby providing a foolproof identification. Each terminal could be equipped with one of these devices, along with a discreet sign reading: "For login, please deposit sample here." This might be an absolutely unbreakable system, but it would probably have a fairly serious user acceptance problem.

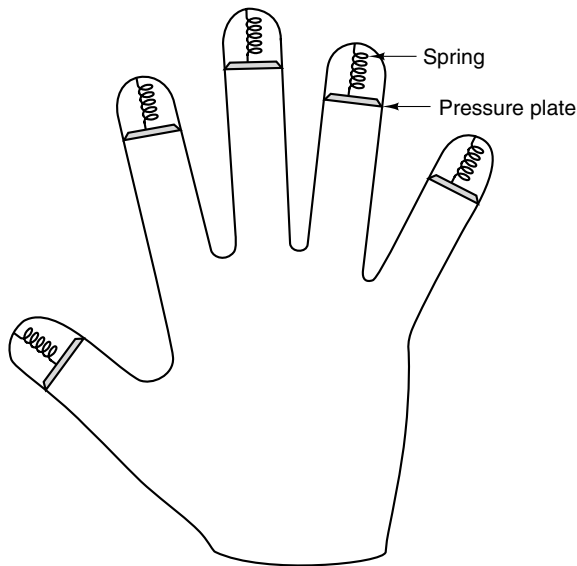


Figure 5-23. A device for measuring finger length.

The same could be said of a system consisting of a thumbtack and a small spectrograph. The user would be requested to jab his thumb against the thumbtack, thus extracting a drop of blood for spectrographic analysis. The point is that any authentication scheme must be psychologically acceptable to the user community. Finger-length measurements probably will not cause any problem, but even something as nonintrusive as storing fingerprints on line may be unacceptable to many people.

Countermeasures

Computer installations that are really serious about security—and few are until the day after an intruder has broken in and done major damage—often take steps to make unauthorized entry much harder. For example, each user could be allowed to log in only from a specific terminal, and only during certain days of the week and hours of the day.

Dial-up telephone lines could be made to work as follows. Anyone can dial up and log in, but after a successful login, the system immediately breaks the connection and calls the user back at an agreed upon number. This measure means that an intruder cannot just try breaking in from any phone line; only the user's (home) phone will do. In any event, with or without call back, the system should take at least 10 seconds to check any password typed in on a dial-up line, and should increase this time after several consecutive unsuccessful login attempts, in

order to reduce the rate at which intruders can try. After three failed login attempts, the line should be disconnected for 10 minutes and security personnel notified.

All logins should be recorded. When a user logs in, the system should report the time and terminal of the previous login, so he can detect possible break ins.

The next step up is laying baited traps to catch intruders. A simple scheme is to have one special login name with an easy password (e.g., login name: guest, password: guest). Whenever anyone logs in using this name, the system security specialists are immediately notified. Other traps can be easy-to-find bugs in the operating system and similar things, designed for the purpose of catching intruders in the act. Stoll (1989) has written an entertaining account of the traps he set to track down a spy who broke into a university computer in search of military secrets.

5.5 PROTECTION MECHANISMS

In the previous sections we have looked at many potential problems, some of them technical, some of them not. In the following sections we will concentrate on some of the detailed technical ways that are used in operating systems to protect files and other things. All of these techniques make a clear distinction between policy (whose data are to be protected from whom) and mechanism (how the system enforces the policy). The separation of policy and mechanism is discussed by Sandhu (1993). Our emphasis will be on mechanisms, not policies.

In some systems, protection is enforced by a program called a **reference monitor**. Every time an access to a potentially protected resource is attempted, the system first asks the reference monitor to check its legality. The reference monitor then looks at its policy tables and makes a decision. Below we will describe the environment in which a reference monitor operates.

5.5.1 Protection Domains

A computer system contains many “objects” that need to be protected. These objects can be hardware (e.g., CPUs, memory segments, disk drives, or printers), or they can be software (e.g., processes, files, databases, or semaphores).

Each object has a unique name by which it is referenced, and a finite set of operations that processes are allowed to carry out on it. The read and write operations are appropriate to a file; up and down make sense on a semaphore.

It is obvious that a way is needed to prohibit processes from accessing objects that they are not authorized to access. Furthermore, this mechanism must also make it possible to restrict processes to a subset of the legal operations when that is needed. For example, process *A* may be entitled to read, but not write, file *F*.

In order to discuss different protection mechanisms, it is useful to introduce the concept of a domain. A **domain** is a set of (object, rights) pairs. Each pair specifies an object and some subset of the operations that can be performed on it. A **right** in this context means permission to perform one of the operations. Often a domain corresponds to a single user, telling what the user can do and not do, but a domain can also be more general than just one user.

Figure 5-24 shows three domains, showing the objects in each domain and the rights [Read, Write, eXecute] available on each object. Note that *Printer1* is in two domains at the same time. Although not shown in this example, it is possible for the same object to be in multiple domains, with *different* rights in each one.

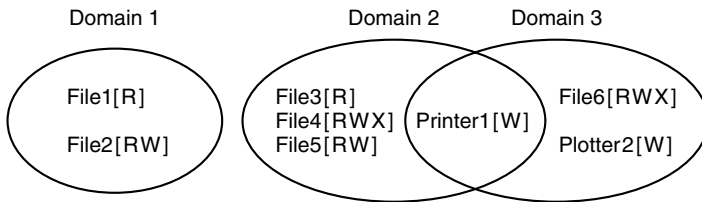


Figure 5-24. Three protection domains.

At every instant of time, each process runs in some protection domain. In other words, there is some collection of objects it can access, and for each object it has some set of rights. Processes can also switch from domain to domain during execution. The rules for domain switching are highly system dependent.

To make the idea of a protection domain more concrete, let us look at UNIX. In UNIX, the domain of a process is defined by its UID and GID. Given any (UID, GID) combination, it is possible to make a complete list of all objects (files, including I/O devices represented by special files, etc.) that can be accessed, and whether they can be accessed for reading, writing, or executing. Two processes with the same (UID, GID) combination will have access to exactly the same set of objects. Processes with different (UID, GID) values will have access to a different set of files, although there may be considerable overlap in most cases.

Furthermore, each process in UNIX has two halves: the user part and the kernel part. When the process does a system call, it switches from the user part to the kernel part. The kernel part has access to a different set of objects from the user part. For example, the kernel can access all the pages in physical memory, the entire disk, and all the other protected resources. Thus, a system call causes a domain switch.

When a process does an exec on a file with the SETUID or SETGID bit on, it acquires a new effective UID or GID. With a different (UID, GID) combination, it has a different set of files and operations available. Running a program with SETUID or SETGID is also a domain switch, since the rights available change.

An important question is how the system keeps track of which object belongs to which domain. Conceptually, at least, one can envision a large matrix, with the

rows being domains and the columns being objects. Each box lists the rights, if any, that the domain contains for the object. The matrix for Fig. 5-24 is shown in Fig. 5-25. Given this matrix and the current domain number, the system can tell if an access to a given object in a particular way from a specified domain is allowed.

Domain	Object							
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 5-25. A protection matrix.

Domain switching itself can be easily included in the matrix model by realizing that a domain is itself an object, with the operation enter. Figure 5-26 shows the matrix of Fig. 5-25 again, only now with the three domains as objects themselves. Processes in domain 1 can switch to domain 2, but once there, they cannot go back. This situation models executing a SETUID program in UNIX. No other domain switches are permitted in this example.

Domain	Object										
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Figure 5-26. A protection matrix with domains as objects.

5.5.2 Access Control Lists

In practice, actually storing the matrix of Fig. 5-26 is rarely done because it is large and sparse. Most domains have no access at all to most objects, so storing a very large, mostly empty, matrix is a waste of disk space. Two methods that are practical, however, are storing the matrix by rows or by columns, and then storing

only the nonempty elements. The two approaches are surprisingly different. In this section we will look at storing it by column; in the next one we will study storing it by row.

The first technique consists of associating with each object an (ordered) list containing all the domains that may access the object, and how. This list is called the **Access Control List** or **ACL** and is illustrated in Fig. 5-27. Here we see three processes, each belonging to a different domain. *A*, *B*, and *C*, and three files *F1*, *F2*, and *F3*. For simplicity, we will assume that each domain corresponds to exactly one user, in this case, users *A*, *B*, and *C*. Often in the security literature, the users are called **subjects** or **principals**, to contrast them with the things owned, the **objects**, such as files.

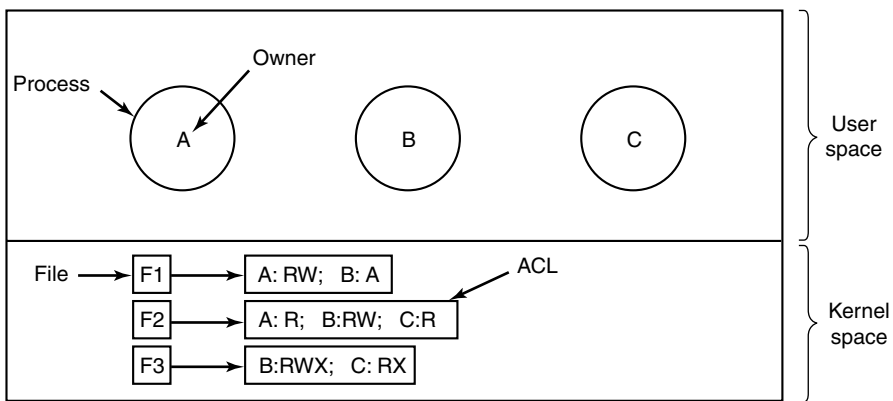


Figure 5-27. Use of access control lists to manage file access.

Each file has an ACL associated with it. File *F1* has two entries in its ACL (separated by a semicolon). The first entry says that any process owned by user *A* may read and write the file. The second entry says that any process owned by user *B* may read the file. All other accesses by these users and all accesses by other users are forbidden. Note that the rights are granted by user, not by process. As far as the protection system goes, any process owned by user *A* can read and write file *F1*. It does not matter if there is one such process or 100 of them. It is the owner, not the process ID, that matters.

File *F2* has three entries in its ACL: *A*, *B*, and *C* can all read the file, and in addition *B* can also write it. No other accesses are allowed. File *F3* is apparently an executable program, since *B* and *C* can both read and execute it. *B* can also write it.

This example illustrates the most basic form of protection with ACLs. More sophisticated systems are often used in practice. To start with, we have only shown three rights so far: read, write, and execute. There may be additional rights as well. Some of these may be generic, that is, apply to all objects, and some may be object specific. Examples of generic rights are destroy object and copy object.

These could hold for any object, no matter what type it is. Object-specific rights might include `append message` for a mailbox object and `sort alphabetically` for a directory object.

So far, our ACL entries have been for individual users. Many systems support the concept of a **group** of users. Groups have names and can be included in ACLs. Two variations on the semantics of groups are possible. In some systems, each process has a user ID (UID) and group ID (GID). In such systems, an ACL entry contains entries of the form

UID1, GID1: rights1; UID2, GID2: rights2; ...

Under these conditions, when a request is made to access an object, a check is made using the caller's UID and GID. If they are present in the ACL, the rights listed are available. If the (UID, GID) combination is not in the list, the access is not permitted.

Using groups this way effectively introduces the concept of a **role**. Consider an installation in which Tana is system administrator, and thus in the group *sysadm*. However, suppose that the company also has some clubs for employees and Tana is a member of the pigeon fanciers club. Club members belong to the group *pigfan* and have access to the company's computers for managing their pigeon database. A portion of the ACL might be as shown in Fig. 5-28.

File	Access control list
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Figure 5-28. Two access control lists.

If Tana tries to access one of these files, the result depends on which group she is currently logged in as. When she logs in, the system may ask her to choose which of her groups she is currently using, or there might even be different login names and/or passwords to keep them separate. The point of this scheme is to prevent Tana from accessing the password file when she currently has her pigeon fancier's hat on. She can only do that when logged in as the system administrator.

In some cases, a user may have access to certain files independent of which group she is currently logged in as. That case can be handled by introducing **wildcards**, which mean everyone. For example, the entry

tana, *: RW

for the password file would give Tana access no matter which group she was currently in as.

Yet another possibility is that if a user belongs to any of the groups that have certain access rights, the access is permitted. In this case, a user belonging to multiple groups does not have to specify which group to use at login time. All of

them count all of the time. A disadvantage of this approach is that it provides less encapsulation: Tana can edit the password file during a pigeon club meeting.

The use of groups and wildcards introduces the possibility of selectively blocking a specific user from accessing a file. For example, the entry

```
virgil, *: (none); *, *: RW
```

gives the entire world except for Virgil read and write access to the file. This works because the entries are scanned in order, and the first one that applies is taken; subsequent entries are not even examined. A match is found for Virgil on the first entry and the access rights, in this case, (none) are found and applied. The search is terminated at that point. The fact that the rest of the world has access is never even seen.

The other way of dealing with groups is not to have ACL entries consist of (UID, GID) pairs, but to have each entry be a UID or a GID. For example, an entry for the file *pigeon_data* could be

```
debbie: RW; phil: RW; pigfan: RW
```

meaning that Debbie and Phil, and all members of the *pigfan* group have read and write access to the file.

It sometimes occurs that a user or a group has certain permissions with respect to a file that the file owner later wishes to revoke. With access control lists, it is relatively straightforward to revoke a previously granted access. All that has to be done is edit the ACL to make the change. However, if the ACL is checked only when a file is opened, most likely the change will only take effect on future calls to open. Any file that is already open will continue to have the rights it had when it was opened, even if the user is no longer authorized to access the file at all.

5.5.3 Capabilities

The other way of slicing up the matrix of Fig. 5-26 is by rows. When this method is used, associated with each process is a list of objects that may be accessed, along with an indication of which operations are permitted on each, in other words, its domain. This list is called a **capability list** or **C-list** and the individual items on it are called **capabilities** (Dennis and Van Horn, 1966; Fabry, 1974). A set of three processes and their capability lists is shown in Fig. 5-29.

Each capability grants the owner certain rights on a certain object. In Fig. 5-29, the process owned by user *A* can read files *F1* and *F2*, for example. Usually, a capability consists of a file (or more generally, an object) identifier and a bitmap for the various rights. In a UNIX-like system, the file identifier would probably be the i-node number. Capability lists are themselves objects and may be pointed to from other capability lists, thus facilitating sharing of subdomains.

It is fairly obvious that capability lists must be protected from user tampering. Three methods of protecting them are known. The first way requires a **tagged**

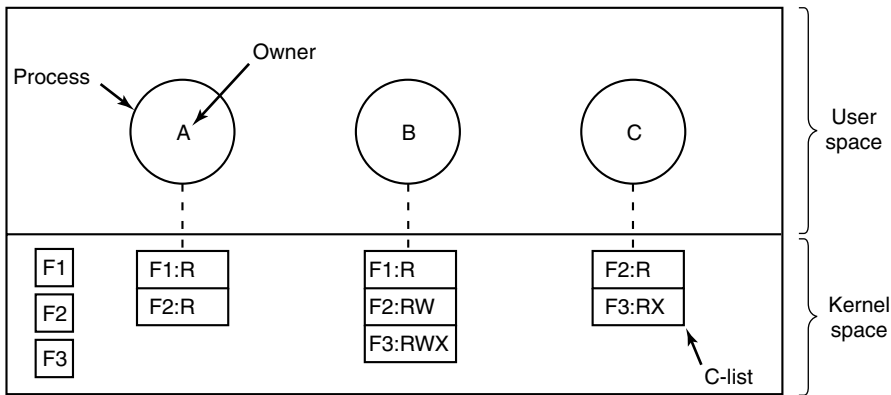


Figure 5-29. When capabilities are used, each process has a capability list.

architecture, a hardware design in which each memory word has an extra (or tag) bit that tells whether the word contains a capability or not. The tag bit is not used by arithmetic, comparison, or similar ordinary instructions, and it can be modified only by programs running in kernel mode (i.e., the operating system). Tagged-architecture machines have been built and can be made to work well (Feustal, 1972). The IBM AS/400 is a popular example.

The second way is to keep the C-list inside the operating system. Capabilities are then referred to by their position in the capability list. A process might say: “Read 1 KB from the file pointed to by capability 2.” This form of addressing is similar to using file descriptors in UNIX. Hydra worked this way (Wulf et al., 1974).

The third way is to keep the C-list in user space, but manage the capabilities cryptographically so that users cannot tamper with them. This approach is particularly suited to distributed systems and works as follows. When a client process sends a message to a remote server, for example, a file server, to create an object for it, the server creates the object and generates a long random number, the check field, to go with it. A slot in the server’s file table is reserved for the object and the check field is stored there along with the addresses of the disk blocks, etc. In UNIX terms, the check field is stored on the server in the i-node. It is not sent back to the user and never put on the network. The server then generates and returns a capability to the user of the form shown in Fig. 5-30.

Server	Object	Rights	f(Objects,Rights,Check)
--------	--------	--------	-------------------------

Figure 5-30. A cryptographically-protected capability.

The capability returned to the user contains the server’s identifier, the object number (the index into the server’s tables, essentially, the i-node number), and the

rights, stored as a bitmap. For a newly created object, all the rights bits are turned on. The last field consists of the concatenation of the object, rights, and check field run through a cryptographically-secure one-way function, f , of the kind we discussed earlier.

When the user wishes to access the object, it sends the capability to the server as part of the request. The server then extracts the object number to index into its tables to find the object. It then computes $f(\text{Object}, \text{Rights}, \text{Check})$ taking the first two parameters from the capability itself and the third one from its own tables. If the result agrees with the fourth field in the capability, the request is honored; otherwise, it is rejected. If a user tries to access someone else's object, he will not be able to fabricate the fourth field correctly since he does not know the check field, and the request will be rejected.

A user can ask the server to produce and return a weaker capability, for example, for read-only access. First the server verifies that the capability is valid. If so, it computes $f(\text{Object}, \text{New_rights}, \text{Check})$ and generates a new capability putting this value in the fourth field. Note that the original *Check* value is used because other outstanding capabilities depend on it.

This new capability is sent back to the requesting process. The user can now give this to a friend by just sending it in a message. If the friend turns on rights bits that should be off, the server will detect this when the capability is used since the f value will not correspond to the false rights field. Since the friend does not know the true check field, he cannot fabricate a capability that corresponds to the false rights bits. This scheme was developed for the Amoeba system and used extensively there (Tanenbaum et al., 1990).

In addition to the specific object-dependent rights, such as read and execute, capabilities (both kernel and cryptographically-protected) usually have **generic rights** which are applicable to all objects. Examples of generic rights are

1. Copy capability: create a new capability for the same object.
2. Copy object: create a duplicate object with a new capability.
3. Remove capability: delete an entry from the C-list; object unaffected.
4. Destroy object: permanently remove an object and a capability.

A last remark worth making about capability systems is that revoking access to an object is quite difficult in the kernel-managed version. It is hard for the system to find all the outstanding capabilities for any object to take them back, since they may be stored in C-lists all over the disk. One approach is to have each capability point to an indirect object, rather than to the object itself. By having the indirect object point to the real object, the system can always break that connection, thus invalidating the capabilities. (When a capability to the indirect object is later presented to the system, the user will discover that the indirect object is now pointing to a null object.)

In the Amoeba scheme, revocation is easy. All that needs to be done is change the check field stored with the object. In one blow, all existing capabilities are invalidated. However, neither scheme allows selective revocation, that is, taking back, say, John's permission, but nobody else's. This defect is generally recognized to be a problem with all capability systems.

Another general problem is making sure the owner of a valid capability does not give a copy to 1000 of his best friends. Having the kernel manage capabilities, as in Hydra, solves this problem, but this solution does not work well in a distributed system such as Amoeba.

On the other hand, capabilities solve the problem of sandboxing mobile code very elegantly. When a foreign program is started, it is given a capability list containing only those capabilities that the machine owner wants to give it, such as the ability to write on the screen and the ability to read and write files in one scratch directory just created for it. If the mobile code is put into its own process with only these limited capabilities, it will not be able to access any other system resources and thus be effectively confined to a sandbox without the need to modify its code or run it interpretively. Running code with as few access rights as possible is known as the **principle of least privilege** and is a powerful guideline for producing secure systems.

Briefly summarized, ACLs and capabilities have somewhat complementary properties. Capabilities are very efficient because if a process says "Open the file pointed to by capability 3," no checking is needed. With ACLs, a (potentially long) search of the ACL may be needed. If groups are not supported, then granting everyone read access to a file requires enumerating all users in the ACL. Capabilities also allow a process to be encapsulated easily, whereas ACLs do not. On the other hand, ACLs allow selective revocation of rights, which capabilities do not. Finally, if an object is removed and the capabilities are not or the capabilities are removed and an object is not, problems arise. ACLs do not suffer from this problem.

5.5.4 Covert Channels

Even with access control lists and capabilities, security leaks can still occur. In this section we discuss how information can still leak out even when it has been rigorously proven that such leakage is mathematically impossible. These ideas are due to Lampson (1973).

Lampson's model was originally formulated in terms of a single timesharing system, but the same ideas can be adapted to LANs and other multiuser environments. In the purest form, it involves three processes on some protected machine. The first process is the client, which wants some work performed by the second one, the server. The client and the server do not entirely trust each other. For example, the server's job is to help clients with filling out their tax forms. The

clients are worried that the server will secretly record their financial data, for example, maintaining a secret list of who earns how much, and then selling the list. The server is worried that the clients will try to steal the valuable tax program.

The third process is the collaborator, which is conspiring with the server to indeed steal the client's confidential data. The collaborator and server are typically owned by the same person. These three processes are shown in Fig. 5-31. The object of this exercise is to design a system in which it is impossible for the server process to leak to the collaborator process the information that it has legitimately received from the client process. Lampson called this the **confinement problem**.

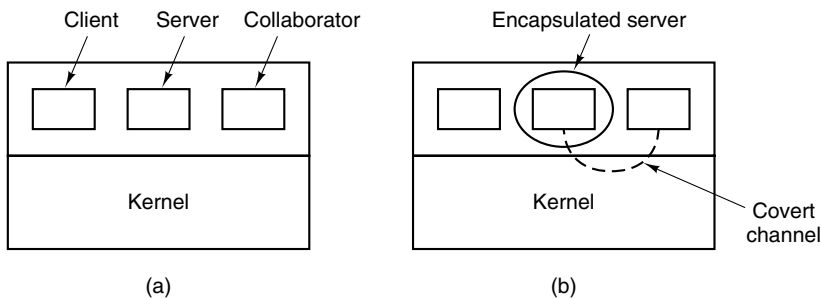


Figure 5-31. (a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator via covert channels.

From the system designer's point of view, the goal is to encapsulate or confine the server in such a way that it cannot pass information to the collaborator. Using a protection matrix scheme we can easily guarantee that the server cannot communicate with the collaborator by writing a file to which the collaborator has read access. We can probably also ensure that the server cannot communicate with the collaborator using the system's normal interprocess communication mechanism.

Unfortunately, more subtle communication channels may be available. For example, the server can try to communicate a binary bit stream as follows: To send a 1 bit, it computes as hard as it can for a fixed interval of time. To send a 0 bit, it goes to sleep for the same length of time.

The collaborator can try to detect the bit stream by carefully monitoring its response time. In general, it will get better response when the server is sending a 0 than when the server is sending a 1. This communication channel is known as a **covert channel**, and is illustrated in Fig. 5-31(b).

Of course, the covert channel is a noisy channel, containing a lot of extraneous information, but information can be reliably sent over a noisy channel by using an error-correcting code (e.g., a Hamming code, or even something more sophisticated). The use of an error-correcting code reduces the already low band-

width of the covert channel even more, but it still may be enough to leak substantial information. It is fairly obvious that no protection model based on a matrix of objects and domains is going to prevent this kind of leakage.

Modulating the CPU usage is not the only covert channel. The paging rate can also be modulated (many page faults for a 1, no page faults for a 0). In fact, almost any way of degrading system performance in a clocked way is a candidate. If the system provides a way of locking files, then the server can lock some file to indicate a 1, and unlock it to indicate a 0. On some systems, it may be possible for a process to detect the status of a lock even on a file that it cannot access. This covert channel is illustrated in Fig. 5-32, with the file locked or unlocked for some fixed time interval known to both the server and collaborator. In this example, the secret bit stream 11010100 is being transmitted.

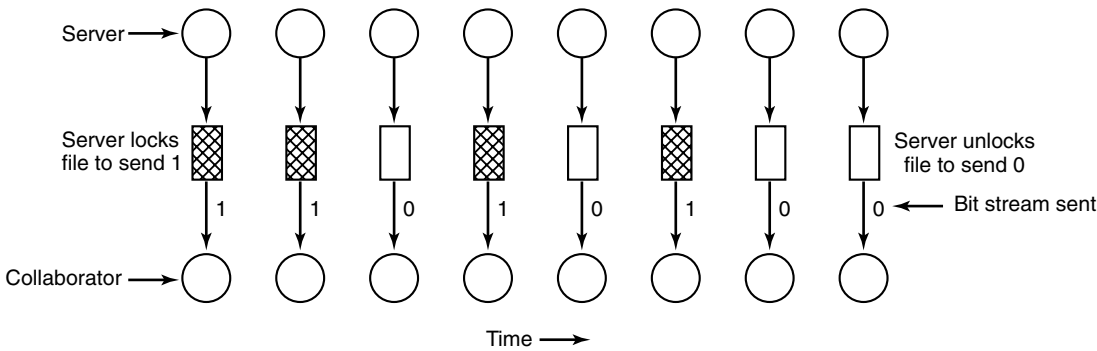


Figure 5-32. A covert channel using file locking.

Locking and unlocking a prearranged file, *S* is not an especially noisy channel, but it does require fairly accurate timing unless the bit rate is very low. The reliability and performance can be increased even more using an acknowledgement protocol. This protocol uses two more files, *F1* and *F2*, locked by the server and collaborator, respectively to keep the two processes synchronized. After the server locks or unlocks *S*, it flips the lock status of *F1* to indicate that a bit has been sent. As soon as the collaborator has read out the bit, it flips *F2*'s lock status to tell the server it is ready for another bit and waits until *F1* is flipped again to indicate that another bit is present in *S*. Since timing is no longer involved, this protocol is fully reliable, even in a busy system and can proceed as fast as the two processes can get scheduled. To get higher bandwidth, why not use two files per bit time, or make it a byte-wide channel with eight signaling files, *S0* through *S7*.

Acquiring and releasing dedicated resources (tape drives, plotters, etc.) can also be used for signaling. The server acquires the resource to send a 1 and releases it to send a 0. In UNIX, the server could create a file to indicate a 1 and remove it to indicate a 0; the collaborator could use the `access` system call to see

if the file exists. This call works even though the collaborator has no permission to use the file. Unfortunately, many other covert channels exist.

Lampson also mentioned a way of leaking information to the (human) owner of the server process. Presumably the server process will be entitled to tell its owner how much work it did on behalf of the client, so the client can be billed. If the actual computing bill is, say, \$100 and the client's income is \$53,000 dollars, the server could report the bill as \$100.53 to its owner.

Just finding all the covert channels, let alone blocking them, is extremely difficult. In practice, there is little that can be done. Introducing a process that causes page faults at random, or otherwise spends its time degrading system performance in order to reduce the bandwidth of the covert channels is not an attractive proposition.

5.6 OVERVIEW OF THE MINIX 3 FILE SYSTEM

Like any file system, the MINIX 3 file system must deal with all the issues we have just studied. It must allocate and deallocate space for files, keep track of disk blocks and free space, provide some way to protect files against unauthorized usage, and so on. In the remainder of this chapter we will look closely at MINIX 3 to see how it accomplishes these goals.

In the first part of this chapter, we have repeatedly referred to UNIX rather than to MINIX 3 for the sake of generality, although the external interfaces of the two is virtually identical. Now we will concentrate on the internal design of MINIX 3. For information about the UNIX internals, see Thompson (1978), Bach (1987), Lions (1996), and Vahalia (1996).

The MINIX 3 file system is just a big C program that runs in user space (see Fig. 2-29). To read and write files, user processes send messages to the file system telling what they want done. The file system does the work and then sends back a reply. The file system is, in fact, a network file server that happens to be running on the same machine as the caller.

This design has some important implications. For one thing, the file system can be modified, experimented with, and tested almost completely independently of the rest of MINIX 3. For another, it is very easy to move the file system to any computer that has a C compiler, compile it there, and use it as a free-standing UNIX-like remote file server. The only changes that need to be made are in the area of how messages are sent and received, which differs from system to system.

In the following sections, we will present an overview of many of the key areas of the file system design. Specifically, we will look at messages, the file system layout, the bitmaps, i-nodes, the block cache, directories and paths, file descriptors, file locking, and special files (plus pipes). After studying these topics, we will show a simple example of how the pieces fit together by tracing what happens when a user process executes the `read` system call.

Messages from users	Input parameters	Reply value
access	File name, access mode	Status
chdir	Name of new working directory	Status
chmod	File name, new mode	Status
chown	File name, new owner, group	Status
chroot	Name of new root directory	Status
close	File descriptor of file to close	Status
creat	Name of file to be created, mode	File descriptor
dup	File descriptor (for dup2, two fds)	New file descriptor
fcntl	File descriptor, function code, arg	Depends on function
fstat	Name of file, buffer	Status
ioctl	File descriptor, function code, arg	Status
link	Name of file to link to, name of link	Status
lseek	File descriptor, offset, whence	New position
mkdir	File name, mode	Status
mknod	Name of dir or special, mode, address	Status
mount	Special file, where to mount, ro flag	Status
open	Name of file to open, r/w flag	File descriptor
pipe	Pointer to 2 file descriptors (modified)	Status
read	File descriptor, buffer, how many bytes	# Bytes read
rename	File name, file name	Status
rmdir	File name	Status
stat	File name, status buffer	Status
stime	Pointer to current time	Status
sync	(None)	Always OK
time	Pointer to place where current time goes	Status
times	Pointer to buffer for process and child times	Status
umask	Complement of mode mask	Always OK
umount	Name of special file to unmount	Status
unlink	Name of file to unlink	Status
utime	File name, file times	Always OK
write	File descriptor, buffer, how many bytes	# Bytes written
Messages from PM	Input parameters	Reply value
exec	Pid	Status
exit	Pid	Status
fork	Parent pid, child pid	Status
setgid	Pid, real and effective gid	Status
setsid	Pid	Status
setuid	Pid, real and effective uid	Status
Other messages	Input parameters	Reply value
revive	Process to revive	(No reply)
unpause	Process to check	(See text)

Figure 5-33. File system messages. File name parameters are always pointers to the name. The code status as reply value means *OK* or *ERROR*.

5.6.1 Messages

The file system accepts 39 types of messages requesting work. All but two are for MINIX 3 system calls. The two exceptions are messages generated by other parts of MINIX 3. Of the system calls, 31 are accepted from user processes. Six system call messages are for system calls which are handled first by the process manager, which then calls the file system to do a part of the work. Two other messages are also handled by the file system. The messages are shown in Fig. 5-33.

The structure of the file system is basically the same as that of the process manager and all the I/O device drivers. It has a main loop that waits for a message to arrive. When a message arrives, its type is extracted and used as an index into a table containing pointers to the procedures within the file system that handle all the types. Then the appropriate procedure is called, it does its work and returns a status value. The file system then sends a reply back to the caller and goes back to the top of the loop to wait for the next message.

5.6.2 File System Layout

A MINIX 3 file system is a logical, self-contained entity with i-nodes, directories, and data blocks. It can be stored on any block device, such as a floppy disk or a hard disk partition. In all cases, the layout of the file system has the same structure. Figure 5-34 shows this layout for a floppy disk or a small hard disk partition with 64 i-nodes and a 1-KB block size. In this simple example, the zone bitmap is just one 1-KB block, so it can keep track of no more than 8192 1-KB zones (blocks), thus limiting the file system to 8 MB. Even for a floppy disk, only 64 i-nodes puts a severe limit on the number of files, so rather than the four blocks reserved for i-nodes in the figure, more would probably be used. Reserving eight blocks for i-nodes would be more practical but our diagram would not look as nice. For a modern hard disk, both the i-node and zone bitmaps will be much larger than 1 block, of course. The relative size of the various components in Fig. 5-34 may vary from file system to file system, depending on their sizes, how many files are allowed maximum, and so on. But all the components are always present and in the same order.

Each file system begins with a **boot block**. This contains executable code. The size of a boot block is always 1024 bytes (two disk sectors), even though MINIX 3 may (and by default does) use a larger block size elsewhere. When the computer is turned on, the hardware reads the boot block from the boot device into memory, jumps to it, and begins executing its code. The boot block code begins the process of loading the operating system itself. Once the system has been booted, the boot block is not used any more. Not every disk drive can be used as a boot device, but to keep the structure uniform, every block device has a block reserved for boot block code. At worst this strategy wastes one block. To prevent the hardware from trying to boot an unbootable device, a **magic number**

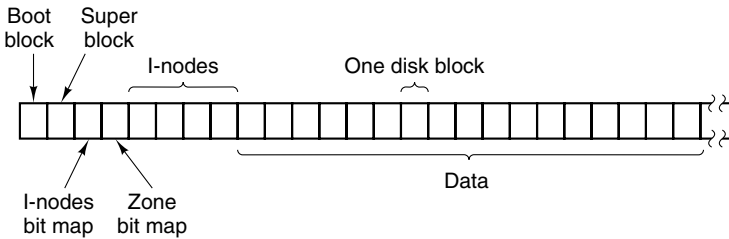


Figure 5-34. Disk layout for a floppy disk or small hard disk partition, with 64 i-nodes and a 1-KB block size (i.e., two consecutive 512-byte sectors are treated as a single block).

is placed at a known location in the boot block when and only when the executable code is written to the device. When booting from a device, the hardware (actually, the BIOS code) will refuse to attempt to load from a device lacking the magic number. Doing this prevents inadvertently using garbage as a boot program.

The **superblock** contains information describing the layout of the file system. Like the boot block, the superblock is always 1024 bytes, regardless of the block size used for the rest of the file system. It is illustrated in Fig. 5-35.

The main function of the superblock is to tell the file system how big the various pieces of the file system are. Given the block size and the number of i-nodes, it is easy to calculate the size of the i-node bitmap and the number of blocks of i-nodes. For example, for a 1-KB block, each block of the bitmap has 1024 bytes (8192 bits), and thus can keep track of the status of up to 8192 i-nodes. (Actually the first block can handle only up to 8191 i-nodes, since there is no 0th i-node, but it is given a bit in the bitmap, anyway). For 10,000 i-nodes, two bitmap blocks are needed. Since i-nodes each occupy 64 bytes, a 1-KB block holds up to 16 i-nodes. With 64 i-nodes, four disk blocks are needed to contain them all.

We will explain the difference between zones and blocks in detail later, but for the time being it is sufficient to say that disk storage can be allocated in units (zones) of 1, 2, 4, 8, or in general 2^n blocks. The zone bitmap keeps track of free storage in zones, not blocks. For all standard disks used by MINIX 3 the zone and block sizes are the same (4 KB by default), so to a first approximation a zone is the same as a block on these devices. Until we come to the details of storage allocation later in the chapter, it is adequate to think “block” whenever you see “zone.”

Note that the number of blocks per zone is not stored in the superblock, as it is never needed. All that is needed is the base 2 logarithm of the zone to block ratio, which is used as the shift count to convert zones to blocks and vice versa. For example, with 8 blocks per zone, $\log_2 8 = 3$, so to find the zone containing block 128 we shift 128 right 3 bits to get zone 16.

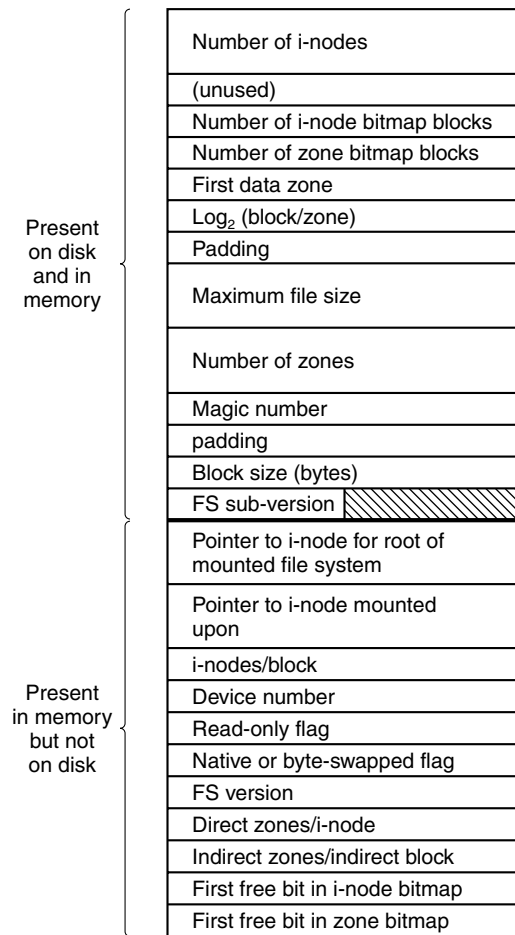


Figure 5-35. The MINIX 3 superblock.

The zone bitmap includes only the data zones (i.e., the blocks used for the bitmaps and i-nodes are not in the map), with the first data zone designated zone 1 in the bitmap. As with the i-node bitmap, bit 0 in the map is unused, so the first block in the zone bitmap can map 8191 zones and subsequent blocks can map 8192 zones each. If you examine the bitmaps on a newly formatted disk, you will find that both the i-node and zone bitmaps have 2 bits set to 1. One is for the nonexistent 0th i-node or zone; the other is for the i-node and zone used by the root directory on the device, which is placed there when the file system is created.

The information in the superblock is redundant because sometimes it is needed in one form and sometimes in another. With 1 KB devoted to the superblock, it makes sense to compute this information in all the forms it is needed, rather than having to recompute it frequently during execution. The zone number

of the first data zone on the disk, for example, can be calculated from the block size, zone size, number of i-nodes, and number of zones, but it is faster just to keep it in the superblock. The rest of the superblock is wasted anyhow, so using up another word of it costs nothing.

When MINIX 3 is booted, the superblock for the root device is read into a table in memory. Similarly, as other file systems are mounted, their superblocks are also brought into memory. The superblock table holds a number of fields not present on the disk. These include flags that allow a device to be specified as read-only or as following a byte-order convention opposite to the standard, and fields to speed access by indicating points in the bitmaps below which all bits are marked used. In addition, there is a field describing the device from which the superblock came.

Before a disk can be used as a MINIX 3 file system, it must be given the structure of Fig. 5-34. The utility program *mkfs* has been provided to build file systems. This program can be called either by a command like

```
mkfs /dev/fd1 1440
```

to build an empty 1440 block file system on the floppy disk in drive 1, or it can be given a prototype file listing directories and files to include in the new file system. This command also puts a magic number in the superblock to identify the file system as a valid MINIX file system. The MINIX file system has evolved, and some aspects of the file system (for instance, the size of i-nodes) were different previously. The magic number identifies the version of *mkfs* that created the file system, so differences can be accommodated. Attempts to mount a file system not in MINIX 3 format, such as an MS-DOS diskette, will be rejected by the mount system call, which checks the superblock for a valid magic number and other things.

5.6.3 Bitmaps

MINIX 3 keeps tracks of which i-nodes and zones are free by using two bitmaps. When a file is removed, it is then a simple matter to calculate which block of the bitmap contains the bit for the i-node being freed and to find it using the normal cache mechanism. Once the block is found, the bit corresponding to the freed i-node is set to 0. Zones are released from the zone bitmap in the same way.

Logically, when a file is to be created, the file system must search through the bit-map blocks one at a time for the first free i-node. This i-node is then allocated for the new file. In fact, the in-memory copy of the superblock has a field which points to the first free i-node, so no search is necessary until after a node is used, when the pointer must be updated to point to the new next free i-node, which will often turn out to be the next one, or a close one. Similarly, when an i-node is freed, a check is made to see if the free i-node comes before the currently-pointed-to one, and the pointer is updated if necessary. If every i-node slot on the

disk is full, the search routine returns a 0, which is why i-node 0 is not used (i.e., so it can be used to indicate the search failed). (When *mkfs* creates a new file system, it zeroes i-node 0 and sets the lowest bit in the bitmap to 1, so the file system will never attempt to allocate it.) Everything that has been said here about the i-node bitmaps also applies to the zone bitmap; logically it is searched for the first free zone when space is needed, but a pointer to the first free zone is maintained to eliminate most of the need for sequential searches through the bitmap.

With this background, we can now explain the difference between zones and blocks. The idea behind zones is to help ensure that disk blocks that belong to the same file are located on the same cylinder, to improve performance when the file is read sequentially. The approach chosen is to make it possible to allocate several blocks at a time. If, for example, the block size is 1 KB and the zone size is 4 KB, the zone bitmap keeps track of zones, not blocks. A 20-MB disk has 5K zones of 4 KB, hence 5K bits in its zone map.

Most of the file system works with blocks. Disk transfers are always a block at a time, and the buffer cache also works with individual blocks. Only a few parts of the system that keep track of physical disk addresses (e.g., the zone bitmap and the i-nodes) know about zones.

Some design decisions had to be made in developing the MINIX 3 file system. In 1985, when MINIX was conceived, disk capacities were small, and it was expected that many users would have only floppy disks. A decision was made to restrict disk addresses to 16 bits in the V1 file system, primarily to be able to store many of them in the indirect blocks. With a 16-bit zone number and a 1-KB zone, only 64-KB zones can be addressed, limiting disks to 64 MB. This was an enormous amount of storage in those days, and it was thought that as disks got larger, it would be easy to switch to 2-KB or 4-KB zones, without changing the block size. The 16-bit zone numbers also made it easy to keep the i-node size to 32 bytes.

As MINIX developed, and larger disks became much more common, it was obvious that changes were desirable. Many files are smaller than 1 KB, so increasing the block size would mean wasting disk bandwidth, reading and writing mostly empty blocks and wasting precious main memory storing them in the buffer cache. The zone size could have been increased, but a larger zone size means more wasted disk space, and it was still desirable to retain efficient operation on small disks. Another reasonable alternative would have been to have different zone sizes on large and small devices.

In the end it was decided to increase the size of disk pointers to 32 bits. This made it possible for the MINIX V2 file system to deal with device sizes up to 4 terabytes with 1-KB blocks and zones and 16 TB with 4-KB blocks and zones (the default value now). However, other factors restrict this size (e.g., with 32-bit pointers, raw devices are limited to 4 GB). Increasing the size of disk pointers required an increase in the size of i-nodes. This is not necessarily a bad thing—it means the MINIX V2 (and now, V3) i-node is compatible with standard UNIX i-

nodes, with room for three time values, more indirect and double indirect zones, and room for later expansion with triple indirect zones.

Zones also introduce an unexpected problem, best illustrated by a simple example, again with 4-KB zones and 1-KB blocks. Suppose that a file is of length 1-KB, meaning that one zone has been allocated for it. The three blocks between offsets 1024 and 4095 contain garbage (residue from the previous owner), but no structural harm is done to the file system because the file size is clearly marked in the i-node as 1 KB. In fact, the blocks containing garbage will not be read into the block cache, since reads are done by blocks, not by zones. Reads beyond the end of a file always return a count of 0 and no data.

Now someone seeks to 32,768 and writes 1 byte. The file size is now set to 32,769. Subsequent seeks to byte 1024 followed by attempts to read the data will now be able to read the previous contents of the block, a major security breach.

The solution is to check for this situation when a write is done beyond the end of a file, and explicitly zero all the not-yet-allocated blocks in the zone that was previously the last one. Although this situation rarely occurs, the code has to deal with it, making the system slightly more complex.

5.6.4 I-Nodes

The layout of the MINIX 3 i-node is given in Fig. 5-36. It is almost the same as a standard UNIX i-node. The disk zone pointers are 32-bit pointers, and there are only 9 pointers, 7 direct and 2 indirect. The MINIX 3 i-nodes occupy 64 bytes, the same as standard UNIX i-nodes, and there is space available for a 10th (triple indirect) pointer, although its use is not supported by the standard version of the FS. The MINIX 3 i-node access, modification time and i-node change times are standard, as in UNIX. The last of these is updated for almost every file operation except a read of the file.

When a file is opened, its i-node is located and brought into the *inode* table in memory, where it remains until the file is closed. The *inode* table has a few additional fields not present on the disk, such as the i-node's device and number, so the file system knows where to rewrite the i-node if it is modified while in memory. It also has a counter per i-node. If the same file is opened more than once, only one copy of the i-node is kept in memory, but the counter is incremented each time the file is opened and decremented each time the file is closed. Only when the counter finally reaches zero is the i-node removed from the table. If it has been modified since being loaded into memory, it is also rewritten to the disk.

The main function of a file's i-node is to tell where the data blocks are. The first seven zone numbers are given right in the i-node itself. For the standard distribution, with zones and blocks both 1 KB, files up to 7 KB do not need indirect blocks. Beyond 7 KB, indirect zones are needed, using the scheme of Fig. 5-10,

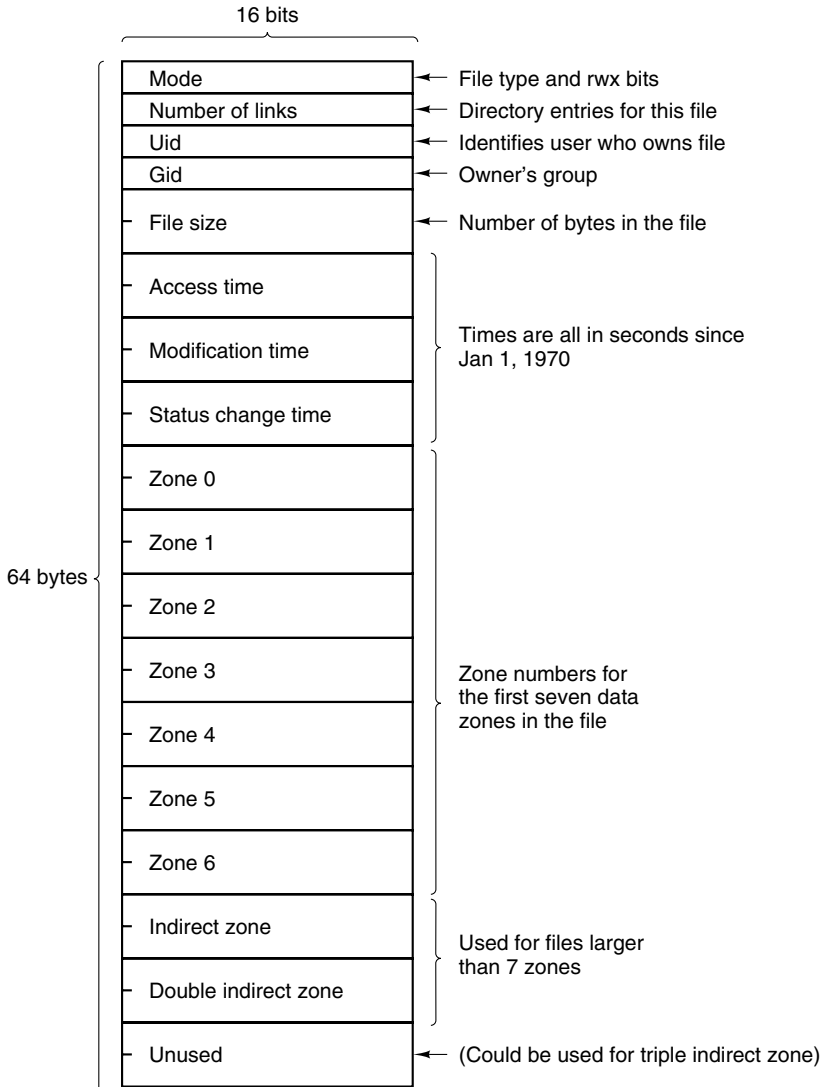


Figure 5-36. The MINIX i-node.

except that only the single and double indirect blocks are used. With 1-KB blocks and zones and 32-bit zone numbers, a single indirect block holds 256 entries, representing a quarter megabyte of storage. The double indirect block points to 256 single indirect blocks, giving access to up to 64 megabytes. With 4-KB blocks, the double indirect block leads to 1024×1024 blocks, which is over a million 4-KB blocks, making the maximum file size over 4 GB. In practice the use of 32-bit numbers as file offsets limits the maximum file size to $2^{32} - 1$ bytes. As a

consequence of these numbers, when 4-KB disk blocks are used MINIX 3 has no need for triple indirect blocks; the maximum file size is limited by the pointer size, not the ability to keep track of enough blocks.

The i-node also holds the mode information, which tells what kind of a file it is (regular, directory, block special, character special, or pipe), and gives the protection and SETUID and SETGID bits. The *link* field in the i-node records how many directory entries point to the i-node, so the file system knows when to release the file's storage. This field should not be confused with the counter (present only in the *inode* table in memory, not on the disk) that tells how many times the file is currently open, typically by different processes.

As a final note on i-nodes, we mention that the structure of Fig. 5-36 may be modified for special purposes. An example used in MINIX 3 is the i-nodes for block and character device special files. These do not need zone pointers, because they don't have to reference data areas on the disk. The major and minor device numbers are stored in the *Zone-0* space in Fig. 5-36. Another way an i-node could be used, although not implemented in MINIX 3, is as an immediate file with a small amount of data stored in the i-node itself.

5.6.5 The Block Cache

MINIX 3 uses a block cache to improve file system performance. The cache is implemented as a fixed array of buffers, each consisting of a header containing pointers, counters, and flags, and a body with room for one disk block. All the buffers that are not in use are chained together in a double-linked list, from most recently used (MRU) to least recently used (LRU) as illustrated in Fig. 5-37.

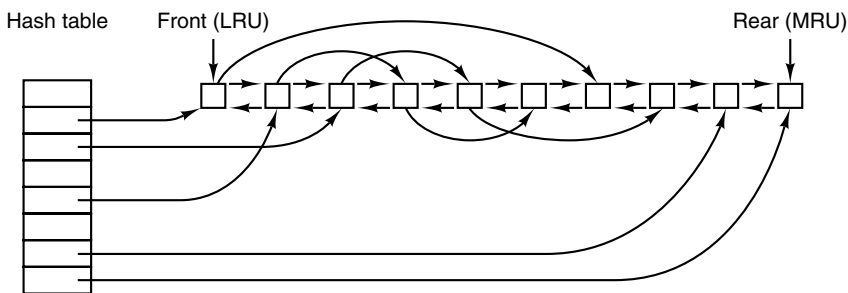


Figure 5-37. The linked lists used by the block cache.

In addition, to be able to quickly determine if a given block is in the cache or not, a hash table is used. All the buffers containing a block that has hash code k are linked together on a single-linked list pointed to by entry k in the hash table. The hash function just extracts the low-order n bits from the block number, so

blocks from different devices appear on the same hash chain. Every buffer is on one of these chains. When the file system is initialized after MINIX 3 is booted, all buffers are unused, of course, and all are in a single chain pointed to by the 0th hash table entry. At that time all the other hash table entries contain a null pointer, but once the system starts, buffers will be removed from the 0th chain and other chains will be built.

When the file system needs to acquire a block, it calls a procedure, *get_block*, which computes the hash code for that block and searches the appropriate list. *Get_block* is called with a device number as well as a block number, and the search compares both numbers with the corresponding fields in the buffer chain. If a buffer containing the block is found, a counter in the buffer header is incremented to show that the block is in use, and a pointer to it is returned. If a block is not found on the hash list, the first buffer on the LRU list can be used; it is guaranteed not to be still in use, and the block it contains may be evicted to free up the buffer.

Once a block has been chosen for eviction from the block cache, another flag in its header is checked to see if the block has been modified since being read in. If so, it is rewritten to the disk. At this point the block needed is read in by sending a message to the disk driver. The file system is suspended until the block arrives, at which time it continues and a pointer to the block is returned to the caller.

When the procedure that requested the block has completed its job, it calls another procedure, *put_block*, to free the block. Normally, a block will be used immediately and then released, but since it is possible that additional requests for a block will be made before it has been released, *put_block* decrements the use counter and puts the buffer back onto the LRU list only when the use counter has gone back to zero. While the counter is nonzero, the block remains in limbo.

One of the parameters to *put_block* tells what class of block (e.g., i-nodes, directory, data) is being freed. Depending on the class, two key decisions are made:

1. Whether to put the block on the front or rear of the LRU list.
2. Whether to write the block (if modified) to disk immediately or not.

Almost all blocks go on the rear of the list in true LRU fashion. The exception is blocks from the RAM disk; since they are already in memory there is little advantage to keeping them in the block cache.

A modified block is not rewritten until either one of two events occurs:

1. It reaches the front of the LRU chain and is evicted.
2. A sync system call is executed.

Sync does not traverse the LRU chain but instead indexes through the array of

buffers in the cache. Even if a buffer has not been released yet, if it has been modified, `sync` will find it and ensure that the copy on disk is updated.

Policies like this invite tinkering. In an older version of MINIX a superblock was modified when a file system was mounted, and was always rewritten immediately to reduce the chance of corrupting the file system in the event of a crash. Superblocks are modified only if the size of a RAM disk must be adjusted at startup time because the RAM disk was created bigger than the RAM image device. However, the superblock is not read or written as a normal block, because it is always 1024 bytes in size, like the boot block, regardless of the block size used for blocks handled by the cache. Another abandoned experiment is that in older versions of MINIX there was a *ROBUST* macro definable in the system configuration file, *include/minix/config.h*, which, if defined, caused the file system to mark i-node, directory, indirect, and bit-map blocks to be written immediately upon release. This was intended to make the file system more robust; the price paid was slower operation. It turned out this was not effective. A power failure occurring when all blocks have not been yet been written is going to cause a headache whether it is an i-node or a data block that is lost.

Note that the header flag indicating that a block has been modified is set by the procedure within the file system that requested and used the block. The procedures *get_block* and *put_block* are concerned just with manipulating the linked lists. They have no idea which file system procedure wants which block or why.

5.6.6 Directories and Paths

Another important subsystem within the file system manages directories and path names. Many system calls, such as `open`, have a file name as a parameter. What is really needed is the i-node for that file, so it is up to the file system to look up the file in the directory tree and locate its i-node.

A MINIX directory is a file that in previous versions contained 16-byte entries, 2 bytes for an i-node number and 14 bytes for the file name. This design limited disk partitions to 64-KB files and file names to 14 characters, the same as V7 UNIX. As disks have grown file names have also grown. In MINIX 3 the V3 file system provides 64 bytes directory entries, with 4 bytes for the i-node number and 60 bytes for the file name. Having up to 4 billion files per disk partition is effectively infinite and any programmer choosing a file name longer than 60 characters should be sent back to programming school.

Note that *paths* such as

/usr/ast/course_material_for_this_year/operating_systems/examination-1.ps

are not limited to 60 characters—just the individual component names. The use of fixed-length directory entries, in this case, 64 bytes, is an example of a trade-off involving simplicity, speed, and storage. Other operating systems typically

organize directories as a heap, with a fixed header for each file pointing to a name on the heap at the end of the directory. The MINIX 3 scheme is very simple and required practically no code changes from V2. It is also very fast for both looking up names and storing new ones, since no heap management is ever required. The price paid is wasted disk storage, because most files are much shorter than 60 characters.

It is our firm belief that optimizing to save disk storage (and some RAM storage since directories are occasionally in memory) is the wrong choice. Code simplicity and correctness should come first and speed should come second. With modern disks usually exceeding 100 GB, saving a small amount of disk space at the price of more complicated and slower code is generally not a good idea. Unfortunately, many programmers grew up in an era of tiny disks and even tinier RAMs, and were trained from day 1 to resolve all trade-offs between code complexity, speed, and space in favor of minimizing space requirements. This implicit assumption really has to be reexamined in light of current realities.

Now let us see how the path `/usr/ast/mbox/` is looked up. The system first looks up `usr` in the root directory, then it looks up `ast` in `/usr/`, and finally it looks up `mbox` in `/usr/ast/`. The actual lookup proceeds one path component at a time, as illustrated in Fig. 5-16.

The only complication is what happens when a mounted file system is encountered. The usual configuration for MINIX 3 and many other UNIX-like systems is to have a small root file system containing the files needed to start the system and to do basic system maintenance, and to have the majority of the files, including users' directories, on a separate device mounted on `/usr`. This is a good time to look at how mounting is done. When the user types the command

```
mount /dev/c0d1p2 /usr
```

on the terminal, the file system contained on hard disk 1, partition 2 is mounted on top of `/usr/` in the root file system. The file systems before and after mounting are shown in Fig. 5-38.

The key to the whole mount business is a flag set in the memory copy of the i-node of `/usr` after a successful mount. This flag indicates that the i-node is mounted on. The mount call also loads the superblock for the newly mounted file system into the *super_block* table and sets two pointers in it. Furthermore, it puts the root i-node of the mounted file system in the *inode* table.

In Fig. 5-35 we see that superblocks in memory contain two fields related to mounted file systems. The first of these, the *i-node-for-root-of-mounted-file-system*, is set to point to the root i-node of the newly mounted file system. The second, the *i-node-mounted-upon*, is set to point to the i-node mounted on, in this case, the i-node for `/usr`. These two pointers serve to connect the mounted file system to the root and represent the “glue” that holds the mounted file system to the root [shown as the dots in Fig. 5-38(c)]. This glue is what makes mounted file systems work.

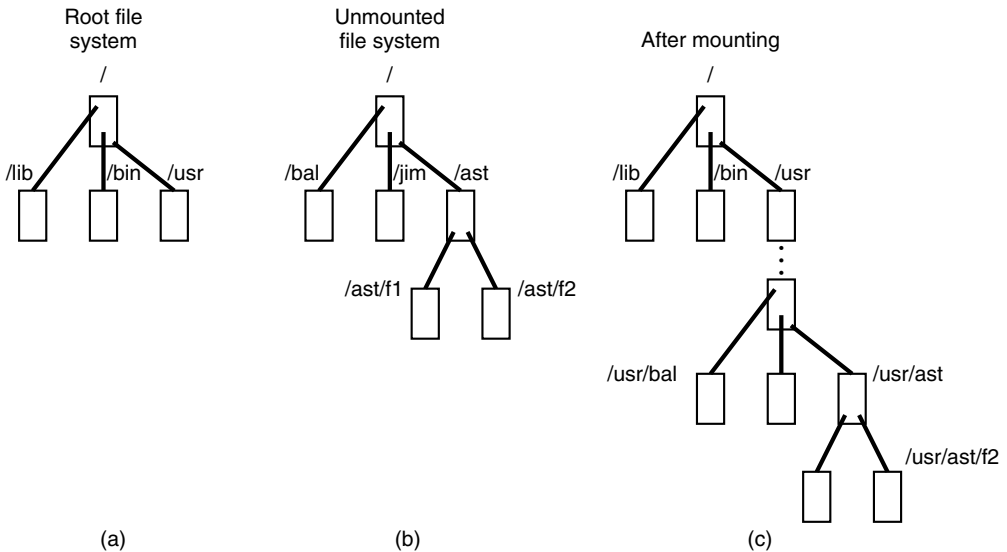


Figure 5-38. (a) Root file system. (b) An unmounted file system. (c) The result of mounting the file system of (b) on `/usr/`.

When a path such as `/usr/ast/f2` is being looked up, the file system will see a flag in the i-node for `/usr/` and realize that it must continue searching at the root i-node of the file system mounted on `/usr/`. The question is: “How does it find this root i-node?”

The answer is straightforward. The system searches all the superblocks in memory until it finds the one whose *i-node mounted on* field points to `/usr/`. This must be the superblock for the file system mounted on `/usr/`. Once it has the superblock, it is easy to follow the other pointer to find the root i-node for the mounted file system. Now the file system can continue searching. In this example, it looks for `ast` in the root directory of hard disk partition 2.

5.6.7 File Descriptors

Once a file has been opened, a file descriptor is returned to the user process for use in subsequent read and write calls. In this section we will look at how file descriptors are managed within the file system.

Like the kernel and the process manager, the file system maintains part of the process table within its address space. Three of its fields are of particular interest. The first two are pointers to the i-nodes for the root directory and the working directory. Path searches, such as that of Fig. 5-16, always begin at one or the other, depending on whether the path is absolute or relative. These pointers are

changed by the `chroot` and `chdir` system calls to point to the new root or new working directory, respectively.

The third interesting field in the process table is an array indexed by file descriptor number. It is used to locate the proper file when a file descriptor is presented. At first glance, it might seem sufficient to have the k -th entry in this array just point to the i -node for the file belonging to file descriptor k . After all, the i -node is fetched into memory when the file is opened and kept there until it is closed, so it is sure to be available.

Unfortunately, this simple plan fails because files can be shared in subtle ways in MINIX 3 (as well as in UNIX). The trouble arises because associated with each file is a 32-bit number that indicates the next byte to be read or written. It is this number, called the **file position**, that is changed by the `lseek` system call. The problem can be stated easily: “Where should the file pointer be stored?”

The first possibility is to put it in the i -node. Unfortunately, if two or more processes have the same file open at the same time, they must all have their own file pointers, since it would hardly do to have an `lseek` by one process affect the next read of a different process. Conclusion: the file position cannot go in the i -node.

What about putting it in the process table? Why not have a second array, paralleling the file descriptor array, giving the current position of each file? This idea does not work either, but the reasoning is more subtle. Basically, the trouble comes from the semantics of the `fork` system call. When a process forks, both the parent and the child are required to share a single pointer giving the current position of each open file.

To better understand the problem, consider the case of a shell script whose output has been redirected to a file. When the shell forks off the first program, its file position for standard output is 0. This position is then inherited by the child, which writes, say, 1 KB of output. When the child terminates, the shared file position must now be 1024.

Now the shell reads some more of the shell script and forks off another child. It is essential that the second child inherit a file position of 1024 from the shell, so it will begin writing at the place where the first program left off. If the shell did not share the file position with its children, the second program would overwrite the output from the first one, instead of appending to it.

As a result, it is not possible to put the file position in the process table. It really must be shared. The solution used in UNIX and MINIX 3 is to introduce a new, shared table, *filp*, which contains all the file positions. Its use is illustrated in Fig. 5-39. By having the file position truly shared, the semantics of `fork` can be implemented correctly, and shell scripts work properly.

Although the only thing that the *filp* table really must contain is the shared file position, it is convenient to put the i -node pointer there, too. In this way, all that the file descriptor array in the process table contains is a pointer to a *filp* entry. The *filp* entry also contains the file mode (permission bits), some flags indicating

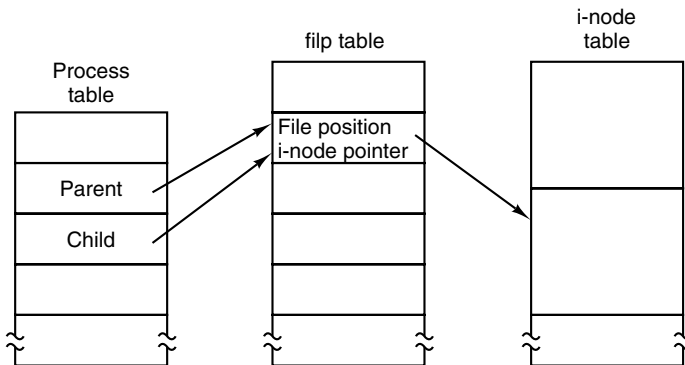


Figure 5-39. How file positions are shared between a parent and a child.

whether the file was opened in a special mode, and a count of the number of processes using it, so the file system can tell when the last process using the entry has terminated, in order to reclaim the slot.

5.6.8 File Locking

Yet another aspect of file system management requires a special table. This is file locking. MINIX 3 supports the POSIX interprocess communication mechanism of **advisory file locking**. This permits any part, or multiple parts, of a file to be marked as locked. The operating system does not enforce locking, but processes are expected to be well behaved and to look for locks on a file before doing anything that would conflict with another process.

The reasons for providing a separate table for locks are similar to the justifications for the *filp* table discussed in the previous section. A single process can have more than one lock active, and different parts of a file may be locked by more than one process (although, of course, the locks cannot overlap), so neither the process table nor the *filp* table is a good place to record locks. Since a file may have more than one lock placed upon it, the i-node is not a good place either.

MINIX 3 uses another table, the *file_lock* table, to record all locks. Each slot in this table has space for a lock type, indicating if the file is locked for reading or writing, the process ID holding the lock, a pointer to the i-node of the locked file, and the offsets of the first and last bytes of the locked region.

5.6.9 Pipes and Special Files

Pipes and special files differ from ordinary files in an important way. When a process tries to read or write a block of data from a disk file, it is almost certain that the operation will complete within a few hundred milliseconds at most. In the worst case, two or three disk accesses might be needed, not more. When reading

from a pipe, the situation is different: if the pipe is empty, the reader will have to wait until some other process puts data in the pipe, which might take hours. Similarly, when reading from a terminal, a process will have to wait until somebody types something.

As a consequence, the file system's normal rule of handling a request until it is finished does not work. It is necessary to suspend these requests and restart them later. When a process tries to read or write from a pipe, the file system can check the state of the pipe immediately to see if the operation can be completed. If it can be, it is, but if it cannot be, the file system records the parameters of the system call in the process table, so it can restart the process when the time comes.

Note that the file system need not take any action to have the caller suspended. All it has to do is refrain from sending a reply, leaving the caller blocked waiting for the reply. Thus, after suspending a process, the file system goes back to its main loop to wait for the next system call. As soon as another process modifies the pipe's state so that the suspended process can complete, the file system sets a flag so that next time through the main loop it extracts the suspended process' parameters from the process table and executes the call.

The situation with terminals and other character special files is slightly different. The i-node for each special file contains two numbers, the major device and the minor device. The major device number indicates the device class (e.g., RAM disk, floppy disk, hard disk, terminal). It is used as an index into a file system table that maps it onto the number of the corresponding I/O device driver. In effect, the major device determines which I/O driver to call. The minor device number is passed to the driver as a parameter. It specifies which device is to be used, for example, terminal 2 or drive 1.

In some cases, most notably terminal devices, the minor device number encodes some information about a category of devices handled by a driver. For instance, the primary MINIX 3 console, */dev/console*, is device 4, 0 (major, minor). Virtual consoles are handled by the same part of the driver software. These are devices */dev/ttyc1* (4,1), */dev/ttyc2* (4,2), and so on. Serial line terminals need different low-level software, and these devices, */dev/tty00*, and */dev/tty01* are assigned device numbers 4, 16 and 4, 17. Similarly, network terminals use pseudo-terminal drivers, and these also need different low-level software. In MINIX 3 these devices, *ttyp0*, *ttyp1*, etc., are assigned device numbers such as 4, 128 and 4, 129. These pseudo devices each have an associated device, *ptyp0*, *ptyp1*, etc. The major, minor device number pairs for these are 4,192 and 4,193, etc. These numbers are chosen to make it easy for the device driver to call the low-level functions required for each group of devices. It is not expected that anyone is going to equip a MINIX 3 system with 192 or more terminals.

When a process reads from a special file, the file system extracts the major and minor device numbers from the file's i-node, and uses the major device number as an index into a file system table to map it onto the process number of the corresponding device driver. Once it has identified the driver, the file system

sends it a message, including as parameters the minor device, the operation to be performed, the caller's process number and buffer address, and the number of bytes to be transferred. The format is the same as in Fig. 3-15, except that *POSITION* is not used.

If the driver is able to carry out the work immediately (e.g., a line of input has already been typed on the terminal), it copies the data from its own internal buffers to the user and sends the file system a reply message saying that the work is done. The file system then sends a reply message to the user, and the call is finished. Note that the driver does not copy the data to the file system. Data from block devices go through the block cache, but data from character special files do not.

On the other hand, if the driver is not able to carry out the work, it records the message parameters in its internal tables, and immediately sends a reply to the file system saying that the call could not be completed. At this point, the file system is in the same situation as having discovered that someone is trying to read from an empty pipe. It records the fact that the process is suspended and waits for the next message.

When the driver has acquired enough data to complete the call, it transfers them to the buffer of the still-blocked user and then sends the file system a message reporting what it has done. All the file system has to do is send a reply message to the user to unblock it and report the number of bytes transferred.

5.6.10 An Example: The *READ* System Call

As we shall see shortly, most of the code of the file system is devoted to carrying out system calls. Therefore, it is appropriate that we conclude this overview with a brief sketch of how the most important call, *read*, works.

When a user program executes the statement

```
n = read(fd, buffer, nbytes);
```

to read an ordinary file, the library procedure *read* is called with three parameters. It builds a message containing these parameters, along with the code for *read* as the message type, sends the message to the file system, and blocks waiting for the reply. When the message arrives, the file system uses the message type as an index into its tables to call the procedure that handles reading.

This procedure extracts the file descriptor from the message and uses it to locate the *filp* entry and then the i-node for the file to be read (see Fig. 5-39). The request is then broken up into pieces such that each piece fits within a block. For example, if the current file position is 600 and 1024 bytes have been requested, the request is split into two parts, for 600 to 1023, and for 1024 to 1623 (assuming 1-KB blocks).

For each of these pieces in turn, a check is made to see if the relevant block is in the cache. If the block is not present, the file system picks the least recently

used buffer not currently in use and claims it, sending a message to the disk device driver to rewrite it if it is dirty. Then the disk driver is asked to fetch the block to be read.

Once the block is in the cache, the file system sends a message to the system task asking it to copy the data to the appropriate place in the user's buffer (i.e., bytes 600 to 1023 to the start of the buffer, and bytes 1024 to 1623 to offset 424 within the buffer). After the copy has been done, the file system sends a reply message to the user specifying how many bytes have been copied.

When the reply comes back to the user, the library function *read* extracts the reply code and returns it as the function value to the caller.

One extra step is not really part of the *read* call itself. After the file system completes a read and sends a reply, it initiates reading additional blocks, provided that the read is from a block device and certain other conditions are met. Since sequential file reads are common, it is reasonable to expect that the next blocks in a file will be requested in the next read request, and this makes it likely that the desired block will already be in the cache when it is needed. The number of blocks requested depends upon the size of the block cache; as many as 32 additional blocks may be requested. The device driver does not necessarily return this many blocks, and if at least one block is returned a request is considered successful.

5.7 IMPLEMENTATION OF THE MINIX 3 FILE SYSTEM

The MINIX 3 file system is relatively large (more than 100 pages of C) but quite straightforward. Requests to carry out system calls come in, are carried out, and replies are sent. In the following sections we will go through it a file at a time, pointing out the highlights. The code itself contains many comments to aid the reader.

In looking at the code for other parts of MINIX 3 we have generally looked at the main loop of a process first and then looked at the routines that handle the different message types. We will organize our approach to the file system differently. First we will go through the major subsystems (cache management, i-node management, etc.). Then we will look at the main loop and the system calls that operate upon files. Next we will look at systems call that operate upon directories, and then, we will discuss the remaining system calls that fall into neither category. Finally we will see how device special files are handled.

5.7.1 Header Files and Global Data Structures

Like the kernel and process manager, various data structures and tables used in the file system are defined in header files. Some of these data structures are placed in system-wide header files in *include/* and its subdirectories. For instance,

include/sys/stat.h defines the format by which system calls can provide i-node information to other programs and the structure of a directory entry is defined in *include/sys/dir.h*. Both of these files are required by POSIX. The file system is affected by a number of definitions contained in the global configuration file *include/minix/config.h*, such as *NR_BUFS* and *NR_BUF_HASH*, which control the size of the block cache.

File System Headers

The file system's own header files are in the file system source directory *src/fs/*. Many file names will be familiar from studying other parts of the MINIX 3 system. The FS master header file, *fs.h* (line 20900), is quite analogous to *src/kernel/kernel.h* and *src/pm/pm.h*. It includes other header files needed by all the C source files in the file system. As in the other parts of MINIX 3, the file system master header includes the file system's own *const.h*, *type.h*, *proto.h*, and *glo.h*. We will look at these next.

Const.h (line 21000) defines some constants, such as table sizes and flags, that are used throughout the file system. MINIX 3 already has a history. Earlier versions of MINIX had different file systems. Although MINIX 3 does not support the old V1 and V2 file systems, some definitions have been retained, both for reference and in expectation that someone will add support for these later. Support for older versions is useful not only for accessing files on older MINIX file systems, it may also be useful for exchanging files.

Other operating systems may use older MINIX file systems—for instance, Linux originally used and still supports MINIX file systems. (It is perhaps somewhat ironic that Linux still supports the original MINIX file system but MINIX 3 does not.) Some utilities are available for MS-DOS and Windows to access older MINIX directories and files. The superblock of a file system contains a **magic number** to allow the operating system to identify the file system's type; the constants *SUPER_MAGIC*, *SUPER_V2*, and *SUPER_V3* define these numbers for the three versions of the MINIX file system. There are also *_REV*-suffixed versions of these for V1 and V2, in which the bytes of the magic number are reversed. These were used with ports of older MINIX versions to systems with a different byte order (little-endian rather than big-endian) so a removable disk written on a machine with a different byte order could be identified as such. As of the release of MINIX 3.1.0 defining a *SUPER_V3_REV* magic number has not been necessary, but it is likely this definition will be added in the future.

Type.h (line 21100) defines both the old V1 and new V2 i-node structures as they are laid out on the disk. The i-node is one structure that did not change in MINIX 3, so the V2 i-node is used with the V-3 file system. The V2 i-node is twice as big as the old one, which was designed for compactness on systems with no hard drive and 360-KB diskettes. The new version provides space for the three time fields which UNIX systems provide. In the V1 i-node there was only one

time field, but a `stat` or `fstat` would “fake it” and return a `stat` structure containing all three fields. There is a minor difficulty in providing support for the two file system versions. This is flagged by the comment on line 21116. Older MINIX 3 software expected the `gid_t` type to be an 8-bit quantity, so `d2_gid` must be declared as type `u16_t`.

Proto.h (line 21200) provides function prototypes in forms acceptable to either old K&R or newer ANSI Standard C compilers. It is a long file, but not of great interest. However, there is one point to note: because there are so many different system calls handled by the file system, and because of the way the file system is organized, the various `do_XXX` functions are scattered through a number of files. *Proto.h* is organized by file and is a handy way to find the file to consult when you want to see the code that handles a particular system call.

Finally, *glo.h* (line 21400) defines global variables. The message buffers for the incoming and reply messages are also here. The now-familiar trick with the *EXTERN* macro is used, so these variables can be accessed by all parts of the file system. As in the other parts of MINIX 3, the storage space will be reserved when *table.c* is compiled.

The file system’s part of the process table is contained in *fproc.h* (line 21500). The *fproc* array is declared with the *EXTERN* macro. It holds the mode mask, pointers to the i-nodes for the current root directory and working directory, the file descriptor array, uid, gid, and terminal number for each process. The process id and the process group id are also found here. The process id is duplicated in the part of the process table located in the process manager.

Several fields are used to store the parameters of those system calls that may be suspended part way through, such as reads from an empty pipe. The fields *fp_suspended* and *fp_revived* actually require only single bits, but nearly all compilers generate better code for characters than bit fields. There is also a field for the *FD_CLOEXEC* bits called for by the POSIX standard. These are used to indicate that a file should be closed when an `exec` call is made.

Now we come to files that define other tables maintained by the file system. The first, *buf.h* (line 21600), defines the block cache. The structures here are all declared with *EXTERN*. The array *buf* holds all the buffers, each of which contains a data part, *b*, and a header full of pointers, flags, and counters. The data part is declared as a union of five types (lines 21618 to 21632) because sometimes it is convenient to refer to the block as a character array, sometimes as a directory, etc.

The truly proper way to refer to the data part of buffer 3 as a character array is *buf[3].b.b_ _data* because *buf[3].b* refers to the union as a whole, from which the *b_ _data* field is selected. Although this syntax is correct, it is cumbersome, so on line 21649 we define a macro *b_ _data*, which allows us to write *buf[3].b_ _data* instead. Note that *b_ _data* (the field of the union) contains two underscores, whereas *b_ _data* (the macro) contains just one, to distinguish them. Macros for other ways of accessing the block are defined on lines 21650 to 21655.

The buffer hash table, *buf_hash*, is defined on line 21657. Each entry points to a list of buffers. Originally all the lists are empty. Macros at the end of *buf.h* define different block types. The *WRITE_IMMED* bit signals that a block must be rewritten to the disk immediately if it is changed, and the *ONE_SHOT* bit is used to indicate a block is unlikely to be needed soon. Neither of these is used currently but they remain available for anyone who has a bright idea about improving performance or reliability by modifying the way blocks in the cache are queued.

Finally, in the last line *HASH_MASK* is defined, based upon the value of *NR_BUF_HASH* configured in *include/minix/config.h*. *HASH_MASK* is ANDed with a block number to determine which entry in *buf_hash* to use as the starting point in a search for a block buffer.

File.h (line 21700) contains the intermediate table *filp* (declared as *EXTERN*), used to hold the current file position and i-node pointer (see Fig. 5-39). It also tells whether the file was opened for reading, writing, or both, and how many file descriptors are currently pointing to the entry.

The file locking table, *file_lock* (declared as *EXTERN*), is in *lock.h* (line 21800). The size of the array is determined by *NR_LOCKS*, which is defined as 8 in *const.h*. This number should be increased if it is desired to implement a multi-user data base on a MINIX 3 system.

In *inode.h* (line 21900) the i-node table *inode* is declared (using *EXTERN*). It holds i-nodes that are currently in use. As we said earlier, when a file is opened its i-node is read into memory and kept there until the file is closed. The *inode* structure definition provides for information that is kept in memory, but is not written to the disk i-node. Notice that there is only one version, and nothing is version-specific here. When the i-node is read in from the disk, differences between V1 and V2/V3 file systems are handled. The rest of the file system does not need to know about the file system format on the disk, at least until the time comes to write back modified information.

Most of the fields should be self-explanatory at this point. However, *i_seek* deserves some comment. It was mentioned earlier that, as an optimization, when the file system notices that a file is being read sequentially, it tries to read blocks into the cache even before they are asked for. For randomly accessed files there is no read ahead. When an *lseek* call is made, the field *i_seek* is set to inhibit read ahead.

The file *param.h* (line 22000) is analogous to the file of the same name in the process manager. It defines names for message fields containing parameters, so the code can refer to, for example, *m_in.buffer*, instead of *m_in.m1_p1*, which selects one of the fields of the message buffer *m_in*.

In *super.h* (line 22100), we have the declaration of the superblock table. When the system is booted, the superblock for the root device is loaded here. As file systems are mounted, their superblocks go here as well. As with other tables, *super_block* is declared as *EXTERN*.

File System Storage Allocation

The last file we will discuss in this section is not a header. However, just as we did when discussing the process manager, it seems appropriate to discuss *table.c* immediately after reviewing the header files, since they are all included when *table.c* (line 22200) is compiled. Most of the data structures we have mentioned—the block cache, the *filp* table, and so on—are defined with the *EXTERN* macro, as are also the file system's global variables and the file system's part of the process table. In the same way we have seen in other parts of the MINIX 3 system, the storage is actually reserved when *table.c* is compiled. This file also contains one major initialized array. *Call_vector* contains the pointer array used in the main loop for determining which procedure handles which system call number. We saw a similar table inside the process manager.

5.7.2 Table Management

Associated with each of the main tables—blocks, i-nodes, superblocks, and so forth—is a file that contains procedures that manage the table. These procedures are heavily used by the rest of the file system and form the principal interface between tables and the file system. For this reason, it is appropriate to begin our study of the file system code with them.

Block Management

The block cache is managed by the procedures in the file *cache.c*. This file contains the nine procedures listed in Fig. 5-40. The first one, *get_block* (line 22426), is the standard way the file system gets data blocks. When a file system procedure needs to read a user data block, a directory block, a superblock, or any other kind of block, it calls *get_block*, specifying the device and block number.

When *get_block* is called, it first examines the block cache to see if the requested block is there. If so, it returns a pointer to it. Otherwise, it has to read the block in. The blocks in the cache are linked together on *NR_BUF_HASH* linked lists. *NR_BUF_HASH* is a tunable parameter, along with *NR_BUFS*, the size of the block cache. Both of these are set in *include/minix/config.h*. At the end of this section we will say a few words about optimizing the size of the block cache and the hash table. The *HASH_MASK* is *NR_BUF_HASH* - 1. With 256 hash lists, the mask is 255, so all the blocks on each list have block numbers that end with the same string of 8 bits, that is 00000000, 00000001, ..., or 11111111.

The first step is usually to search a hash chain for a block, although there is a special case, when a hole in a sparse file is being read, where this search is skipped. This is the reason for the test on line 22454. Otherwise, the next two

Procedure	Function
<code>get_block</code>	Fetch a block for reading or writing
<code>put_block</code>	Return a block previously requested with <code>get_block</code>
<code>alloc_zone</code>	Allocate a new zone (to make a file longer)
<code>free_zone</code>	Release a zone (when a file is removed)
<code>rw_block</code>	Transfer a block between disk and cache
<code>invalidate</code>	Purge all the cache blocks for some device
<code>flushall</code>	Flush all dirty blocks for one device
<code>rw_scattered</code>	Read or write scattered data from or to a device
<code>rm_lru</code>	Remove a block from its LRU chain

Figure 5-40. Procedures used for block management.

lines set *bp* to point to the start of the list on which the requested block would be, if it were in the cache, applying *HASH_MASK* to the block number. The loop on the next line searches this list to see if the block can be found. If it is found and is not in use, it is removed from the LRU list. If it is already in use, it is not on the LRU list anyway. The pointer to the found block is returned to the caller on line 22463.

If the block is not on the hash list, it is not in the cache, so the least recently used block from the LRU list is taken. The buffer chosen is removed from its hash chain, since it is about to acquire a new block number and hence belongs on a different hash chain. If it is dirty, it is rewritten to the disk on line 22495. Doing this with a call to *flushall* rewrites any other dirty blocks for the same device. This call is the way most blocks get written. Blocks that are currently in use are never chosen for eviction, since they are not on the LRU chain. Blocks will hardly ever be found to be in use, however; normally a block is released by *put_block* immediately upon being used.

As soon as the buffer is available, all of the fields, including *b_dev*, are updated with the new parameters (lines 22499 to 22504), and the block may be read in from the disk. However, there are two occasions when it may not be necessary to read the block from the disk. *Get_block* is called with a parameter *only_search*. This may indicate that this is a prefetch. During a prefetch an available buffer is found, writing the old contents to the disk if necessary, and a new block number is assigned to the buffer, but the *b_dev* field is set to *NO_DEV* to signal there are as yet no valid data in this block. We will see how this is used when we discuss the *rw_scattered* function. *Only_search* can also be used to signal that the file system needs a block just to rewrite all of it. In this case it is wasteful to first read the old version in. In either of these cases the parameters are updated, but the actual disk read is omitted (lines 22507 to 22513). When the new block has been read in, *get_block* returns to its caller with a pointer to it.

Suppose that the file system needs a directory block temporarily, to look up a file name. It calls *get_block* to acquire the directory block. When it has looked up its file name, it calls *put_block* (line 22520) to return the block to the cache, thus making the buffer available in case it is needed later for a different block.

Put_block takes care of putting the newly returned block on the LRU list, and in some cases, rewriting it to the disk. At line 22544 a decision is made to put it on the front or rear of the LRU list. Blocks on a RAM disk are always put on the front of the queue. The block cache does not really do very much for a RAM disk, since its data are already in memory and accessible without actual I/O. The *ONE_SHOT* flag is tested to see if the block has been marked as one not likely to be needed again soon, and such blocks are put on the front, where they will be reused quickly. However, this is used rarely, if at all. Almost all blocks except those from the RAM disk are put on the rear, in case they are needed again soon.

After the block has been repositioned on the LRU list, another check is made to see if the block should be rewritten to disk immediately. Like the previous test, the test for *WRITE_IMMED* is a vestige of an abandoned experiment; currently no blocks are marked for immediate writing.

As a file grows, from time to time a new zone must be allocated to hold the new data. The procedure *alloc_zone* (line 22580) takes care of allocating new zones. It does this by finding a free zone in the zone bitmap. There is no need to search through the bitmap if this is to be the first zone in a file; the *s_zsearch* field in the superblock, which always points to the first available zone on the device, is consulted. Otherwise an attempt is made to find a zone close to the last existing zone of the current file, in order to keep the zones of a file together. This is done by starting the search of the bitmap at this last zone (line 22603). The mapping between the bit number in the bitmap and the zone number is handled on line 22615, with bit 1 corresponding to the first data zone.

When a file is removed, its zones must be returned to the bitmap. *Free_zone* (line 22621) is responsible for returning these zones. All it does is call *free_bit*, passing the zone map and the bit number as parameters. *Free_bit* is also used to return free i-nodes, but then with the i-node map as the first parameter, of course.

Managing the cache requires reading and writing blocks. To provide a simple disk interface, the procedure *rw_block* (line 22641) has been provided. It reads or writes one block. Analogously, *rw_inode* exists to read and write i-nodes.

The next procedure in the file is *invalidate* (line 22680). It is called when a disk is unmounted, for example, to remove from the cache all the blocks belonging to the file system just unmounted. If this were not done, then when the device were reused (with a different floppy disk), the file system might find the old blocks instead of the new ones.

We mentioned earlier that *flushall* (line 22694), called from *get_block* whenever a dirty block is removed from the LRU list, is the function responsible for writing most data. It is also called by the sync system call to flush to disk all dirty buffers belonging to a specific device. Sync is activated periodically by the

update daemon, and calls *flushall* once for each mounted device. *Flushall* treats the buffer cache as a linear array, so all dirty buffers are found, even ones that are currently in use and are not in the LRU list. All buffers in the cache are scanned, and those that belong to the device to be flushed and that need to be written are added to an array of pointers, *dirty*. This array is declared as *static* to keep it off the stack. It is then passed to *rw_scattered*.

In MINIX 3 scheduling of disk writing has been removed from the disk device drivers and made the sole responsibility of *rw_scattered* (line 22711). This function receives a device identifier, a pointer to an array of pointers to buffers, the size of the array, and a flag indicating whether to read or write. The first thing it does is sort the array it receives on the block numbers, so the actual read or write operation will be performed in an efficient order. It then constructs vectors of contiguous blocks to send to the device driver with a call to *dev_io*. The driver does not have to do any additional scheduling. It is likely with a modern disk that the drive electronics will further optimize the order of requests, but this is not visible to MINIX 3. *Rw_scattered* is called with the *WRITING* flag only from the *flushall* function described above. In this case the origin of these block numbers is easy to understand. They are buffers which contain data from blocks previously read but now modified. The only call to *rw_scattered* for a read operation is from *rahead* in *read.c*. At this point, we just need to know that before calling *rw_scattered*, *get_block* has been called repeatedly in prefetch mode, thus reserving a group of buffers. These buffers contain block numbers, but no valid device parameter. This is not a problem, since *rw_scattered* is called with a device parameter as one of its arguments.

There is an important difference in the way a device driver may respond to a read (as opposed to a write) request, from *rw_scattered*. A request to write a number of blocks *must* be honored completely, but a request to read a number of blocks may be handled differently by different drivers, depending upon what is most efficient for the particular driver. *Rahead* often calls *rw_scattered* with a request for a list of blocks that may not actually be needed, so the best response is to get as many blocks as can be gotten easily, but not to go wildly seeking all over a device that may have a substantial seek time. For instance, the floppy driver may stop at a track boundary, and many other drivers will read only consecutive blocks. When the read is complete, *rw_scattered* marks the blocks read by filling in the device number field in their block buffers.

The last function in Fig. 5-40 is *rm_lru* (line 22809). This function is used to remove a block from the LRU list. It is used only by *get_block* in this file, so it is declared *PRIVATE* instead of *PUBLIC* to hide it from procedures outside the file.

Before we leave the block cache, let us say a few words about fine-tuning it. *NR_BUF_HASH* must be a power of 2. If it is larger than *NR_BUFS*, the average length of a hash chain will be less than one. If there is enough memory for a large number of buffers, there is space for a large number of hash chains, so the usual choice is to make *NR_BUF_HASH* the next power of 2 greater than

NR_BUFS. The listing in the text shows settings of 128 blocks and 128 hash lists. The optimal size depends upon how the system is used, since that determines how much must be buffered. The full source code used to compile the standard MINIX 3 binaries that are installed from the CD-ROM that accompanies this text has settings of 1280 buffers and 2048 hash chains. Empirically it was found that increasing the number of buffers beyond this did not improve performance when recompiling the MINIX 3 system, so apparently this is large enough to hold the binaries for all compiler passes. For some other kind of work a smaller size might be adequate or a larger size might improve performance.

The buffers for the standard MINIX 3 system on the CD-ROM occupy more than 5 MB of RAM. An additional binary, designated *image_small* is provided that was compiled with just 128 buffers in the block cache, and the buffers for this system need only a little more than 0.5 MB. This one can be installed on a system with only 8 MB of RAM. The standard version requires 16 MB of RAM. With some tweaking, it could no doubt be shoehorned into a memory of 4 MB or smaller.

I-Node Management

The block cache is not the only file system table that needs support procedures. The i-node table does, too. Many of the procedures are similar in function to the block management procedures. They are listed in Fig. 5-41.

Procedure	Function
<code>get_inode</code>	Fetch an i-node into memory
<code>put_inode</code>	Return an i-node that is no longer needed
<code>alloc_inode</code>	Allocate a new i-node (for a new file)
<code>wipe_inode</code>	Clear some fields in an i-node
<code>free_inode</code>	Release an i-node (when a file is removed)
<code>update_times</code>	Update time fields in an i-node
<code>rw_inode</code>	Transfer an i-node between memory and disk
<code>old_icopy</code>	Convert i-node contents to write to V1 disk i-node
<code>new_icopy</code>	Convert data read from V1 file system disk i-node
<code>dup_inode</code>	Indicate that someone else is using an i-node

Figure 5-41. Procedures used for i-node management.

The procedure *get_inode* (line 22933) is analogous to *get_block*. When any part of the file system needs an i-node, it calls *get_inode* to acquire it. *Get_inode* first searches the *inode* table to see if the i-node is already present. If so, it increments the usage counter and returns a pointer to it. This search is contained on

lines 22945 to 22955. If the i-node is not present in memory, the i-node is loaded by calling *rw_inode*.

When the procedure that needed the i-node is finished with it, the i-node is returned by calling the procedure *put_inode* (line 22976), which decrements the usage count *i_count*. If the count is then zero, the file is no longer in use, and the i-node can be removed from the table. If it is dirty, it is rewritten to disk.

If the *i_link* field is zero, no directory entry is pointing to the file, so all its zones can be freed. Note that the usage count going to zero and the number of links going to zero are different events, with different causes and different consequences. If the i-node is for a pipe, all the zones must be released, even though the number of links may not be zero. This happens when a process reading from a pipe releases the pipe. There is no sense in having a pipe for one process.

When a new file is created, an i-node must be allocated by *alloc_inode* (line 23003). MINIX 3 allows mounting of devices in read-only mode, so the superblock is checked to make sure the device is writable. Unlike zones, where an attempt is made to keep the zones of a file close together, any i-node will do. In order to save the time of searching the i-node bitmap, advantage is taken of the field in the superblock where the first unused i-node is recorded.

After the i-node has been acquired, *get_inode* is called to fetch the i-node into the table in memory. Then its fields are initialized, partly in-line (lines 23038 to 23044) and partly using the procedure *wipe_inode* (line 23060). This particular division of labor has been chosen because *wipe_inode* is also needed elsewhere in the file system to clear certain i-node fields (but not all of them).

When a file is removed, its i-node is freed by calling *free_inode* (line 23079). All that happens here is that the corresponding bit in the i-node bitmap is set to 0 and the superblock's record of the first unused i-node is updated.

The next function, *update_times* (line 23099), is called to get the time from the system clock and change the time fields that require updating. *Update_times* is also called by the *stat* and *fstat* system calls, so it is declared *PUBLIC*.

The procedure *rw_inode* (line 23125) is analogous to *rw_block*. Its job is to fetch an i-node from the disk. It does its work by carrying out the following steps:

1. Calculate which block contains the required i-node.
2. Read in the block by calling *get_block*.
3. Extract the i-node and copy it to the *inode* table.
4. Return the block by calling *put_block*.

Rw_inode is a bit more complex than the basic outline given above, so some additional functions are needed. First, because getting the current time requires a kernel call, any need for a change to the time fields in the i-node is only marked by setting bits in the i-node's *i_update* field while the i-node is in memory. If this field is nonzero when an i-node must be written, *update_times* is called.

Second, the history of MINIX adds a complication: in the old *V1* file system the i-nodes on the disk have a different structure from *V2*. Two functions, *old_icopy* (line 23168) and *new_icopy* (line 23214) are provided to take care of the conversions. The first converts between i-node information in memory and the format used by the *V1* filesystem. The second does the same conversion for *V2* and *V3* filesystem disks. Both of these functions are called only from within this file, so they are declared *PRIVATE*. Each function handles conversions in both directions (disk to memory or memory to disk).

Older versions of MINIX were ported to systems which used a different byte order from Intel processors and MINIX 3 is also likely to be ported to such architectures in the future. Every implementation uses the native byte order on its disk; the *sp->native* field in the superblock identifies which order is used. Both *old_icopy* and *new_icopy* call functions *conv2* and *conv4* to swap byte orders, if necessary. Of course, much of what we have just described is not used by MINIX 3, since it does not support the *V1* filesystem to the extent that *V1* disks can be used. And as of this writing nobody has ported MINIX 3 to a platform that uses a different byte order. But these bits and pieces remain in place for the day when someone decides to make MINIX 3 more versatile.

The procedure *dup_inode* (line 23257) just increments the usage count of the i-node. It is called when an open file is opened again. On the second open, the i-node need not be fetched from disk again.

Superblock Management

The file *super.c* contains procedures that manage the superblock and the bitmaps. Six procedures are defined in this file, listed in Fig. 5-42.

Procedure	Function
<i>alloc_bit</i>	Allocate a bit from the zone or i-node map
<i>free_bit</i>	Free a bit in the zone or i-node map
<i>get_super</i>	Search the superblock table for a device
<i>get_block_size</i>	Find block size to use
<i>mounted</i>	Report whether given i-node is on a mounted (or root) file system
<i>read_super</i>	Read a superblock

Figure 5-42. Procedures used to manage the superblock and bitmaps.

When an i-node or zone is needed, *alloc_inode* or *alloc_zone* is called, as we have seen above. Both of these call *alloc_bit* (line 23324) to actually search the relevant bitmap. The search involves three nested loops, as follows:

1. The outer one loops on all the blocks of a bitmap.
2. The middle one loops on all the words of a block.
3. The inner one loops on all the bits of a word.

The middle loop works by seeing if the current word is equal to the one's complement of zero, that is, a complete word full of 1s. If so, it has no free i-nodes or zones, so the next word is tried. When a word with a different value is found, it must have at least one 0 bit in it, so the inner loop is entered to find the free (i.e., 0) bit. If all the blocks have been tried without success, there are no free i-nodes or zones, so the code *NO_BIT* (0) is returned. Searches like this can consume a lot of processor time, but the use of the superblock fields that point to the first unused i-node and zone, passed to *alloc_bit* in *origin*, helps to keep these searches short.

Freeing a bit is simpler than allocating one, because no search is required. *Free_bit* (line 23400) calculates which bitmap block contains the bit to free and sets the proper bit to 0 by calling *get_block*, zeroing the bit in memory and then calling *put_block*.

The next procedure, *get_super* (line 23445), is used to search the superblock table for a specific device. For example, when a file system is to be mounted, it is necessary to check that it is not already mounted. This check can be performed by asking *get_super* to find the file system's device. If it does not find the device, then the file system is not mounted.

In MINIX 3 the file system server is capable of handling file systems with different block sizes, although within a given disk partition only a single block size can be used. The *get_block_size* function (line 23467) is meant to determine the block size of a file system. It searches the superblock table for the given device and returns the block size of the device if it is mounted. Otherwise the minimum block size, *MIN_BLOCK_SIZE* is returned.

The next function, *mounted* (line 23489), is called only when a block device is closed. Normally, all cached data for a device are discarded when it is closed. But, if the device happens to be mounted, this is not desirable. *Mounted* is called with a pointer to the i-node for a device. It just returns *TRUE* if the device is the root device, or if it is a mounted device.

Finally, we have *read_super* (line 23509). This is partially analogous to *rw_block* and *rw_inode*, but it is called only to read. The superblock is not read into the block cache at all, a request is made directly to the device for 1024 bytes starting at an offset of the same amount from the beginning of the device. Writing a superblock is not necessary in the normal operation of the system. *Read_super* checks the version of the file system from which it has just read and performs conversions, if necessary, so the copy of the superblock in memory will have the standard structure even when read from a disk with a different superblock structure or byte order.

Even though it is not currently used in MINIX 3, the method of determining whether a disk was written on a system with a different byte order is clever and worth noting. The magic number of a superblock is written with the native byte order of the system upon which the file system was created, and when a superblock is read a test for reversed-byte-order superblocks is made.

File Descriptor Management

MINIX 3 contains special procedures to manage file descriptors and the *filp* table (see Fig. 5-39). They are contained in the file *filedes.c*. When a file is created or opened, a free file descriptor and a free *filp* slot are needed. The procedure *get_fd* (line 23716) is used to find them. They are not marked as in use, however, because many checks must first be made before it is known for sure that the *creat* or *open* will succeed.

Get_filp (line 23761) is used to see if a file descriptor is in range, and if so, returns its *filp* pointer.

The last procedure in this file is *find_filp* (line 23774). It is needed to find out when a process is writing on a broken pipe (i.e., a pipe not open for reading by any other process). It locates potential readers by a brute force search of the *filp* table. If it cannot find one, the pipe is broken and the write fails.

File Locking

The POSIX record locking functions are shown in Fig. 5-43. A part of a file can be locked for reading and writing, or for writing only, by an *fcntl* call specifying a *F_SETLK* or *F_SETLKW* request. Whether a lock exists over a part of a file can be determined using the *F_GETLK* request.

Operation	Meaning
<i>F_SETLK</i>	Lock region for both reading and writing
<i>F_SETLKW</i>	Lock region for writing
<i>F_GETLK</i>	Report if region is locked

Figure 5-43. The POSIX advisory record locking operations. These operations are requested by using an *FCNTL* system call.

The file *lock.c* contains only two functions. *Lock_op* (line 23820) is called by the *fcntl* system call with a code for one of the operations shown in Fig. 5-43. It does some error checking to be sure the region specified is valid. When a lock is being set, it must not conflict with an existing lock, and when a lock is being cleared, an existing lock must not be split in two. When any lock is cleared, the other function in this file, *lock_revive* (line 23964), is called. It wakes up all the processes that are blocked waiting for locks.

This strategy is a compromise; it would take extra code to figure out exactly which processes were waiting for a particular lock to be released. Those processes that are still waiting for a locked file will block again when they start. This strategy is based on an assumption that locking will be used infrequently. If a major multiuser data base were to be built upon a MINIX 3 system, it might be desirable to reimplement this.

Lock_revive is also called when a locked file is closed, as might happen, for instance, if a process is killed before it finishes using a locked file.

5.7.3 The Main Program

The main loop of the file system is contained in file *main.c*, (line 24040). After a call to *fs_init* for initialization, the main loop is entered. Structurally, this is very similar to the main loop of the process manager and the I/O device drivers. The call to *get_work* waits for the next request message to arrive (unless a process previously suspended on a pipe or terminal can now be handled). It also sets a global variable, *who*, to the caller's process table slot number and another global variable, *call_nr*, to the number of the system call to be carried out.

Once back in the main loop the variable *fp* is pointed to the caller's process table slot, and the *super_user* flag tells whether the caller is the superuser or not. Notification messages are high priority, and a *SYS_SIG* message is checked for first, to see if the system is shutting down. The second highest priority is a *SYN_ALARM*, which means that a timer set by the file system has expired. A *NOTIFY_MESSAGE* means a device driver is ready for attention, and is dispatched to *dev_status*. Then comes the main attraction—the call to the procedure that carries out the system call. The procedure to call is selected by using *call_nr* as an index into the array of procedure pointers, *call_vecs*.

When control comes back to the main loop, if *dont_reply* has been set, the reply is inhibited (e.g., a process has blocked trying to read from an empty pipe). Otherwise a reply is sent by calling *reply* (line 24087). The final statement in the main loop has been designed to detect that a file is being read sequentially and to load the next block into the cache before it is actually requested, to improve performance.

Two other functions in this file are intimately involved with the file system's main loop. *Get_work* (line 24099) checks to see if any previously blocked procedures have now been revived. If so, these have priority over new messages. When there is no internal work to do the file system calls the kernel to get a message, on line 24124. Skipping ahead a few lines, we find *reply* (line 24159) which is called after a system call has been completed, successfully or otherwise. It sends a reply back to the caller. The process may have been killed by a signal, so the status code returned by the kernel is ignored. In this case there is nothing to be done anyway.

Initialization of the File System

The functions that remain to be discussed in *main.c* are used at system startup. The major player is *fs_init*, which is called by the file system before it enters its main loop during startup of the entire system. In the context of discussing process scheduling in Chapter 2 we showed in Fig. 2-43 the initial queueing of processes as the MINIX 3 system starts up. The file system is scheduled on a queue with lower priority than the process manager, so we can be sure that at startup time the process manager will get a chance to run before the file system. In Chapter 4 we examined the initialization of the process manager. As the PM builds its part of the process table, adding entries for itself and all other processes in the boot image, it sends a message to the file system for each one so the FS can initialize the corresponding entry in the FS part of the file system. Now we can see the other half of this interaction.

When the file system starts it immediately enters a loop of its own in *fs_init*, on lines 24189 to 24202. The first statement in the loop is a call to *receive*, to get a message sent at line 18235 in the PM's *pm_init* initialization function. Each message contains a process number and a PID. The first is used as an index into the file system's process table and the second is saved in the *fp_pid* field of each selected slot. Following this the real and effective uid and gid for the superuser and a ~0 (all bits set) umask is set up for each selected slot. When a message with the symbolic value *NONE* in the process number field is received the loop terminates and a message is sent back to the process manager to tell it all is OK.

Next, the file system's own initialization is completed. First important constants are tested for valid values. Then several other functions are invoked to initialize the block cache and the device table, to load the RAM disk if necessary, and to load the root device superblock. At this point the root device can be accessed, and another loop is made through the FS part of the process table, so each process loaded from the boot image will recognize the root directory and use the root directory as its working directory (lines 24228 to 24235).

The first function called by *fs_init* after it finishes its interaction with the process manager is *buf_pool*, which begins on line 24132. It builds the linked lists used by the block cache. Figure 5-37 shows the normal state of the block cache, in which all blocks are linked on both the LRU chain and a hash chain. It may be helpful to see how the situation of Fig. 5-37 comes about. Immediately after the cache is initialized by *buf_pool*, all the buffers will be on the LRU chain, and all will be linked into the 0th hash chain, as in Fig. 5-44(a). When a buffer is requested, and while it is in use, we have the situation of Fig. 5-44(b), in which we see that a block has been removed from the LRU chain and is now on a different hash chain.

Normally, blocks are released and returned to the LRU chain immediately. Figure 5-44(c) shows the situation after the block has been returned to the LRU chain. Although it is no longer in use, it can be accessed again to provide the

same data, if need be, and so it is retained on the hash chain. After the system has been in operation for awhile, almost all of the blocks can be expected to have been used and to be distributed among the different hash chains at random. Then the LRU chain will look like Fig. 5-37.

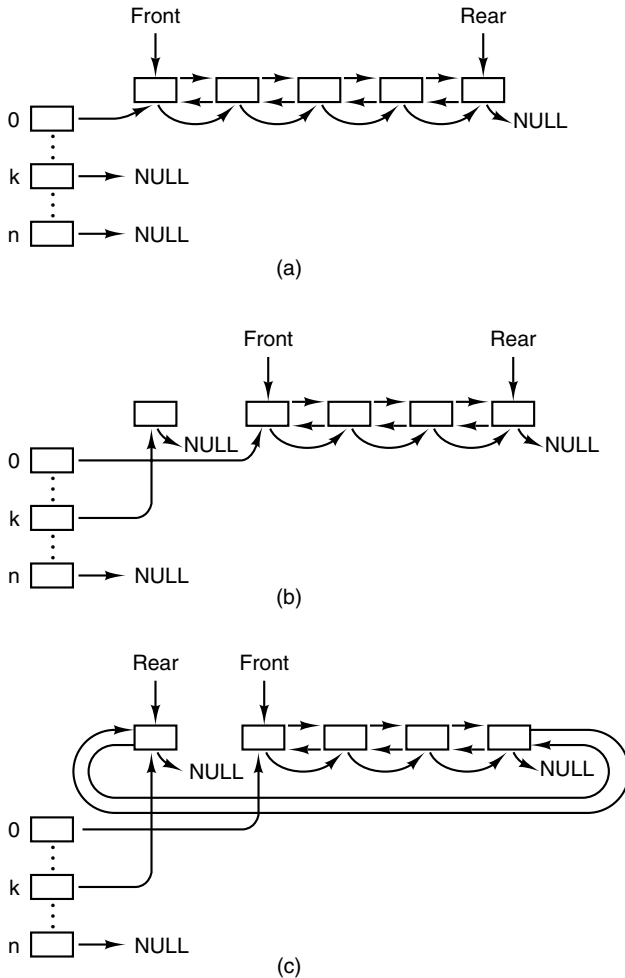


Figure 5-44. Block cache initialization. (a) Before any buffers have been used. (b) After one block has been requested. (c) After the block has been released.

The next thing called after *buf_pool* is *build_dmap*, which we will describe later, along with other functions dealing with device files. After that, *load_ram* is called, which uses the next function we will examine, *igetenv* (line 2641). This

function retrieves a numeric device identifier from the kernel, using the name of a boot parameter as a key. If you have used the *sysenv* command to look at the boot parameters on a working MINIX 3 system, you have seen that *sysenv* reports devices numerically, displaying strings like

```
rootdev=912
```

The file system uses numbers like this to identify devices. The number is simply $256 \times \text{major} + \text{minor}$, where *major* and *minor* are the major and minor device numbers. In this example, the major, minor pair is 3, 144, which corresponds to */dev/c0d1p0s0*, a typical place to install MINIX 3 on a system with two disk drives.

Load_ram (line 24260) allocates space for a RAM disk, and loads the root file system on it, if required by the boot parameters. It uses *igetenv* to get the *rootdev*, *ramimagedev*, and *ramsize* parameters set in the boot environment (lines 24278 to 24280). If the boot parameters specify

```
rootdev = ram
```

the root file system is copied from the device named by *ramimagedev* to the RAM disk block by block, starting with the boot block, with no interpretation of the various file system data structures. If the *ramsize* boot parameter is smaller than the size of *ramimagedev*, the RAM disk is made large enough to hold it. If *ramsize* specifies a size larger than the boot device file system the requested size is allocated and the RAM disk file system is adjusted to use the full size specified (lines 24404 to 24420). This is the only time that the file system ever writes a superblock, but, just as with reading a superblock, the block cache is not used and the data is written directly to the device using *dev_io*.

Two items merit note at this point. The first is the code on lines 24291 to 24307 which deals with the case of booting from a CD-ROM. The *cdprobe* function, not discussed in this text, is used. Interested readers are referred to the code in *fs/cdprobe.c*, which can be found on the CD-ROM or the Web site. Second, regardless of the disk block size used by MINIX 3 for ordinary disk access, the boot block is always a 1 KB block and the superblock is loaded from the second 1 KB of the disk device. Anything else would be complicated, since the block size cannot be known until the superblock has been loaded.

Load_ram allocates space for an empty RAM disk if a nonzero *ramsize* is specified without a request to use the RAM disk as the root file system. In this case, since no file system structures are copied, the RAM device cannot be used as a file system until it has been initialized by the *mkfs* command. Alternatively, such a RAM disk can be used for a secondary cache if support for this is compiled into the file system.

The last function in *main.c* is *load_super* (line 24426). It initializes the superblock table and reads in the superblock of the root device.

5.7.4 Operations on Individual Files

In this section we will look at the system calls that operate on individual files one at a time (as opposed to, say, operations on directories). We will start with how files are created, opened, and closed. After that we will examine in some detail the mechanism by which files are read and written. Then that we will look at pipes and how operations on them differ from those on files.

Creating, Opening, and Closing Files

The file *open.c* contains the code for six system calls: *creat*, *open*, *mknod*, *mknod*, *close*, and *lseek*. We will examine *creat* and *open* together, and then look at each of the others.

In older versions of UNIX, the *creat* and *open* calls had distinct purposes. Trying to open a file that did not exist was an error, and a new file had to be created with *creat*, which could also be used to truncate an existing file to zero length. The need for two distinct calls is no longer present in a POSIX system, however. Under POSIX, the *open* call now allows creating a new file or truncating an old file, so the *creat* call now represents a subset of the possible uses of the *open* call and is really only necessary for compatibility with older programs. The procedures that handle *creat* and *open* are *do_creat* (line 24537) and *do_open* (line 24550). (As in the process manager, the convention is used in the file system that system call XXX is performed by procedure *do_XXX*.) Opening or creating a file involves three steps:

1. Finding the i-node (allocating and initializing if the file is new).
2. Finding or creating the directory entry.
3. Setting up and returning a file descriptor for the file.

Both the *creat* and the *open* calls do two things: they fetch the name of a file and then they call *common_open* which takes care of tasks common to both calls.

Common_open (line 24573) starts by making sure that free file descriptor and *filp* table slots are available. If the calling function specified creation of a new file (by calling with the *O_CREAT* bit set), *new_node* is called on line 24594. *New_node* returns a pointer to an existing i-node if the directory entry already exists; otherwise it will create both a new directory entry and i-node. If the i-node cannot be created, *new_node* sets the global variable *err_code*. An error code does not always mean an error. If *new_node* finds an existing file, the error code returned will indicate that the file exists, but in this case that error is acceptable (line 24597). If the *O_CREAT* bit is not set, a search is made for the i-node using an alternative method, the *eat_path* function in *path.c*, which we will discuss further on. At this point, the important thing to understand is that if an i-node is

not found or successfully created, *common_open* will terminate with an error before line 24606 is reached. Otherwise, execution continues here with assignment of a file descriptor and claiming of a slot in the *filp* table. Following this, if a new file has just been created, lines 24612 to 24680 are skipped.

If the file is not new, then the file system must test to see what kind of a file it is, what its mode is, and so on, to determine whether it can be opened. The call to *forbidden* on line 24614 first makes a general check of the *rw**x* bits. If the file is a regular file and *common_open* was called with the *O_TRUNC* bit set, it is truncated to length zero and *forbidden* is called again (line 24620), this time to be sure the file may be written. If the permissions allow, *wipe_inode* and *rw_inode* are called to re-initialize the i-node and write it to the disk. Other file types (directories, special files, and named pipes) are subjected to appropriate tests. In the case of a device, a call is made on line 24640 (using the *dmap* structure) to the appropriate routine to open the device. In the case of a named pipe, a call is made to *pipe_open* (line 24646), and various tests relevant to pipes are made.

The code of *common_open*, as well as many other file system procedures, contains a large amount of code that checks for various errors and illegal combinations. While not glamorous, this code is essential to having an error-free, robust file system. If something is wrong, the file descriptor and *filp* slot previously allocated are deallocated and the i-node is released (lines 24683 to 24689). In this case the value returned by *common_open* will be a negative number, indicating an error. If there are no problems the file descriptor, a positive value, is returned.

This is a good place to discuss in more detail the operation of *new_node* (line 24697), which does the allocation of the i-node and the entering of the path name into the file system for *creat* and *open* calls. It is also used for the *mknod* and *mkdir* calls, yet to be discussed. The statement on line 24711 parses the path name (i.e., looks it up component by component) as far as the final directory; the call to *advance* three lines later tries to see if the final component can be opened.

For example, on the call

```
fd = creat("/usr/ast/foobar", 0755);
```

last_dir tries to load the i-node for */usr/ast/* into the tables and return a pointer to it. If the file does not exist, we will need this i-node shortly in order to add *foobar* to the directory. All the other system calls that add or delete files also use *last_dir* to first open the final directory in the path.

If *new_node* discovers that the file does not exist, it calls *alloc_inode* on line 24717 to allocate and load a new i-node, returning a pointer to it. If no free i-nodes are left, *new_node* fails and returns *NIL_INODE*.

If an i-node can be allocated, the operation continues at line 24727, filling in some of the fields, writing it back to the disk, and entering the file name in the final directory (on line 24732). Again we see that the file system must constantly check for errors, and upon encountering one, carefully release all the resources, such as i-nodes and blocks that it is holding. If we were prepared to just let

MINIX 3 panic when we ran out of, say, i-nodes, rather than undoing all the effects of the current call and returning an error code to the caller, the file system would be appreciably simpler.

As mentioned above, pipes require special treatment. If there is not at least one reader/writer pair for a pipe, *pipe_open* (line 24758) suspends the caller. Otherwise, it calls *release*, which looks through the process table for processes that are blocked on the pipe. If it is successful, the processes are revived.

The *mknod* call is handled by *do_mknod* (line 24785). This procedure is similar to *do_creat*, except that it just creates the i-node and makes a directory entry for it. In fact, most of the work is done by the call to *new_node* on line 24797. If the i-node already exists, an error code will be returned. This is the same error code that was an acceptable result from *new_node* when it was called by *common_open*; in this case, however, the error code is passed back to the caller, which presumably will act accordingly. The case-by-case analysis we saw in *common_open* is not needed here.

The *mkdir* call is handled by the function *do_mkdir* (line 24805). As with the other system calls we have discussed here, *new_node* plays an important part. Directories, unlike files, always have links and are never completely empty because every directory must contain two entries from the time of its creation: the “.” and “..” entries that refer to the directory itself and to its parent directory. The number of links a file may have is limited, it is *LINK_MAX* (defined in *include/limits.h* as *SHRT_MAX*, 32767 for MINIX 3 on a standard 32-bit Intel system). Since the reference to a parent directory in a child is a link to the parent, the first thing *do_mkdir* does is to see if it is possible to make another link in the parent directory (lines 24819 and 24820). Once this test has been passed, *new_node* is called. If *new_node* succeeds, then the directory entries for “.” and “..” are made (lines 24841 and 24842). All of this is straightforward, but there could be failures (for instance, if the disk is full), so to avoid making a mess of things provision is made for undoing the initial stages of the process if it can not be completed.

Closing a file is easier than opening one. The work is done by *do_close* (line 24865). Pipes and special files need some attention, but for regular files, almost all that needs to be done is to decrement the *filp* counter and check to see if it is zero, in which case the i-node is returned with *put_inode*. The final step is to remove any locks and to revive any process that may have been suspended waiting for a lock on the file to be released.

Note that returning an i-node means that its counter in the *inode* table is decremented, so it can be removed from the table eventually. This operation has nothing to do with freeing the i-node (i.e., setting a bit in the bitmap saying that it is available). The i-node is only freed when the file has been removed from all directories.

The final procedure in *open.c* is *do_lseek* (line 24939). When a seek is done, this procedure is called to set the file position to a new value. On line 24968

reading ahead is inhibited; an explicit attempt to seek to a position in a file is incompatible with sequential access.

Reading a File

Once a file has been opened, it can be read or written. Many functions are used during both reading and writing. These are found in the file *read.c*. We will discuss these first and then proceed to the following file, *write.c*, to look at code specifically used for writing. Reading and writing differ in a number of ways, but they have enough similarities that all that is required of *do_read* (line 25030) is to call the common procedure *read_write* with a flag set to *READING*. We will see in the next section that *do_write* is equally simple.

Read_write begins on line 25038. Some special code on lines 25063 to 25066 is used by the process manager to have the file system load entire segments in user space for it. Normal calls are processed starting on line 25068. Some validity checks follow (e.g., reading from a file opened only for writing) and some variables are initialized. Reads from character special files do not go through the block cache, so they are filtered out on line 25122.

The tests on lines 25132 to 25145 apply only to writes and have to do with files that may get bigger than the device can hold, or writes that will create a hole in the file by writing *beyond* the end-of-file. As we discussed in the MINIX 3 overview, the presence of multiple blocks per zone causes problems that must be dealt with explicitly. Pipes are also special and are checked for.

The heart of the read mechanism, at least for ordinary files, is the loop starting on line 25157. This loop breaks the request up into chunks, each of which fits in a single disk block. A chunk begins at the current position and extends until one of the following conditions is met:

1. All the bytes have been read.
2. A block boundary is encountered.
3. The end-of-file is hit.

These rules mean that a chunk never requires two disk blocks to satisfy it. Figure 5-45 shows three examples of how the chunk size is determined, for chunk sizes of 6, 2, and 1 bytes, respectively. The actual calculation is done on lines 25159 to 25169.

The actual reading of the chunk is done by *rw_chunk*. When control returns, various counters and pointers are incremented, and the next iteration begins. When the loop terminates, the file position and other variables may be updated (e.g., pipe pointers).

Finally, if read ahead is called for, the i-node to read from and the position to read from are stored in global variables, so that after the reply message is sent to the user, the file system can start getting the next block. In many cases the file

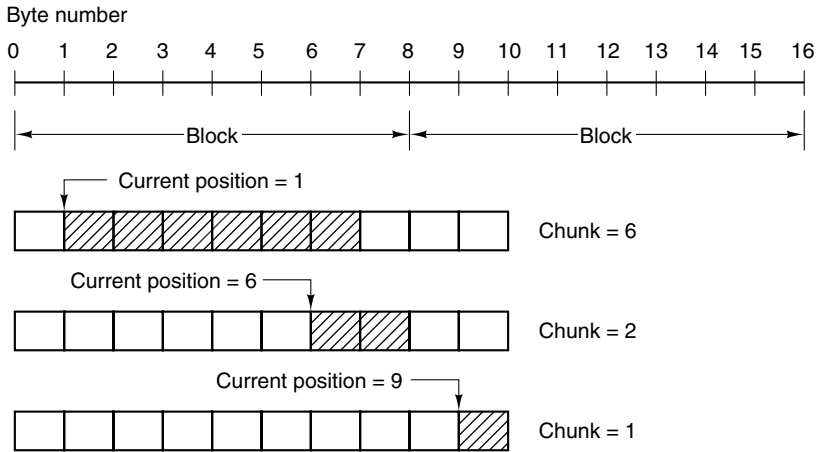


Figure 5-45. Three examples of how the first chunk size is determined for a 10-byte file. The block size is 8 bytes, and the number of bytes requested is 6. The chunk is shown shaded.

system will block, waiting for the next disk block, during which time the user process will be able to work on the data it just received. This arrangement overlaps processing and I/O and can improve performance substantially.

The procedure *rw_chunk* (line 25251) is concerned with taking an i-node and a file position, converting them into a physical disk block number, and requesting the transfer of that block (or a portion of it) to the user space. The mapping of the relative file position to the physical disk address is done by *read_map*, which understands about i-nodes and indirect blocks. For an ordinary file, the variables *b* and *dev* on line 25280 and line 25281 contain the physical block number and device number, respectively. The call to *get_block* on line 25303 is where the cache handler is asked to find the block, reading it in if need be. Calling *rahead* on line 25295 then ensures that the block is read into the cache.

Once we have a pointer to the block, the *sys_vircopy* kernel call on line 25317 takes care of transferring the required portion of it to the user space. The block is then released by *put_block*, so that it can be evicted from the cache later. (After being acquired by *get_block*, it will not be in the LRU queue and it will not be returned there while the counter in the block's header shows that it is in use, so it will be exempt from eviction; *put_block* decrements the counter and returns the block to the LRU queue when the counter reaches zero.) The code on line 25327 indicates whether a write operation filled the block. However, the value passed to *put_block* in *n* does not affect how the block is placed on the queue; all blocks are now placed on the rear of the LRU chain.

Read_map (line 25337) converts a logical file position to the physical block number by inspecting the i-node. For blocks close enough to the beginning of the

file that they fall within one of the first seven zones (the ones right in the i-node), a simple calculation is sufficient to determine which zone is needed, and then which block. For blocks further into the file, one or more indirect blocks may have to be read.

Rd_indir (line 25400) is called to read an indirect block. The comments for this function are a bit out of date; code to support the 68000 processor has been removed and the support for the MINIX V1 file system is not used and could also be dropped. However, it is worth noting that if someone wanted to add support for other file system versions or other platforms where data might have a different format on the disk, problems of different data types and byte orders could be relegated to this file. If messy conversions were necessary, doing them here would let the rest of the file system see data in only one form.

Read_ahead (line 25432) converts the logical position to a physical block number, calls *get_block* to make sure the block is in the cache (or bring it in), and then returns the block immediately. It cannot do anything with the block, after all. It just wants to improve the chance that the block is around if it is needed soon,

Note that *read_ahead* is called only from the main loop in *main*. It is not called as part of the processing of the read system call. It is important to realize that the call to *read_ahead* is performed *after* the reply is sent, so that the user will be able to continue running even if the file system has to wait for a disk block while reading ahead.

Read_ahead by itself is designed to ask for just one more block. It calls the last function in *read.c*, *rahead*, to actually get the job done. *Rahead* (line 25451) works according to the theory that if a little more is good, a lot more is better. Since disks and other storage devices often take a relatively long time to locate the first block requested but then can relatively quickly read in a number of adjacent blocks, it may be possible to get many more blocks read with little additional effort. A prefetch request is made to *get_block*, which prepares the block cache to receive a number of blocks at once. Then *rw_scattered* is called with a list of blocks. We have previously discussed this; recall that when the device drivers are actually called by *rw_scattered*, each one is free to answer only as much of the request as it can efficiently handle. This all sounds fairly complicated, but the complications make possible a significant speedup of applications which read large amounts of data from the disk.

Figure 5-46 shows the relations between some of the major procedures involved in reading a file—in particular, who calls whom.

Writing a File

The code for writing to files is in *write.c*. Writing a file is similar to reading one, and *do_write* (line 25625) just calls *read_write* with the *WRITING* flag. A major difference between reading and writing is that writing requires allocating new disk blocks. *Write_map* (line 25635) is analogous to *read_map*, only instead

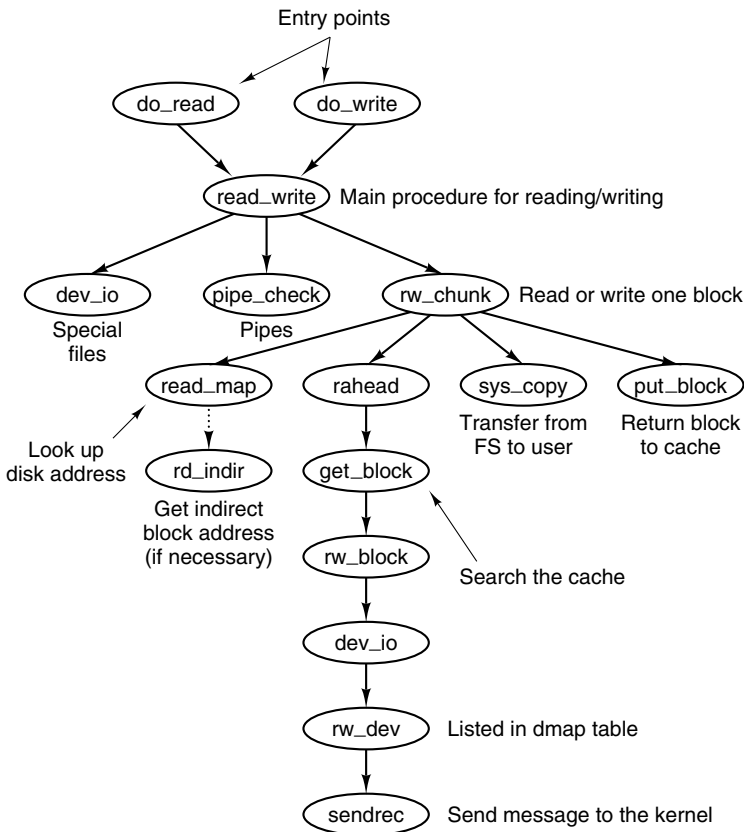


Figure 5-46. Some of the procedures involved in reading a file.

of looking up physical block numbers in the i-node and its indirect blocks, it enters new ones there (to be precise, it enters zone numbers, not block numbers).

The code of `write_map` is long and detailed because it must deal with several cases. If the zone to be inserted is close to the beginning of the file, it is just inserted into the i-node on (line 25658).

The worst case is when a file exceeds the size that can be handled by a single-indirect block, so a double-indirect block is now required. Next, a single-indirect block must be allocated and its address put into the double-indirect block. As with reading, a separate procedure, `wr_indir`, is called. If the double-indirect block is acquired correctly, but the disk is full so the single-indirect block cannot be allocated, then the double one must be returned to avoid corrupting the bitmap.

Again, if we could just toss in the sponge and panic at this point, the code would be much simpler. However, from the user's point of view it is much nicer that running out of disk space just returns an error from `write`, rather than crashing the computer with a corrupted file system.

Wr_indir (line 25726) calls the conversion routines, *conv4* to do any necessary data conversion and puts a new zone number into an indirect block. (Again, there is leftover code here to handle the old V1 filesystem, but only the V2 code is currently used.) Keep in mind that the name of this function, like the names of many other functions that involve reading and writing, is not literally true. The actual writing to the disk is handled by the functions that maintain the block cache.

The next procedure in *write.c* is *clear_zone* (line 25747), which takes care of the problem of erasing blocks that are suddenly in the middle of a file. This happens when a seek is done beyond the end of a file, followed by a write of some data. Fortunately, this situation does not occur very often.

New_block (line 25787) is called by *rw_chunk* whenever a new block is needed. Figure 5-47 shows six successive stages of the growth of a sequential file. The block size is 1-KB and the zone size is 2-KB in this example.

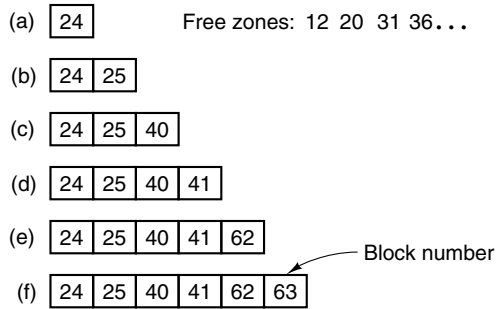


Figure 5-47. (a) – (f) The successive allocation of 1-KB blocks with a 2-KB zone.

The first time *new_block* is called, it allocates zone 12 (blocks 24 and 25). The next time it uses block 25, which has already been allocated but is not yet in use. On the third call, zone 20 (blocks 40 and 41) is allocated, and so on. *Zero_block* (line 25839) clears a block, erasing its previous contents. This description is considerably longer than the actual code.

Pipes

Pipes are similar to ordinary files in many respects. In this section we will focus on the differences. The code we will discuss is all in *pipe.c*.

First of all, pipes are created differently, by the *pipe* call, rather than the *creat* call. The *pipe* call is handled by *do_pipe* (line 25933). All *do_pipe* really does is allocate an i-node for the pipe and return two file descriptors for it. Pipes are owned by the system, not by the user, and are located on the designated pipe de-

vice (configured in *include/minix/config.h*), which could very well be a RAM disk, since pipe data do not have to be preserved permanently.

Reading and writing a pipe is slightly different from reading and writing a file, because a pipe has a finite capacity. An attempt to write to a pipe that is already full will cause the writer to be suspended. Similarly, reading from an empty pipe will suspend the reader. In effect, a pipe has two pointers, the current position (used by readers) and the size (used by writers), to determine where data come from or go to.

The various checks to see if an operation on a pipe is possible are carried out by *pipe_check* (line 25986). In addition to the above tests, which may lead to the caller being suspended, *pipe_check* calls *release* to see if a process previously suspended due to no data or too much data can now be revived. These revivals are done on line 26017 and line 26052, for sleeping writers and readers, respectively. Writing on a broken pipe (no readers) is also detected here.

The act of suspending a process is done by *suspend* (line 26073). All it does is save the parameters of the call in the process table and set the flag *dont_reply* to *TRUE*, to inhibit the file system's reply message.

The procedure *release* (line 26099) is called to check if a process that was suspended on a pipe can now be allowed to continue. If it finds one, it calls *revive* to set a flag so that the main loop will notice it later. This function is not a system call, but is listed in Fig. 5-33(c) because it uses the message-passing mechanism.

The last procedure in *pipe.c* is *do_unpause* (line 26189). When the process manager is trying to signal a process, it must find out if that process is hanging on a pipe or special file (in which case it must be awakened with an *EINTR* error). Since the process manager knows nothing about pipes or special files, it sends a message to the file system to ask. That message is processed by *do_unpause*, which revives the process, if it is blocked. Like *revive*, *do_unpause* has some similarity to a system call, although it is not one.

The last two functions in *pipe.c*, *select_request_pipe* (line 26247) and *select_match_pipe* (line 26278), support the *select* call, which is not discussed here.

5.7.5 Directories and Paths

We have now finished looking at how files are read and written. Our next task is to see how path names and directories are handled.

Converting a Path to an I-Node

Many system calls (e.g., *open*, *unlink*, and *mount*) have path names (i.e., file names) as a parameter. Most of these calls must fetch the i-node for the named file before they can start working on the call itself. How a path name is converted