chapter

16

File Systems

The fourth main resource in a computer system is persistent storage, that is, a file system. By *persistent storage*, we mean storage that will continue to exist (and, of course, retain its contents) after the program that uses or creates it completes. The main memory allocated to a program is returned to the operating system after the program execution completes and the contents are lost. Persistent storage allows a program to create persistent data.

The most common hardware device that offers persistent storage is the magnetic disk. Other hardware devices also provide persistent storage (ROM, tape, CD). A file system provides an improved version of disk devices. A file system is not strictly necessary for computation, but it makes it much more convenient. There are some special-purpose operating systems that do not have a file system at all (for example, an operating system on a computer aboard a satellite).

16.1 THE NEED FOR FILES

16.1.1 USING DISKS DIRECTLY FOR PERSISTENT STORAGE

What is a disk like as a hardware device? Well, first of all, it *is* persistent. The data written on a disk is still there after the process that writes it has completed and been destroyed. The data written on the disk stay there indefinitely after it is written and does not require continuous power.

We saw in Chapter 15 that disks store data in large, fixed-size blocks that are addressed numerically as triples (cylinder, track, sector) or with a sector number. The disk is controlled by writing words into its control registers. Disk errors are uncommon but not rare, and so errors must be dealt with. The error reporting and recovery is complex.

Overall, the only nice thing about a disk is that it provides persistent storage. Everything else about it is complex and hard to use: the units of reading and writing are too large, the addressing and the operations are too complex, and the reliability is too low.

In order to make up for the poor user friendliness of disks, the operating system abstracts the disk resource into a much nicer persistent storage resource. This new resource will allow reading and writing in arbitrary, variable-sized units; it will have more convenient names (character strings); it will provide some structure to the storage, including concepts of ownership of storage and protection; it will provide reliable data storage; and it will provide simple, easy-to-use operations to access the data.

16.1.2 FILES

What we are talking about is a file system which implements files using disks. The basic concept in the file system is that of the file, which is a collection of information. A *file* is a sequence of bytes of arbitrary length (from 0 to a very large upper limit). Each file is owned by some user and is protected from unauthorized use by other users. Each file has a name, which is a character string and which shows where the file fits in to a hierarchical structure.

Files are implemented by the operating system to provide persistent storage.

16.1.3 LEVELS IN THE FILE SYSTEM

Figure 16.1 shows the levels of the file system abstraction. There is the logical specification of the system call interface to the file system objects and operations, there is the implementation of the file system interface on the I/O system interface, and there is the implementation of the I/O system interface on the disk hardware. We

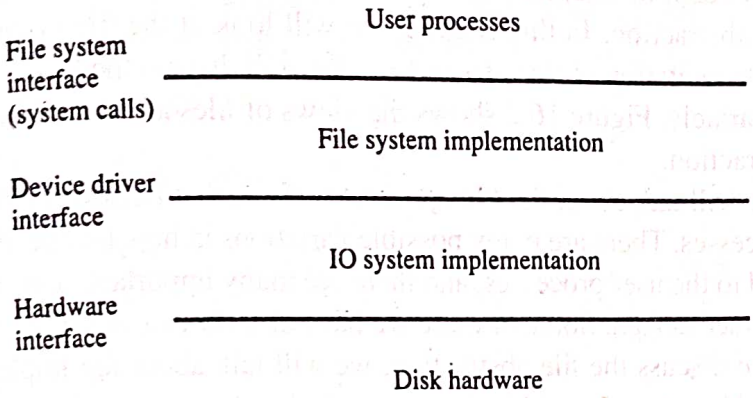


Figure 16.1 File system levels

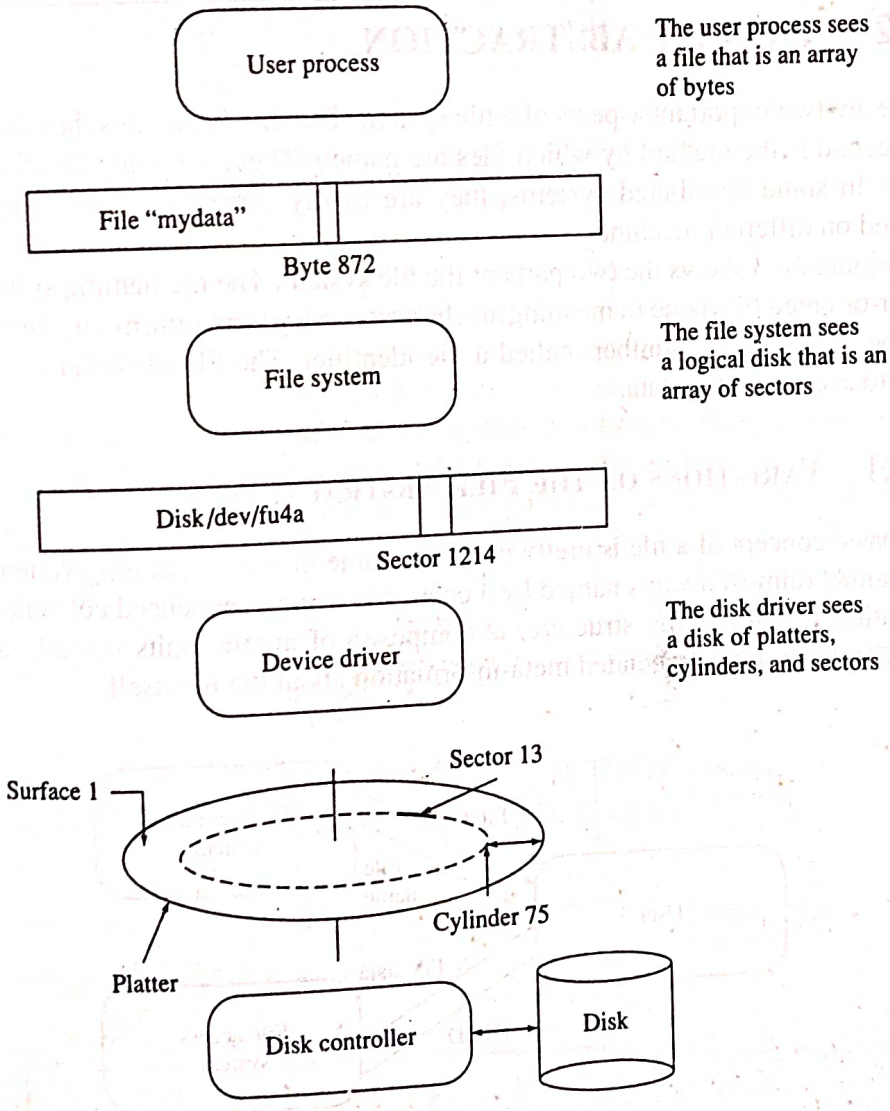


Figure 16.2 Views of a file at each level

have already seen, in Chapter 15, the I/O system abstraction and the implementation of that abstraction. In this chapter, we will look at the file system abstraction and the implementation of that abstraction. We will discuss these two aspects of file systems separately. Figure 16.2 shows the views of files at each of the levels of file system abstraction.

First we will talk about the file system interface, that is, how the file system appears to processes. There are many possible variations in how the file abstraction can be presented to the user processes, and there are many important design issues in file system interface design. Some of these we have already discussed in Chapter 3.

After we discuss the file abstraction, we will talk about the implementation of file systems. Here, too, we will look at many design decisions and the different ways we can solve each design problem. We will see that the implementation of files has some similarities to the implementation of virtual memory.

16.2 THE FILE ABSTRACTION

There are two important aspects of a file system. The first is the files themselves, and the second is the method by which files are named. These are logically distinct services. In some distributed systems, they are totally separated, even being implemented on different machines.

Figure 16.3 shows the two parts of the file system. The file naming system takes a user-oriented file name (a meaningful character string) and returns an internal name for the file (a binary number) called a file identifier. The file identifier can then be used to access the file data.

16.2.1 VARIATIONS ON THE FILE ABSTRACTION

The basic concept of a file is pretty much the same in every operating system. A file is a named (almost always named by a character string), sequenced collection of information that has some structure, is composed of atomic units (usually a single byte), and has some associated meta-information about the file itself.

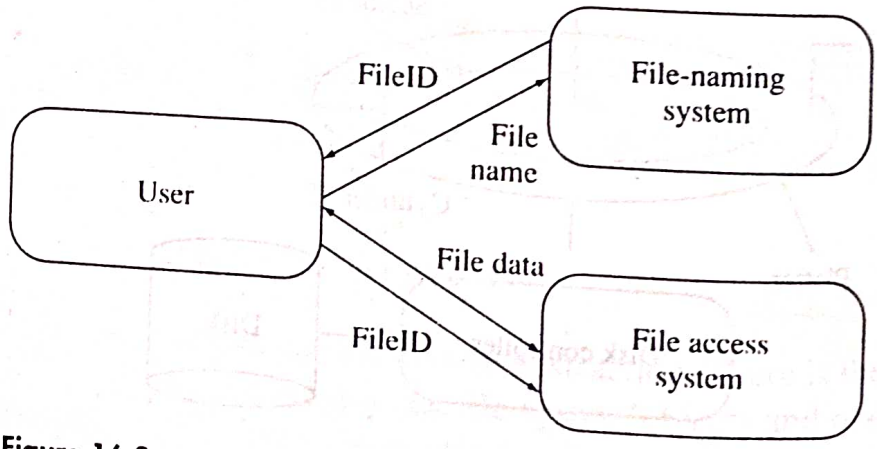


Figure 16.3 The two major parts of a file system

We will look at the file abstraction first, looking at each aspect of the file concept to see the possible variations in how it can be presented to user processes. Then we will turn our attention to the naming of files.

16.2.2 Logical File Structure

What do files look like to processes? First we need to decide what kind of internal structure files will have. There are three possible forms of atomic units in a file:

- Bytes,
- Fixed-length records, or
- Variable-length records.

Figure 16.4 shows the three forms of record structure in a file.

The simplest atomic unit an operating system can provide is files that are simply a sequence of bytes. This is sometimes called a *flat file* because it has no internal record structure. flat file

Many older operating systems required files to be divided into fixed-length records. Each record is a collection of information about one thing, and is analogous to a C++ structure. A record might represent an employee, a vehicle, a building, a city, an invoice, etc.

Fixed-length records are easy for an operating system to deal with, but do not reflect the realities of data. Students have taken different numbers of courses; invoices have different numbers of items; employees have different lengths of work history. To meet these needs, most operating systems also provide variable-length records.

One problem with a record structure is that it is hard to find anything unless you know where it is in the file, that is, which record it is in. For example, suppose we had a file where each record was an employee, and we wanted to find the record for Rita Mondragon. If we did not know which record it was in, we would have to search the entire file until we found it. To deal with this problem, some operating systems have a form of file called a *keyed file*. Each record has a specified field (a *field* is a part of a record, like a component in a C++ structure or class) that is the key keyed file

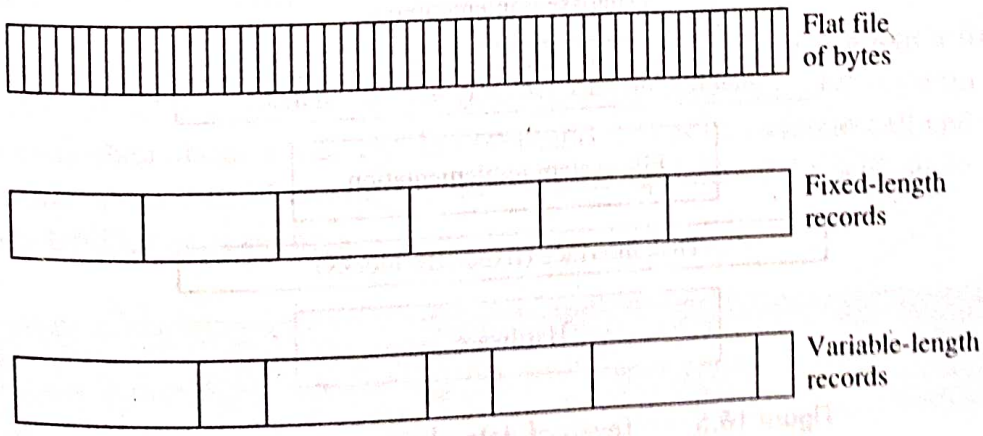


Figure 16.4 Record structures in a file

field. You can request a record by key rather than record number, and the operating system will find the record for you. Various search data structures have been used by operating systems to implement keyed files. B-trees are the most common.

Gradually, designers realized that data management was a complex and sophisticated problem that should not be solved in the operating system itself but in a separate program called a database management system. All record structuring is handled by the database management system, and it is only necessary for the operating system to provide basic, flat files for the database management system to build on.

As a consequence, most modern operating systems provide files that are simply a sequence of bytes, and the application programs, not the operating system, impose structure on the files. Figure 16.5 shows the levels involved. The operating system provides flat files, and the database management system builds complex data structures on top of that. In some cases, the operating system will provide a more efficient, lower-level interface to the disk. Since the performance requirements for database management systems are very high, in some cases it might be allocated an entire disk or disk partition and be given direct access to the disk device driver interface. That way it can avoid the overhead of the file system.

Most operating systems implement flat files, which are essentially arrays of bytes stored on the disk.

16.2.3 File Size and Granularity

An operating system designer would like to allow files to be of any length, with no fixed maximum size. Normally this is too difficult to implement, and so they settle

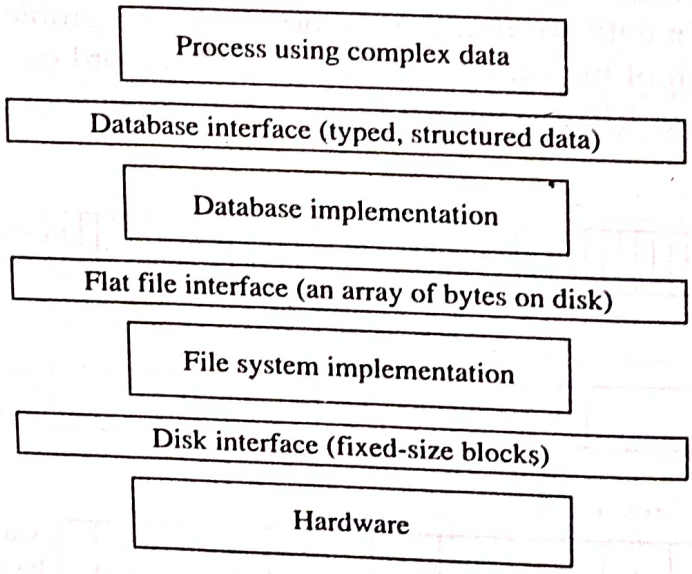


Figure 16.5 Levels of data abstraction

for a very large maximum file size. For example, the standard UNIX file system has a maximum file size of 1 billion blocks. With 4K disk blocks, that would be a maximum file size of 4 trillion bytes (4 Tbyte). This is the theoretical maximum, however, since the file pointers are 32-bit integers on most UNIX systems and they only go to 4 billion bytes (4 Gbyte). As 64-bit architectures become common and languages start using 64-bit integers, this size limit will be eliminated. It is fairly easy to implement a very large maximum file size, and so file size limits are not a problem in modern operating systems.

Granularity of Files Some file systems to not allow files of just any length, but only in some fixed units such as disk blocks. This reflects the inherent granularity of the disks themselves. But most operating systems hide this granularity from the user and allow files of any size from 0 bytes on up.

16.2.4 File Meta-Data

A file contains information, but the file system also keeps information about the file, that is, meta-data (information about information). A file system will keep track of things like:

- The name of the file.
- The type of the file.
- The size of the file.
- The owner of the file.
- The group(s) of users that have special access to the file.
- What access to the file is allowed to various users.
- The last time the file was read.
- The last time the file was written.
- The time the file was created.
- Which disk the file is stored on.
- Where the file is stored on disk.

There is generally a system call that will return all the meta-data about a file, and a command to display this information. In UNIX, the `stat` and `fstat` system calls return the meta-data about a file. The `ls` command makes this system call and will display the file meta-data. Looking at the various options of `ls` will show you the meta-data UNIX keeps on files.

Files contain data and have associated meta-data describing the file data.

16.3 FILE NAMING

path name
component name
separator character

Files in an operating system are generally named using a hierarchical naming system based on directories (as described in Section 3.2.2). Virtually all modern operating systems use a hierarchical naming system. In such a system, a *path name* consists of a series of *component names* separated with a *separator character*. In UNIX the separator is the slash ("/") character, and in MS/DOS the separator is the backwards slash ("\") character.[1]

16.3.1 COMPONENT NAMES

There are two main issues about the component names:

- What characters are allowed to be in the name?
- How long can the name be?

Most naming schemes have some disallowed characters. For example, it is inconvenient to allow the separator character to be in a component name. Sometimes spaces are not allowed because they are used as separators when file names are used on command lines.

Naturally, we would like component names to be as long as the user wants, but most systems place limits on the length because it takes a lot of programming to handle names of unlimited length. UNIX used to have a limit of 14 characters, but now allows longer names (from 28 to 255 characters depending on the UNIX version). The Macintosh OS allows names of up to 31 characters. MS/DOS limits names to 11 characters in length.

16.3.2 DIRECTORIES

We talked about hierarchical file systems and directories in Section 3.2.2. In that terminology, a directory is a name space, and the associated name map maps each name into either a file or another directory. We will see later that directories are usually implemented as files, and so there is not really a distinction between files and directories from the operating system's point of view. But users do think of them differently, and so it is useful to consider them as separate kinds of objects.

16.3.3 PATH NAMES

absolute path name

An *absolute path name* is a name that gives the path from the root directory to the file we are naming. This is an unambiguous way to name files, but absolute path names tend to be long. When you use a hierarchical file system to classify your files, you find that you create many levels of subdirectories, and hence your path names become long.

[1] In some operating systems, the delimiter between components is not the same for every level. VMS and MS/DOS are examples of this. The reason for this is generally because the system is an extension of a simpler naming system.

In actual practice, however, people tend to use files in one directory together. This means that you generally have one directory that you are working in. Most operating systems have a concept of a *working directory* (sometimes called the *current directory*), that is, the directory that you are currently most interested in. The working directory allows the use of relative path names. A *relative path name* is a path name that starts at the working directory rather than the root directory. Files in the working directory have short names and are easy to work with.

working directory
current directory
relative path name

This is an example of the *locality principle*, that is, that use tends to cluster in a small set of objects rather than being uniformly distributed over the set of all the objects you might be using. We saw in Section 11.8.2 that the locality principle applies to the use of memory also.

Figure 16.6 shows a directory tree. This directory tree could appear in a number of operating systems, but the path names would look slightly different from each other. The file represented as a black circle in Figure 16.6 would be named as follows in several operating systems:

- `/nfs/chaco/u1/crowley/os/book/ch02`—UNIX: full path name.
- `nfs/chaco/u2/maccabe/osbook/Ch2`—UNIX: full path name of the alias to the file.
- `book/ch02`—UNIX: relative path name.
- `E:\u1\crowley\os\book\ch02`—MS/DOS: full path name (assuming the disk mounted from chaco is disk E).

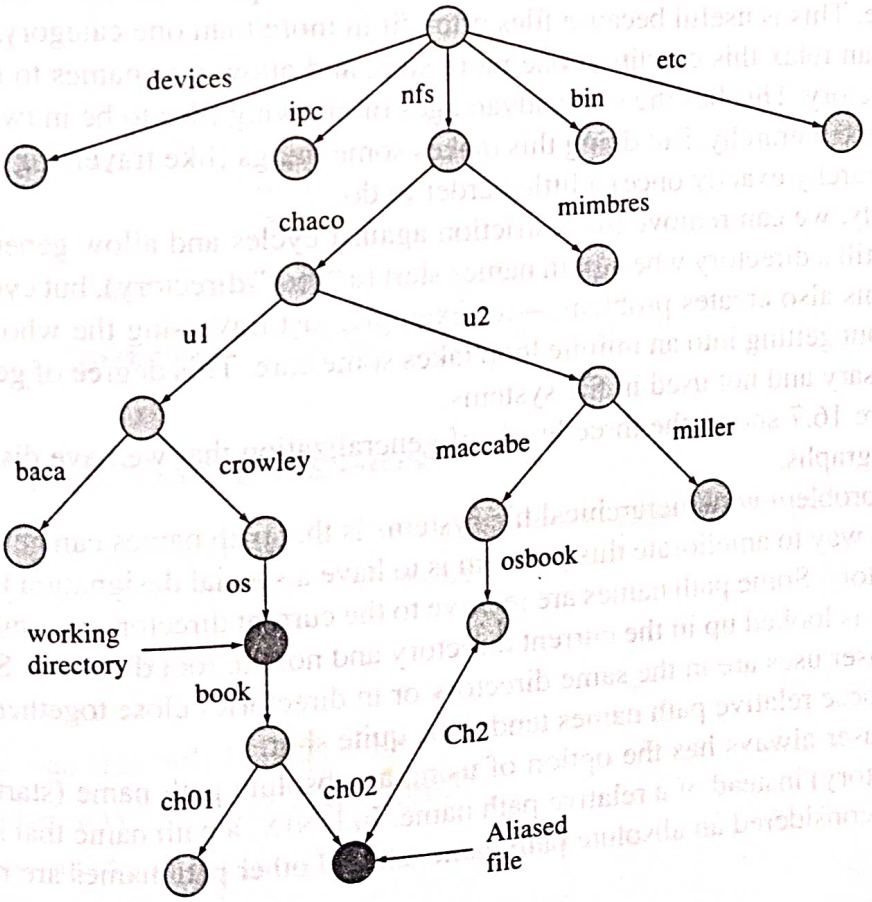


Figure 16.6 A directory tree (with aliases)

- `os\book\ch02`—MS/DOS: relative path name.
- `Users:ul:crowley:os:book:ch02`—Macintosh OS: full path name (assuming the disk mounted from chaco is disk "Users").
- `chaco:[crowley/os/book]ch02`—VMS: full path name.
- `[book]ch01`—VMS: relative path name.

UNIX has a regular path structure and / is always the delimiter. MS/DOS uses the back slash (\) as the delimiter, and Macintosh uses a letter and colon notation to describe different disk volumes. VMS has special delimiters for the directory part of the name.

Path names provide unique names for files. They are made up of component names, where each component is in the directory named by the path that precedes it.

16.3.4 Variations and Generalizations

A hierarchical name space is structured in a tree. A tree is a directed graph with no cycles, such that there is only one path from any node to the root. We can relax the conditions requiring unique paths, and allow two path names to map to the same object as long as the object is not a directory. This allows the leaves of the tree to be shared, thus allowing us to place a file in two different places in the hierarchy at the same time. This is useful because files often fit in more than one category.

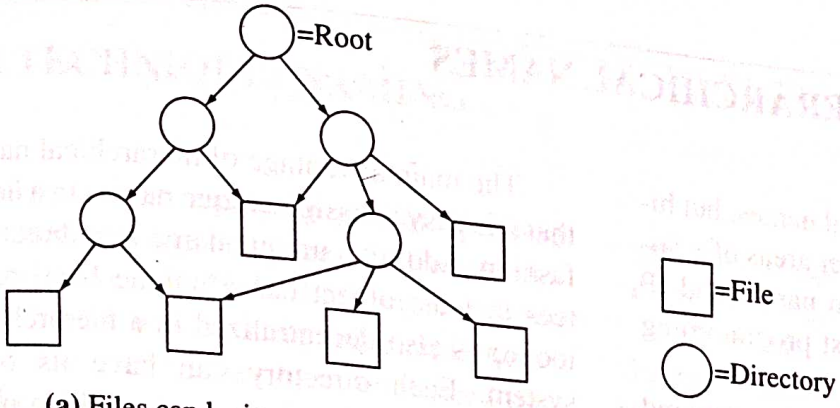
We can relax this condition one more step, and allow two names to map to the same directory. This has the same advantages of allowing files to be in two different places in the hierarchy. But doing this makes some things (like traversing all the files in the hierarchy exactly once) a little harder to do.

Finally, we can remove the restriction against cycles and allow general graphs. There is still a directory where path names start (a "root" directory), but cycles are allowed. This also creates problems—for example, just traversing the whole file system without getting into an infinite loop takes some care. This degree of generality is not necessary and not used in file systems.

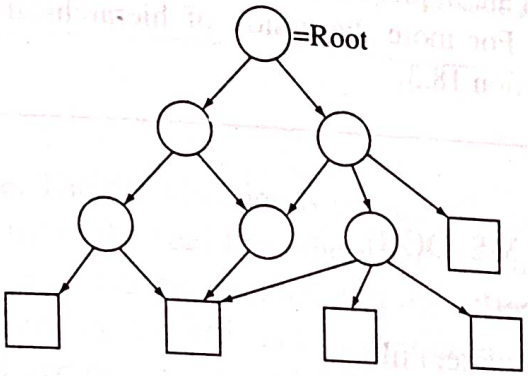
Figure 16.7 shows the three levels of generalization that we have discussed for directory graphs.

One problem with hierarchical file systems is that path names can get to be very long. One way to ameliorate this problem is to have a special designation for the current directory. Some path names are relative to the current directory, meaning that the first name is looked up in the current directory and not the root directory. Since most names a user uses are in the same directory or in directories close together in the hierarchy, these relative path names tend to be quite short.

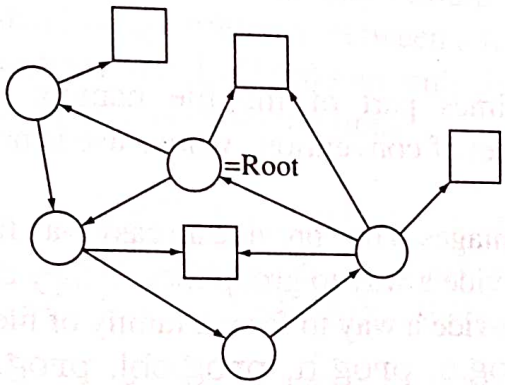
The user always has the option of using an absolute path name (starting at the root directory) instead of a relative path name. In UNIX, a path name that starts with a slash is considered an absolute path name, and all other path names are relative.



(a) Files can be in several directories (tree with shared leaves)



(b) Directories can be in several directories (directed acyclic graph)



(c) Cycles are possible (general graph)

Figure 16.7 Three name space topologies

16.3.5 File Name Extensions

File names are often ended with suffixes that indicate the type of file that they are. Some examples (from several systems) are:

- file.c—a C program file.
- file.txt—a text file.
- file.s—an assembly language file.
- file.obj—an object file (in MS/DOS).
- file.o—an object file (in UNIX).

DESIGN TECHNIQUE: HIERARCHICAL NAMES

All modern file systems use hierarchical names, but hierarchical names are used in many other areas of computer science as well. Internet domain names and IP addresses are hierarchical names. Most programming languages use hierarchical names.

Hierarchical naming systems use compound names with several parts. Each part is looked up in a flat name space, usually called a directory. If this lookup leads to another directory, then the next part of the compound name is looked up in that directory.

The main advantage of hierarchical name space is that it is easy to assign unique names in a decentralized fashion. Adding a unique name to a directory guarantees that the object has a unique local name. Name lookup is also decentralized in a hierarchical naming system. Each directory can have its own lookup method. This makes it easy to combine name spaces that are implemented in different ways.

For more discussion of hierarchical names, see Section 18.3.

- file.exe—an executable file (in MS/DOS).
- file.wk1—spreadsheet worksheet.
- file.tex—tex or latex (a text formatter) file.
- file.mif—Framemaker interchange file.
- file.scm—Scheme program file.
- file.tmp—a temporary file.

file extension

The *file extensions* are sometimes part of the file naming scheme(as in MS/DOS), and sometimes merely a set of conventions whose use is not required by the operating system.

File extensions have several advantages. They provide an easy way for users to see the type of a file at a glance. They provide a way to group files so they can be moved, copied, or deleted as a group. They provide a way to form a family of files with a base name and several extensions (e.g., `prog.c`, `prog.h`, `prog.obj`, `prog.exe`).

File name extensions are a means of indicating the type of the file. They also allow families of related files.

16.3.6 ALIASES

Hierarchical directory systems provide a means for classifying and grouping files. All the files in a single directory tend to be related in some way, but sometimes a file is related to several different groups and it is an inconvenient restriction to have to place it in just one group. This problem can be handled with the concept of a file alias. A *file alias* is a file name that refers to a file that also has another name. This means that a file will have two or more absolute path names (one for the original name and one for each alias). Aliases are shown in Figure 16.6.

file alias

Every file has a "real" path name, that is, a path name that leads directly to the file descriptor. Most systems implement aliases as a file that contains the real path

DESIGN TECHNIQUE: NAMING

Naming of objects is an important part of operating systems and programming languages. There are various types of names and methods of naming. A naming system has to have a way of assigning unique names and a way of determining the object referenced by a name (called *name resolution*).

Internal names are assigned by the system and are generally integers. External names are assigned by users and are generally character strings.

There are two methods of generating unique names. In the uniqueness check method, you propose a name and ask if it is unique. In the name assignment method, you request a unique name from the naming system.

For more discussion of naming, see Section 18.4.

name of the file. The file also needs some indication that it is an alias so that the file system will jump to the real file whenever the alias file is opened. In this case, one of the names is the real path name, and the other names are aliases and are treated differently in some cases. This is true of aliases in the Macintosh operating system, and it is true of *symbolic links* in UNIX.

symbolic link

UNIX does not keep the file descriptor in the directory entry for a file, so it can have a different kind of link that is called a *hard link*. With hard links, all path names are "real" and there is no difference between a file name and an alias. Because of the way they are implemented, hard links are only possible within a single disk partition, and so are not as useful as symbolic links.

hard link

Aliases allow you to place files in more than one location in the file naming hierarchy.

16.4 FILE SYSTEM OBJECTS AND OPERATIONS

We have seen that the basic abstraction provided by the operating system is the file, that is, a named array of bytes on disk. For operating on files, we will add a second abstraction which we will call an open file. An *open file* is a source of bytes (for a file opened for reading) and a sink for bytes (for a file opened for writing). The open command creates an open file object that is attached to the file that was opened. Bytes read from the open file object come from that file, and bytes written to the open file object go to that file.

open file

An important reason for making the distinction between files and open files is to allow us to generalize this concept and allow for an open file that is attached to a terminal keyboard, an output window, a network connection, etc. (We saw how this worked in Chapter 3.) That is, anything that is a source of bytes or can accept bytes can be connected to an open file object. This simplification makes I/O more consistent.

So the file system will implement three objects:

- Files—that hold persistent data.
- Open files—that allow access to files.
- Directories—that name the files.

Each object has some data associated with it and some operations that can be performed on it. The file data consist of the contents of the file itself and the meta-data that the operating system keeps about the file. The open file data consists of the file it is associated with (or the device or other object, in the case of our extended open files) and the file pointer (or the current location we are reading from or writing to). The directory data is the list of names in the directory and the objects they refer to (or a pointer to the object they refer to). File names will not be objects in and of themselves. They are constructed out of sequences of directory entries.

Now we will look at the operations possible on these objects. We will take as an example the UNIX system call interface, since it is representative of the types of file system operations. Figure 16.8 is a table of UNIX operations applicable to files.

The *open* and *create* (`creat` in UNIX) calls prepare to access the contents of the file.[2] The *status* (`stat` in UNIX) call reads the meta-data about the file, and the *change owner* (`chown` in UNIX) and *change mode* (`chmod` in UNIX) change parts of the meta-data about the file (the owner and access permissions respectively). Some of the meta-data is changed implicitly by other operations. For example, opening a file will change its time of last access.

Figure 16.9 is a table of UNIX operations applicable to open files. The *read* and *write* calls read and write the file contents. The *seek* call sets the file location for the next read or write. The *close* call breaks the connection between the open file and the file, and destroys the open file. The *duplicate* call is necessary only because of the way UNIX passes open files to child processes and its convention that standard input and output are open files 0 and 1.[3] The *file lock* (`flock` in UNIX) call ensures exclusive access to the file. In this case, the file system is providing process synchronization operations. The *file control* (`fcntl` in UNIX) call controls various aspects of using the open file, such as whether reads from the open file are blocking or nonblocking. The *file sync* (`sync` in UNIX) call is related to the file system implementation and ensures that all important information about the file is written to permanent storage on the disk. The *file status* (`fstat` in UNIX) call has the same functionality as the one for files. It is useful when you have an open file but do not know the name of the file it is associated with. Finally, the *pipe* system call sets up a form of interprocess communication. It is included here only because pipes merge IPC into the file access protocols.

Figure 16.10 is a table of UNIX operations applicable to directories and file names. The file naming system is logically separate from the file storage system. However, directories are usually implemented as files, so their implementations are intertwined. These operations allow you to attach an alternative name to a file (*link*), unattach a name from a file (*unlink*), change the name of a file (*rename*), and create

[2]Actually, in UNIX, file creation is done using the *open* system call, with a flag telling it to create the file.
[3]Note that it is the open file that is duplicated, not the file itself.

Operation	Arguments	Return Value	Comments
Open	File name	Open file	Creates an open file attached to this file
Create	File name	Open file	Creates a new file and an open file attached to it
Status	File name	File info	Returns the file's meta-data
Access	File name, access type	Legal?	Checks if the access violates the protection rules
Change mode	File name, new mode	Success?	Changes the file's protection
Change owner	File name, new owner	Success?	Changes the file's owner and group

Figure 16.8 Operations on files in UNIX

Operation	Arguments	Return Value	Comments
Read	Open file	Success	Reads the file
Write	Open file	Success	Writes the file
Seek	Open file	New position	Changes the file pointer
Close	Open file	Success	Destroys the open file and updates the attached file
Duplicate	Open file	Success	Makes a copy of the open file
File lock	Open file	Success	Locks or unlocks the file the open file is attached to
File control	Open file	Success	Various control operations
File sync	Open file	Success	Write out file data to disk
File status	Open file	File info	Returns the open file's meta-data
Pipe	—	Two open files	Creates two open files that are attached to each other

Figure 16.9 Operations on open files in UNIX

Operation	Arguments	Return Value	Comments
Link	File name Alias name	Success	Create a directory entry that points to a file
Unlink	File name	Success	Remove a directory entry that points to a file
Rename	Old name New name	Success	Change the name of a file in a directory
Make dir	Dir name	Success	Create a directory
Remove dir	Dir name	Success	Remove a directory

Figure 16.10 Operations on directories in UNIX

(*make directory*—`mkdir` in UNIX) and destroy (*remove directory*—`rmdir` in UNIX) directories.

The *create* call creates a new file and also adds its name to the file name system. It was listed as a file operation, but it also affects a directory.

Like all the parts of an operating system, the file system implements objects and operations on those objects.

16.5 FILE SYSTEM IMPLEMENTATION

To implement the file system, we have to implement the three objects the file system provides for users: files, open files, and directories. Each one will be implemented as a data structure and a set of procedures to manipulate the data structure.

16.5.1 FILE SYSTEM DATA STRUCTURES

Figure 16.11 shows the main data structures in a file system and how they are connected to each other.

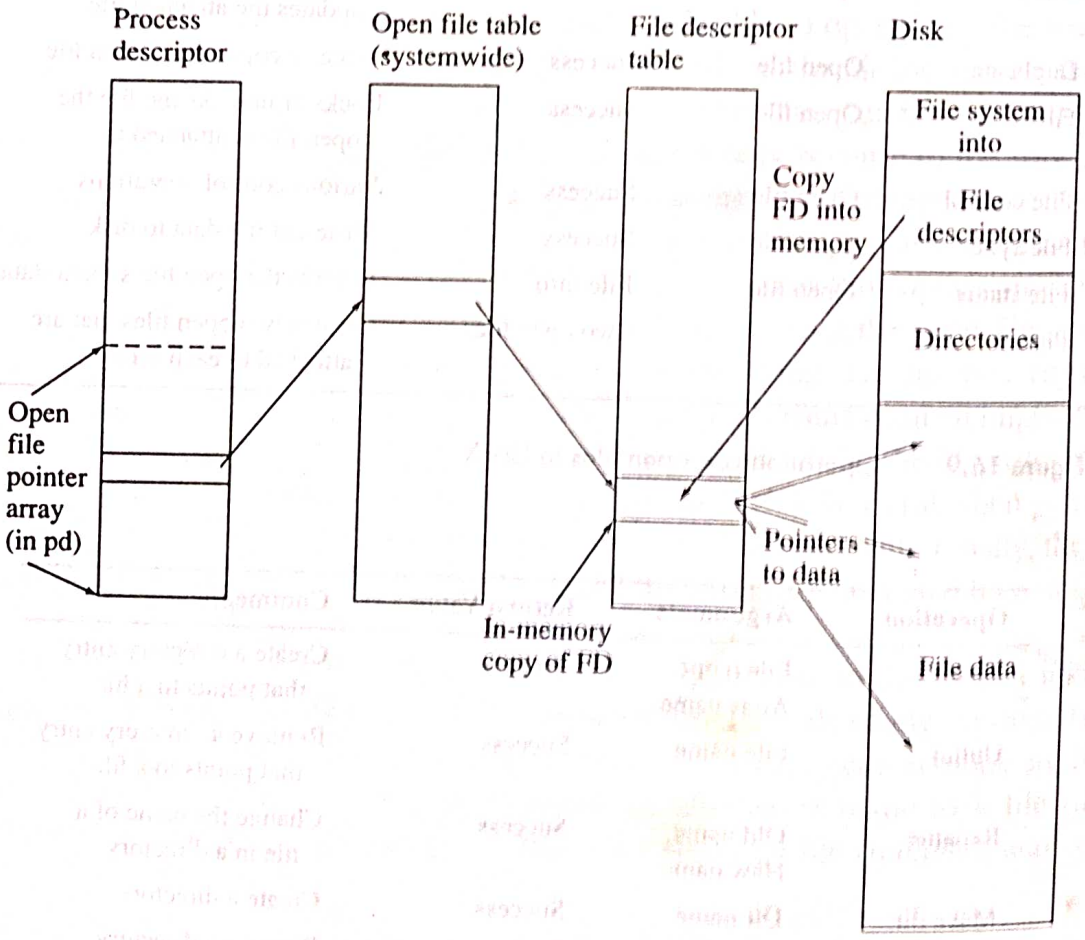


Figure 16.11 File system data structures

The *open file table* contains a structure for each open file. An open file structure contains the following information: open file table

- The current file position (the location in the file of the next read or write).
- Other status information about the open file—whether it is open for reading or writing, the type of open file (file, device, pipe), whether the file is locked, etc.
- A pointer to the file descriptor for the file that is open (or to the device driver if the open file is connected to a device, or to a pipe data structure if the open file is connected to a pipe, etc.).

An open file is connected to a data source or data sink. Normally that is a file, but it can also be a device or a pipe, or possibly something else. Open files are created when an `open` or `creat` system call is made, and destroyed when a `close` system call is made. These calls allocate and free open file data structures. The diagram shows them in an array, but they may be in a linked list or other data structure.

A file is opened by a process and is connected with that process. When a file (or device or pipe) is opened, the open file structure is allocated and a pointer to that structure is placed in a table in the process descriptor of the process opening the file. This is called the *open file pointer table*, or sometime the *per process open file table*, and consists of an array of pointers to open file structures. The file identifier returned by the `open` system call is an index into that array. So when you use the file in a `read` system call, the system will index into the array and get a pointer to the open file structure. open file pointer table per process open file table

Most open files are connected to files. Files exist on disk and consist of two parts. The first part, the data structure that represents the file, is the *file descriptor*, and it contains all the meta-data about the file. The second part is the file data itself, kept in disk blocks. Some of the items in the file descriptor point to the file data. A file descriptor data structure contains the following information: file descriptor

- The owner of the file.
- File protection information.
- Time of creation, last modification, and last use.
- Other file meta-data.
- The location of the file data—this includes the device the file is stored on (that is, which device driver to use to read and write the file blocks) and the disk block addresses on the disk where the file data is located.

Principally, the file descriptor contains all the file meta-data. In addition it contains information about the location of the data in the file. This will consist of the addresses of one or more disk blocks. We will see later (in Section 17.3) how this information is stored. For now, we will just assume that the information is sufficient to find that file data.

Files are persistent, and so the file descriptors must be kept on disk along with the file data. But when you access a file, you have to refer to the information in the file descriptor over and over again, and so it is more efficient to store it in memory while the file is open. So, when a file is opened, the file descriptor is read into memory and put in a file descriptor table. This is shown in the diagram as an array, but it

is more likely to be a linked list and not in contiguous storage. This table is a cache for the file descriptors on disk. When the file is closed, the updated file descriptor is written back out to disk. Since there is a copy in memory of the file descriptor for every open file, the open file structure can point to the file descriptor.

In Section 17.1, we will examine the data structures of a file system on disk, but we have shown a rough version of them in Figure 16.11. There will be some information about the entire file system—a file system descriptor. The file descriptors will be on disk. They are shown here all together, but that may or may not be the case. In some systems, the file descriptors are kept in the directories. The directories are also shown all together, but more likely they are actually kept in the data block area. The file data blocks take up the bulk of the disk space. This is where all the file data is kept (plus maybe the file descriptors and the directories).

All persistent file system structures are stored on disk, but they are cached into main memory when they are in active use.

16.5.2 File System Organization and Control Flow

Figure 16.12 shows the control flow for the `open` system call. The left side of the diagram shows the data structures affected. The right side of the diagram shows the file system modules that are called. The vertical arrows connecting the modules represent procedure calls and returns (down for calls and up for returns). The dashed arrows between the two sides show which data structures the modules access and modify.

The open system call is handled by the file naming module of the file system. The first thing it does is to allocate the open file data structure and link it up to an entry in the process's open file pointer array (which is kept in the process descriptor). Then the file must be found from the file name. This activity takes the file name, which is a path name with one or more components, and finds the file descriptor for that file by searching a sequence of directories for successive component names in the file path name. We will defer a more complete description to Section 16.5.7, where we will describe in more detail how directories are implemented.

After this lookup has finished, the file descriptor for the named file is in the file descriptor table in memory, and the open file structure will be linked to it. The open system call returns the index into the open file pointer array in the process descriptor.

Figure 16.13 shows the control flow for the `read` system call. The left side of the diagram shows the data structures and how they are linked together. The right side of the diagram shows the file system modules that use these data structures. The downward arrows represent procedure calls. Each downward arrow is broken into two parts, with the procedure call in between the two parts. The upward arrow beside it represents the return from the procedure call. Sometimes the upward arrow is labeled with the name of the returned value. The lines from the right side

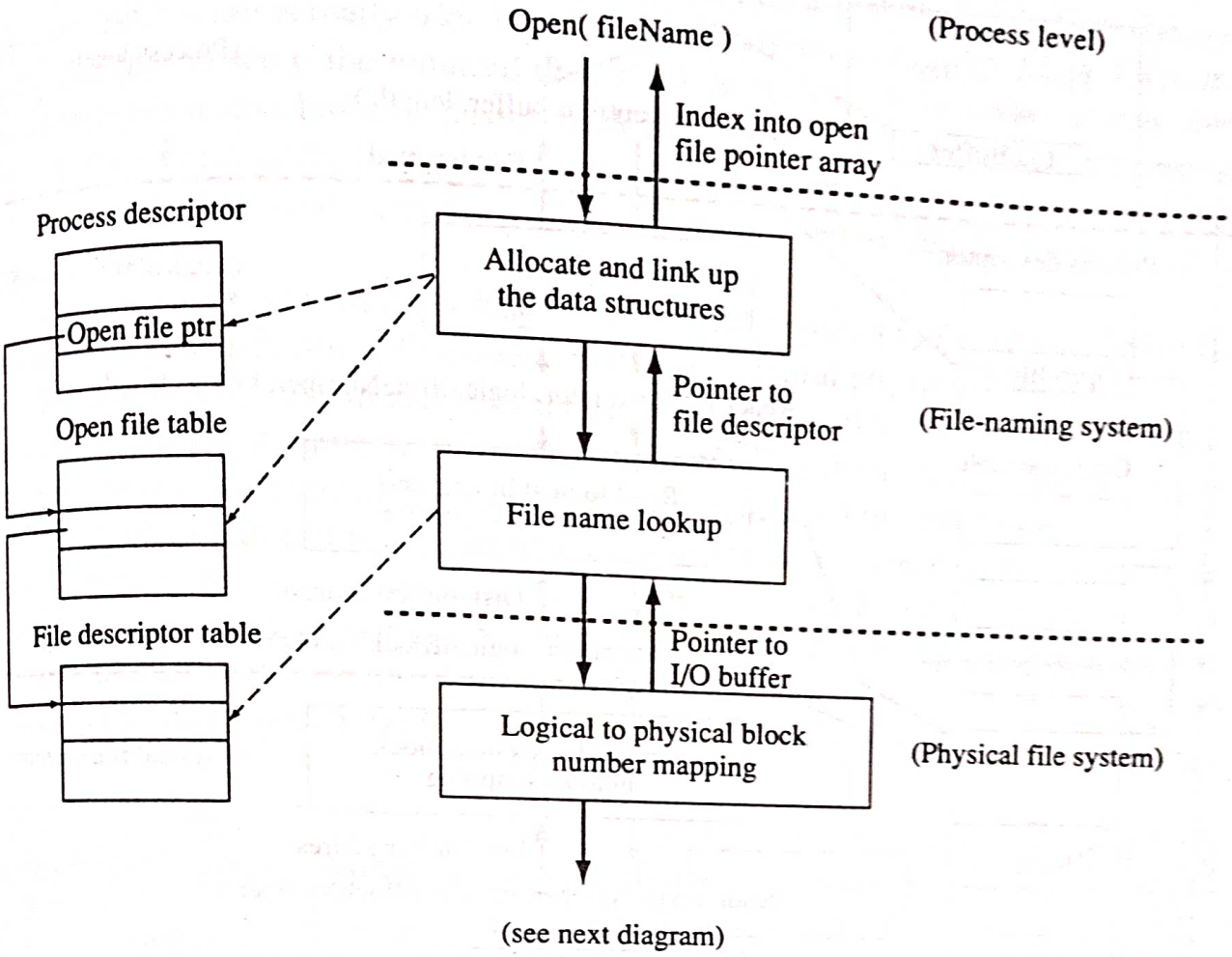


Figure 16.12 File system control flow for an open

to the left side indicate which data structures are accessed and modified by the various modules.

The horizontal, dotted lines divide the levels of the file system. The process level initiated the system call. The *logical file system* deals with files at the logical level. This includes logical byte offsets, logical blocks, and logical block numbers. The *physical file system* deals with physical blocks. The *I/O system* deals with devices.

Logical File System The arguments of the `read` system call are the file identifier and the buffer. The file system uses the file identifier to find the open file structure. At that point, it checks to be sure the access is legal (that the file is open for reading in this case), picks up the logical byte number in the file to read from (this is kept in the open file structure), follows the link to get to the file descriptor, and passes the read on to the next level of the file system.

The next level is in charge of sequencing the read. It figures out which logical blocks in the file will be required. A read request might span several logical blocks or it might be only a part of a logical block. It will read these logical blocks one at a time and transfer the requested data to the user's buffer. Only parts of the logical blocks may be required. The reading of each logical block is handled by the next level down in the file system.

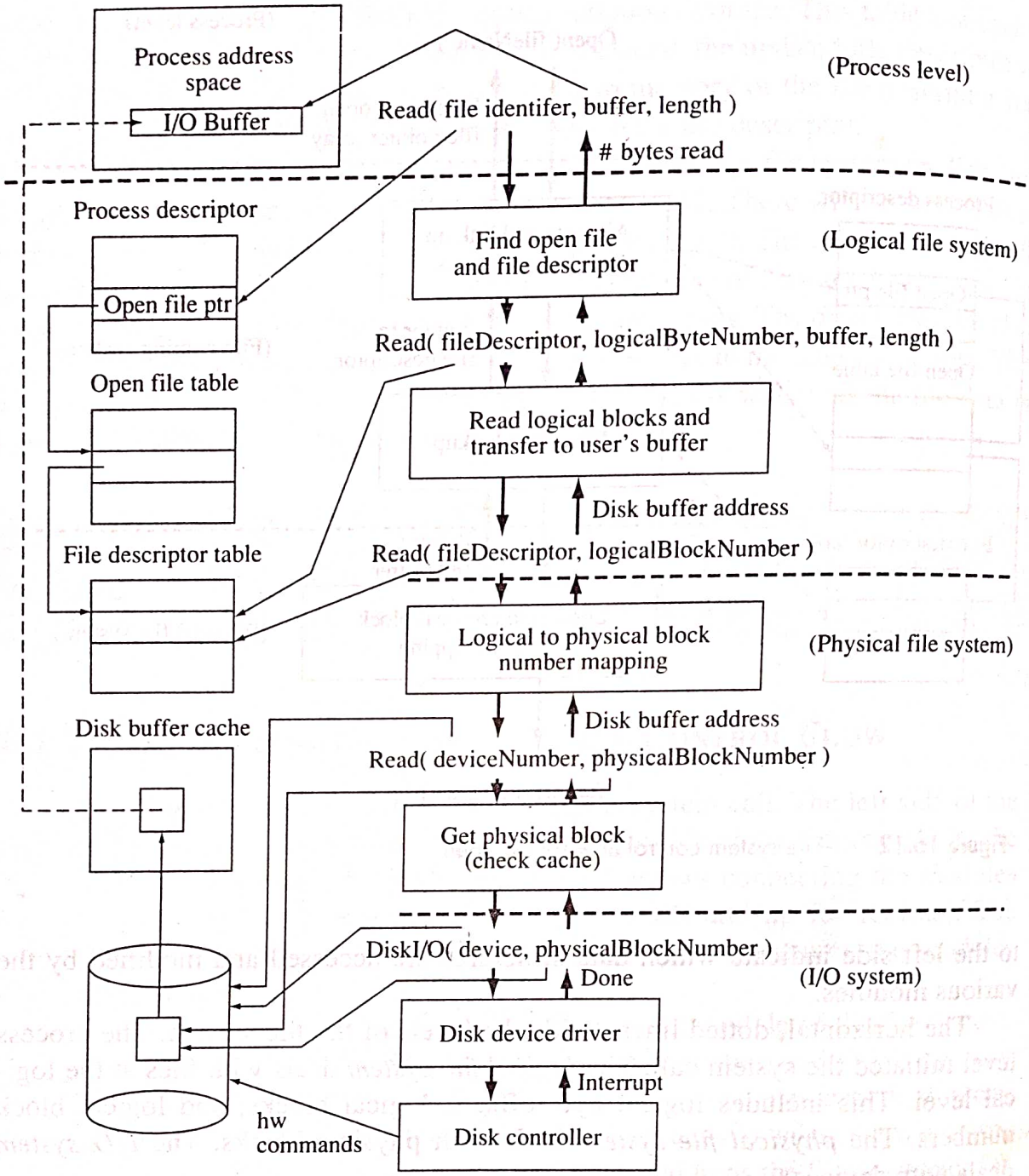


Figure 16.13 File system control flow for a read

Physical File System Now we move from the logical file system (which deals in logical blocks) to the physical file system (which deals with physical blocks). First, the logical block number is converted to a device number and a physical block number on that device. This translation uses the information on file block locations found in the file descriptor. The format of this information varies, depending on how the file system has decided to manage disk storage. We will examine the possible design choices in Section 17.3.[4] The result of this stage is a device number and a block number on that device, and this is passed down to the next level of the physical file system.

[4] It is likely that other disk blocks will have to be read in order to do the conversion from logical block number to physical block number.

Now the task is to get the required disk block into a system disk buffer. The system disk buffers are actually a disk cache (as described in Section 15.7). First, the module checks to see if the required disk block is already in the cache. If it is, then the address of the disk buffer is passed back immediately, and no disk I/O is done. If it is not, then a disk cache buffer is allocated to this block (and so another disk block is evicted from the cache), and the device driver is called to transfer the block from disk to memory.

The logical file system level may make several requests to the physical file system in order to satisfy one I/O request (if it spans several physical blocks).

The file naming system also uses the physical I/O level to search directories during a name search. A name search on a long path name through some large directories may involve the reading of dozens (or even hundreds) of disk blocks.

A `write` system call is not handled much differently; except for the fact that the data is going in the other direction, most of the steps are the same. If a partial disk block is written, then the current contents of the entire disk block must be read first, the parts that are written to are changed, and the whole block is written back out. This is necessary since the disk will only write in full block units.

The file system is implemented in levels: the logical file system, the physical file system, and the I/O system.

16.5.3 Connecting to Device Drivers

A remaining question is how the physical file system knows which device driver to call. Devices are identified by a device number. There is a file system data structure called a *device switch*, which is an array of structures, each describing a device driver. The structures in the device switch have the addresses of each entry point in the driver. This includes the entry points for open, close, strategy, read, write, device control, etc.

device switch

If the file system has the device number, it can use it as an index into the device switch and get the address of the device driver entry point it needs to call. The file descriptor contains the device number the file is on, and this is used to find the correct device driver to call. Figure 16.14 shows how this works.

Remember that *special files* were names in the file system that referred to devices rather than files. So to set up a special file connected to a device, you need to do three things:

1. Load the device driver into the operating system.
2. Put an entry into the device switch that points to the driver.
3. Put a name in the file system of a special file which contains the index of the entry for the device in the device switch.

Steps 1 and 2 are usually done when the operating system is linked, but they can be done dynamically by dynamically loading the device driver and modifying the device switch. Once this is set up, then you can use the device by opening the special file.

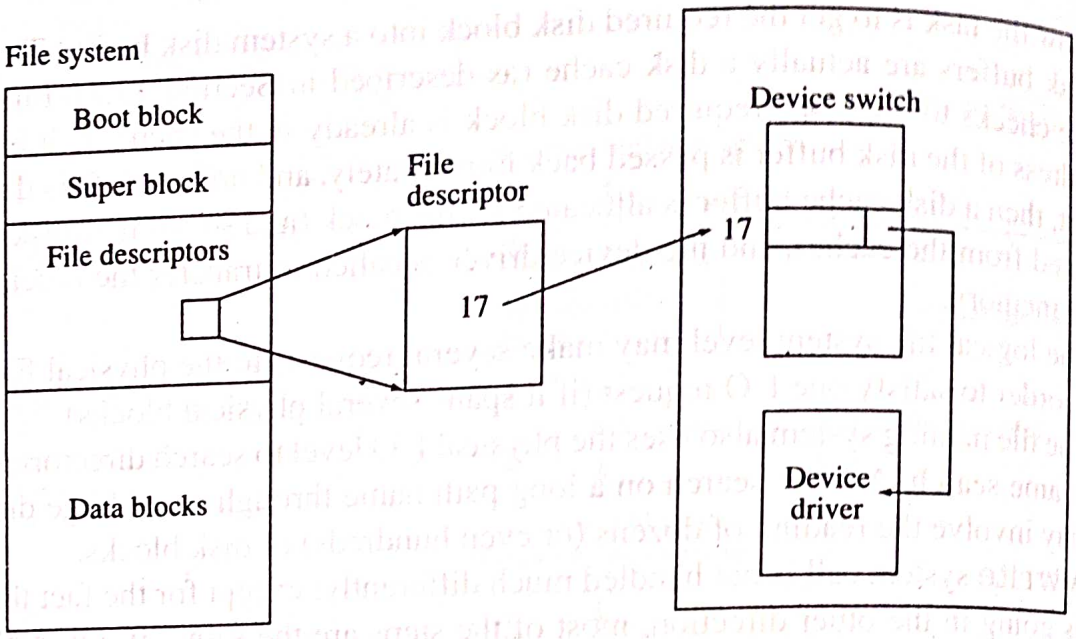


Figure 16.14 Connecting files and devices through the device switch

16.5.4 Reading and Writing from Special Files

In Section 16.5.2, we looked at how to read from a file. In this section, we will discuss what happens if you read from a special file. The special file might be a device or a pipe.

Once the file system finds the file descriptor, it looks at the mode flags of the file descriptor. These will indicate whether the descriptor is for a file, a block device, a character device, or a pipe. For a file, you proceed as described in Section 16.5.2. For a block device or a character device, you get the device number from the file descriptor and call the read entry point for the appropriate device driver. You get this entry point from the device switch. For a pipe, you call the pipe processing code in the file system.

16.5.5 Other File System Calls

Figure 16.15 shows the effect of the `fork` system call on the open files in the system. Each open file in the forking process is duplicated, and the duplicate is placed in the open file pointer array of the forked process. This means that each open file structure in the systemwide open file table is pointed to twice. The structure will not be deallocated until both processes have closed the file.

Figure 16.16 shows the effect of the `stat`, `fstat`, `access`, `chmod`, `chown`, and `seek` system calls on the file system data structures. The `stat` and `fstat` system calls find the file descriptor and return the meta-data in it. The `access` system call finds the file descriptor and checks the protection information. The `chmod` and `chown` system calls change the meta-data in the file descriptor and arrange for it to be written back out to disk. The `seek` system call changes the file pointer in the open file structure.[5] The `file control` and `file lock` system calls change information in the file descriptor.

[5] It is an interesting and ironic fact that the seek system call is just about the only file system call that will *never* read the disk or cause a disk seek.

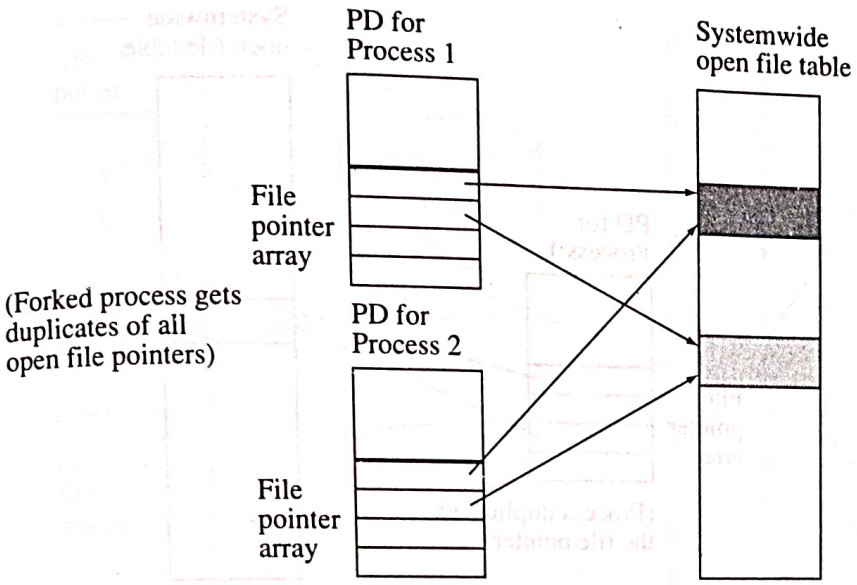


Figure 16.15 The effect of a fork on the file system data structures

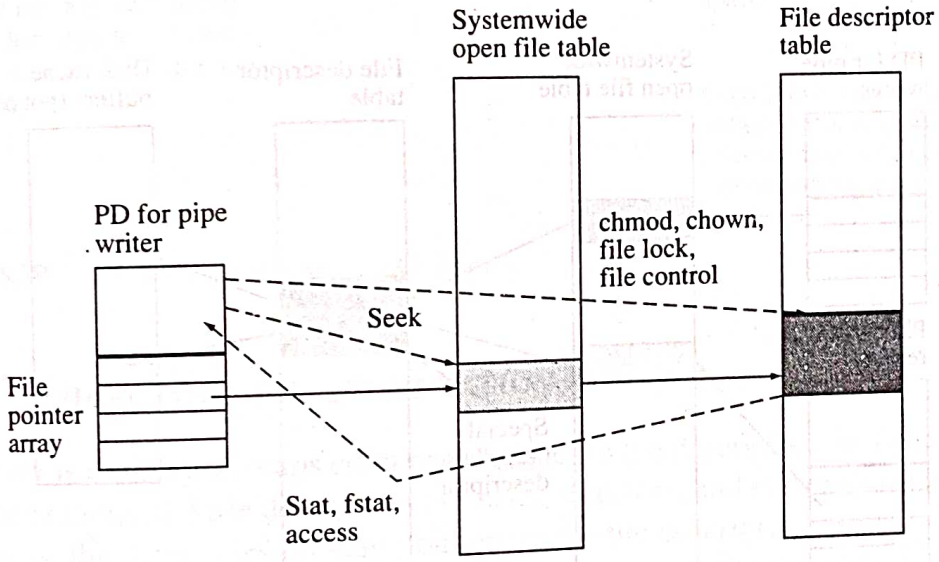


Figure 16.16 How some system calls use and change the file system data structures

The duplicate system call allocates an entry in the process's open file array, and initializes it to point to the same open file structure as the old one did. Figure 16.17 shows the effect of the duplicate system call on the file system data structures.

The pipe system call allocates two open file structures and a file descriptor structure. This is a special type of file descriptor that is not associated with a file, but instead represents the pipe. Figure 16.18 shows the effect of the pipe system call on the file system data structures.

The file-related system calls, like all system calls, access and update various system tables. The read and write system calls also transfer data.

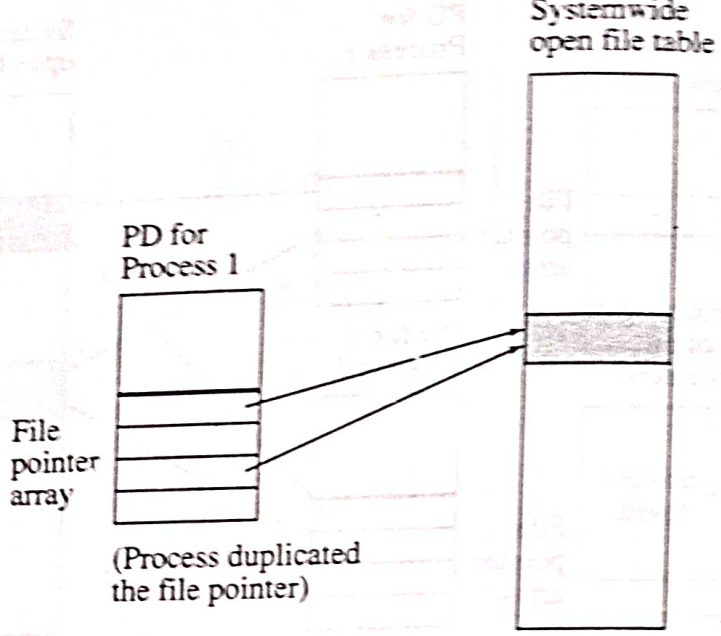


Figure 16.17 Effect of the duplicate system call on the file system data structures

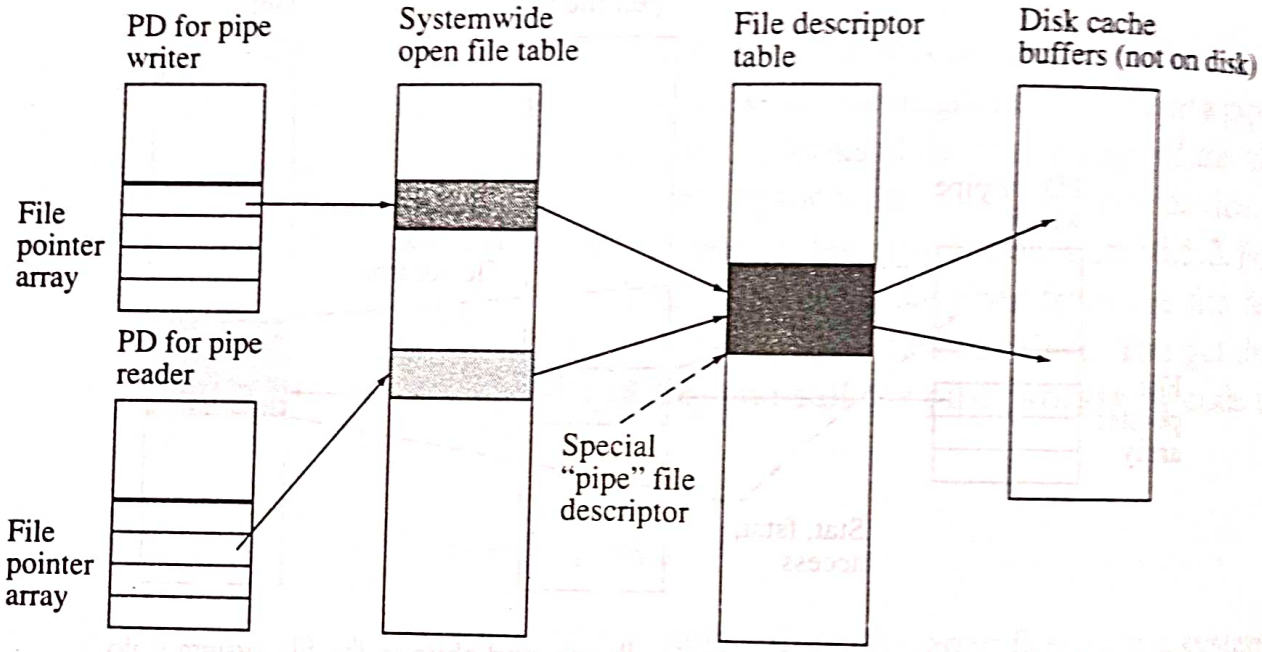


Figure 16.18 Effect of the pipe system call on the file system data structures

16.5.6 Avoiding Copying Data

Using the method of implementing read (and write) by using disk cache buffers implies a memory-to-memory copy of the data (from the disk cache buffer to the user's I/O buffer). We could avoid this copying by writing data directly into the user's buffers, or we could use the virtual memory system to move the data from the system's address space to the user's virtual address space by just changing the user's page table, without any data movement. Figure 16.19 shows these two ways of avoiding copying. These techniques can only be used for full disk blocks.

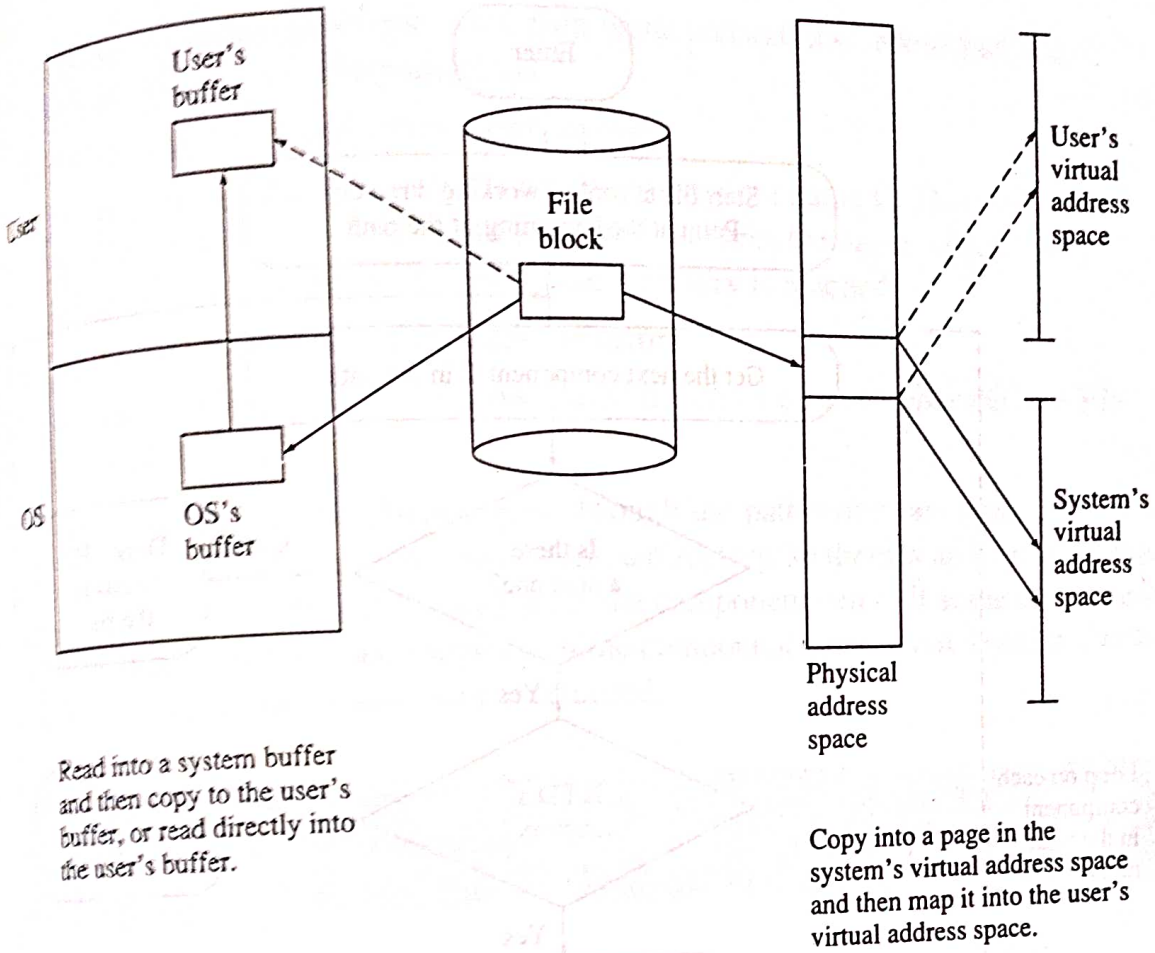


Figure 16.19 Avoiding copying of I/O data

16.5.7 Directory Implementation

A directory is a table that maps component names to file descriptors. In some file system organizations, the file descriptors are in the directory, and in some file system organizations, the directories contain pointers to the file descriptors. We will examine the tradeoffs of this issue in Section 17.2.1. In either case, the directory might have only a few names in it, or it might have thousands. We cannot predict how large it will be, so it is difficult to allocate the space for the directory. It turns out that the easiest way to implement directories is by keeping each directory in a file.

We will assume that the directory contains a table of component names and a pointer to the file descriptor the name identifies. We keep this information in a file, and we look up a name in a directory by reading the file. This way, we can use all the mechanisms for handling files that we are going to need anyway to implement files.

Figure 16.20 shows the flow chart for a path name lookup, and the algorithm presented next describes the process in words.

1. If the path name starts with the '/' then let FD be the root directory and move the name pointer past the '/'. Otherwise, let FD be the current working directory and start at the beginning of the path name.
2. If we are at the end of path name, then return FD.

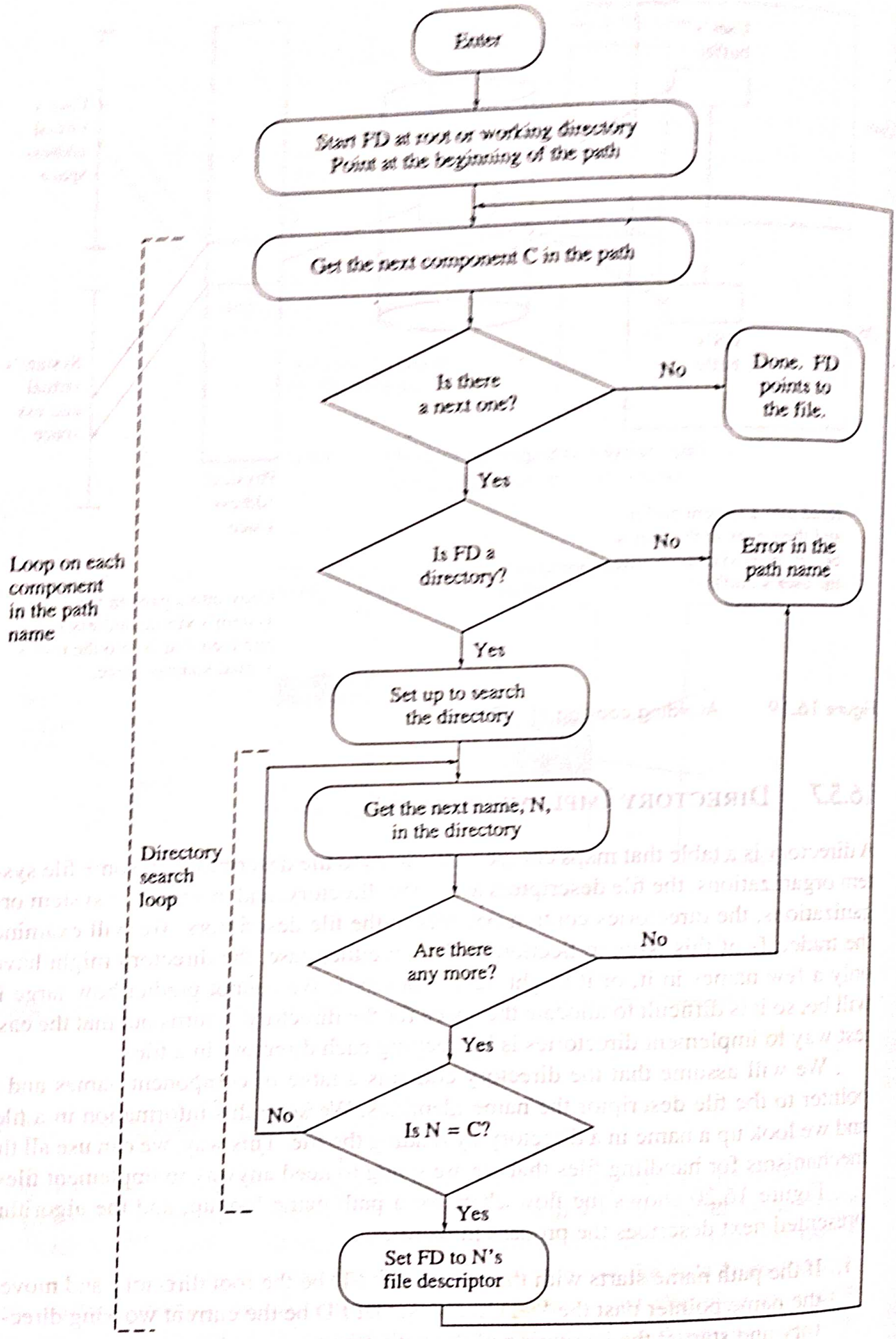


Figure 16.20 Algorithm for a path name lookup

3. Isolate the next component in the path name and call it C. Move past the component name in the path name.
4. If FD is not a directory, then return an error.
5. Search through the directory FD for the component name C. This involves a loop that reads the next name/fd pair and compares the name with C. Loop until a match is found or the end of the directory is reached.
6. If no match was found, then return an error.
7. If a match was found, then its associated file descriptor becomes the new FD. Go back to step 2.

The outer loop of the process goes through the path name one component at a time. Each component name is looked up in a directory, so there is an inner loop that searches through the directory looking for the component name. If some component (other than the last) is not a directory or if the component name is not found in the directory, then a bad path name error is returned.

Directories are usually implemented as files.

*16.6 AN EXAMPLE FILE SYSTEM IMPLEMENTATION

In this section, we will look at some of the code necessary to implement a file system. We have explained the principles behind the code already, and this should help in understanding the code.

16.6.1 System Constants and Global Data

First we need a few more constants and type definitions. The ones shown here are used in several places in the file system code. Other constants and type definitions are made in later sections, so they will be closer to the code that uses them.

SYSTEM CONSTANTS AND GLOBAL DATA

```
const int BlockSize = 4096;
// only one disk for now
const int DiskNumber = 0;
// type for a disk block in memory
typedef char * BlockBuffer[BlockSize];
// type for block numbers = disk block addresses
typedef int BlockNumber;
```

We will write the code to allow for more than one disk, but whenever a disk number is required we will just use the constant DiskNumber because our hardware has only one disk. The BlockBuffer type defines a disk block in memory. Block numbers are usually 16- or 32-bit integers. This type will apply to both logical block numbers and to physical block numbers. It would probably be useful to distinguish between them in the type system. In C++, this would require us to define a class for each.

16.6.2 Disk Cache

We will keep a cache of recently used disk blocks in memory. This is called the *disk cache*. The disk cache buffers will also serve as I/O buffers to read in and write out disk blocks.

DISK CACHE

```
const int DiskCacheSize = 200;
const int BlockBufferHashTableSize = 200;

// type for a disk block header
struct BlockBufferHeader {
   int useCount; // how many processes are using this block
   int wasModified; // indicates the buffer has been written since
                    // it was read from disk
   BlockBuffer * block; // the buffer itself
   int DiskNumber; // the disk it comes from
   BlockNumber blockNumber; // the block number on the disk
   BlockBufferHeader * next; // the hash table link
};
// the disk cache
BlockBufferHeader diskCache[DiskCacheSize]; //the buffers
int nextHeaderToReplace; // for FIFO replacement of cache buffers

//the hash table for looking up disk blocks quickly
BlockBufferHeader * blockBufferHashTable[BlockBufferHashTableSize];

int
HashBlockAddress( DiskNumber dn, BlockNumber pbn ) {
   return (dn + pbn) % BlockBufferHashTableSize;
}

BlockBufferHeader *
LookUpInDiskCache( DiskNumber dn, BlockNumber pbn ) {
   int hash = HashBlockAddress( db, pbn );
   BlockBufferHeader * header = blockBufferHashTable[hash];
   // get the linked list for this hash bucket and search it
   while( header != 0 && (header->bn != pbn || header->dn != dn) ) {
      header = header->nextHeader;
   }
   return header;
}
```

```
BlockBufferHeader *
FindCacheBufferToReuse( void ) {
  for( int i = 0; i < DiskCacheSize; ++i) {
    if( ++nextDiskCache >= DiskCacheSize )
      nextDiskCache = 0;
    if( diskCache[nextDiskCache].useCount == 0 )
      break;
  }
  if( diskCache[nextDiskCache].useCount != 0 )
    return 0; // no free cache buffers to reuse
  else {
    BlockBufferHeader * header = &(diskCache[nextDiskCache]);
    if( header->wasModified )
      DiskIO( WriteDisk, header->blockNumber, header->buffer );
    return &(diskCache[nextDiskCache]);
  }
}

BlockBufferHeader *
GetDiskBlock( DiskNumber dn, BlockNumber pbn ) {
  // See if the disk block is already in the disk cache
  BlockBufferHeader * header = LookUpInDiskCache( dn, pbn );
  if( header == 0 ) {
    // It's not in the disk cache so read it in.
    header = FindCacheBufferToReuse();
    header->useCount = 0;
    header->dn = dn;
    header->pbn = pbn;
    DiskIO( ReadDisk, pbn, header->buffer );
  }
  ++header.useCount;
  return header;
}

void
FreeDiskBlock( BlockBufferHeader * header ) {
  --header->useCount;
}
```

We need to keep some meta-information about each disk block buffer, so we define a `BlockBufferHeader` structure. It will contain a pointer to the actual disk buffer. It also contains the disk number and block number of the disk block contained in the buffer. We also have to keep track of whether the buffer has been modified since it was read from the disk. If so, we will have to write it out again before we reuse the buffer.

The disk cache will contain the most recently used blocks. Some of these blocks will be in active use and some will not. The ones that are not in active use are kept in the disk cache in case they are needed again soon. We only replace a block in the disk cache when we need to make space for another block that is needed immediately. We keep a use count to keep track of the buffers that are in active use and cannot be replaced.

When a kernel-mode process needs to use a disk block, it will call `GetDiskBlock`. `GetDiskBlock` will return after the disk block is in memory (in a

disk block buffer). The kernel-mode process will use the block for a while, and then indicate that it is done with the disk block by calling FreeDiskBlock. We maintain a use count of how many kernel-mode processes are actively using the disk block. While this use count is positive, we will not reuse the buffer.

GetDiskBlock will first check to see if the requested block is already in a buffer. This is done by the procedure LookUpInDiskCache. We use a simple hash table to make this search faster. A hash table entry contains the address of a linked list of buffers that hash to that value. This simple scheme makes block lookup in the cache quite fast.

If the block is not found in a buffer, then we have to choose a buffer to reuse for this block. This is done by the procedure FindCacheBufferToReuse. We use a first-in, first-out scheme and look for a buffer whose use count is zero.

Once we have a buffer to read into, we call the disk driver, DiskIO, to read in the block. DiskIO will not return until the block transfer is complete.

FreeDiskBlock only has to decrement the use count. The block will be reused when needed if the use count is zero.

16.6.3 File Descriptors

File descriptors are kept on disk with the file data, but when a file is in active use we read the file descriptor into memory and keep it in a table of active file descriptors. We could read it in each time we needed it, but this would be very slow.

File descriptors are usually kept in a special area of the disk. This special area will start with physical block 2 of the disk. File descriptors do not use an entire disk block each, but are packed (in this system) 32 to a block. We will learn more about how file systems are structured in Section 17.1.2.

FILE DESCRIPTORS

```
const int DirectBlocksInFD = 10;
const int NumFileDescriptors = 100;
const int BlockNumberSize = sizeof(BlockNumber);
const int BlocksMappedByIndirectBlock = BlockSize/BlockNumberSize;
const int BlocksMappedByDouble IndirectBlock
    = BlocksMappedByIndirectBlock * BlocksMappedByIndirectBlock;
// type for indirect blocks which are an array of disk block numbers
typedef BlockNumber IndirectBlock[BlocksMappedByIndirectBlock];

//type for a file descriptor in memory
struct FileDescriptor {
    int length; // length of the file in bytes
    int nlinks; // number of links to the file
    BlockNumber direct[DirectBlocksInFD]; // direct blocks
    BlockNumber single_indirect; // single indirect block
    BlockNumber double_indirect; // double indirect block
    // we won't use these fields but they are typically present
    int owner;
    int group;
    int time_created;
```

```
    int time_last_modified;
    int time_last_read;
    int pad[13]; // to pad it out to 128 bytes = 32 words
    // The following fields are not part of the file descriptor as it
    // is kept on disk. They are only used for the in-memory version.
    int useCount; // how many open files point to this file descriptor
    int disk; // disk number the file descriptor comes from
    int fd_number; // file descriptor number of the disk
};
const int FileDescriptorSize = 128;
const int FileDescriptorsPerBlock = BlockSize / FileDescriptorSize;

// the in-memory table of file descriptors
FileDescriptor fileDescriptor[NumFileDescriptors];

FileDescriptor *
GetFileDescriptor( int disk, int fd_number ) {
    // find the fd (or a free slot) the file descriptor table is in
    int i;
    free_slot = -1;
    for( i = 0; i < NumFileDescriptors; ++i ) {
        if( fileDescriptor[i].disk == disk
            && fileDescriptor[i].fd_number == fd_number ) {
            ++(fileDescriptor[i].useCount);
            return &(fileDescriptor[i]);
        }
        if( free_slot < 0 && fileDescriptor[i].useCount == 0 )
            free_slot = i;
    }
    if( free_slot < 0 ) {
        return 0;
    }
    // find the physical block the file descriptor is in.
    // the 2 + is because the file descriptor blocks start after 2 blocks
    int fd_block = 2 + (fd_number / FileDescriptorsPerBlock);
    int fd_offset
        = (fd_number % FileDescriptorsPerBlock) * FileDescriptorSize;
    BlockBufferHeader * fd_buffer = GetDiskBlock( disk, fd_block );
    FileDescriptor * fd = (FileDescriptor *)&(fd_buffer->block +
    fd_offset;
    MemoryCopy( (char *)fd, (char *)&(fileDescriptor[free_slot]),
        FileDescriptorSize);
    FreeDiskBlock( fd_buffer );
    fd->useCount = 1;
    return fd;
}
void
MemoryCopy( char * from, char * to, int count) {
    while( count-->0 ) *to++ = *from++;
}

void
FreeFileDescriptor( FileDescriptor * fd ) {
    --(fd->useCount);
}
```

A file descriptor contains everything that we need to know about the file. This includes where on the disk the data blocks are located. We will defer the explanation and use of this part of the file descriptor until the next chapter, after we discuss a range of strategies for keeping this information.

The in-memory version of the file descriptor will contain three extra fields not contained in the disk version of the file descriptor. Two of these record the disk number and the file descriptor number. Naturally, these are not needed in the disk version because you need these to find the disk version. We also keep track of how many open files are using this file descriptor. Normally this use count is 1.

GetFileDescriptor fetches file descriptors from the disk and puts them in the in-memory table of active file descriptors. The first thing it does when it gets a request is to see if the file descriptor is already in the table. It does this with a linear search. If you expected a large table, you might use a hash table search like we used for the disk cache buffers.

While it is looking for the file descriptor, it also looks for free slots in case it does not find the file descriptor and has to read it into the table from the disk.

If the file descriptor is not found in the table and no table slots are free, then we return an error. This should not happen. Normally, we find a free slot and read the file descriptor into it.

We figure out the physical block number of the block that contains this particular file descriptor and read in that block. This uses the same GetDiskBlock procedure that is used everywhere in the file system to read disk blocks. Then we copy the file descriptor into the table.

We keep a use count for each file descriptor so we know when we can replace it if we need a table slot.

16.6.4 Open Files

This code maintains the systemwide open file table and the per-process open file tables.

OPEN FILES

```
const int NumOpenFiles = 150;
const int OpenFilesPerProcess = 20;
// type for the open file table entries
struct OpenFile {
    int useCount;
    int openMode;
    int filePosition;
    FileDescriptor * fd;
};
// openMode is one or more of these ORed together
const int ReadMode = 0x1;
const int WriteMode = 0x2;

// the in-memory table of open files
OpenFile openFile[NumOpenFiles];

// some new fields in the process descriptor
```

```
struct processDescriptor {
    // . . . all the fields we had before plus:
    OpenFile * openFile[OpenFilesPerProcess];
        //these are all initialized to 0
    int currentDirectoryFD;

};

int
GetProcessOpenFileSlot( int pid ) {
    for( int i = 0; i < OpenFilesPerProcess; ++i ) {
        if( pd[pid].openFile[i] == 0 ) {
            return i;
        }
    }
    return -1;// no free open file slots left to allocate
}

int
GetSystemOpenFileSlot( void ) {
    for( int i = 0; i < NumOpenFiles; ++i ) {
        if( openFile[i].useCount == 0 ) {
            return i;
        }
    }
    return -1;// no free open file slots left to allocate
}
```

The open file structure has four fields. Like we do for all these structures, we maintain a use count for open file table entries so we know when we can replace them. We remember the mode the file was opened in so we can check later reads and writes to be sure they are allowed. We also keep the file position here and a pointer to the file descriptor of the file that is open.

Each process contains an array of pointers to open file table slots. The index into this array is the file identifier used by processes.

`GetSystemOpenFileSlot` finds a free slot in the open file table.

16.6.5 Directories

A directory, in this file system, is a file that contains directory entries. Each directory entry contains the name and the number of the file descriptor of the file the name refers to. We are allowing for a maximum of 60 characters in a file name and 250 characters in a path name.

We record the file descriptor number of the root directory.

DIRECTORIES

```
const int FileNameSize = 60;
const int MaxPathNameSize = 250;
// type of a directory entry
```

```
struct DirectoryEntry {
    int FDNumber;
    char name[FileNameSize];
};
const int DirectoryEntriesPerBlock = BlockSize / sizeof(DirectoryEntry);

int rootFD = 0;// the first FD is always the root directory
```

16.6.6 File System Initialization

We need to initialize the various tables we use, mainly setting all the use counts to zero.

INITIALIZATION

```
void
FileSystemInitialization( void ) {
    int i;

    // initialize the disk cache
    for( i = 0; i < DiskCacheSize; ++i ) {
        diskCache[i].block = &(diskBuffer[i]);
        diskCache[i].blockNumber = -1;
        diskCache[i].useCount = 0;
    }
    nextHeaderToReplace = DiskCacheSize;
    // initialize the file descriptor table
    for( i = 0; i < NumFileDescriptors; ++i ) {
        fileDescriptor[i].useCount = 0;
        fileDescriptor[i].fd_number = -1;
    }
    // initialize the open file table
    for( i = 0; i < NumOpenFiles; ++i ) {
        openFile[i].useCount = 0;
    }
}
```

16.6.7 File-Related System Calls

Now we are ready to implement the file system-related system calls. Here are the new cases in the case statement of the `SystemCallInterruptHandler` procedure.

SYSTEM CALL INTERRUPT HANDLER

```
void SystemCallInterruptHandler( void ) {
    // . . . initial part as before

    case OpenSystemCall:
```

```
        char * fileName; asm { store r9,fileName }
        int openMode; asm { store r10,openMode }
        pd[current_process].sa.reg[1]
                = Open( current_process, fileName, openMode );
        break;

    case CreatSystemCall:
        // . . . Not implemented in this code
        break;

    case ReadSystemCall:
        int fid; asm {store r9,fid }
        char * userBuffer; asm { store r10,userBuffer }
        int count; asm { store r11,count }
        pd[current_process].sa.reg[1]
            =Read( current_process, fid, userBuffer, count );
        break;

    case WriteSystemCall:
        // . . . not shown, nearly the same as read

    case LseekSystemCall:
        int fid; asm { store r9,fid }
        int offset; asm { store r10,offset }
        int how; asm { store r11,how }
        pd[current_process].sa.reg[1]
            =Lseek( current_process, fid, offset, how );
        break;

    case CloseSystemCall:
        int fid; asm { store r9,fid }
        int ret_value;
        OpenFile * of = pd[pid].openFile[fid];
        if( of==0 )
            ret_value = -1;
        else {
            if(--(of->useCount) == 0 ) {
               --(of->fd->useCount);
            }
            ret_value = 0;
        }
        pd[current_process].sa.reg[1] = 0;
        break;

    }
    Dispatcher();
}
```

Most of these cases just get the arguments and call a procedure. The close file system call is so short we put it in line in the switch statement. The close decrements the use count of the open file. If this use count goes to zero, we also decrement the use count of the file descriptor it is pointing to.

16.6.8 System Call Procedures

Now we will go through each of the procedures implementing a file system-related system call. We will start with the Open procedure.

OPEN FILE

```
int
Open( int pid, char * fileNameIn, int openMode ) {
    // find slots in the per-process and systemwide open file tables
    int process_ofslot = GetProcessOpenFileSlot(pid);
    if( process_ofslot < 0 ) return -1;

    int ofslot = GetSystemOpenFileSlot(pid);
    if( ofslot < 0 ) return -2;

    char fileName[MaxPathNameSize];
    CopyToSystemSpace( pid, fileNameIn, fileName, MaxPathNameSize );

    char * current_path = fileName;
    int fd_number;
    if( *fileName == '/' ) {
        fd_number = rootFD;
        ++current_path
    } else {
        fd_number = pd[pid].currentDirectoryFD;
    }

    // This is the loop to look up the file in the directory tree
    while( 1 ) {
        // are we at the end of the pathname yet?
        if( *current_path == '\0' ) {
            // we are at the end of the path
            break;
        }
        // isolate the file name component
        current_component = current_path;
        while( 1 ) {
            ch = *current_path;
            if( ch == '/' || ch =='\0' )
                break;
            ++current_path;
        }
        char save_char = *current_path;
        *current_path = '\0'; // temporarily put in end of string marker

        // get the file descriptor for the next directory
        FileDescriptor * fd = GetFileDescriptor( DiskNumber, fd_number );

        // search the directory for the name
        int dir_entry_number = 0;
        Directory Entry * dir_entry;
        while(1) {
```

```
        BlockNumber lbn
            = dir_entry_number / DirectoryEntriesPerBlock;
        // have we gotten to the end of the directory yet?
        if(dir_entry_number*sizeof(DirectoryEntry) >=fd->length ) {
            FreeFileDescriptor( fd );
            // the component name was not found
            return -1;
        }
        BlockNumber pbn = LogicalToPhysicalBlock( fd, lbn);
        BlockBufferHeader * dir_buffer
            = GetDiskBlock( DiskNumber, pbn);
        int dir_offset = (dir_entry_number % DirectoryEntriesPerBlock)
            * sizeof(DirectoryEntry);
        dir_entry = (DirectoryEntry *)&(dir_buffer->buffer + dir_offset);
        // compare the names;
        if(strncmp(dir_entry->name,current_component,
           FileNameSize)==0)
            break;
        FreeDiskBlock( dir_buffer );
        ++dir_entry_number;
    }
    FreeFileDescriptor( fd );
    // pick out the fd number of this file
    fd_number = dir_entry->FDNumber;
    FreeDiskBlock( dir_buffer );
    // move to the next component of the name
    *current_path = save_char;
    if( save_char == '/' )
        ++current_path; // skip past the "/"
}
// read in the fd for this file and put it in the open file table
fd = GetFileDescriptor( DiskNumber, fd_number );
openFile[ofslot].fd = fd;
openFile[ofslot].filePosition = 0;
openFile[ofslot].useCount = 1;
pd[pid].openFile[process_ofslot] = &(openFile[ofslot]);
return ofslot;
```

a)

The first thing the open procedure does is allocate slots for the process in the open file table and in the systemwide open file table. If either of these allocations fail, then the system call returns an error.

Next we copy in the path name from the user's address space. If the name starts with a slash, then we start at the root directory; otherwise we start and use the current directory for the current process.

Now we can begin the major loop. In each iteration of this loop, we will process one component of the path name. We do this by searching for the name in the directory that the name is supposed to be in. The variable `fd_number` contains the file descriptor number of the directory we will search.

If we are at the end of the path name, then `fd_number` is the number of the file descriptor we are searching for. This might refer to a file or a directory.

If we are not at the end of the path name, then we isolate the string that is the next component of the path name. We do this by writing a null at the end of the name. We save (and later restore) the character we write over, so the file name is not permanently changed. This is not necessary in this code, because we do not use the path name for anything else.

Next, we have to search the directory for the component name. We get ready to do this by getting the file descriptor into the file descriptor table with GetFileDescriptor. Then we enter an inner loop where we search the directory one entry at a time.

For each directory entry, we figure out which logical disk block it is in. Then we convert that into a physical block number. We will show this code in the next chapter. We get the block and compare the name in the directory entry with the component name we are looking for. If they match, we have found what we are looking for and we drop out of the loop. Otherwise we continue searching with the next directory entry. If we get to the end of the directory, we have an error in the path name, and so we return an error code.

Once we have found the component name, we get its file descriptor number, make that the new value of fd_number, and continue our major loop through the components of the path name.

When we get to the end of the path name, we have the file descriptor number of the file we want to open. We get this file descriptor and set up the various linked structures.

The next system call is the read system call. The process can read bytes from the file starting from any position, and can read as few or as many bytes as it needs. Since we have to read the disk in complete disk blocks, we have to do some buffering and adjusting. The read might begin in the middle of a disk block and end in the middle of a disk block. It might be entirely in one disk block, or it might span several disk blocks.

We handle this with a loop that reads the required disk blocks one at a time and transfers the data from each block.

READ FILE

```
int
Read( int pid, int fid, char * userBuffer, int count ) {
    OpenFile * of = pd[pid].openFile[fid];
    if( of == 0 )
        return -1;

    if( !(of->openMode & ReadMode) ) // check if the read is allowed
        return -2;

    int filepos = of->filePosition;

    // check against the file length and adjust if near EOF
    if( (filepos+count) > of->fd->length ) {
        count = (of->fd->length) - filepos;
```

```
}
if( count <= 0 )
    return 0;
int bytesRead = 0;

// Get the bytes one block at a time.
// We may not use some bytes at the begining of the first block
// and at the end of the last block.
while( count > 0) {
    BlockNumber lbn = filepos / BlockSize;
    int offsetInBlock = filepos % BlockSize;
    int leftInBlock = BlockSize - offsetInBlock;
    int lengthToCopy;
    if( leftInBlock < count ) {
        lengthToCopy = leftInBlock;
    } else {
        lengthToCopy = count;
    }
    BlockNumber pbn = LogicalToPhysicalBlock( of->fd, lbn );
    BlockBufferHeader * header = GetDiskBlock( DiskNumber, pbn );
    CopyFromSystemSpace( pid, userBuffer, (header->block) +
        offsetInBlock, lengthToCopy);
    FreeDiskBlock( header );
    filepos += lengthToCopy;
    userBuffer  += lengthToCopy;
    count -= lengthToCopy;
    bytesRead += lengthToCopy;
  }
  return bytesRead;
}
```

The read starts with some error checking. It is legal for a process to try to read past the end of the file. That is why the read system call returns how many bytes it actually read. If the process is trying to read past the end of the file, we adjust the count to read to the end of the file exactly.

If there is nothing to read, we return immediately. Otherwise, we start a loop which will perform one iteration for each disk block we need to read.

We figure out which disk block we need, where in the block to start reading, and how many bytes to read from this block. This code takes care of the special cases with the first and last block. Then we convert the logical block number to a physical block number. (This procedure is defined in the next chapter.) Then we read the block and copy the parts we need. This copy involves a transfer between the system's address space and the user's address space, and so must be done by a procedure that knows how to do that. Finally, we adjust all the counts for the loop, and iterate until there are no more bytes to read.

The write system call is similar, but actually a bit more complex. This is because it may have to write a partial block. To do this correctly, it has to first read the block in and then change part of it. Other than this, read and write are just about the same.

The final system call we will show is the seek system call. This call adjusts the value of the file pointer.

LSEEK

```
int
Lseek( int pid, int fid, int offset, int how ) {
    OpenFile * of = pd[pid].openFile[fid];
    if( of == 0 )
        return -1;
    switch( how ) {
        case 0: // from beginning of file
            // nothing to do, offset is what we want
            break;
        case 1: // from current
            offset += of->filePosition;
            break;
        case 2:
            offset += of->fd->length;
            break;
    }
    // do not allow negative file positions
    if( offset < 0 )
        return -2;
    of->filePosition = offset;
    return offset;
}
```

The adjustment of the file pointer depends on the how argument. It can be from the beginning of the file, from the current file pointer, or from the end of the file. File pointers are not allowed to be negative, so if this change would make that happen, it is rejected and an error code is returned. Otherwise, the value of the new file pointer is returned.

16.7 SUMMARY

It is very useful to have persistent storage in a computer system that stays around even after the process that created it has exited. Disks provide persistent storage, but with an inconvenient user interface. A file system is an abstraction implemented by the operating system on top of the logical disk system. A file system provides a user-friendly interface to persistent storage. File systems provide reliable, convenient access to files of arbitrary length, which are named with a hierarchical naming system. The principal objects in a file system are files, open files, and directories.

All file systems provide the same general types of objects and operations, but there are many possible ways to present this functionality to the user. Most operating