6

FILE SYSTEMS

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

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

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

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

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

The usual solution to all these problems is to store information on disks and other external media in units called **files**. Processes can then read them and write new

IBM Ex-1009
Page 293

ones if need be. Information stored in files must be **persistent**, that is, not be affected by process creation and termination. A file should only disappear when its owner explicitly removes it.

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

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

6.1 FILES

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

6.1.1 File Naming

Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work.

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

The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of one to eight letters as legal file names. Thus *andrea*, *bruce*, and *cathy* are possible file names. Frequently digits and special characters are also permitted, so names like *2*, *urgent!*, and *Fig.2-14* are often valid as well. Many file systems support names as long as 255 characters.

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

IBM Ex-1009
Page 294

An aside on file systems is probably in order here. Windows 95 and Windows 98 both use the MS-DOS file system, and thus inherit many of its properties, such as how file names are constructed. In addition, Windows NT and Windows 2000 support the MS-DOS file system and thus also inherit its properties. However, the latter two systems also have a native file system (NTFS) that has different properties (such as file names in Unicode). In this chapter, when we refer to the Windows file system, we mean the MS-DOS file system, which is the only file system supported by all versions of Windows. We will discuss the Windows 2000 native file system in Chap. 11.

Many operating systems support two-part file names, with the two parts separated by a period, as in *prog.c*. The part following the period is called the **file extension** and usually indicates something about the file. In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in *prog.c.Z*, where *.Z* is commonly used to indicate that the file (*prog.c*) has been compressed using the Ziv-Lempel compression algorithm. Some of the more common file extensions and their meanings are shown in Fig. 6-1.

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

Figure 6-1. Some typical file extensions.

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

IBM Ex-1009
Page 295

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

In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system and specify for each one which program "owns" that extension. When a user double clicks on a file name, the program assigned to its file extension is launched with the file as parameter. For example, double clicking on *file.doc* starts Microsoft *Word* with *file.doc* as the initial file to edit.

6.1.2 File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 6-2. The file in Fig. 6-2(a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.

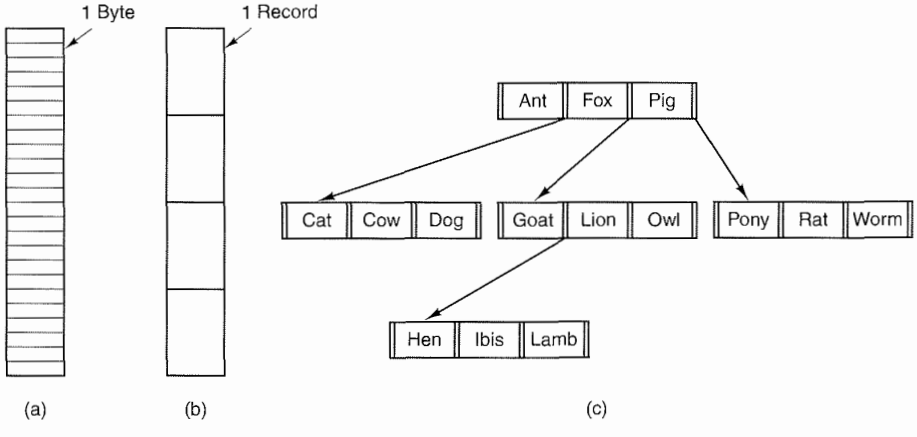


Figure 6-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

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

not help, but it also does not get in the way. For users who want to do unusual things, the latter can be very important.

The first step up in structure is shown in Fig. 6-2(b). In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record. As a historical note, in decades gone by, when the 80-column punched card was king, many (mainframe) operating systems based their file systems on files consisting of 80-character records, in effect, card images. These systems also supported files of 132-character records, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of 80 characters and wrote it in units of 132 characters, although the final 52 could be spaces, of course. No current general-purpose system works this way.

The third kind of file structure is shown in Fig. 6-2(c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a **key** field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

The basic operation here is not to get the "next" record, although that is also possible, but to get the record with a specific key. For the zoo file of Fig. 6-2(c), one could ask the system to get the record whose key is *pony*, for example, without worrying about its exact position in the file. Furthermore, new records can be added to the file, with the operating system, and not the user, deciding where to place them. This type of file is clearly quite different from the unstructured byte streams used in UNIX and Windows but is widely used on the large mainframe computers still used in some commercial data processing.

6.1.3 File Types

Many operating systems support several types of files. UNIX and Windows, for example, have regular files and directories. UNIX also has character and block special files. **Regular files** are the ones that contain user information. All the files of Fig. 6-2 are regular files. **Directories** are system files for maintaining the structure of the file system. We will study directories below. **Character special files** are related to input/output and used to model serial I/O devices such as terminals, printers, and networks. **Block special files** are used to model disks. In this chapter we will be primarily interested in regular files.

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

The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers

of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines. (The interprocess plumbing is not any easier, but interpreting the information certainly is if a standard convention, such as ASCII, is used for expressing it.)

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

For example, in Fig. 6-3(a) we see a simple executable binary file taken from a version of UNIX. Although technically the file is just a sequence of bytes, the operating system will only execute a file if it has the proper format. It has five sections: header, text, data, relocation bits, and symbol table. The header starts with a so-called **magic number**, identifying the file as an executable file (to prevent the accidental execution of a file not in this format). Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits. Following the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is used for debugging.

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

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

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

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

IBM Ex-1009
Page 298

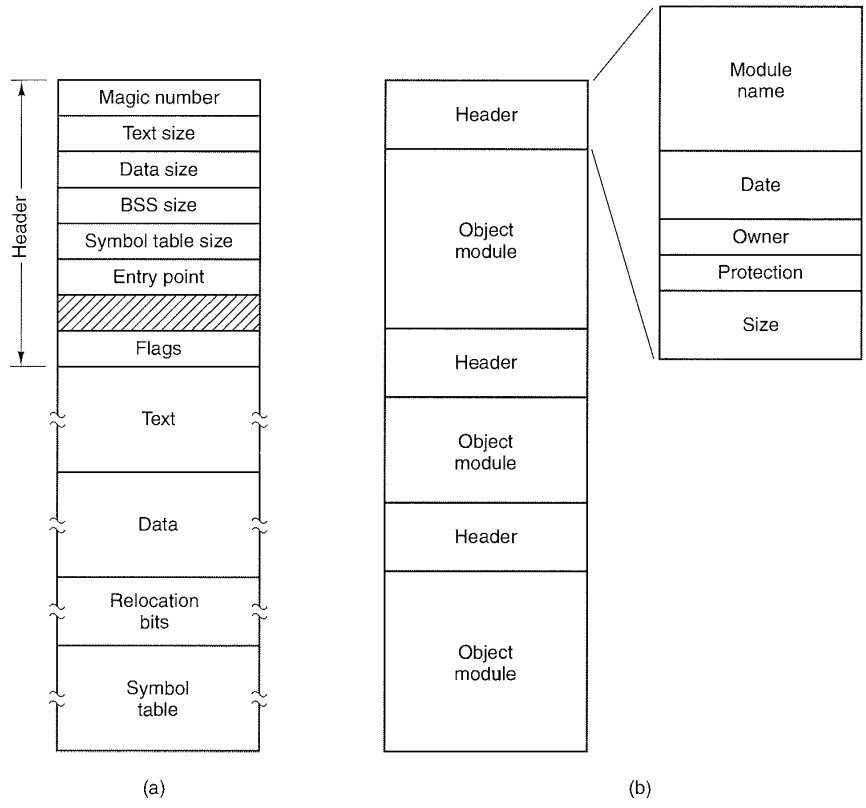


Figure 6-3. (a) An executable file. (b) An archive.

6.1.4 File Access

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

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

IBM Ex-1009
Page 299

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

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

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

6.1.5 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was created and the file's size. We will call these extra items the file's **attributes**. The list of attributes varies considerably from system to system. The table of Fig. 6-4 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.

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

The flags are bits or short fields that control or enable some specific property. Hidden files, for example, do not appear in listings of all the files. The archive flag is a bit that keeps track of whether the file has been backed up. The backup program clears it, and the operating system sets it whenever a file is changed. In this way, the backup program can tell which files need backing up. The temporary flag allows a file to be marked for automatic deletion when the process that created it terminates.

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

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

The current size tells how big the file is at present. Some old mainframe operating systems require the maximum size to be specified when the file is created, in order to let the operating system reserve the maximum amount of storage in ad-

IBM Ex-1009
Page 300

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

Figure 6-4. Some possible file attributes.

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

6.1.6 File Operations

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

1. Create. The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
2. Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. Open. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.

5. Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.

6. Write. Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

7. Append. This call is a restricted form of write. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.

8. Seek. For random access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

9. Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When *make* is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

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

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

6.1.7 An Example Program Using File System Calls

In this section we will examine a simple UNIX program that copies one file from its source file to a destination file. It is listed in Fig. 6-5. The program has minimal functionality and even worse error reporting, but it gives a reasonable idea of how some of the system calls related to files work.

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

```
copyfile abc xyz
```

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

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

The first *#define* statement is a macro definition that defines the string *BUF_SIZE* as a macro that expands into the number 4096. The program will read and write in chunks of 4096 bytes. It is considered good programming practice to give names to constants like this and to use the names instead of the constants. Not only does this convention make programs easier to read, but it also makes them easier to maintain. The second *#define* statement determines who can access the output file.

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

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

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

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

The first actual statement checks *argc* to see if it is 3. If not, it exits with status code 1. Any status code other than 0 means that an error has occurred. The

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

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

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

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

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

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

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

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

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

Figure 6-5. A simple program to copy a file.

IBM Ex-1009
Page 304

status code is the only error reporting present in this program. A production version would normally print error messages as well.

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

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

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

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

Although the Windows system calls are different from those of UNIX, the general structure of a command-line Windows program to copy a file is moderately similar to that of Fig. 6-5. We will examine the Windows 2000 calls in Chap. 11.

6.1.8 Memory-Mapped Files

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

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

IBM Ex-1009
Page 305

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

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

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

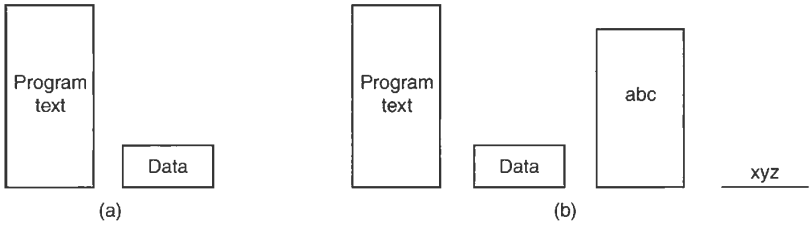


Figure 6-6. (a) A segmented process before mapping files into its address space. (b) The process after mapping an existing file *abc* into one segment and creating a new segment for file *xyz*.

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

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

IBM Ex-1009
Page 306

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

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

6.2 DIRECTORIES

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

6.2.1 Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory**, but since it is the only one, the name does not matter much. On early personal computers, this system was common, in part because there was only one user. Interestingly enough, the world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once. This decision was no doubt made to keep the software design simple.

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

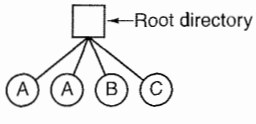


Figure 6-7. A single-level directory system containing four files, owned by three different people, *A*, *B*, and *C*.

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

IBM Ex-1009
Page 307

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

6.2.2 Two-level Directory Systems

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

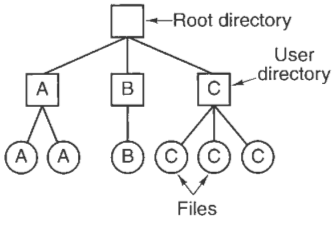


Figure 6-8. A two-level directory system. The letters indicate the owners of the directories and files.

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

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

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

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

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

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

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

IBM Ex-1009
Page 308

6.2.3 Hierarchical Directory Systems

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

What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Fig. 6-9. Here, the directories *A*, *B*, and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

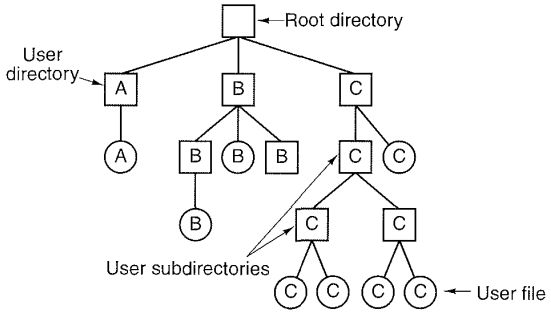


Figure 6-9. A hierarchical directory system.

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

6.2.4 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox* means that the

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

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

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

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

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

and the command

```
cp mailbox mailbox.bak
```

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

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

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

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

IBM Ex-1009
Page 310

this reason, library procedures rarely change the working directory, and when they must, they always change it back again before returning.

Most operating systems that support a hierarchical directory system have two special entries in every directory, "." and "..", generally pronounced "dot" and "dotdot." Dot refers to the current directory; dotdot refers to its parent. To see how these are used, consider the UNIX file tree of Fig. 6-10. A certain process has */usr/ast* as its working directory. It can use .. to go up the tree. For example, it can copy the file */usr/lib/dictionary* to its own directory using the command

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

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

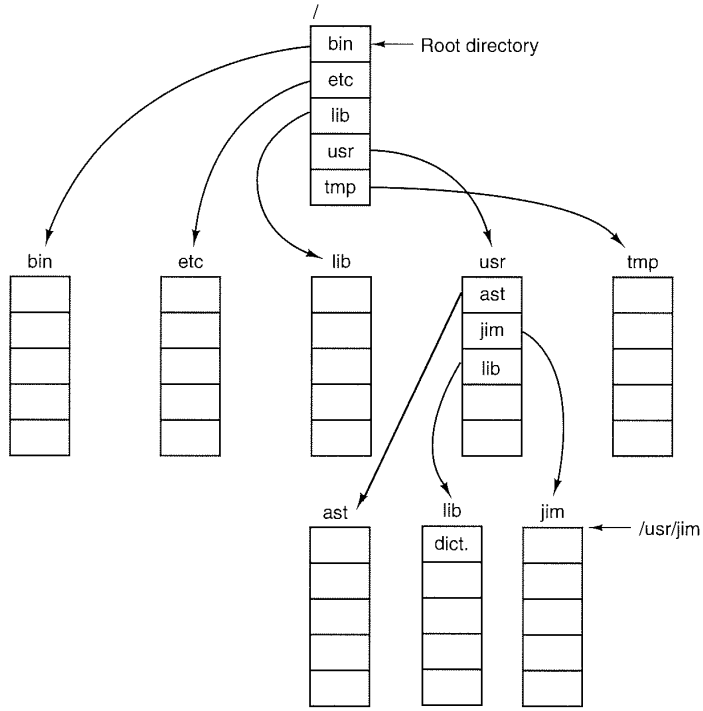


Figure 6-10. A UNIX directory tree.

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

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

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

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

also works fine, as does

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

All of these do exactly the same thing.

6.2.5 Directory Operations

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

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

IBM Ex-1009
Page 312

8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

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

6.3 FILE SYSTEM IMPLEMENTATION

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

6.3.1 File System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR** (**Master Boot Record**) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, called the **boot block**, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the future, so reserving a boot block is a good idea anyway.

Other than starting with a boot block, the layout of a disk partition varies strongly from file system to file system. Often the file system will contain some of the items shown in Fig. 6-11. The first one is the **superblock**. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file system type, the number of blocks in the file system, and other key administrative information.

Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers. This might be followed by the i-

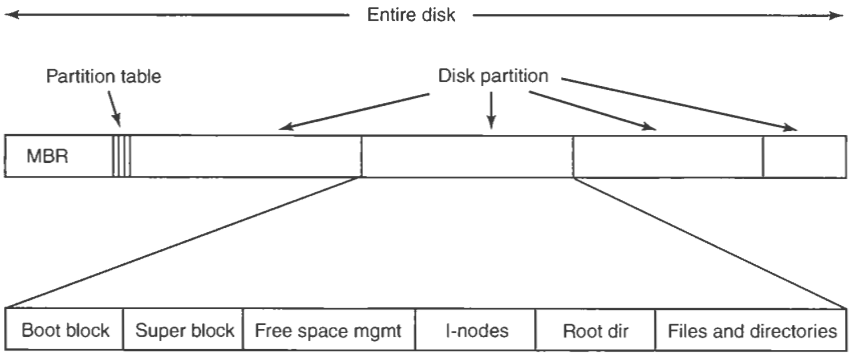


Figure 6-11. A possible file system layout.

nodes, an array of data structures, one per file, telling all about the file. After that might come the root directory, which contains the top of the file system tree. Finally, the remainder of the disk typically contains all the other directories and files.

6.3.2 Implementing Files

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

Contiguous Allocation

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

We see an example of contiguous storage allocation in Fig. 6-12(a). Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file *A*, of length four blocks was written to disk starting at the beginning (block 0). After that a six-block file, *B*, was written starting right after the end of file *A*. Note that each file begins at the start of a new block, so that if file *A* was really 3½ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart.

Contiguous disk space allocation has two significant advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced

IBM Ex-1009
Page 314

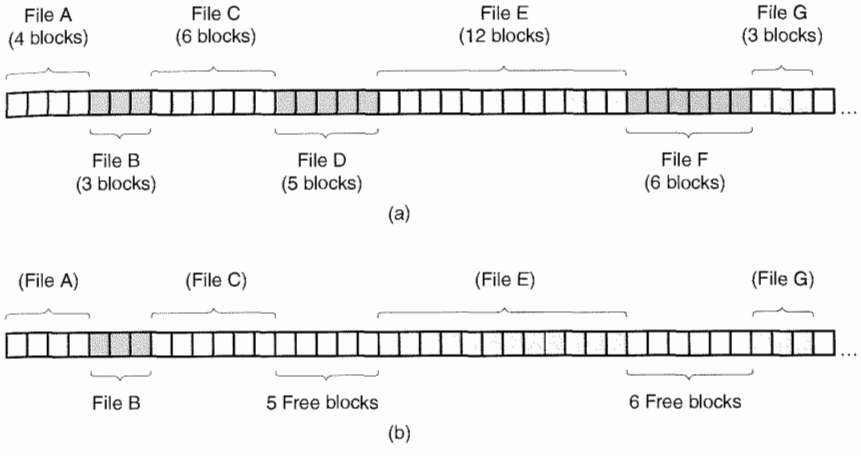


Figure 6-12. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.

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

Unfortunately, contiguous allocation also has a significant drawback: in time, the disk becomes fragmented. To see how this comes about, examine Fig. 6-12(b). Here two files, *D* and F have been removed. When a file is removed, its blocks are freed, leaving a run of free blocks on the disk. The disk is not compacted on the spot to squeeze out the hole since that would involve copying all the blocks following the hole, potentially millions of blocks. As a result, the disk ultimately consists of files and holes, as illustrated in the figure.

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

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

IBM Ex-1009
Page 315

how many bytes the final document will be. The question must be answered or the program will not continue. If the number given ultimately proves too small, the program has to terminate prematurely because the disk hole is full and there is no place to put the rest of the file. If the user tries to avoid this problem by giving an unrealistically large number as the final size, say, 100 MB, the editor may be unable to find such a large hole and announce that the file cannot be created. Of course, the user would be free to start the program again and say 50 MB this time, and so on until a suitable hole was located. Still, this scheme is not likely to lead to happy users.

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

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

Linked List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 6-13. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

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

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

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

IBM Ex-1009
Page 316

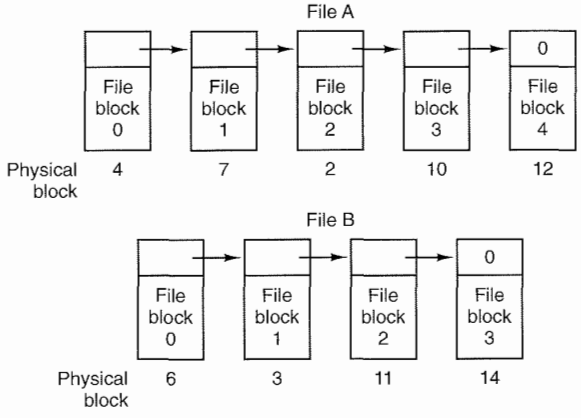


Figure 6-13. Storing a file as a linked list of disk blocks.

Linked List Allocation Using a Table in Memory

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

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

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time. Conceivably the table could be put in pageable memory, but it would still occupy a great deal of virtual memory and disk space as well as generating extra paging traffic.

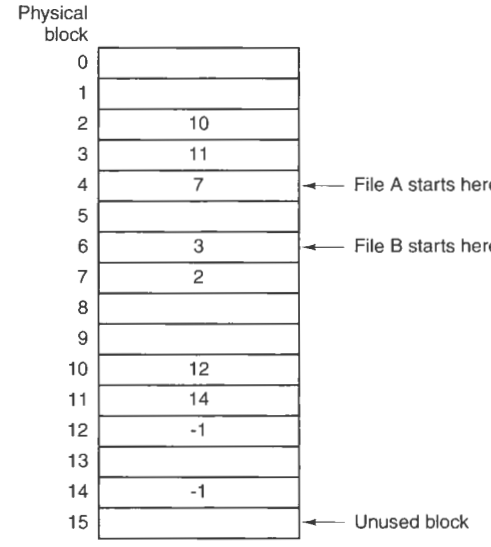


Figure 6-14. Linked list allocation using a file allocation table in main memory.

I-nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node** (**index-node**), which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Fig. 6-15. Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only kn bytes. Only this much space need be reserved in advance.

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

One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address

IBM Ex-1009
Page 318

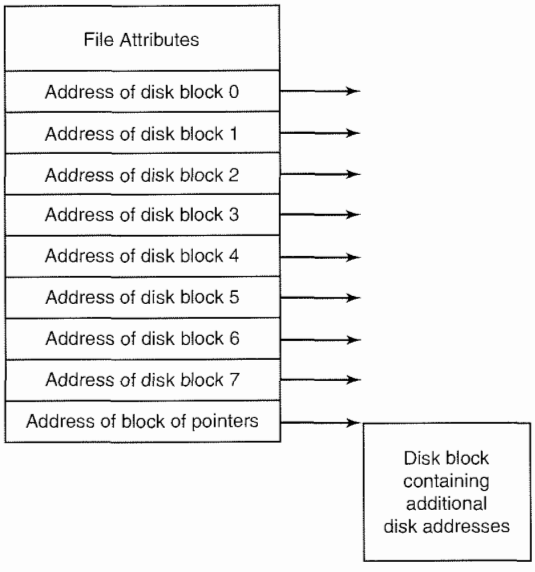


Figure 6-15. An example i-node.

of a block containing more disk block addresses, as shown in Fig. 6-15. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses. We will come back to i-nodes when studying UNIX later.

6.3.3 Implementing Directories

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

A closely related issue is where the attributes should be stored. Every file system maintains file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Many systems do precisely that. This option is shown in Fig. 6-16(a) In this simple design, a directory consists of a list of fixed-size

entries, one per file, containing a (fixed-length) file name, a structure of the file attributes, and one or more disk addresses (up to some maximum) telling where the disk blocks are.

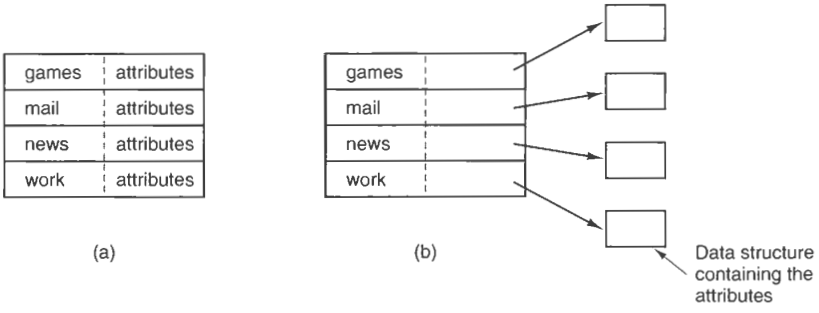


Figure 6-16. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number. This approach is illustrated in Fig. 6-16(b). As we shall see later, this method has certain advantages over putting them in the directory entry. The two approaches shown in Fig. 6-16 correspond to MS-DOS/Windows and UNIX, respectively, as we will see later in this chapter.

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

The simplest approach is to set a limit on file name length, typically 255 characters, and then use one of the designs of Fig. 6-16 with 255 characters reserved for each file name. This approach is simple, but wastes a great deal of directory space, since few files have such long names. For efficiency reasons, a different structure is desirable.

One alternative is to give up the idea that all directory entries are the same size. With this method, each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes. This fixed-length header is followed by the actual file name, however long it may be, as shown in Fig. 6-17(a) in big-endian format (e.g., SPARC). In this example we have three files, *project-budget*, *personnel*, and *foo*. Each file name is terminated by a special character (usually 0), which is represented in the

IBM Ex-1009
Page 320

figure by a box with a cross in it. To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words, shown by shaded boxes in the figure.

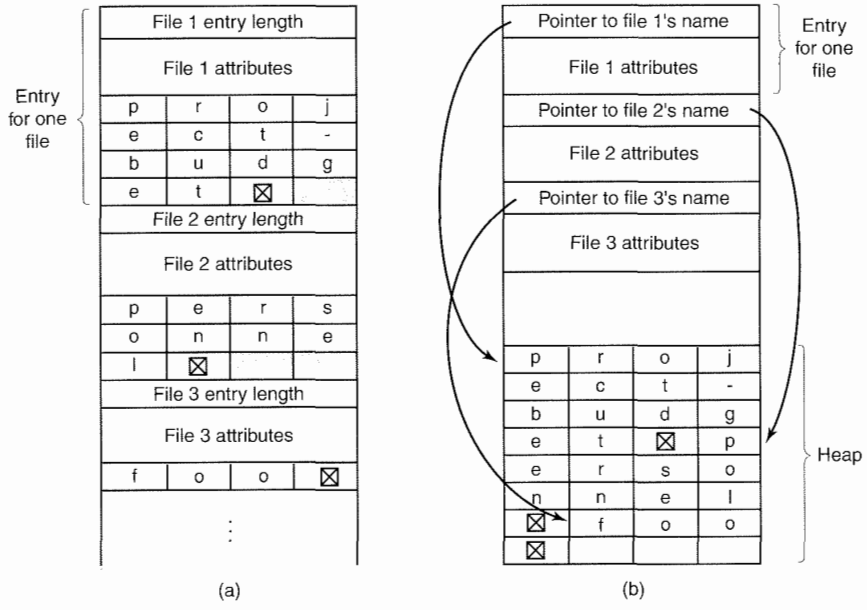


Figure 6-17. Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

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

Another way to handle variable-length names is to make the directory entries themselves all fixed length and keep the file names together in a heap at the end of the directory, as shown in Fig. 6-17(b). This method has the advantage that when an entry is removed, the next file entered will always fit there. Of course, the heap must be managed and page faults can still occur while processing file names. One minor win here is that there is no longer any real need for file names to begin at word boundaries, so no filler characters are needed after file names in Fig. 6-17(b) and they are in Fig. 6-17(a).

In all of the designs so far, directories are searched linearly from beginning to end when a file name has to be looked up. For extremely long directories, linear

IBM Ex-1009
Page 321

searching can be slow. One way to speed up the search is to use a hash table in each directory. Call the size of the table n. To enter a file name, the name is hashed onto a value between 0 and $n - 1$, for example, by dividing it by n and taking the remainder. Alternatively, the words comprising the file name can be added up and this quantity divided by n, or something similar.

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

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

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

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

6.3.4 Shared Files

When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Figure 6-18 shows the file system of Fig. 6-9 again, only with one of C's files now present in one of B's directories as well. The connection between B's directory and the shared file is called a **link**. The file system itself is now a **Directed Acyclic Graph**, or **DAG**, rather than a tree.

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

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

IBM Ex-1009
Page 322

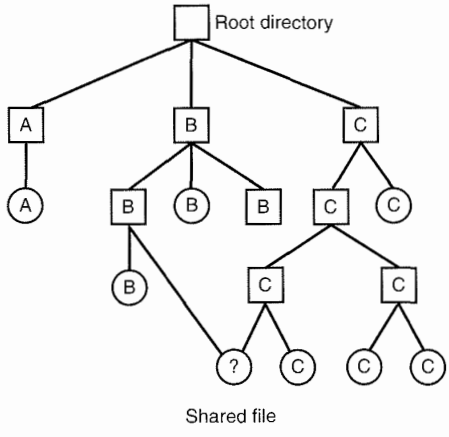


Figure 6-18. File system containing a shared file.

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

Each of these methods has its drawbacks. In the first method, at the moment that *B* links to the shared file, the i-node records the file's owner as *C*. Creating a link does not change the ownership (see Fig. 6-19), but it does increase the link count in the i-node, so the system knows how many directory entries currently point to the file.

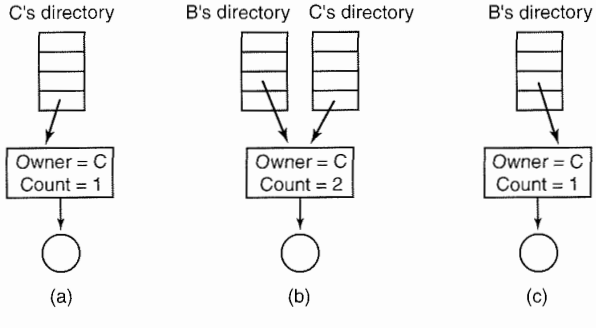


Figure 6-19. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

IBM Ex-1009
Page 323

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

The only thing to do is remove *C*'s directory entry, but leave the i-node intact, with count set to 1, as shown in Fig. 6-19(c). We now have a situation in which *B* is the only user having a directory entry for a file owned by *C*. If the system does accounting or has quotas, *C* will continue to be billed for the file until *B* decides to remove it, if ever, at which time the count goes to 0 and the file is deleted.

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

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

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

6.3.5 Disk Space Management

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

IBM Ex-1009
Page 324

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

Block Size

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

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

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

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

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

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

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

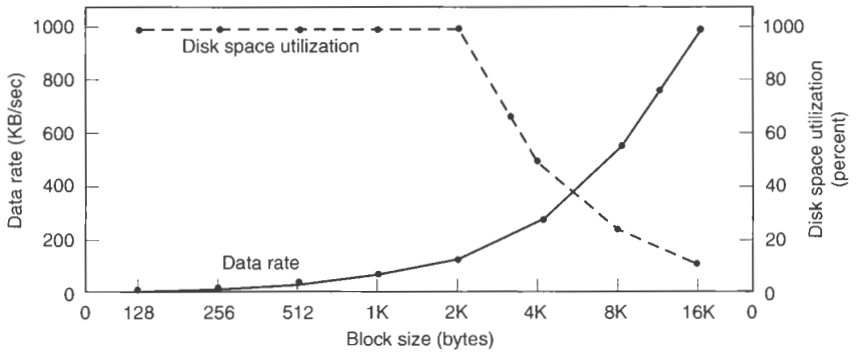


Figure 6-20. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All files are 2 KB.

the transfer time begins to dominate). With small blocks that are powers of two and 2-KB files, no space is wasted in a block. However, with 2-KB files and 4 KB or larger blocks, some disk space is wasted. In reality, few files are a multiple of the disk block size, so some space is always wasted in the last block of a file.

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

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

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

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

IBM Ex-1009
Page 326

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used, as shown in Fig. 6-21. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is needed for the pointer to the next block). A 16-GB disk needs a free list of maximum 16,794 blocks to hold all 2^{24} disk block numbers. Often free blocks are used to hold the free list.

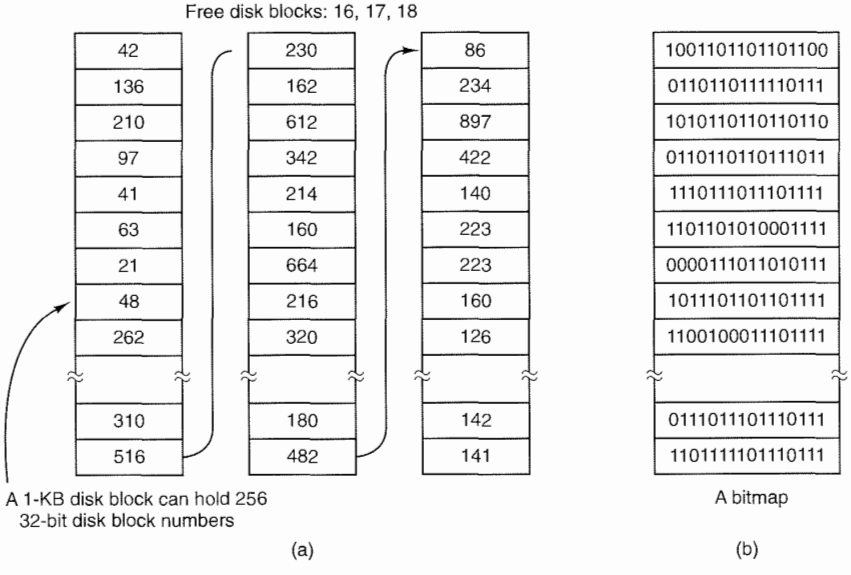


Figure 6-21. (a) Storing the free list on a linked list. (b) A bitmap.

The other free space management technique is the bitmap. A disk with n blocks requires a bitmap with n bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa). A 16-GB disk has 2^{24} 1-KB blocks and thus requires 2^{24} bits for the map, which requires 2048 blocks. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked list scheme require fewer blocks than the bitmap. On the other hand, if there are many blocks free, some of them can be borrowed to hold the free list without any loss of disk capacity.

When the free list method is used, only one block of pointers need be kept in main memory. When a file is created, the needed blocks are taken from the block

IBM Ex-1009
Page 327

of pointers. When it runs out, a new block of pointers is read in from the disk. Similarly, when a file is deleted, its blocks are freed and added to the block of pointers in main memory. When this block fills up, it is written to disk.

Under certain circumstances, this method leads to unnecessary disk I/O. Consider the situation of Fig. 6-22(a), in which the block of pointers in memory has room for only two more entries. If a three-block file is freed, the pointer block overflows and has to be written to disk, leading to the situation of Fig. 6-22(b). If a three-block file is now written, the full block of pointers has to be read in again, taking us back to Fig. 6-22(a). If the three-block file just written was a temporary file, when it is freed, another disk write is needed to write the full block of pointers back to the disk. In short, when the block of pointers is almost empty, a series of short-lived temporary files can cause a lot of disk I/O.

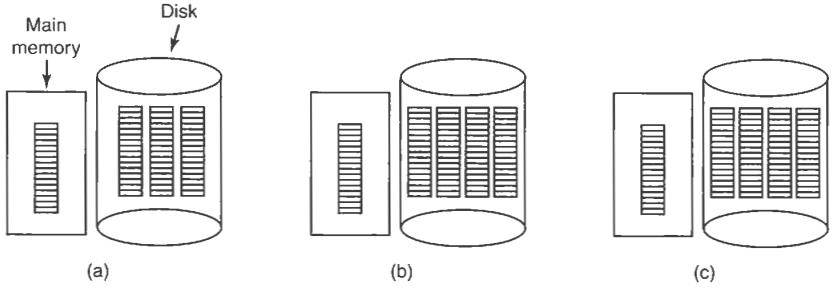


Figure 6-22. (a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

An alternative approach that avoids most of this disk I/O is to split the full block of pointers. Thus instead of going from Fig. 6-22(a) to Fig. 6-22(b), we go from Fig. 6-22(a) to Fig. 6-22(c) when three blocks are freed. Now the system can handle a series of temporary files without doing any disk I/O. If the block in memory fills up, it is written to the disk and the half-full block from disk is read in. The idea here is to keep most of the pointer blocks on disk full (to minimize disk usage), but keep the one in memory about half full, so it can handle both file creation and file removal without disk I/O on the free list.

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

IBM Ex-1009
Page 328

Disk Quotas

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

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

A second table contains the quota record for every user with a currently open file, even if the file was opened by someone else. This table is shown in Fig. 6-23. It is an extract from a quota file on disk for the users whose files are currently open. When all the files are closed, the record is written back to the quota file.

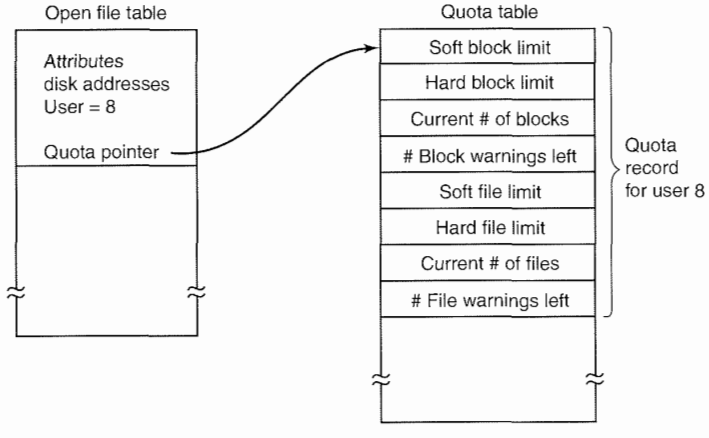


Figure 6-23. Quotas are kept track of on a per-user basis in a quota table.

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

When a user attempts to log in, the system checks the quota file to see if the user has exceeded the soft limit for either number of files or number of disk blocks. If either limit has been violated, a warning is displayed, and the count of

warnings remaining is reduced by one. If the count ever gets to zero, the user has ignored the warning one time too many, and is not permitted to log in. Getting permission to log in again will require some discussion with the system administrator.

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

6.3.6 File System Reliability

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

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

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

Backups

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

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

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

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

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

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

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

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

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

Fifth and last, making backups introduces many nontechnical problems into an organization. The best online security system in the world may be useless if the system administrator keeps all the backup tapes in his office and leaves it open and unguarded whenever he walks down the hall to get output from the printer. All a spy has to do is pop in for a second, put one tiny tape in his pocket, and saunter off jauntily. Goodbye security. Also, making a daily backup has little use if the fire that burns down the computers also burns up all the backup tapes. For this reason, backup tapes should be kept off-site, but that introduces more security risks. For a thorough discussion of these and other practical administration issues, see (Nemeth et al., 2000). Below we will discuss only the technical issues involved in making file system backups.

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

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

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

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

IBM Ex-1009
Page 332

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

Since logical dumping is the most common form, let us examine a common algorithm in detail using the example of Fig. 6-24 to guide us. Most UNIX systems use this algorithm. In the figure we see a file tree with directories (squares) and files (circles). The shaded items have been modified since the base date and thus need to be dumped. The unshaded ones do not need to be dumped.

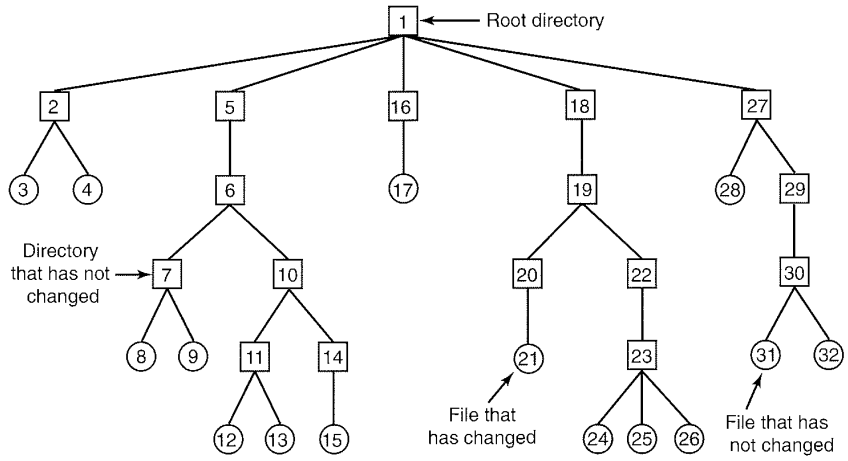


Figure 6-24. A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

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

The second reason for dumping unmodified directories above modified files is to make it possible to incrementally restore a single file (possibly to handle recovery from stupidity). Suppose that a full file system dump is done Sunday evening and an incremental dump is done on Monday evening. On Tuesday the directory */usr/jhs/proj/nr3* is removed, along with all the directories and files under it. On Wednesday morning bright and early the user wants to restore the file */usr/jhs/proj/nr3/plans/summary* However, is not possible to just restore the

file *summary* because there is no place to put it. The directories *nr3* and *plans* must be restored first. To get their owners, modes, times, etc., correct, these directories must be present on the dump tape even though they themselves were not modified since the previous full dump.

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

At the end of phase 1, all modified files and all directories have been marked in the bitmap, as shown (by shading) in Fig. 6-25(a). Phase 2 conceptually recursively walks the tree again, unmarking any directories that have no modified files or directories in them or under them. This phase leaves the bitmap as shown in Fig. 6-25(b). Note that directories 10, 11, 14, 27, 29, and 30 are now unmarked because they contain nothing under them that has been modified. They will not be dumped. In contrast, directories 5 and 6 will be dumped even though they themselves have not been modified because they will be needed to restore today's changes to a fresh machine. For efficiency, phases 1 and 2 can be combined in one tree walk.

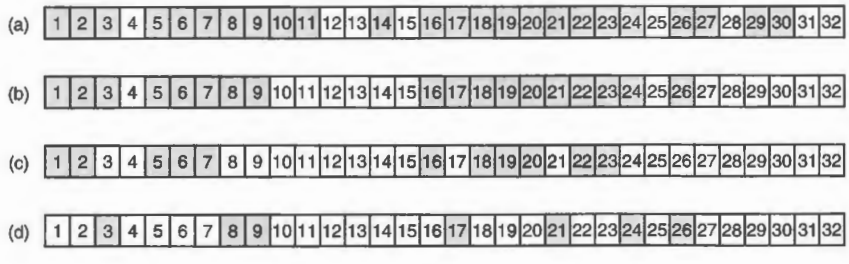


Figure 6-25. Bit maps used by the logical dumping algorithm.

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

Restoring a file system from the dump tapes is straightforward. To start with, an empty file system is created on the disk. Then the most recent full dump is restored. Since the directories appear first on the tape, they are all restored first,

giving a skeleton of the file system. Then the files themselves are restored. This process is then repeated with the first incremental dump made after the full dump, then the next one, and so on.

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

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

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

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

File System Consistency

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

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

Two kinds of consistency checks can be made: blocks and files. To check for block consistency, the program builds two tables, each one containing a counter for each block, initially set to 0. The counters in the first table keep track of how

many times each block is present in a file; the counters in the second table record how often each block is present in the free list (or the bitmap of free blocks).

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

If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 6-26(a). However, as a result of a crash, the tables might look like Fig. 6-26(b), in which block 2 does not occur in either table. It will be reported as being a **missing block**. While missing blocks do no real harm, they do waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

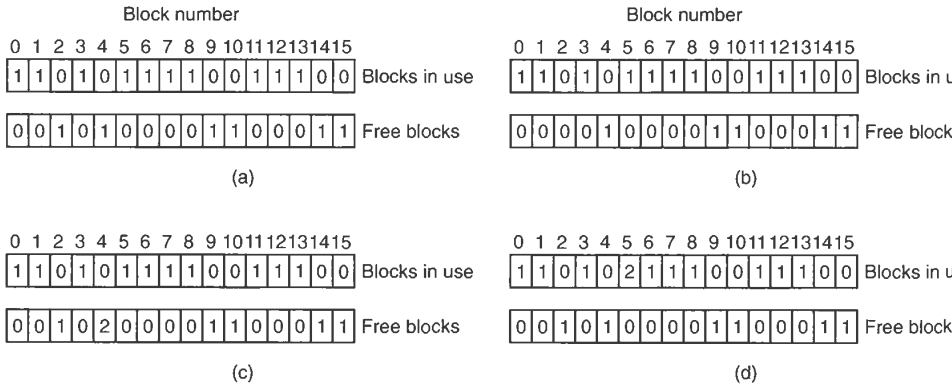


Figure 6-26. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

Another situation that might occur is that of Fig. 6-26(c). Here we see a block, number 4, that occurs twice in the free list. (Duplicates can occur only if the free list is really a list; with a bitmap it is impossible.) The solution here is also simple: rebuild the free list.

The worst thing that can happen is that the same data block is present in two or more files, as shown in Fig. 6-26(d) with block 5. If either of these files is removed, block 5 will be put on the free list, leading to a situation in which the same block is both in use and free at the same time. If both files are removed, the block will be put onto the free list twice.

The appropriate action for the file system checker to take is to allocate a free block, copy the contents of block 5 into it, and insert the copy into one of the files.

IBM Ex-1009
Page 336

In this way, the information content of the files is unchanged (although almost assuredly one is garbled), but the file system structure is at least made consistent. The error should be reported, to allow the user to inspect the damage.

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

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

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

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

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

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

IBM Ex-1009
Page 337

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

```
rm *.o
```

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

```
rm * .o
```

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

6.3.7 File System Performance

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

Caching

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

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

Operation of the cache is illustrated in Fig. 6-27. Since there are many (often thousands of) blocks in the cache, some way is needed to determine quickly if a

given block is present. The usual way is to hash the device and disk address and look up the result in a hash table. All the blocks with the same hash value are chained together on a linked list so the collision chain can be followed.

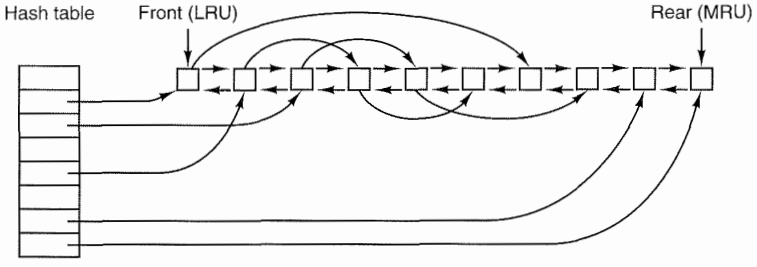


Figure 6-27. The buffer cache data structures.

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

In Fig. 6-27, we see that in addition to the collision chains starting at the hash table, there is also a bidirectional list running through all the blocks in the order of usage, with the least recently used block on the front of this list and the most recently used block at the end of this list. When a block is referenced, it can be removed from its position on the bidirectional list and put at the end. In this way, exact LRU order can be maintained.

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

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

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

For both questions, blocks can be divided into categories such as i-node blocks, indirect blocks, directory blocks, full data blocks, and partially full data blocks.

IBM Ex-1009
Page 339

Blocks that will probably not be needed again soon go on the front, rather than the rear of the LRU list, so their buffers will be reused quickly. Blocks that might be needed again soon, such as a partly full block that is being written, go on the end of the list, so they will stay around for a long time.

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

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

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

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

A consequence of this difference in caching strategy is that just removing a (floppy) disk from a UNIX system without doing a sync will almost always result in lost data, and frequently in a corrupted file system as well. With MS-DOS, no problem arises. These differing strategies were chosen because UNIX was developed in an environment in which all disks were hard disks and not removable, whereas MS-DOS started out in the floppy disk world. As hard disks became the norm, the UNIX approach, with its better efficiency, became the norm, and is also used now on Windows for hard disks.

Block Read Ahead

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

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

Reducing Disk Arm Motion

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

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

IBM Ex-1009
Page 341

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

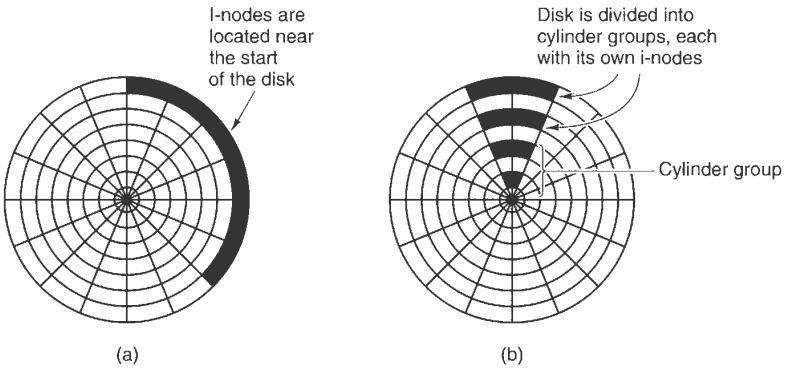


Figure 6-28. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

One easy performance improvement is to put the i-nodes in the middle of the disk, rather than at the start, thus reducing the average seek between the i-node and the first block by a factor of two. Another idea, shown in Fig. 6-28(b), is to divide the disk into cylinder groups, each with its own i-nodes, blocks, and free list (McKusick et al., 1984). When creating a new file, any i-node can be chosen, but an attempt is made to find a block in the same cylinder group as the i-node. If none is available, then a block in a nearby cylinder group is used.

6.3.8 Log-Structured File Systems

Changes in technology are putting pressure on current file systems. In particular, CPUs keep getting faster, disks are becoming much bigger and cheaper (but not much faster), and memories are growing exponentially in size. The one parameter that is not improving by leaps and bounds is disk seek time. The combination of these factors means that a performance bottleneck is arising in many file systems. Research done at Berkeley attempted to alleviate this problem by designing a completely new kind of file system, LFS (the **Log-structured File System**). In this section we will briefly describe how LFS works. For a more complete treatment, see (Rosenblum and Ousterhout, 1991).

The idea that drove the LFS design is that as CPUs get faster and RAM memories get larger, disk caches are also increasing rapidly. Consequently, it is now

possible to satisfy a very substantial fraction of all read requests directly from the file system cache, with no disk access needed. It follows from this observation, that in the future, most disk accesses will be writes, so the read-ahead mechanism used in some file systems to fetch blocks before they are needed no longer gains much performance.

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

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

From this reasoning, the LFS designers decided to re-implement the UNIX file system in such a way as to achieve the full bandwidth of the disk, even in the face of a workload consisting in large part of small random writes. The basic idea is to structure the entire disk as a log. Periodically, and when there is a special need for it, all the pending writes being buffered in memory are collected into a single segment and written to the disk as a single contiguous segment at the end of the log. A single segment may thus contain i-nodes, directory blocks, and data blocks, all mixed together. At the start of each segment is a segment summary, telling what can be found in the segment. If the average segment can be made to be about 1 MB, almost the full bandwidth of the disk can be utilized.

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

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

If disks were infinitely large, the above description would be the entire story. However, real disks are finite, so eventually the log will occupy the entire disk, at which time no new segments can be written to the log. Fortunately, many existing segments may have blocks that are no longer needed, for example, if a file is

IBM Ex-1009
Page 343

overwritten, its i-node will now point to the new blocks, but the old ones will still be occupying space in previously written segments.

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

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

6.4 EXAMPLE FILE SYSTEMS

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

6.4.1 CD-ROM File Systems

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

The ISO 9660 File System

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

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

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

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

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

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

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

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

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

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

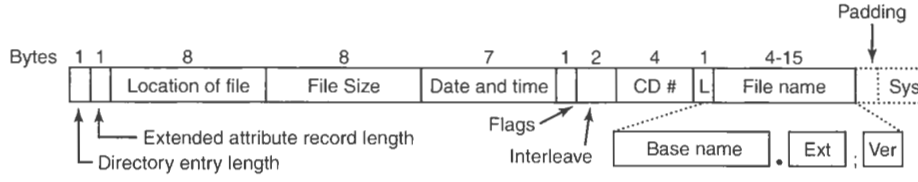


Figure 6-29. The ISO 9660 directory enty.

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

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

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

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

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

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

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

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

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

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

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

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

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

IBM Ex-1009
Page 347

Rock Ridge Extensions

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

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

The extensions are divided up into the following fields:

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

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

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

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

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

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

IBM Ex-1009
Page 348

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

Joliet Extensions

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

The major extensions provided by Joliet are

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

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

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

6.4.2 The CP/M File System

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

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

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

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

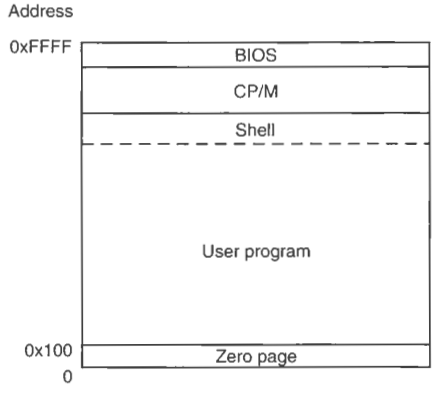


Figure 6-30. Memory layout of CP/M.

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

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

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

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

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

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

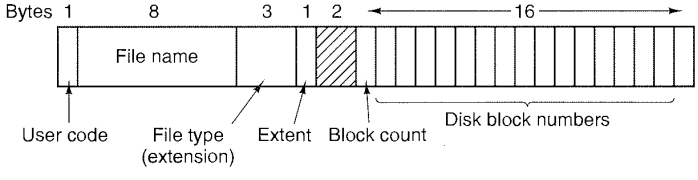


Figure 6-31. The CP/M directory entry format.

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

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

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

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

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

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

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

6.4.3 The MS-DOS File System

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

IBM Ex-1009
Page 352

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

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

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

Although MS-DOS directories are variable sized, like CP/M, they use a fixed-size 32-byte directory entry. The format of an MS-DOS directory entry is shown in Fig. 6-32. It contains the file name, attributes, creation date and time, starting block, and exact file size. File names shorter than 8 + 3 characters are left justified and padded with spaces on the right, in each field separately. The *Attributes* field is new and contains bits to indicate that a file is read-only, needs to be archived, is hidden, or is a system file. Read-only files cannot be written. This is to protect them from accidental damage. The archived bit has no actual operating system function (i.e., MS-DOS does not examine or set it). The intention is to allow user-level archive programs to clear it upon archiving a file and to have other programs set it when modifying a file. In this way, a backup program can just examine this attribute bit on every file to see which files to back up. The hidden bit can be set to prevent a file from appearing in directory listings. Its main use is to avoid confusing novice users with files they might not understand. Finally, the system bit also hides files. In addition, system files cannot accidentally be deleted using the *del* command. The main components of MS-DOS have this bit set.

The directory entry also contains the date and time the file was created or last modified. The time is accurate only to ±2 sec because it is stored in a 2-byte field, which can store only 65,536 unique values (a day contains 86,400 unique seconds). The time field is subdivided into seconds (5 bits), minutes (6 bits), and hours (5 bits). The date counts in days using three subfields: day (5 bits), month (4 bits), and year–1980 (7 bits). With a 7-bit number for the year and time

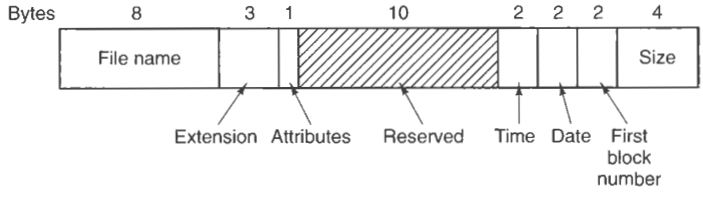


Figure 6-32. The MS-DOS directory entry.

beginning in 1980, the highest expressible year is 2107. Thus MS-DOS has a built-in Y2108 problem. To avoid catastrophe, MS-DOS users should begin with Y2108 compliance as early as possible. If MS-DOS had used the combined date and time fields as a 32-bit seconds counter, it could have represented every second exactly and delayed the catastrophe until 2116.

Unlike CP/M, which does not store the exact file size, MS-DOS does. Since a 32-bit number is used for the file size, in theory files can be as large as 4 GB. However, other limits (described below) restrict the maximum file size to 2 GB or less. A surprising large part of the entry (10 bytes) is unused.

Another way in which MS-DOS differs from CP/M is that it does not store a file's disk addresses in its directory entry, probably because the designers realized that large hard disks (by then common on minicomputers) would some day reach the MS-DOS world. Instead, MS-DOS keeps track of file blocks via a file allocation table in main memory. The directory entry contains the number of the first file block. This number is used as an index into a 64K entry FAT in main memory. By following the chain, all the blocks can be found. The operation of the FAT is illustrated in Fig. 6-14.

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

For all FATs, the disk block can be set to some multiple of 512 bytes (possibly different for each partition), with the set of allowed block sizes (called **cluster sizes** by Microsoft) being different for each variant. The first version of MS-DOS used FAT-12 with 512-byte blocks, giving a maximum partition size of $2^{12} \times 512$ bytes (actually only 4086×512 bytes because 10 of the disk addresses were used as special markers, such as end of file, bad block, etc. With these parameters, the maximum disk partition size was about 2 MB and the size of the FAT table in memory was 4096 entries of 2 bytes each. Using a 12-bit table entry would have been too slow.

This system worked well for floppy disks, but when hard disks came out, it became a problem. Microsoft solved the problem by allowing additional block

IBM Ex-1009
Page 354

sizes of 1 KB, 2 KB, and 4 KB. This change preserved the structure and size of the FAT-12 table, but allowed disk partitions of up to 16 MB.

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

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

Starting with the second release of Windows 95, the FAT-32 file system, with its 28-bit disk addresses, was introduced and the version of MS-DOS underlying Windows 95 was adapted to support FAT-32. In this system, partitions could theoretically be $2^{28} \times 2^{15}$ bytes, but they are actually limited to 2 TB (2048 GB) because internally the system keeps track of partition sizes in 512-byte sectors using a 32-bit number and $2^9 \times 2^{32}$ is 2 TB. The maximum partition size for various block sizes and all three FAT types is shown in Fig. 6-33.

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

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

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

The other advantage of FAT-32 over FAT-16 is that for a given size disk partition, a smaller block size can be used. For example, for a 2-GB disk partition, FAT-16 must use 32-KB blocks, otherwise with only 64K available disk addresses, it cannot cover the whole partition. In contrast, FAT-32 can use, for example, 4-KB blocks for a 2-GB disk partition. The advantage of the smaller block size is that most files are much shorter than 32 KB. If the block size is 32 KB, a file of 10 bytes ties up 32 KB of disk space. If the average file is, say, 8 KB, then with a 32-KB block, ¾ of the disk will be wasted, not a terribly efficient way to use the disk. With an 8-KB file and a 4-KB block, there is no disk wastage, but the price paid is more RAM eaten up by the FAT. With a 4-KB block and a 2-GB disk partition, there are 512K blocks, so the FAT must have 512K entries in memory (occupying 2 MB of RAM).

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

6.4.4 The Windows 98 File System

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

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

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

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

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

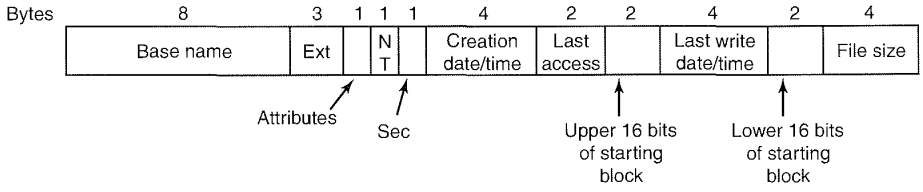


Figure 6-34. The extended MOS-DOS directory entry used in Windows 98.

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

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

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

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

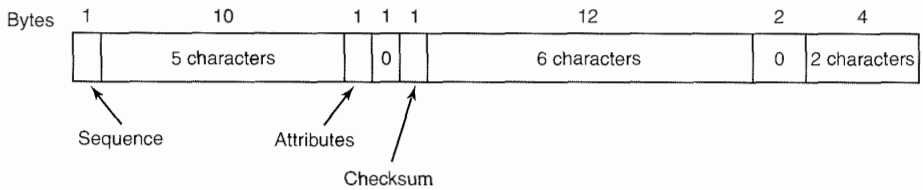


Figure 6-35. An entry for (part of) a long file name in Windows 98.

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

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

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

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

IBM Ex-1009
Page 358

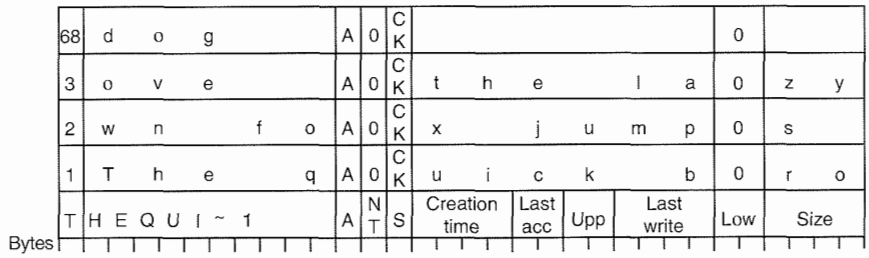


Figure 6-36. An example of how a long name is stored in Windows 98.

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

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

6.4.5 The UNIX V7 File System

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

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

A UNIX directory entry contains one entry for each file in that directory. Each entry is extremely simple because UNIX uses the i-node scheme illustrated in Fig. 6-15. A directory entry contains only two fields: the file name (14 bytes) and the number of the i-node for that file (2 bytes), as shown in Fig. 6-37. These parameters limit the number of files per file system to 64K.

Like the i-node of Fig. 6-15, the UNIX i-nodes contains some attributes. The attributes contain the file size, three times (creation, last access, and last modification), owner, group, protection information, and a count of the number of directory entries that point to the i-node. The latter field is needed due to links. Whenever a new link is made to an i-node, the count in the i-node is increased. When a

IBM Ex-1009
Page 359

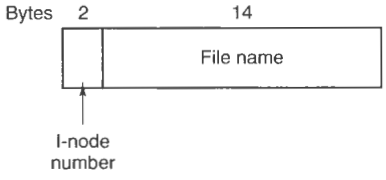


Figure 6-37. A UNIX V7 directory entry.

link is removed, the count is decremented. When it gets to 0, the i-node is reclaimed and the disk blocks are put back in the free list.

Keeping track of disk blocks is done using a generalization of Fig. 6-15 in order to handle very large files. The first 10 disk addresses are stored in the i-node itself, so for small files, all the necessary information is right in the i-node, which is fetched from disk to main memory when the file is opened. For somewhat larger files, one of the addresses in the i-node is the address of a disk block called a **single indirect block**. This block contains additional disk addresses. If this still is not enough, another address in the i-node, called a **double indirect block**, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a **triple indirect block** can also be used. The complete picture is given in Fig. 6-38.

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

Then it reads the root directory and looks up the first component of the path, *usr*, in the root directory to find the i-node number of the file */usr*. Locating an i-node from its number is straightforward, since each one has a fixed location on the disk. From this i-node, the system locates the directory for */usr* and looks up the next component, *ast*, in it. When it has found the entry for *ast*, it has the i-node for the directory */usr/ast*. From this i-node it can find the directory itself and look up *mbox*. The i-node for this file is then read into memory and kept there until the file is closed. The lookup process is illustrated in Fig. 6-39.

Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of starting from the root directory. Every directory has entries for . and .. which are put there when the directory is created. The entry . has the i-node number for the current directory, and the entry for .. has the i-node number for the parent directory. Thus, a procedure looking up *../dick/prog.c* simply looks up .. in the working directory, finds the i-node number

IBM Ex-1009
Page 360

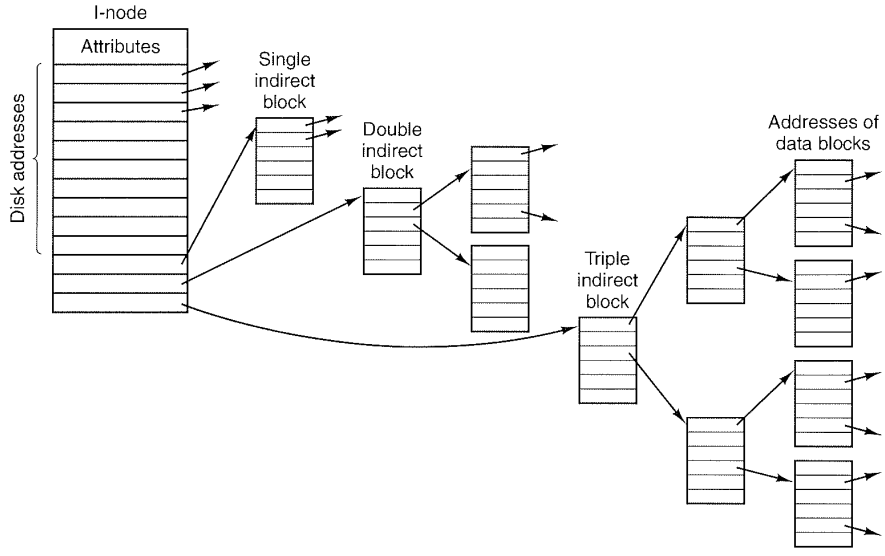


Figure 6-38. A UNIX i-node.

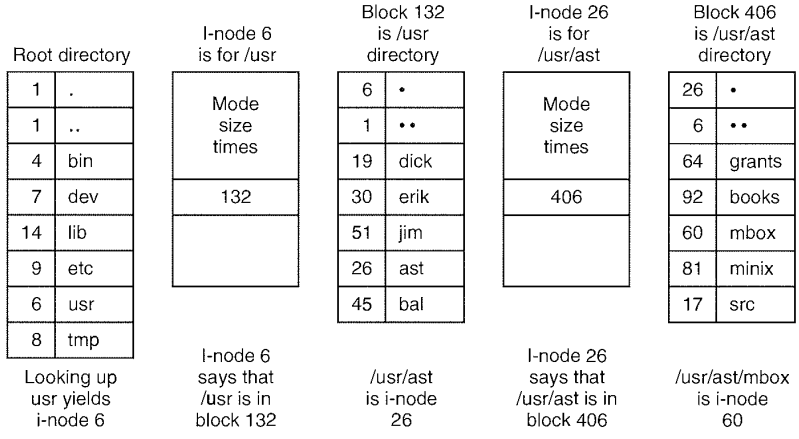


Figure 6-39. The steps in looking up */usr/ast/mbox*.

for the parent directory, and searches that directory for *dick*. No special mechanism is needed to handle these names. As far as the directory system is concerned, they are just ordinary ASCII strings, just the same as any other names.

6.5 RESEARCH ON FILE SYSTEMS

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

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

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

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

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

6.6 SUMMARY

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

IBM Ex-1009
Page 362

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

Examples of file systems include ISO 9660, CP/M, MS-DOS, Windows 98, and UNIX. These differ in many ways, including how they keep track of which blocks go with which file, directory structure, and management of free disk space.

PROBLEMS

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

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

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

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

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

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

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

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

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

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

18. It has been suggested that efficiency could be improved and disk space saved by storing the data of a short file within the i-node. For the i-node of Fig. 6-15, how many bytes of data could be stored inside the i-node?

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

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

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

IBM Ex-1009
Page 364

22. The beginning of a free space bitmap looks like this after the disk partition is first formatted: 1000 0000 0000 0000 (the first block is used by the root directory). The system always searches for free blocks starting at the lowest numbered block, so after writing file *A*, which uses 6 blocks, the bitmap looks like this: 1111 1110 0000 0000. Show the bitmap after each of the following additional actions:

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

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

24. Oliver Owl's night job at the university computing center is to change the tapes used for overnight data backups. While waiting for each tape to complete, he works on writing his thesis that proves Shakespeare's plays were written by extraterrestrial visitors. His text processor runs on the system being backed up since that is the only one they have. Is there a problem with this arrangement?

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

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

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

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

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

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

31. Consider the idea behind Fig. 6-20, but now for a disk with a mean seek time of 8 msec, a rotational rate of 15,000 rpm, and 262,144 bytes per track. What are the data rates for block sizes of 1 KB, 2 KB, and 4 KB, respectively?

IBM Ex-1009
Page 365

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

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

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

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

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

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

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

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

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

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

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

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

8

MULTIPLE PROCESSOR SYSTEMS

Since its inception, the computer industry has been driven by an endless quest for more and more computing power. The ENIAC could perform 300 operations per second, easily 1000 times faster than any calculator before it, yet people were not satisfied. We now have machines a million times faster than the ENIAC and still there is a demand for yet more horsepower. Astronomers are trying to make sense of the universe, biologists are trying to understand the implications of the human genome, and aeronautical engineers are interested in building safer and more efficient aircraft, and all want more CPU cycles. However much computing power there is, it is never enough.

In the past, the solution was always to make the clock run faster. Unfortunately, we are beginning to hit some fundamental limits on clock speed. According to Einstein's special theory of relativity, no electrical signal can propagate faster than the speed of light, which is about 30 cm/nsec in vacuum and about 20 cm/nsec in copper wire or optical fiber. This means that in a computer with a 10-GHz clock, the signals cannot travel more than 2 cm in total. For a 100-GHz computer the total path length is at most 2 mm. A 1-THz (1000 GHz) computer will have to be smaller than 100 microns, just to let the signal get from one end to the other and back once with a single clock cycle.

Making computers this small may be possible, but then we hit another fundamental problem: heat dissipation. The faster runs the computer, the more heat it generates, and the smaller the computer, the harder it is to get rid of this heat.

IBM Ex-1009
Page 367

Already on high-end Pentium systems, the CPU cooler is bigger than the CPU itself. All in all, going from 1 MHz to 1 GHz simply required incrementally better engineering of the chip manufacturing process. Going from 1 GHz to 1 THz is going to require a radically different approach.

One approach to greater speed is through massively parallel computers. These machines consist of many CPUs, each of which runs at "normal" speed (whatever that may mean in a given year), but which collectively have far more computing power than a single CPU. Systems with 1000 CPUs are now commercially available. Systems with 1 million CPUs are likely to be built in the coming decade. While there are other potential approaches to greater speed, such as biological computers, in this chapter we will focus on systems with multiple conventional CPUs.

Highly parallel computers are often used for heavy number crunching. Problems such as predicting the weather, modeling airflow around an aircraft wing, simulating the world economy, or understanding drug-receptor interactions in the brain are all computationally intensive. Their solutions require long runs on many CPUs at once. The multiple processor systems discussed in this chapter are widely-used for these and similar problems in science and engineering, among other areas.

Another relevant development is the incredibly rapid growth of the Internet. It was originally designed as a prototype for a fault-tolerant military control system, then became popular among academic computer scientists, and has recently acquired many new uses. One of these is linking up thousands of computers all over the world to work together on large scientific problems. In a sense, a system consisting of 1000 computers spread all over the world is no different than one consisting of 1000 computers in a single room, although the delay and other technical characteristics are different. We will also consider these systems in this chapter.

Putting 1 million unrelated computers in a room is easy to do provided that you have enough money and a sufficiently large room. Spreading 1 million unrelated computers around the world is even easier since it finesses the second problem. The trouble comes in when you want them to communicate with one another to work together on a single problem. As a consequence, a great deal of work has been done on the interconnection technology, and different interconnect technologies have led to qualitatively different kinds of systems and different software organizations.

All communication between electronic (or optical) components ultimately comes down to sending messages—well-defined bit strings—between them. The differences are in the time scale, distance scale, and logical organization involved. At one extreme are the shared-memory multiprocessors, systems in which somewhere between two and about 1000 CPUs communicate via a shared memory. In this model, every CPU has equal access to the entire physical memory, and can read and write individual words using LOAD and STORE instructions. Accessing a

memory word usually takes 10–50 nsec. While this model, illustrated in Fig. 8-1(a), sounds simple, actually implementing it is far from simple and usually involves considerable message passing under the covers, as we will explain shortly.

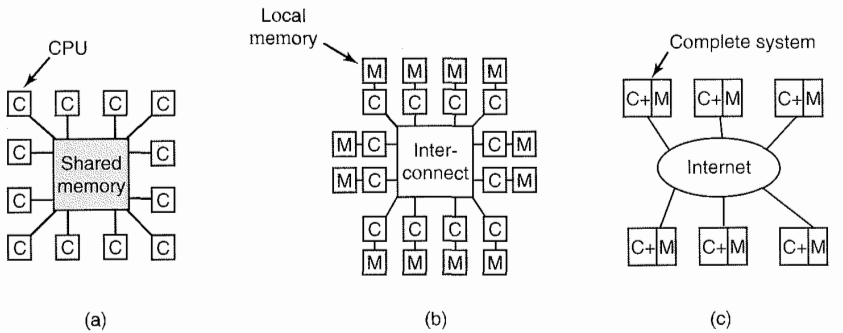


Figure 8-1. (a) A shared-memory multiprocessor. (b) A message-passing multicomputer. (c) A wide area distributed system.

Next comes the system of Fig. 8-1(b) in which a number of CPU-memory pairs are connected by some kind of high-speed interconnect. This kind of system is called a message-passing multicomputer. Each memory is local to a single CPU and can be accessed only by that CPU. The machines communicate by sending multiword messages over the interconnect. With a good interconnect, a short message can be sent in 10–50 μsec, but still far longer than the memory access time of Fig. 8-1(a). There is no shared global memory in this design. Multicomputers (i.e., message-passing systems) are much easier to build than (shared-memory) multiprocessors but they are harder to program. Thus each genre has its fans.

The third model, illustrated in Fig. 8-1(c), connects complete computer systems over a wide area network, such as the Internet, to form a **distributed system**. Each of these has its own memory, of course, and the systems communicate by message passing. The only real difference between Fig. 8-1(c) and Fig. 8-1(b) is that in the former, complete computers are used and message times are often 10–50 msec. This long delay forces these **loosely-coupled** systems to be used in different ways than the **tightly-coupled** systems of Fig. 8-1(b). The three types of systems differ in their delays by something like three orders of magnitude. That is the difference between a day and three years.

This chapter has three major sections, corresponding to the three models of Fig. 8-1. In each one, we start out with a brief introduction to the relevant hardware. Then we move on to the software, especially the operating system issues for that type of system. As we will see, in each case different issues are present.

IBM Ex-1009
Page 369

8.1 MULTIPROCESSORS

A **shared-memory multiprocessor** (or just multiprocessor henceforth) is a computer system in which two or more CPUs share full access to a common RAM. A program running on any of the CPUs sees a normal (usually paged) virtual address space. The only unusual property this system has is that the CPU can write some value into a memory word and then read the word back and get a different value (because another CPU has changed it). When organized correctly, this property forms the basis of interprocessor communication: one CPU writes some data into memory and another one reads the data out.

For the most part, multiprocessor operating systems are just regular operating systems. They handle system calls, do memory management, provide a file system, and manage I/O devices. Nevertheless, there are some areas in which they have unique features. These include process synchronization, resource management, and scheduling. Below we will first take a brief look at multiprocessor hardware and then move on to these operating systems issues.

8.1.1 Multiprocessor Hardware

Although all multiprocessors have the property that every CPU can address all of memory, some multiprocessors have the additional property that every memory word can be read as fast as every other memory word. These machines are called **UMA (Uniform Memory Access)** multiprocessors. In contrast, **NUMA (Nonuniform Memory Access)** multiprocessors do not have this property. Why this difference exists will become clear later. We will first examine UMA multiprocessors and then move on to NUMA multiprocessors.

UMA Bus-Based SMP Architectures

The simplest multiprocessors are based on a single bus, as illustrated in Fig. 8-2(a). Two or more CPUs and one or more memory modules all use the same bus for communication. When a CPU wants to read a memory word, it first checks to see if the bus is busy. If the bus is idle, the CPU puts the address of the word it wants on the bus, asserts a few control signals, and waits until the memory puts the desired word on the bus.

If the bus is busy when a CPU wants to read or write memory, the CPU just waits until the bus becomes idle. Herein lies the problem with this design. With two or three CPUs, contention for the bus will be manageable; with 32 or 64 it will be unbearable. The system will be totally limited by the bandwidth of the bus, and most of the CPUs will be idle most of the time.

The solution to this problem is to add a cache to each CPU, as depicted in Fig. 8-2(b). The cache can be inside the CPU chip, next to the CPU chip, on the

IBM Ex-1009
Page 370

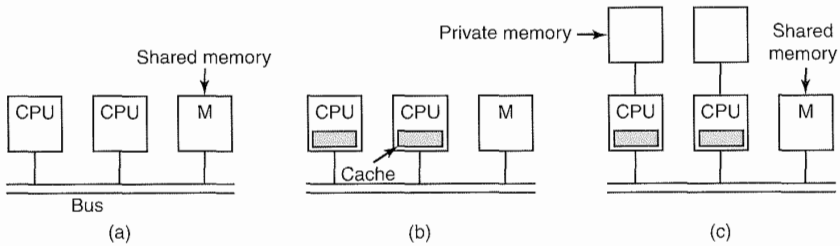


Figure 8-2. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

processor board, or some combination of all three. Since many reads can now be satisfied out of the local cache, there will be much less bus traffic, and the system can support more CPUs. In general, caching is not done on an individual word basis but on the basis of 32- or 64-byte blocks. When a word is referenced, its entire block is fetched into the cache of the CPU touching it.

Each cache block is marked as being either read-only (in which case it can be present in multiple caches at the same time), or as read-write (in which case it may not be present in any other caches). If a CPU attempts to write a word that is in one or more remote caches, the bus hardware detects the write and puts a signal on the bus informing all other caches of the write. If other caches have a "clean" copy, that is, an exact copy of what is in memory, they can just discard their copies and let the writer fetch the cache block from memory before modifying it. If some other cache has a "dirty" (i.e., modified) copy, it must either write it back to memory before the write can proceed or transfer it directly to the writer over the bus. Many cache transfer protocols exist.

Yet another possibility is the design of Fig. 8-2(c), in which each CPU has not only a cache, but also a local, private memory which it accesses over a dedicated (private) bus. To use this configuration optimally, the compiler should place all the program text, strings, constants and other read-only data, stacks, and local variables in the private memories. The shared memory is then only used for writable shared variables. In most cases, this careful placement will greatly reduce bus traffic, but it does require active cooperation from the compiler.

UMA Multiprocessors Using Crossbar Switches

Even with the best caching, the use of a single bus limits the size of a UMA multiprocessor to about 16 or 32 CPUs. To go beyond that, a different kind of interconnection network is needed. The simplest circuit for connecting n CPUs to k memories is the **crossbar switch**, shown in Fig. 8-3. Crossbar switches have been used for decades within telephone switching exchanges to connect a group of incoming lines to a set of outgoing lines in an arbitrary way.

IBM Ex-1009
Page 371

At each intersection of a horizontal (incoming) and vertical (outgoing) line is a **crosspoint**. A crosspoint is a small switch that can be electrically opened or closed, depending on whether the horizontal and vertical lines are to be connected or not. In Fig. 8-3(a) we see three crosspoints closed simultaneously, allowing connections between the (CPU, memory) pairs (001, 000), (101, 101), and (110, 010) at the same time. Many other combinations are also possible. In fact, the number of combinations is equal to the number of different ways eight rooks can be safely placed on a chess board.

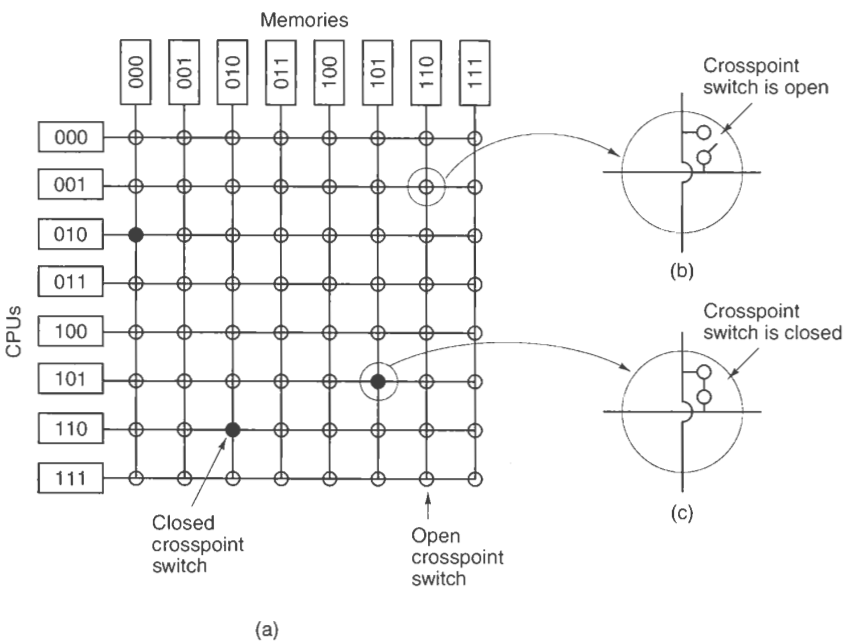


Figure 8-3. (a) An 8 × 8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

One of the nicest properties of the crossbar switch is that it is a **nonblocking network**, meaning that no CPU is ever denied the connection it needs because some crosspoint or line is already occupied (assuming the memory module itself is available). Furthermore, no advance planning is needed. Even if seven arbitrary connections are already set up, it is always possible to connect the remaining CPU to the remaining memory.

One of the worst properties of the crossbar switch is the fact that the number of crosspoints grows as n^2. With 1000 CPUs and 1000 memory modules we need a million crosspoints. Such a large crossbar switch is not feasible. Nevertheless, for medium-sized systems, a crossbar design is workable.

IBM Ex-1009
Page 372

UMA Multiprocessors Using Multistage Switching Networks

A completely different multiprocessor design is based on the humble 2×2 switch shown in Fig. 8-4(a). This switch has two inputs and two outputs. Messages arriving on either input line can be switched to either output line. For our purposes, messages will contain up to four parts, as shown in Fig. 8-4(b). The *Module* field tells which memory to use. The *Address* specifies an address within a module. The *Opcode* gives the operation, such as READ or WRITE. Finally, the optional *Value* field may contain an operand, such as a 32-bit word to be written on a WRITE. The switch inspects the *Module* field and uses it to determine if the message should be sent on *X* or on *Y*.

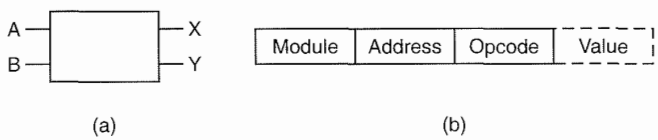


Figure 8-4. (a) A 2×2 switch. (b) A message format.

Our 2×2 switches can be arranged in many ways to build larger **multistage switching networks** (Adams et al., 1987; Bhuyan et al., 1989; and Kumar and Reddy, 1987). One possibility is the no-frills, economy class **omega network**, illustrated in Fig. 8-5. Here we have connected eight CPUs to eight memories using 12 switches. More generally, for n CPUs and n memories we would need $\log_2 n$ stages, with $n/2$ switches per stage, for a total of $(n/2)\log_2 n$ switches, which is a lot better than n^2 crosspoints, especially for large values of n.

The wiring pattern of the omega network is often called the **perfect shuffle**, since the mixing of the signals at each stage resembles a deck of cards being cut in half and then mixed card-for-card. To see how the omega network works, suppose that CPU 011 wants to read a word from memory module 110. The CPU sends a READ message to switch 1D containing 110 in the *Module* field. The switch takes the first (i.e., leftmost) bit of 110 and uses it for routing. A 0 routes to the upper output and a 1 routes to the lower one. Since this bit is a 1, the message is routed via the lower output to 2D.

All the second-stage switches, including 2D, use the second bit for routing. This, too, is a 1, so the message is now forwarded via the lower output to 3D. Here the third bit is tested and found to be a 0. Consequently, the message goes out on the upper output and arrives at memory 110, as desired. The path followed by this message is marked in Fig. 8-5 by the letter *a*.

As the message moves through the switching network, the bits at the left-hand end of the module number are no longer needed. They can be put to good use by recording the incoming line number there, so the reply can find its way back. For

IBM Ex-1009
Page 373

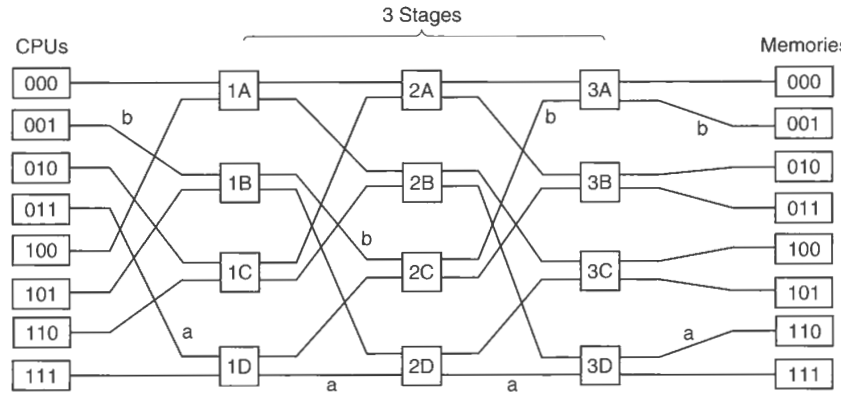


Figure 8-5. An omega switching network.

path *a*, the incoming lines are 0 (upper input to 1D), 1 (lower input to 2D), and 1 (lower input to 3D), respectively. The reply is routed back using 011, only reading it from right to left this time.

At the same time all this is going on, CPU 001 wants to write a word to memory module 001. An analogous process happens here, with the message routed via the upper, upper, and lower outputs, respectively, marked by the letter *b*. When it arrives, its *Module* field reads 001, representing the path it took. Since these two requests do not use any of the same switches, lines, or memory modules, they can proceed in parallel.

Now consider what would happen if CPU 000 simultaneously wanted to access memory module 000. Its request would come into conflict with CPU 001's request at switch 3A. One of them would have to wait. Unlike the crossbar switch, the omega network is a **blocking network**. Not every set of requests can be processed simultaneously. Conflicts can occur over the use of a wire or a switch, as well as between requests *to* memory and replies *from* memory.

It is clearly desirable to spread the memory references uniformly across the modules. One common technique is to use the low-order bits as the module number. Consider, for example, a byte-oriented address space for a computer that mostly accesses 32-bit words. The 2 low-order bits will usually be 00, but the next 3 bits will be uniformly distributed. By using these 3 bits as the module number, consecutively addressed words will be in consecutive modules. A memory system in which consecutive words are in different modules is said to be **interleaved**. Interleaved memories maximize parallelism because most memory references are to consecutive addresses. It is also possible to design switching networks that are nonblocking and which offer multiple paths from each CPU to each memory module, to spread the traffic better.

IBM Ex-1009
Page 374

NUMA Multiprocessors

Single-bus UMA multiprocessors are generally limited to no more than a few dozen CPUs and crossbar or switched multiprocessors need a lot of (expensive) hardware and are not that much bigger. To get to more than 100 CPUs, something has to give. Usually, what gives is the idea that all memory modules have the same access time. This concession leads to the idea of NUMA multiprocessors, as mentioned above. Like their UMA cousins, they provide a single address space across all the CPUs, but unlike the UMA machines, access to local memory modules is faster than access to remote ones. Thus all UMA programs will run without change on NUMA machines, but the performance will be worse than on a UMA machine at the same clock speed.

NUMA machines have three key characteristics that all of them possess and which together distinguish them from other multiprocessors:

1. There is a single address space visible to all CPUs.
2. Access to remote memory is via LOAD and STORE instructions.
3. Access to remote memory is slower than access to local memory.

When the access time to remote memory is not hidden (because there is no caching), the system is called **NC-NUMA**. When coherent caches are present, the system is called **CC-NUMA (Cache-Coherent NUMA)**.

The most popular approach for building large CC-NUMA multiprocessors currently is the **directory-based multiprocessor**. The idea is to maintain a database telling where each cache line is and what its status is. When a cache line is referenced, the database is queried to find out where it is and whether it is clean or dirty (modified). Since this database must be queried on every instruction that references memory, it must be kept in extremely-fast special-purpose hardware that can respond in a fraction of a bus cycle.

To make the idea of a directory-based multiprocessor somewhat more concrete, let us consider as a simple (hypothetical) example, a 256-node system, each node consisting of one CPU and 16 MB of RAM connected to the CPU via a local bus. The total memory is 2^{32} bytes, divided up into 2^{26} cache lines of 64 bytes each. The memory is statically allocated among the nodes, with 0–16M in node 0, 16M–32M in node 1, and so on. The nodes are connected by an interconnection network, as shown in Fig. 8-6(a). Each node also holds the directory entries for the 2^{18} 64-byte cache lines comprising its 2^{24} byte memory. For the moment, we will assume that a line can be held in at most one cache.

To see how the directory works, let us trace a LOAD instruction from CPU 20 that references a cached line. First the CPU issuing the instruction presents it to its MMU, which translates it to a physical address, say, 0x24000108. The MMU splits this address into the three parts shown in Fig. 8-6(b). In decimal, the three

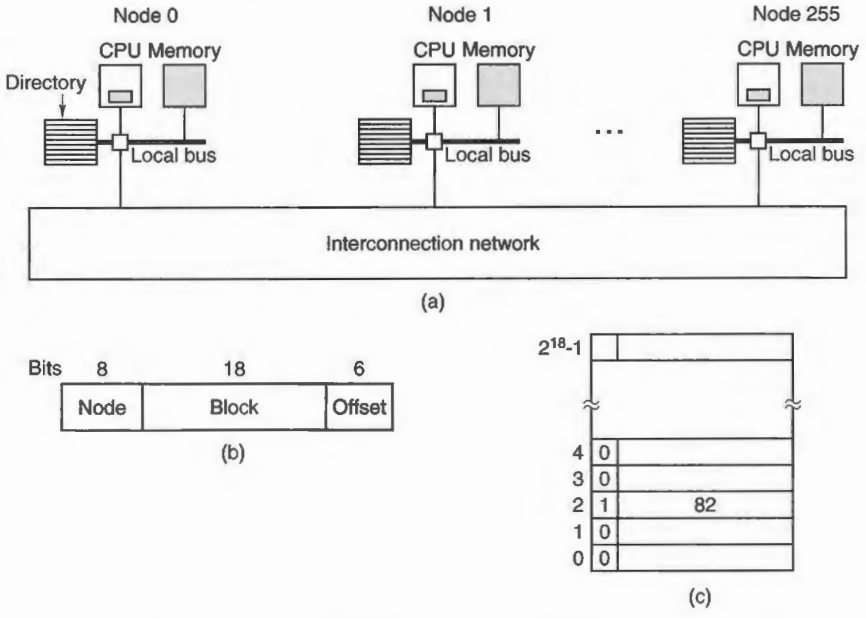


Figure 8-6. (a) A 256-node directory-based multiprocessor. (b) Division of a 32-bit memory address into fields. (c) The directory at node 36.

parts are node 36, line 4, and offset 8. The MMU sees that the memory word referenced is from node 36, not node 20, so it sends a request message through the interconnection network to the line's home node, 36, asking whether its line 4 is cached, and if so, where.

When the request arrives at node 36 over the interconnection network, it is routed to the directory hardware. The hardware indexes into its table of 2^{18} entries, one for each of its cache lines and extracts entry 4. From Fig. 8-6(c) we see that the line is not cached, so the hardware fetches line 4 from the local RAM, sends it back to node 20, and updates directory entry 4 to indicate that the line is now cached at node 20.

Now let us consider a second request, this time asking about node 36's line 2. From Fig. 8-6(c) we see that this line is cached at node 82. At this point the hardware could update directory entry 2 to say that the line is now at node 20 and then send a message to node 82 instructing it to pass the line to node 20 and invalidate its cache. Note that even a so-called "shared-memory multiprocessor" has a lot of message passing going on under the hood.

As a quick aside, let us calculate how much memory is being taken up by the directories. Each node has 16 MB of RAM and 2^{18} 9-bit entries to keep track of that RAM. Thus the directory overhead is about 9×2^{18} bits divided by 16 MB or

IBM Ex-1009
Page 376

about 1.76 percent, which is generally acceptable (although it has to be high-speed memory, which increases its cost). Even with 32-byte cache lines the overhead would only be 4 percent. With 128-byte cache lines, it would be under 1 percent.

An obvious limitation of this design is that a line can be cached at only one node. To allow lines to be cached at multiple nodes, we would need some way of locating all of them, for example, to invalidate or update them on a write. Various options are possible to allow caching at several nodes at the same time, but a discussion of these is beyond the scope of this book.

8.1.2 Multiprocessor Operating System Types

Let us now turn from multiprocessor hardware to multiprocessor software, in particular, multiprocessor operating systems. Various organizations are possible. Below we will study three of them.

Each CPU Has Its Own Operating System

The simplest possible way to organize a multiprocessor operating system is to statically divide memory into as many partitions as there are CPUs and give each CPU its own private memory and its own private copy of the operating system. In effect, the n CPUs then operate as n independent computers. One obvious optimization is to allow all the CPUs to share the operating system code and make private copies of only the data, as shown in Fig. 8-7.

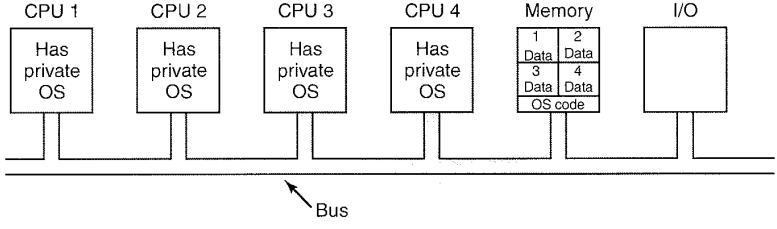


Figure 8-7. Partitioning multiprocessor memory among four CPUs, but sharing a single copy of the operating system code. The boxes marked Data are the operating system's private data for each CPU.

This scheme is still better than having n separate computers since it allows all the machines to share a set of disks and other I/O devices, and it also allows the memory to be shared flexibly. For example, if one day an unusually large program has to be run, one of the CPUs can be allocated an extra large portion of memory for the duration of that program. In addition, processes can efficiently communicate with one another by having, say a producer be able to write data into memory and have a consumer fetch it from the place the producer wrote it.

IBM Ex-1009
Page 377

Still, from an operating systems' perspective, having each CPU have its own operating system is as primitive as it gets.

It is worth explicitly mentioning four aspects of this design that may not be obvious. First, when a process makes a system call, the system call is caught and handled on its own CPU using the data structures in that operating system's tables.

Second, since each operating system has its own tables, it also has its own set of processes that it schedules by itself. There is no sharing of processes. If a user logs into CPU 1, all of his processes run on CPU 1. As a consequence, it can happen that CPU 1 is idle while CPU 2 is loaded with work.

Third, there is no sharing of pages. It can happen that CPU 1 has pages to spare while CPU 2 is paging continuously. There is no way for CPU 2 to borrow some pages from CPU 1 since the memory allocation is fixed.

Fourth, and worst, if the operating system maintains a buffer cache of recently used disk blocks, each operating system does this independently of the other ones. Thus it can happen that a certain disk block is present and dirty in multiple buffer caches at the same time, leading to inconsistent results. The only way to avoid this problem is to eliminate the buffer caches. Doing so is not hard, but it hurts performance considerably.

Master-Slave Multiprocessors

For these reasons, this model is rarely used any more, although it was used in the early days of multiprocessors, when the goal was to port existing operating systems to some new multiprocessor as fast as possible. A second model is shown in Fig. 8-8. Here, one copy of the operating system and its tables are present on CPU 1 and not on any of the others. All system calls are redirected to CPU 1 for processing there. CPU 1 may also run user processes if there is CPU time left over. This model is called **master-slave** since CPU 1 is the master and all the others are slaves.

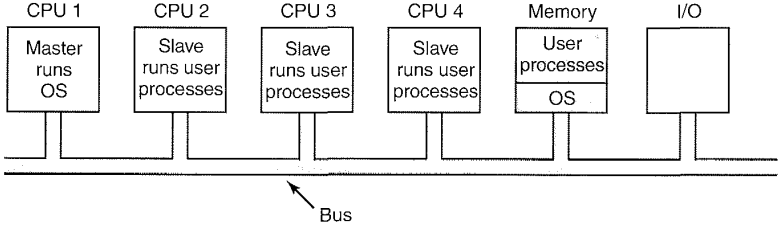


Figure 8-8. A master-slave multiprocessor model.

The master-slave model solves most of the problems of the first model. There is a single data structure (e.g., one list or a set of prioritized lists) that keeps track

of ready processes. When a CPU goes idle, it asks the operating system for a process to run and it is assigned one. Thus it can never happen that one CPU is idle while another is overloaded. Similarly, pages can be allocated among all the processes dynamically and there is only one buffer cache, so inconsistencies never occur.

The problem with this model is that with many CPUs, the master will become a bottleneck. After all, it must handle all system calls from all CPUs. If, say, 10% of all time is spent handling system calls, then 10 CPUs will pretty much saturate the master, and with 20 CPUs it will be completely overloaded. Thus this model is simple and workable for small multiprocessors, but for large ones it fails.

Symmetric Multiprocessors

Our third model, the **SMP (Symmetric MultiProcessor)**, eliminates this asymmetry. There is one copy of the operating system in memory, but any CPU can run it. When a system call is made, the CPU on which the system call was made traps to the kernel and processes the system call. The SMP model is illustrated in Fig. 8-9.

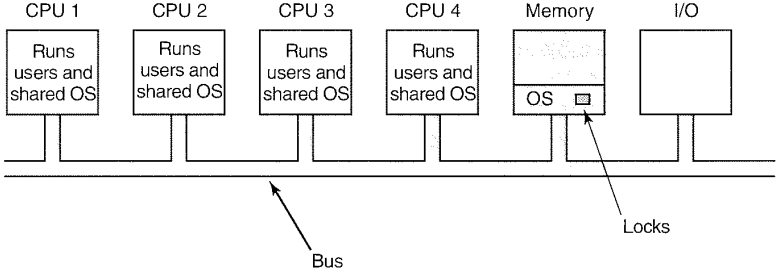


Figure 8-9. The SMP multiprocessor model.

This model balances processes and memory dynamically, since there is only one set of operating system tables. It also eliminates the master CPU bottleneck, since there is no master, but it introduces its own problems. In particular, if two or more CPUs are running operating system code at the same time, disaster will result. Imagine two CPUs simultaneously picking the same process to run or claiming the same free memory page. The simplest way around these problems is to associate a mutex (i.e., lock) with the operating system, making the whole system one big critical region. When a CPU wants to run operating system code, it must first acquire the mutex. If the mutex is locked, it just waits. In this way, any CPU can run the operating system, but only one at a time.

This model works, but is almost as bad as the master-slave model. Again, suppose that 10% of all run time is spent inside the operating system. With 20 CPUs, there will be long queues of CPUs waiting to get in. Fortunately, it is easy

IBM Ex-1009
Page 379

to improve. Many parts of the operating system are independent of one another. For example, there is no problem with one CPU running the scheduler while another CPU is handling a file system call and a third one is processing a page fault.

This observation leads to splitting the operating system up into independent critical regions that do not interact with one another. Each critical region is protected by its own mutex, so only one CPU at a time can execute it. In this way, far more parallelism can be achieved. However, it may well happen that some tables, such as the process table, are used by multiple critical regions. For example, the process table is needed for scheduling, but also for the fork system call and also for signal handling. Each table that may be used by multiple critical regions needs its own mutex. In this way, each critical region can be executed by only one CPU at a time and each critical table can be accessed by only one CPU at a time.

Most modern multiprocessors use this arrangement. The hard part about writing the operating system for such a machine is not that the actual code is so different from a regular operating system. It is not. The hard part is splitting it into critical regions that can be executed concurrently by different CPUs without interfering with one another, not even in subtle, indirect ways. In addition, every table used by two or more critical regions must be separately protected by a mutex and all code using the table must use the mutex correctly.

Furthermore, great care must be taken to avoid deadlocks. If two critical regions both need table *A* and table *B*, and one of them claims *A* first and the other claims *B* first, sooner or later a deadlock will occur and nobody will know why. In theory, all the tables could be assigned integer values and all the critical regions could be required to acquire tables in increasing order. This strategy avoids deadlocks, but it requires the programmer to think very carefully which tables each critical region needs to make the requests in the right order.

As the code evolves over time, a critical region may need a new table it did not previously need. If the programmer is new and does not understand the full logic of the system, then the temptation will be to just grab the mutex on the table at the point it is needed and release it when it is no longer needed. However reasonable this may appear, it may lead to deadlocks, which the user will perceive as the system freezing. Getting it right is not easy and keeping it right over a period of years in the face of changing programmers is very difficult.

8.1.3 Multiprocessor Synchronization

The CPUs in a multiprocessor frequently need to synchronize. We just saw the case in which kernel critical regions and tables have to be protected by mutexes. Let us now take a close look at how this synchronization actually works in a multiprocessor. It is far from trivial, as we will soon see.

IBM Ex-1009
Page 380

To start with, proper synchronization primitives are really needed. If a process on a uniprocessor makes a system call that requires accessing some critical kernel table, the kernel code can just disable interrupts before touching the table. It can then do its work knowing that it will be able to finish without any other process sneaking in and touching the table before it is finished. On a multiprocessor, disabling interrupts affects only the CPU doing the disable. Other CPUs continue to run and can still touch the critical table. As a consequence, a proper mutex protocol must be used and respected by all CPUs to guarantee that mutual exclusion works.

The heart of any practical mutex protocol is an instruction that allows a memory word to be inspected and set in one indivisible operation. We saw how TSL (Test and Set Lock) was used in Fig. 2-22 to implement critical regions. As we discussed earlier, what this instruction does is read out a memory word and store it in a register. Simultaneously, it writes a 1 (or some other nonzero value) into the memory word. Of course, it takes two separate bus cycles to perform the memory read and memory write. On a uniprocessor, as long as the instruction cannot be broken off halfway, TSL always works as expected.

Now think about what could happen on a multiprocessor. In Fig. 8-10 we see the worst case timing, in which memory word 1000, being used as a lock is initially 0. In step 1, CPU 1 reads out the word and gets a 0. In step 2, before CPU 1 has a chance to rewrite the word to 1, CPU 2 gets in and also reads the word out as a 0. In step 3, CPU 1 writes a 1 into the word. In step 4, CPU 2 also writes a 1 into the word. Both CPUs got a 0 back from the TSL instruction, so both of them now have access to the critical region and the mutual exclusion fails.

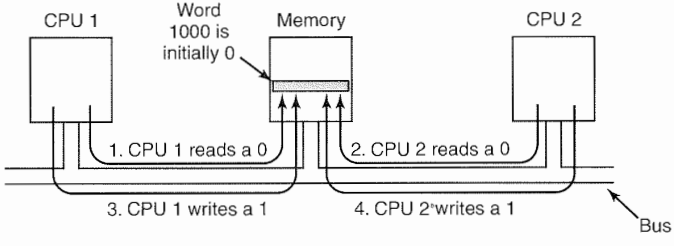


Figure 8-10. The TSL instruction can fail if the bus cannot be locked. These four steps show a sequence of events where the failure is demonstrated.

To prevent this problem, the TSL instruction must first lock the bus, preventing other CPUs from accessing it, then do both memory accesses, then unlock the bus. Typically, locking the bus is done by requesting the bus using the usual bus request protocol, then asserting (i.e., setting to a logical 1) some special bus line until *both* cycles have been completed. As long as this special line is being asserted, no other CPU will be granted bus access. This instruction can only be

IBM Ex-1009
Page 381

implemented on a bus that has the necessary lines and (hardware) protocol for using them. Modern buses have these facilities, but on earlier ones that did not, it was not possible to implement TSL correctly. This is why Peterson's protocol was invented, to synchronize entirely in software (Peterson, 1981).

If TSL is correctly implemented and used, it guarantees that mutual exclusion can be made to work. However, this mutual exclusion method uses a **spin lock** because the requesting CPU just sits in a tight loop testing the lock as fast as it can. Not only does it completely waste the time of the requesting CPU (or CPUs), but it may also put a massive load on the bus or memory, seriously slowing down all other CPUs trying to do their normal work.

At first glance, it might appear that the presence of caching should eliminate the problem of bus contention, but it does not. In theory, once the requesting CPU has read the lock word, it should get a copy in its cache. As long as no other CPU attempts to use the lock, the requesting CPU should be able to run out of its cache. When the CPU owning the lock writes a 1 to it to release it, the cache protocol automatically invalidates all copies of it in remote caches requiring the correct value to be fetched again.

The problem is that caches operate in blocks of 32 or 64 bytes. Usually, the words surrounding the lock are needed by the CPU holding the lock. Since the TSL instruction is a write (because it modifies the lock), it needs exclusive access to the cache block containing the lock. Therefore every TSL invalidates the block in the lock holder's cache and fetches a private, exclusive copy for the requesting CPU. As soon as the lock holder touches a word adjacent to the lock, the cache block is moved to its machine. Consequently, the entire cache block containing the lock is constantly being shuttled between the lock owner and the lock requester, generating even more bus traffic than individual reads on the lock word would have.

If we could get rid of all the TSL-induced writes on the requesting side, we could reduce cache thrashing appreciably. This goal can be accomplished by having the requesting CPU first do a pure read to see if the lock is free. Only if the lock appears to be free does it do a TSL to actually acquire it. The result of this small change is that most of the polls are now reads instead of writes. If the CPU holding the lock is only reading the variables in the same cache block, they can each have a copy of the cache block in shared read-only mode, eliminating all the cache block transfers. When the lock is finally freed, the owner does a write, which requires exclusive access, thus invalidating all the other copies in remote caches. On the next read by the requesting CPU, the cache block will be reloaded. Note that if two or more CPUs are contending for the same lock, it can happen that both see that it is free simultaneously, and both do a TSL simultaneously to acquire it. Only one of these will succeed, so there is no race condition here because the real acquisition is done by the TSL instruction, and this instruction is atomic. Seeing that the lock is free and then trying to grab it immediately with a CX u TSL does not guarantee that you get it. Someone else might win.

IBM Ex-1009
Page 382

Another way to reduce bus traffic is to use the Ethernet binary exponential backoff algorithm (Anderson, 1990). Instead of continuously polling, as in Fig. 2-22, a delay loop can be inserted between polls. Initially the delay is one instruction. If the lock is still busy, the delay is doubled to two instructions, then four instructions and so on up to some maximum. A low maximum gives fast response when the lock is released, but wastes more bus cycles on cache thrashing. A high maximum reduces cache thrashing at the expense of not noticing that the lock is free so quickly. Binary exponential backoff can be used with or without the pure reads preceding the TSL instruction.

An even better idea is to give each CPU wishing to acquire the mutex its own private lock variable to test, as illustrated in Fig. 8-11 (Mellor-Crummey and Scott, 1991). The variable should reside in an otherwise unused cache block to avoid conflicts. The algorithm works by having a CPU that fails to acquire the lock allocate a lock variable and attach itself to the end of a list of CPUs waiting for the lock. When the current lock holder exits the critical region, it frees the private lock that the first CPU on the list is testing (in its own cache). This CPU then enters the critical region. When it is done, it frees the lock its successor is using, and so on. Although the protocol is somewhat complicated (to avoid having two CPUs attach themselves to the end of the list simultaneously), it is efficient and starvation free. For all the details, readers should consult the paper.

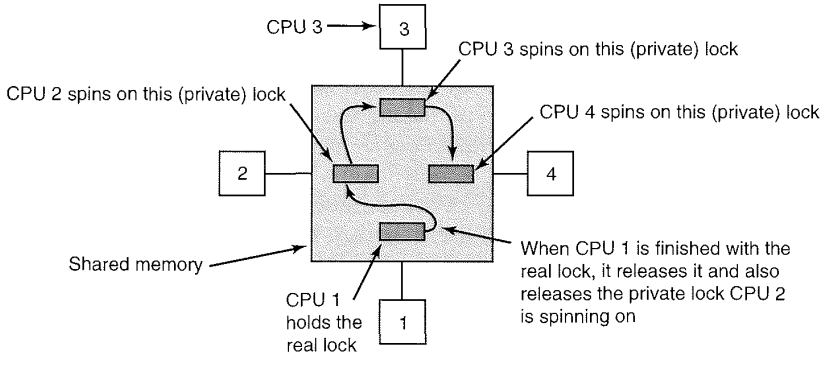


Figure 8-11. Use of multiple locks to avoid cache thrashing.

Spinning versus Switching

So far we have assumed that a CPU needing a locked mutex just waits for it, either by polling continuously, polling intermittently, or attaching itself to a list of waiting CPUs. In some cases, there is no real alternative for the requesting CPU to just waiting. For example, suppose that some CPU is idle and needs to access the shared ready list to pick a process to run. If the ready list is locked, the CPU

IBM Ex-1009
Page 383

cannot just decide to suspend what it is doing and run another process, because doing that would require access to the ready list. It *must* wait until it can acquire the ready list.

However, in other cases, there is a choice. For example, if some thread on a CPU needs to access the file system buffer cache and that is currently locked, the CPU can decide to switch to a different thread instead of waiting. The issue of whether to spin or whether to do a thread switch has been a matter of much research, some of which will be discussed below. Note that this issue does not occur on a uniprocessor because spinning does not make much sense when there is no other CPU to release the lock. If a thread tries to acquire a lock and fails, it is always blocked to give the lock owner a chance to run and release the lock.

Assuming that spinning and doing a thread switch are both feasible options, the trade-off is as follows. Spinning wastes CPU cycles directly. Testing a lock repeatedly is not productive work. Switching, however, also wastes CPU cycles, since the current thread's state must be saved, the lock on the ready list must be acquired, a thread must be selected, its state must be loaded, and it must be started. Furthermore, the CPU cache will contain all the wrong blocks, so many expensive cache misses will occur as the new thread starts running. TLB faults are also likely. Eventually, a switch back to the original thread must take place, with more cache misses following it. The cycles spent doing these two context switches plus all the cache misses are wasted.

If it is known that mutexes are generally held for, say, 50 μsec and it takes 1 msec to switch from the current thread and 1 msec to switch back later, it is more efficient just to spin on the mutex. On the other hand, if the average mutex is held for 10 msec, it is worth the trouble of making the two context switches. The trouble is that critical regions can vary considerably in their duration, so which approach is better?

One design is to always spin. A second design is to always switch. But a third design is to make a separate decision each time a locked mutex is encountered. At the time the decision has to be made, it is not known whether it is better to spin or switch, but for any given system, it is possible to make a trace of all activity and analyze it later offline. Then it can be said in retrospect which decision was the best one and how much time was wasted in the best case. This hindsight algorithm then becomes a benchmark against which feasible algorithms can be measured.

This problem has been studied by researchers (Karlin et al., 1989; Karlin et al., 1991; and Ousterhout, 1982). Most work uses a model in which a thread failing to acquire a mutex spins for some period of time. If this threshold is exceeded, it switches. In some cases the threshold is fixed, typically the known overhead for switching to another thread and then switching back. In other cases it is dynamic, depending on the observed history of the mutex being waited on.

The best results are achieved when the system keeps track of the last few observed spin times and assumes that this one will be similar to the previous ones.

IBM Ex-1009
Page 384

For example, assuming a 1-msec context switch time again, a thread would spin for a maximum of 2 msec, but observe how long it actually spun. If it fails to acquire a lock and sees that on the previous three runs it waited an average of 200 μsec, it should spin for 2 msec before switching. However, it if sees that it spun for the full 2 msec on each of the previous attempts, it should switch immediately and not spin at all. More details can be found in (Karlin et al., 1991).

8.1.4 Multiprocessor Scheduling

On a uniprocessor, scheduling is one dimensional. The only question that must be answered (repeatedly) is: "Which process should be run next?" On a multiprocessor, scheduling is two dimensional. The scheduler has to decide which process to run and which CPU to run it on. This extra dimension greatly complicates scheduling on multiprocessors.

Another complicating factor is that in some systems, all the processes are unrelated whereas in others they come in groups. An example of the former situation is a timesharing system in which independent users start up independent processes. The processes are unrelated and each one can be scheduled without regard to the other ones.

An example of the latter situation occurs regularly in program development environments. Large systems often consist of some number of header files containing macros, type definitions, and variable declarations that are used by the actual code files. When a header file is changed, all the code files that include it must be recompiled. The program *make* is commonly used to manage development. When *make* is invoked, it starts the compilation of only those code files that must be recompiled on account of changes to the header or code files. Object files that are still valid are not regenerated.

The original version of *make* did its work sequentially, but newer versions designed for multiprocessors can start up all the compilations at once. If 10 compilations are needed, it does not make sense to schedule 9 of them quickly and leave the last one until much later since the user will not perceive the work as completed until the last one finishes. In this case it makes sense to regard the processes as a group and to take that into account when scheduling them.

Timesharing

Let us first address the case of scheduling independent processes; later we will consider how to schedule related processes. The simplest scheduling algorithm for dealing with unrelated processes (or threads) is to have a single system-wide data structure for ready processes, possibly just a list, but more likely a set of lists for processes at different priorities as depicted in Fig. 8-12(a). Here the 16 CPUs are all currently busy, and a prioritized set of 14 processes are waiting to run. The first CPU to finish its current work (or have its process block) is CPU 4,

which then locks the scheduling queues and selects the highest priority process, *A*, as shown in Fig. 8-12(b). Next, CPU 12 goes idle and chooses process *B*, as illustrated in Fig. 8-12(c). As long as the processes are completely unrelated, doing scheduling this way is a reasonable choice.

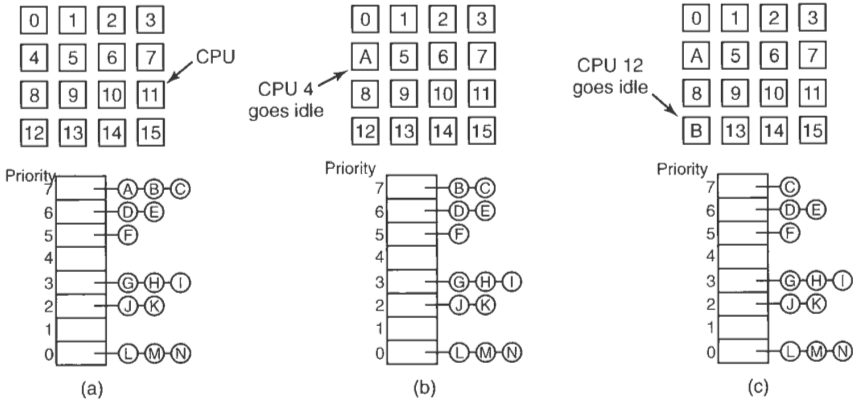


Figure 8-12. Using a single data structure for scheduling a multiprocessor.

Having a single scheduling data structure used by all CPUs timeshares the CPUs, much as they would be in a uniprocessor system. It also provides automatic load balancing because it can never happen that one CPU is idle while others are overloaded. Two disadvantages of this approach are the potential contention for the scheduling data structure as the numbers of CPUs grows and the usual overhead in doing a context switch when a process blocks for I/O.

It is also possible that a context switch happens when a process' quantum expires. On a multiprocessor, that has certain properties not present on a uniprocessor. Suppose that the process holds a spin lock, not unusual on multiprocessors, as discussed above. Other CPUs waiting on the spin lock just waste their time spinning until that process is scheduled again and releases the lock. On a uniprocessor, spin locks are rarely used so if a process is suspended while it holds a mutex, and another process starts and tries to acquire the mutex, it will be immediately blocked, so little time is wasted.

To get around this anomaly, some systems use **smart scheduling**, in which a process acquiring a spin lock sets a process-wide flag to show that it currently has a spin lock (Zahorjan et al., 1991). When it releases the lock, it clears the flag. The scheduler then does not stop a process holding a spin lock, but instead gives it a little more time to complete its critical region and release the lock.

Another issue that plays a role in scheduling is the fact that while all CPUs are equal, some CPUs are more equal. In particular, when process *A* has run for a long time on CPU *k*, CPU *k*'s cache will be full of *A*'s blocks. If *A* gets to run

IBM Ex-1009
Page 386

again soon, it may perform better if it is run on CPU k, because k's cache may still contain some of A's blocks. Having cache blocks preloaded will increase the cache hit rate and thus the process' speed. In addition, the TLB may also contain the right pages, reducing TLB faults.

Some multiprocessors take this effect into account and use what is called **affinity scheduling** (Vaswani and Zahorjan, 1991). The basic idea here is to make a serious effort to have a process run on the same CPU it ran on last time. One way to create this affinity is to use a **two-level scheduling algorithm**. When a process is created, it is assigned to a CPU, for example based on which one has the smallest load at that moment. This assignment of processes to CPUs is the top level of the algorithm. As a result, each CPU acquires its own collection of processes.

The actual scheduling of the processes is the bottom level of the algorithm. It is done by each CPU separately, using priorities or some other means. By trying to keep a process on the same CPU, cache affinity is maximized. However, if a CPU has no processes to run, it takes one from another CPU rather than go idle.

Two-level scheduling has three benefits. First, it distributes the load roughly evenly over the available CPUs. Second, advantage is taken of cache affinity where possible. Third, by giving each CPU its own ready list, contention for the ready lists is minimized because attempts to use another CPU's ready list are relatively infrequent.

Space Sharing

The other general approach to multiprocessor scheduling can be used when processes are related to one another in some way. Earlier we mentioned the example of parallel *make* as one case. It also often occurs that a single process creates multiple threads that work together. For our purposes, a job consisting of multiple related processes or a process consisting of multiple kernel threads are essentially the same thing. We will refer to the schedulable entities as threads here, but the material holds for processes as well. Scheduling multiple threads at the same time across multiple CPUs is called **space sharing**.

The simplest space sharing algorithm works like this. Assume that an entire group of related threads is created at once. At the time it is created, the scheduler checks to see if there are as many free CPUs as there are threads. If there are, each thread is given its own dedicated (i.e., nonmultiprogrammed) CPU and they all start. If there are not enough CPUs, none of the threads are started until enough CPUs are available. Each thread holds onto its CPU until it terminates, at which time the CPU is put back into the pool of available CPUs. If a thread blocks on I/O, it continues to hold the CPU, which is simply idle until the thread wakes up. When the next batch of threads appears, the same algorithm is applied.

At any instant of time, the set of CPUs is statically partitioned into some number of partitions, each one running the threads of one process. In Fig. 8-13,

IBM Ex-1009
Page 387

we have partitions of sizes 4, 6, 8, and 12 CPUs, with 2 CPUs unassigned, for example. As time goes on, the number and size of the partitions will change as processes come and go.

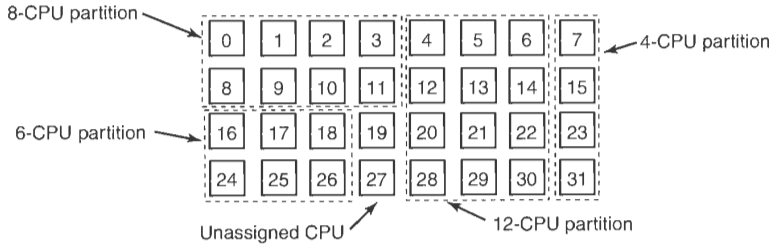


Figure 8-13. A set of 32 CPUs split into four partitions, with two CPUs available.

Periodically, scheduling decisions have to be made. In uniprocessor systems, shortest job first is a well-known algorithm for batch scheduling. The analogous algorithm for a multiprocessor is to choose the process needing the smallest number of CPU cycles, that is the process whose CPU-count × run-time is the smallest of the candidates. However, in practice, this information is rarely available, so the algorithm is hard to carry out. In fact, studies have shown that, in practice, beating first-come, first-served is hard to do (Krueger et al., 1994).

In this simple partitioning model, a process just asks for some number of CPUs and either gets them all or has to wait until they are available. A different approach is for processes to actively manage the degree of parallelism. One way to do manage the parallelism is to have a central server that keeps track of which processes are running and want to run and what their minimum and maximum CPU requirements are (Tucker and Gupta, 1989). Periodically, each CPU polls the central server to ask how many CPUs it may use. It then adjusts the number of processes or threads up or down to match what is available. For example, a Web server can have 1, 2, 5, 10, 20, or any other number of threads running in parallel. If it currently has 10 threads and there is suddenly more demand for CPUs and it is told to drop to 5, when the next 5 threads finish their current work, they are told to exit instead of being given new work. This scheme allows the partition sizes to vary dynamically to match the current workload better than the fixed system of Fig. 8-13.

Gang Scheduling

A clear advantage of space sharing is the elimination of multiprogramming, which eliminates the context switching overhead. However, an equally clear disadvantage is the time wasted when a CPU blocks and has nothing at all to do until it becomes ready again. Consequently, people have looked for algorithms

IBM Ex-1009
Page 388

that attempt to schedule in both time and space together, especially for processes that create multiple threads, which usually need to communicate with one another.

To see the kind of problem that can occur when the threads of a process (or processes of a job) are independently scheduled, consider a system with threads A_0 and A_1 belonging to process A and threads B_0 and B_1 belonging to process B. threads A_0 and B_0 are timeshared on CPU 0; threads A_1 and B_1 are timeshared on CPU 1. threads A_0 and A_1 need to communicate often. The communication pattern is that A_0 sends A_1 a message, with A_1 then sending back a reply to A_0, followed by another such sequence. Suppose that luck has it that A_0 and B_1 start first, as shown in Fig. 8-14.

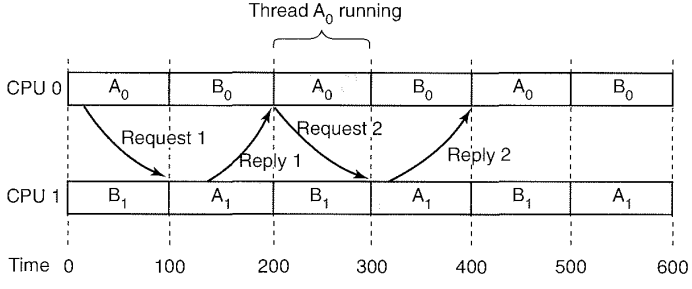


Figure 8-14. Communication between two threads belonging to process A that are running out of phase.

In time slice 0, A_0 sends A_1 a request, but A_1 does not get it until it runs in time slice 1 starting at 100 msec. It sends the reply immediately, but A_0 does not get the reply until it runs again at 200 msec. The net result is one request-reply sequence every 200 msec. Not very good.

The solution to this problem is **gang scheduling**, which is an outgrowth of **co-scheduling** (Ousterhout, 1982). Gang scheduling has three parts:

1. Groups of related threads are scheduled as a unit, a gang.
2. All members of a gang run simultaneously, on different timeshared CPUs.
3. All gang members start and end their time slices together.

The trick that makes gang scheduling work is that all CPUs are scheduled synchronously. This means that time is divided into discrete quanta as we had in Fig. 8-14. At the start of each new quantum, *all* the CPUs are rescheduled, with a new thread being started on each one. At the start of the following quantum, another scheduling event happens. In between, no scheduling is done. If a thread blocks, its CPU stays idle until the end of the quantum.

An example of how gang scheduling works is given in Fig. 8-15. Here we have a multiprocessor with six CPUs being used by five processes, A through E,

IBM Ex-1009
Page 389

with a total of 24 ready threads. During time slot 0, threads A_0 through A_6 are scheduled and run. During time slot 1, Threads B_0, B_1, B_2, C_0, C_1, and C_2 are scheduled and run. During time slot 2, D's five threads and E_0 get to run. The remaining six threads belonging to process E run in time slot 3. Then the cycle repeats, with slot 4 being the same as slot 0 and so on.

Time slot	CPU 0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Figure 8-15. Gang scheduling.

The idea of gang scheduling is to have all the threads of a process run together, so that if one of them sends a request to another one, it will get the message almost immediately and be able to reply almost immediately. In Fig. 8-15, since all the A threads are running together, during one quantum, they may send and receive a very large number of messages in one quantum, thus eliminating the problem of Fig. 8-14.

8.2 MULTICOMPUTERS

Multiprocessors are popular and attractive because they offer a simple communication model: all CPUs share a common memory. Processes can write messages to memory that can then be read by other processes. Synchronization can be done using mutexes, semaphores, monitors, and other well-established techniques. The only fly in the ointment is that large multiprocessors are difficult to build and thus expensive.

To get around these problems, much research has been done on **multicomputers**, which are tightly-coupled CPUs that do not share memory. Each one has its own memory, as shown in Fig. 8-1(b). These systems are also known by a variety of other names, including **cluster computers**, and **COWS** (**Clusters of Workstations**).

Multicomputers are easy to build because the basic component is just a stripped-down PC with the addition of a network interface card. Of course, the secret to getting high performance is to design the interconnection network and

the interface card cleverly. This problem is completely analogous to building the shared memory in a multiprocessor. However, the goal is to send messages on a microsecond time scale, rather than access memory on a nanosecond time scale, so it is simpler, cheaper, and easier to accomplish.

In the following sections, we will first take a brief look at multicomputer hardware, especially the interconnection hardware. Then we will move onto the software, starting with low-level communication software, then high-level communication software. We will also look at a way shared memory can be achieved on systems that do not have it. Finally, we will examine scheduling and load balancing.

8.2.1 Multicomputer Hardware

The basic node of a multicomputer consists of a CPU, memory, a network interface, and sometimes a hard disk. The node may be packaged in a standard PC case, but the graphics adapter, monitor, keyboard, and mouse are nearly always absent. In some cases, the PC contains a 2-way or 4-way multiprocessor board instead of a single CPU, but for simplicity, we will assume that each node has one CPU. Often hundreds or even thousands of nodes are hooked together to form a multicomputer. Below we will say a little about how this hardware is organized.

Interconnection Technology

Each node has a network interface card with one or two cables (or fibers) coming out of it. These cables connect to either other nodes or to switches. In a small system, there may be one switch to which all the nodes are connected in the star topology of Fig. 8-16(a). Modern switched Ethernets use this topology.

As an alternative to the single switch design, the nodes may form a ring, with two wires coming out the network interface card, one going into the node on the left and one going into the node on the right, as shown in Fig. 8-16(b). In this topology, no switches are needed and none are shown.

The **grid** or **mesh** of Fig. 8-16(c) is a two-dimensional design that has been used in many commercial systems. It is highly regular and easy to scale up to large sizes. It has a **diameter**, which is the longest path between any two nodes, and which increases only as the square root of the number of nodes. A variant on the grid is the **double torus** of Fig. 8-16(d), which is a grid with the edges connected. Not only is it more fault tolerant than the grid, but the diameter is also less because the opposite corners can now communicate in only two hops.

The **cube** of Fig. 8-16(e) is a regular three-dimensional topology. We have illustrated a $2 \times 2 \times 2$ cube, but in the general case it could be a $k \times k \times k$ cube. In Fig. 8-16(f) we have a four-dimensional cube constructed from two three-dimensional cubes with the corresponding nodes connected. We could make a

IBM Ex-1009
Page 391

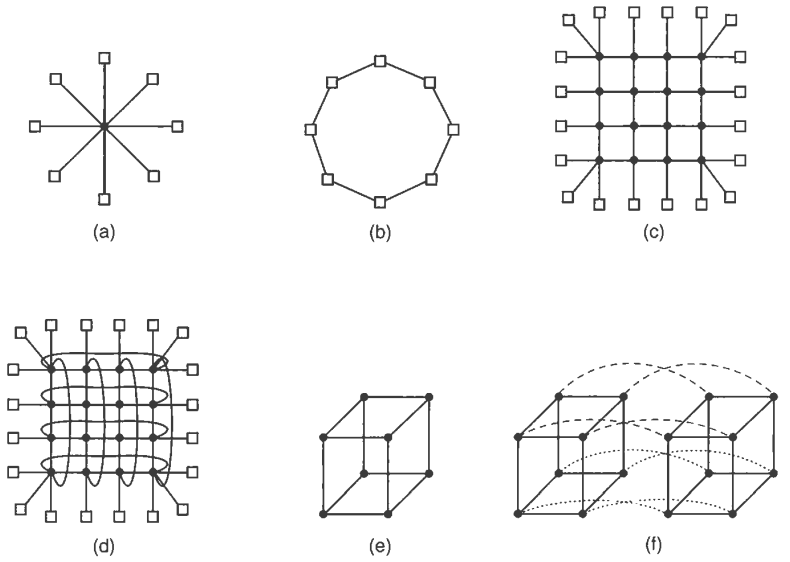


Figure 8-16. Various interconnect topologies. (a) A single switch. (b) A ring. (c) A grid. (d) A double torus. (e) A cube. (f) A 4D hypercube.

five-dimensional cube by cloning the structure of Fig. 8-16(f) and connecting the corresponding nodes to form a block of four cubes. To go to six dimensions, we could replicate the block of four cubes and interconnect the corresponding nodes, and so on. An n-dimensional cube formed this way is called a **hypercube**. Many parallel computers use this topology because the diameter grows linearly with the dimensionality. Put in other words, the diameter is the base 2 logarithm of the number of nodes, so, for example, a 10-dimensional hypercube has 1024 nodes but a diameter of only 10, giving excellent delay properties. Note that in contrast, 1024 nodes arranged as a 32×32 grid has a diameter of 62, more than six times worse than the hypercube. The price paid for the smaller diameter is that the fanout and thus the number of links (and the cost) is much larger for the hypercube.

Two kinds of switching schemes are used in multicomputers. In the first one, each message is first broken up (either by the user software or the network interface) into a chunk of some maximum length called a **packet**. The switching scheme, called **store-and-forward packet switching**, consists of the packet being injected into the first switch by the source node's network interface board, as shown in Fig. 8-17(a). The bits come in one at a time, and when the whole packet has arrived, it is copied to the next switch along the path, as shown in Fig. 8-17(b). When the packet arrives at the switch attached to the destination node, as

IBM Ex-1009
Page 392

shown in Fig. 8-17(c), the packet is copied to that node's network interface board and eventually to its RAM.

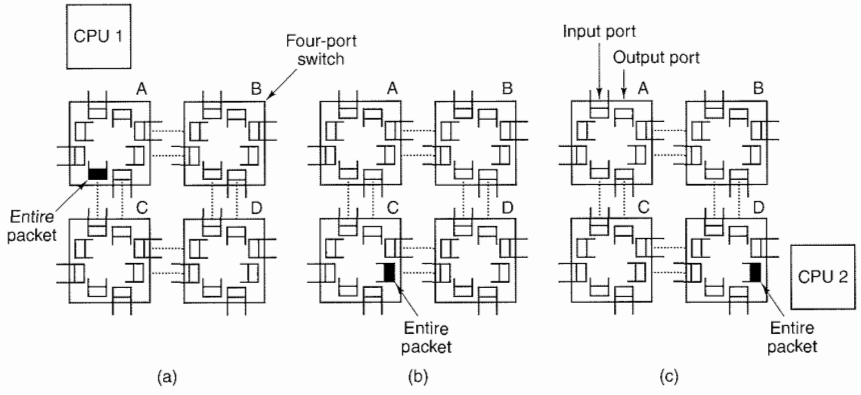


Figure 8-17. Store-and-forward packet switching.

While store-and-forward packet switching is flexible and efficient, it does have the problem of increasing latency (delay) through the interconnection network. Suppose that the time to move a packet one hop in Fig. 8-17 is T nsec. Since the packet must be copied four times to get it from CPU 1 to CPU 2 (to A, to C, to D, and to the destination CPU), and no copy can begin until the previous one is finished, the latency through the interconnection network is $4T$. One way out is to design a hybrid network, with some of the properties of circuit switching and some of the properties of packet switching. For example, each packet can be logically divided into smaller units. As soon as the first unit arrives at a switch, it can be moved to the next switch, even before the tail of the packet has arrived.

The other switching regime, **circuit switching**, consists of the first switch first establishing a path through all the switches to the destination switch. Once that path has been set up, the bits are pumped all the way from the source to the destination nonstop. There is no intermediate buffering at the intervening switches. Circuit switching requires a setup phase, which takes some time, but is faster once the setup has been completed. After the packet has been sent, the path must be torn down again. A variation on circuit switching, called **wormhole routing**, breaks each packet up into subpackets and allows the first subpacket to start flowing even before the full path has been built.

Network Interfaces

All the nodes in a multicomputer have a plug-in board containing the node's connection to the interconnection network that holds the multicomputer together. The way these boards are built and how they connect to the main CPU and RAM

IBM Ex-1009
Page 393

have substantial implications for the operating system. We will now briefly look at some of the issues here. This material is based in part on (Bhoedjang, 2000). Other references are (Buzzard et al., 1996; Pakin et al., 1997; Steenkiste, 1994; and Von Eicken et al., 1992).

In virtually all multicomputers, the interface board contains some RAM for holding outgoing and incoming packets. Usually, an outgoing packet has to be copied to the interface board's RAM before it can be transmitted to the first switch. The reason for this design is that many interconnection networks are synchronous, so that once a packet transmission has started, the bits must continue flowing at a constant rate. If the packet is in the main RAM, this continuous flow out onto the network cannot be guaranteed due to other traffic on the memory bus. Using a dedicated RAM on the interface board eliminates this problem. This design is shown in Fig. 8-18.

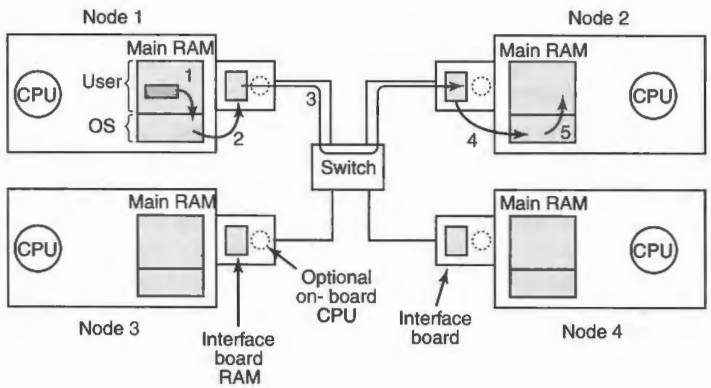


Figure 8-18. Position of the network interface boards in a multicomputer.

The same problem occurs with incoming packets. The bits arrive from the network at a constant and often extremely high rate. If the network interface board cannot store them in real time as they arrive, data will be lost. Again here, trying to go over the system bus (e.g., the PCI bus) to the main RAM is too risky. Since the network board is typically plugged into the PCI bus, this is the only connection it has to the main RAM, so competing for this bus with the disk and every other I/O device is inevitable. It is safer to store incoming packets in the interface board's private RAM and then copy them to the main RAM later.

The interface board may have one or more DMA channels or even a complete CPU on board. The DMA channels can copy packets between the interface board and the main RAM at high speed by requesting block transfers on the system bus, thus transferring several words without having to request the bus separately for each word. However, it is precisely this kind of block transfer, which ties up the

IBM Ex-1009
Page 394

system bus for multiple bus cycles, that makes the interface board RAM necessary in the first place.

Some interface boards have a full CPU on them, possibly in addition to one or more DMA channels. This design means that the main CPU can offload some work to the network board such as handling reliable transmission (if the underlying hardware can lose packets), multicasting (sending a packet to more than one destination), and taking care of protection in a system with multiple processes. However, having two CPUs means that they must synchronize to avoid race conditions, which adds extra overhead and means more work for the operating system.

8.2.2 Low-Level Communication Software

The enemy of high-performance communication in multicomputer systems is excess copying of packets. In the best case, there will be one copy from RAM to the interface board at the source node, one copy from the source interface board to the destination interface board (if no storing and forwarding along the path occurs), and one copy from there to the destination RAM, a total of three copies. However, in many systems it is even worse. In particular, if the interface board is mapped into kernel virtual address space and not user virtual address space, a user process can only send a packet by issuing a system call that traps to the kernel. The kernels may have to copy the packets to their own memory both on output and on input, for example, to avoid page faults while transmitting over the network. Also, the receiving kernel probably does not know where to put incoming packets until it has had a chance to examine them. These five copy steps are illustrated in Fig. 8-18.

If copies to and from RAM dominate the performance, the extra copies to and from the kernel may double the end-to-end delay and cut the bandwidth in half. To avoid this performance hit, many multicomputers map the interface board directly into user space and allow the user process to put the packets on the board directly, without the kernel being involved. While this approach definitely helps performance, it introduces two problems.

First, what if several processes are running on the node and need network access to send packets? Which one gets the interface board in its address space? Having a system call to map the board in and out of a virtual address space is expensive, but if only one process gets the board, how do the other ones send packets? And what happens if the board is mapped into process *A*'s virtual address space and a packet arrives for process *B*, especially if *A* and *B* have different owners, neither of whom wants to put in any effort to help the other?

One solution is to map the interface board into all processes that need it, but then a mechanism is needed to avoid race conditions. For example if *A* claims a buffer on the interface board and then due to a time slice, *B* runs and claims the same buffer, disaster results. Some kind of synchronization mechanism is needed,

IBM Ex-1009
Page 395

but these mechanisms, such as mutexes, only work when the processes are assumed to be cooperating. In a timesharing environment with multiple users all in a hurry to get their work done, one user might just lock the mutex associated with the board and never release it. The conclusion here is that mapping the interface board into user space only really works well when there is just one user process running on each node unless special precautions are taken (for example, different processes get different portions of the interface RAM mapped into their address spaces).

The second problem is that the kernel may well need access to the interconnection network itself, for example, to access the file system on a remote node. Having the kernel share the interface board with any users is not a good idea, even on a timesharing basis. Suppose that while the board was mapped into user space, a kernel packet arrived? Or suppose that the user process sent a packet to a remote machine pretending to be the kernel? The conclusion is that the simplest design is to have two network interface boards, one mapped into user space for application traffic and one mapped into kernel space for use by the operating system. Many multicomputers do precisely this.

Node to Network Interface Communication

Another issue is how to get packets onto the interface board. The fastest way is to use the DMA chip on the board to just copy them in from RAM. The problem with this approach is that DMA uses physical rather than virtual addresses and runs independently of the CPU. To start with, although a user process certainly knows the virtual address of any packet it wants to send, it generally does not know the physical address. Making a system call to do the virtual-to-physical mapping is undesirable, since the point of putting the interface board in user space in the first place was to avoid having to make a system call for each packet to be sent.

In addition, if the operating system decides to replace a page while the DMA chip is copying a packet from it, the wrong data will be transmitted. Worse yet, if the operating system replaces a page while the DMA chip is copying an incoming packet to it, not only will the incoming packet be lost, but also a page of innocent memory will be ruined.

These problems can be avoided by having system calls to pin and unpin pages in memory, marking them as temporarily unpageable. However, having to make a system call to pin the page containing each outgoing packet and then having to make another call later to unpin it is expensive. If packets are small, say, 64 bytes or less, the overhead for pinning and unpinning every buffer is prohibitive. For large packets, say, 1 KB or more, it may be tolerable. For sizes in between, it depends on the details of the hardware (Bhoedjang, 2000).

In theory, the same problem occurs with DMA from a disk or other device, but since these are set up by the operating system to kernel buffers, it is easy for

the system to avoid paging the buffers. The problem here is that the user is setting up and managing the DMA, and the operating system does not know that removing a page could be fatal, something it does know for I/O that it starts itself. The reason using kernel buffers is acceptable for disk I/O and not for multiprocessor communication is that an extra 20-μsec delay is tolerable for disk latency but not for process-to-process communication latency.

The DMA problem can be avoided by having the user process first pin one page at startup and asking for its physical address. Outgoing packets are first copied there and then to the network interface, but this extra copy is just as bad as copying to the kernel.

For these reasons, using programmed I/O to and from the interface board is usually the safest course, since any page faults encountered are just ordinary CPU page faults and can be handled in the usual way by the operating system. When a page fault occurs, the copy loop stops instantly and remains stopped until the operating system has handled the page fault. A more sophisticated scheme is to use programmed I/O for small packets and DMA with pinning and unpinning for large ones.

If the network interface boards have their own CPUs (e.g., as do Myrinet boards), these on-board CPUs can be used to speed up communication. However, care has to be taken to avoid race conditions between the main CPU and the on-board CPU. One way to avoid races is illustrated in Fig. 8-19, where we focus on node 1 sending packets and node 2 receiving them, not necessarily from each other. The key synchronization data structure for senders is the send ring; for receivers it is the receive ring. All nodes have both since they all send and receive. Each ring has room for n packets. There is also a bitmap per ring with n bits, possibly separate (as shown) or possibly integrated into the rings, telling which ring slots are currently valid.

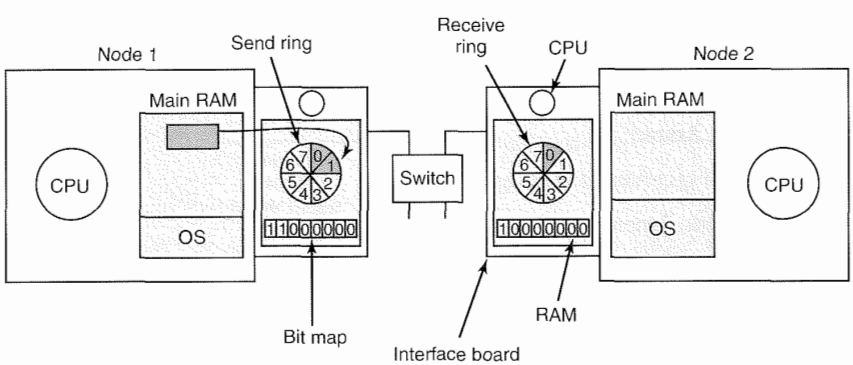


Figure 8-19. Use of send and receive rings to coordinate the main CPU with the on-board CPU.

IBM Ex-1009
Page 397

When a sender has a new packet to send, it first checks to see if there is an available slot in the send ring. If not, it must wait, to prevent overrun. If there is a slot, it copies the packet to the next available slot, and after that is completed, sets the corresponding bit in the bitmap. When the on-board CPU has finished whatever it is doing, it checks the send ring. If it contains any packets, it takes the one there longest and transmits it. When it is done, it clears the corresponding bit in the bitmap. Since the main CPU is the only one that sets the bits and the on-board CPU is the only one that clears them, there are no race conditions. The receive ring works the other way, with the on-board CPU setting a bit to indicate packet arrival and the main CPU clearing it to indicate that it has copied the packet and freed the buffer.

This scheme can also be used even without programmed I/O done by the main CPU. In that case, the send ring entry does not contain the packet itself, but a pointer to the packet in the main RAM. When the on-board CPU is ready to transmit the packet, it fetches the packet to the interface board, either using programmed I/O itself or via DMA. In both cases, this approach works only if the page containing the packet is known to be pinned.

8.2.3 User-Level Communication Software

Processes on different CPUs on a multicomputer communicate by sending messages to one another. In the simplest form, this message passing is exposed to the user processes. In other words, the operating system provides a way to send and receive messages, and library procedures make these underlying calls available to user processes. In a more sophisticated form, the actual message passing is hidden from users by making remote communication look like a procedure call. We will study both of these methods below.

Send and Receive

At the barest minimum, the communication services provided can be reduced to two (library) calls, one for sending messages and one for receiving them. The calling for sending a message might be

```
send(dest, &mptr);
```

and the call for receiving a message might be

```
receive(addr, &mptr);
```

The former sends the message pointed to by *mptr* to a process identified by *dest* and causes the caller to be blocked until the message has been sent. The latter causes the caller to be blocked until a message arrives. When one does, the message is copied to the buffer pointed to by *mptr* and the caller is unblocked. The *addr* parameter specifies the address to which the receiver is listening. Many variants of these two procedures and their parameters are possible.

One issue is how addressing is done. Since multicomputers are static, with the number of CPUs fixed, the easiest way to handle addressing is to make *addr* a

two-part address consisting of a CPU number and a process or port number on the addressed CPU. In this way each CPU can manage its own addresses without potential conflicts.

Blocking versus Nonblocking Calls

The calls described above are **blocking calls** (sometimes called **synchronous calls**). When a process calls *send*, it specifies a destination and a buffer to send to that destination. While the message is being sent, the sending process is blocked (i.e., suspended). The instruction following the call to *send* is not executed until the message has been completely sent, as shown in Fig. 8-20(a). Similarly, a call to *receive* does not return control until a message has actually been received and put in the message buffer pointed to by the parameter. The process remains suspended in *receive* until a message arrives, even if it takes hours. In some systems, the receiver can specify from whom it wishes to receive, in which case it remains blocked until a message from that sender arrives.

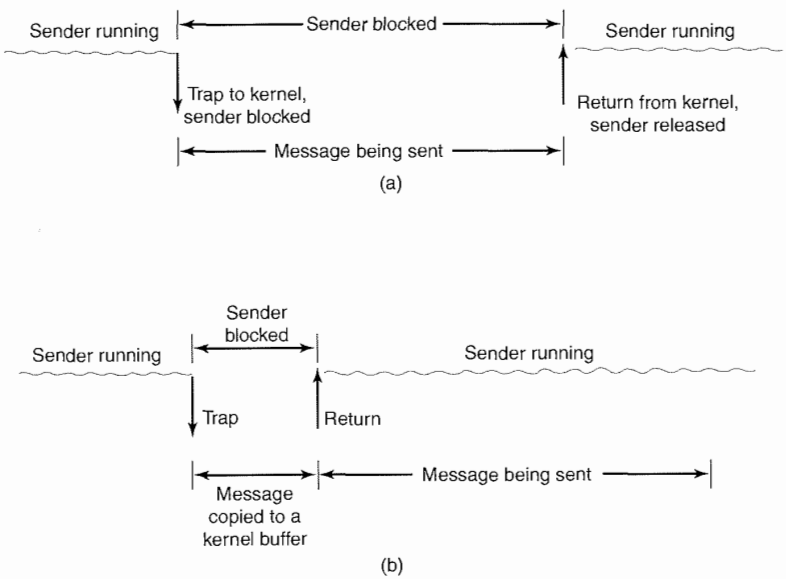


Figure 8-20. (a) A blocking send call. (b) A nonblocking send call.

An alternative to blocking calls are **nonblocking calls** (sometimes called **asynchronous calls**). If *send* is nonblocking, it returns control to the caller immediately, before the message is sent. The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle (assuming no other process is runnable).

IBM Ex-1009
Page 399

The choice between blocking and nonblocking primitives is normally made by the system designers (i.e., either one primitive is available or the other), although in a few systems both are available and users can choose their favorite.

However, the performance advantage offered by nonblocking primitives is offset by a serious disadvantage: the sender cannot modify the message buffer until the message has been sent. The consequences of the process overwriting the message during transmission are too horrible to contemplate. Worse yet, the sending process has no idea of when the transmission is done, so it never knows when it is safe to reuse the buffer. It can hardly avoid touching it forever.

There are three possible ways out. The first solution is to have the kernel copy the message to an internal kernel buffer and then allow the process to continue, as shown in Fig. 8-20(b). From the sender's point of view, this scheme is the same as a blocking call: as soon as it gets control back, it is free to reuse the buffer. Of course, the message will not yet have been sent, but the sender is not hindered by this fact. The disadvantage of this method is that every outgoing message has to be copied from user space to kernel space. With many network interfaces, the message will have to be copied to a hardware transmission buffer later anyway, so the first copy is essentially wasted. The extra copy can reduce the performance of the system considerably.

The second solution is to interrupt the sender when the message has been sent to inform it that the buffer is once again available. No copy is required here, which saves time, but user-level interrupts make programming tricky, difficult, and subject to race conditions, which makes them irreproducible and nearly impossible to debug.

The third solution is to make the buffer copy on write, that is, to mark it as read-only until the message has been sent. If the buffer is reused before the message has been sent, a copy is made. The problem with this solution is that unless the buffer is isolated on its own page, writes to nearby variables will also force a copy. Also, extra administration is needed because the act of sending a message now implicitly affects the read/write status of the page. Finally, sooner or later the page is likely to be written again, triggering a copy that may no longer be necessary.

Thus the choices on the sending side are

1. Blocking send (CPU idle during message transmission).
2. Nonblocking send with copy (CPU time wasted for the extra copy).
3. Nonblocking send with interrupt (makes programming difficult).
4. Copy on write (extra copy probably needed eventually).

Under normal conditions, the first choice is the best one, especially if multiple threads are available, in which case while one thread is blocked trying to send, other threads can continue working. It also does not require any kernel buffers to

IBM Ex-1009
Page 400

be managed. Furthermore, as can be seen from comparing Fig. 8-20(a) to Fig. 8-20(b), the message will usually be out the door faster if no copy is required.

For the record, we would like to point out that some authors use a different criterion to distinguish synchronous from asynchronous primitives. In the alternative view, a call is synchronous only if the sender is blocked until the message has been received and an acknowledgement sent back (Andrews, 1991). In the world of real-time communication, synchronous has yet another meaning, which can lead to confusion, unfortunately.

Just as *send* can be blocking or nonblocking, so can *receive*. A blocking call just suspends the caller until a message has arrived. If multiple threads are available, this is a simple approach. Alternatively, a nonblocking *receive* just tells the kernel where the buffer is and returns control almost immediately. An interrupt can be used to signal that a message has arrived. However, interrupts are difficult to program and are also quite slow, so it may be preferable for the receiver to poll for incoming messages using a procedure, *poll*, that tells whether any messages are waiting. If so, the caller can call *get_message*, which returns the first arrived message. In some systems, the compiler can insert poll calls in the code at appropriate places, although knowing how often to poll is tricky.

Yet another option is a scheme in which the arrival of a message causes a new thread to be created spontaneously in the receiving process' address space. Such a thread is called a **pop-up thread**. It runs a procedure specified in advance and whose parameter is a pointer to the incoming message. After processing the message, it simply exits and is automatically destroyed.

A variant on this idea is to run the receiver code directly in the interrupt handler, without going to the trouble of creating a pop-up thread. To make this scheme even faster, the message itself contains the address of the handler, so when a message arrives, the handler can be called in a few instructions. The big win here is that no copying at all is needed. The handler takes the message from the interface board and processes it on the fly. This scheme is called **active messages** (Von Eicken et al., 1992). Since each message contains the address of the handler, active messages only work when senders and receivers trust each other completely.

8.2.4 Remote Procedure Call

Although the message-passing model provides a convenient way to structure a multicomputer operating system, it suffers from one incurable flaw: the basic paradigm around which all communication is built is input/output. The procedures *send* and *receive* are fundamentally engaged in doing I/O and many people believe that I/O is the wrong programming model.

This problem has long been known, but little was done about it until a paper by Birrell and Nelson (1984) introduced a completely different way of attacking the problem. Although the idea is refreshingly simple (once someone has thought

IBM Ex-1009
Page 401

of it), the implications are often subtle. In this section we will examine the concept, its implementation, its strengths, and its weaknesses.

In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on other CPUs. When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended, and execution of the called procedure takes place on 2. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing or I/O at all is visible to the programmer. This technique is known as **RPC (Remote Procedure Call)** and has become the basis of a large amount of multicomputer software. Traditionally the calling procedure is known as the client and the called procedure is known as the server, and we will use those names here too.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure called the **client stub** that represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the **server stub**. These procedures hide the fact that the procedure call from the client to the server is not local.

The actual steps in making a RPC are shown in Fig. 8-21. Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way. Step 2 is the client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called **marshaling**. Step 3 is the kernel sending the message from the client machine to the server machine. Step 4 is the kernel passing the incoming packet to the server stub (which would normally have called *receive* earlier). Finally, step 5 is the server stub calling the server procedure. The reply traces the same path in the other direction.

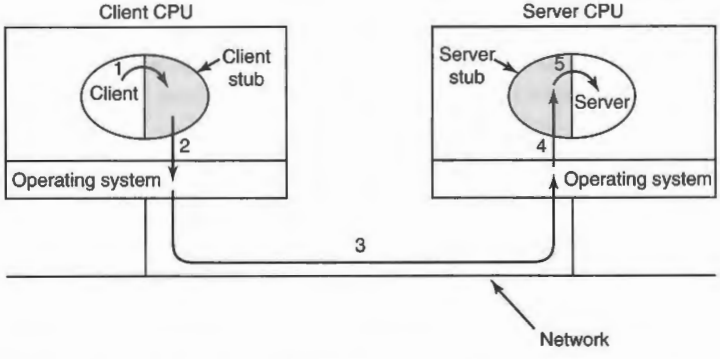


Figure 8-21. Steps in making a remote procedure call. The stubs are shaded gray.

The key item to note here is that the client procedure, written by the user, just makes a normal (i.e., local) procedure call to the client stub, which has the same

name as the server procedure. Since the client procedure and client stub are in the same address space, the parameters are passed in the usual way. Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual. In this way, instead of doing I/O using *send* and *receive*, remote communication is done by faking a normal procedure call.

Implementation Issues

Despite the conceptual elegance of RPC, there are a few snakes hiding under the grass. A big one is the use of pointer parameters. Normally, passing a pointer to a procedure is not a problem. The called procedure can use the pointer the same way the caller can because the two procedures reside in the same virtual address space. With RPC passing pointers is impossible because the client and server are in different address spaces.

In some cases, tricks can be used to make it possible to pass pointers. Suppose that the first parameter is a pointer to an integer, k. The client stub can marshal k and send it along to the server. The server stub then creates a pointer to k and passes it to the server procedure, just as it expects. When the server procedure returns control to the server stub, the latter sends k back to the client where the new k is copied over the old one, just in case the server changed it. In effect, the standard calling sequence of call-by-reference has been replaced by copy-restore. Unfortunately, this trick does not always work, for example, if the pointer points to a graph or other complex data structure. For this reason, some restrictions must be placed on parameters to procedures called remotely.

A second problem is that in weakly-typed languages, like C, it is perfectly legal to write a procedure that computes the inner product of two vectors (arrays), without specifying how large either one is. Each could be terminated by a special value known only to the calling and called procedure. Under these circumstances, it is essentially impossible for the client stub to marshal the parameters: it has no way of determining how large they are.

A third problem is that it is not always possible to deduce the types of the parameters, not even from a formal specification or the code itself. An example is *printf*, which may have any number of parameters (at least one), and they can be an arbitrary mixture of integers, shorts, longs, characters, strings, floating-point numbers of various lengths, and other types. Trying to call *printf* as a remote procedure would be practically impossible because C is so permissive. However, a rule saying that RPC can be used provided that you do not program in C (or C++) would not be popular.

A fourth problem relates to the use of global variables. Normally, the calling and called procedure may communicate using global variables, in addition to via parameters. If the called procedure is now moved to a remote machine, the code will fail because the global variables are no longer shared.

IBM Ex-1009
Page 403

These problems are not meant to suggest that RPC is hopeless. In fact, it is widely used, but some restrictions and care are needed to make it work well in practice.

8.2.5 Distributed Shared Memory

Although RPC has its attractions, many programmers still prefer a model of shared memory and would like to use it, even on a multicomputer. Surprisingly enough, it is possible to preserve the illusion of shared memory reasonably well, even when it does not actually exist, using a technique called **DSM (Distributed Shared Memory)** (Li, 1986; and Li and Hudak, 1989). With DSM, each page is located in one of the memories of Fig. 8-1. Each machine has its own virtual memory and its own page tables. When a CPU does a LOAD or STORE on a page it does not have, a trap to the operating system occurs. The operating system then locates the page and asks the CPU currently holding it to unmap the page and send it over the interconnection network. When it arrives, the page is mapped in and the faulting instruction is restarted. In effect, the operating system is just satisfying page faults from remote RAM instead of from local disk. To the user, the machine looks as if it has shared memory.

The difference between actual shared memory and DSM is illustrated in Fig. 8-22. In Fig. 8-22(a), we see a true multiprocessor with physical shared memory implemented by the hardware. In Fig. 8-22(b), we see DSM, implemented by the operating system. In Fig. 8-22(c), we see yet another form of shared memory, implemented by yet higher levels of software. We will come back to this third option later in the chapter, but for now we will concentrate on DSM.

Let us now look in some detail at how DSM works. In a DSM system, the address space is divided up into pages, with the pages being spread over all the nodes in the system. When a CPU references an address that is not local, a trap occurs, and the DSM software fetches the page containing the address and restarts the faulting instruction, which now completes successfully. This concept is illustrated in Fig. 8-23(a) for an address space with 16 pages and four nodes, each capable of holding four pages.

In this example, if CPU 0 references instructions or data in pages 0, 2, 5, or 9, the references are done locally. References to other pages cause traps. For example, a reference to an address in page 10 will cause a trap to the DSM software, which then moves page 10 from node 1 to node 0, as shown in Fig. 8-23(b).

Replication

One improvement to the basic system that can improve performance considerably is to replicate pages that are read only, for example, program text, read-only constants, or other read-only data structures. For example, if page 10 in Fig. 8-23

IBM Ex-1009
Page 404

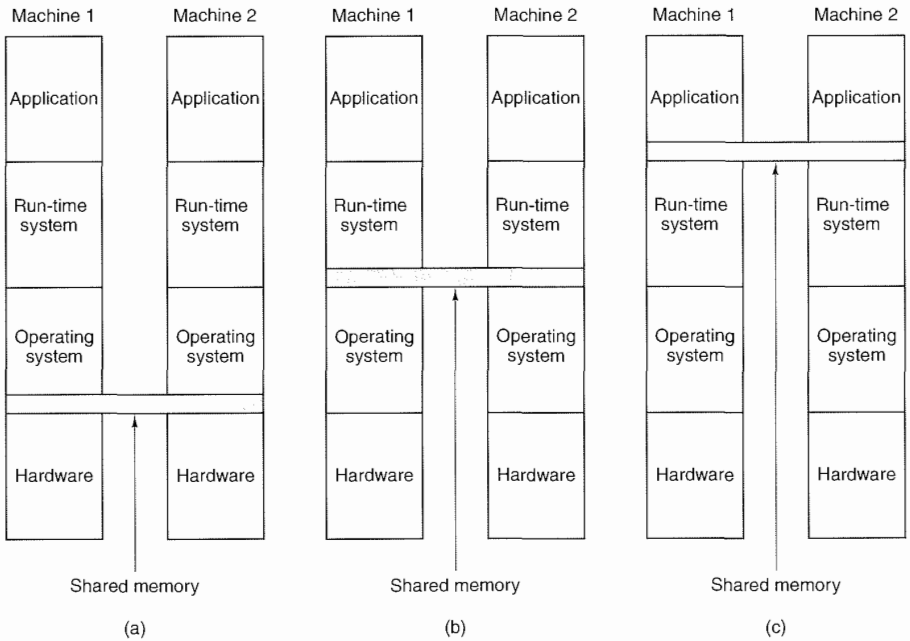


Figure 8-22. Various layers where shared memory can be implemented. (a) The hardware. (b) The operating system. (c) User-level software.

is a section of program text, its use by CPU 0 can result in a copy being sent to CPU 0, without the original in CPU 1's memory being disturbed, as shown in Fig. 8-23(c). In this way, CPUs 0 and 1 can both reference page 10 as often as needed without causing traps to fetch missing memory.

Another possibility is to replicate not only read-only pages, but also all pages. As long as reads are being done, there is effectively no difference between replicating a read-only page and replicating a read-write page. However, if a replicated page is suddenly modified, special action has to be taken to prevent having multiple, inconsistent copies in existence. How inconsistency is prevented will be discussed in the following sections.

False Sharing

DSM systems are similar to multiprocessors in certain key ways. In both systems, when a nonlocal memory word is referenced, a chunk of memory containing the word is fetched from its current location and put on the machine making the reference (main memory or cache, respectively). An important design issue is how big should the chunk be? In multiprocessors, the cache block size is usually

IBM Ex-1009
Page 405

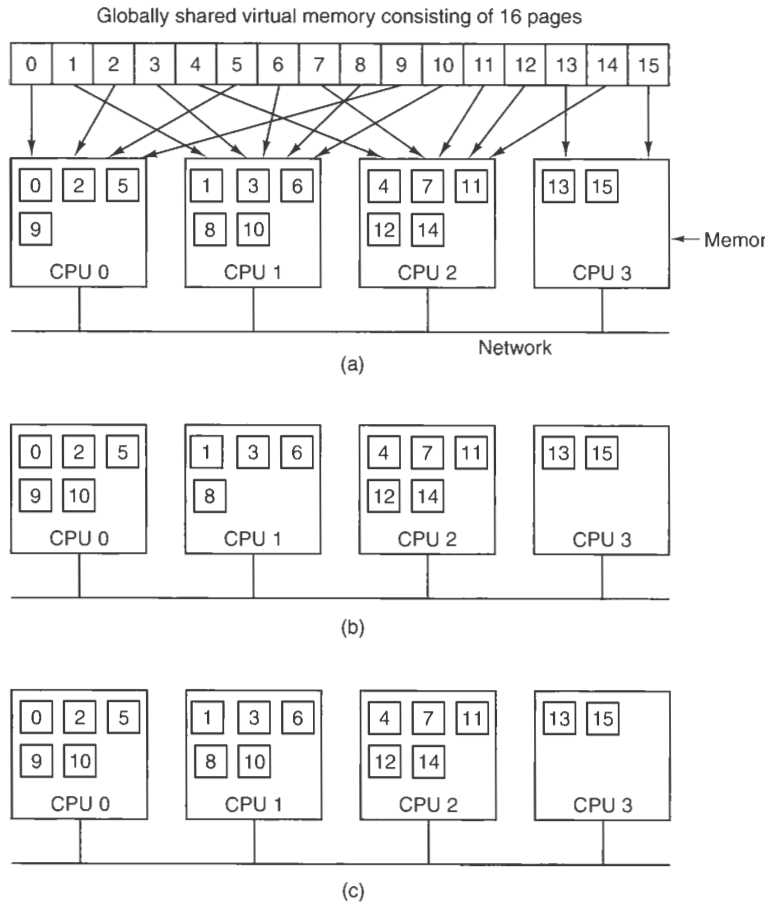


Figure 8-23. (a) Pages of the address space distributed among four machines. (b) Situation after CPU 1 references page 10. (c) Situation if page 10 is read only and replication is used.

32 or 64 bytes, to avoid tying up the bus with the transfer too long. In DSM systems, the unit has to be a multiple of the page size (because the MMU works with pages), but it can be 1, 2, 4, or more pages. In effect, doing this simulates a larger page size.

There are advantages and disadvantages to a larger page size for DSM. The biggest advantage is that because the startup time for a network transfer is fairly substantial, it does not really take much longer to transfer 4096 bytes than it does to transfer 1024 bytes. By transferring data in large units, when a large piece of

IBM Ex-1009
Page 406

address space has to be moved, the number of transfers may often be reduced. This property is especially important because many programs exhibit locality of reference, meaning that if a program has referenced one word on a page, it is likely to reference other words on the same page in the immediate future.

On the other hand, the network will be tied up longer with a larger transfer, blocking other faults caused by other processes. Also, too large an effective page size introduces a new problem, called **false sharing**, illustrated in Fig. 8-24. Here we have a page containing two unrelated shared variables, *A* and *B*. Processor 1 makes heavy use of *A*, reading and writing it. Similarly, process 2 uses *B* frequently. Under these circumstances, the page containing both variables will constantly be traveling back and forth between the two machines.

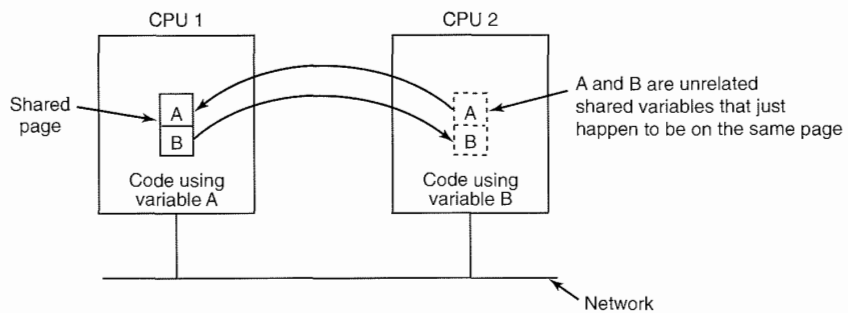


Figure 8-24. False sharing of a page containing two unrelated variables.

The problem here is that although the variables are unrelated, they appear by accident on the same page, so when a process uses one of them, it also gets the other. The larger the effective page size, the more often false sharing will occur, and conversely, the smaller the effective page size, the less often it will occur. Nothing analogous to this phenomenon is present in ordinary virtual memory systems.

Clever compilers that understand the problem and place variables in the address space accordingly can help reduce false sharing and improve performance. However, saying this is easier than doing it. Furthermore, if the false sharing consists of node 1 using one element of an array and node 2 using a different element of the same array, there is little that even a clever compiler can do to eliminate the problem.

Achieving Sequential Consistency

If writable pages are not replicated, achieving consistency is not an issue. There is exactly one copy of each writable page, and it is moved back and forth dynamically as needed. Since it is not always possible to see in advance which

IBM Ex-1009
Page 407

pages are writable, in many DSM systems, when a process tries to read a remote page, a local copy is made and both the local and remote copies are set up in their respective MMUs as read only. As long as all references are reads, everything is fine.

However, if any process attempts to write on a replicated page, a potential consistency problem arises because changing one copy and leaving the others alone is unacceptable. This situation is analogous to what happens in a multiprocessor when one CPU attempts to modify a word that is present in multiple caches. The solution there is for the CPU about to do the write to first put a signal on the bus telling all other CPUs to discard their copy of the cache block. DSM systems typically work the same way. Before a shared page can be written, a message is sent to all other CPUs holding a copy of the page telling them to unmap and discard the page. After all of them have replied that the unmap has finished, the original CPU can now do the write.

It is also possible to tolerate multiple copies of writable pages under carefully restricted circumstances. One way is to allow a process to acquire a lock on a portion of the virtual address space, and then perform multiple read and write operations on the locked memory. At the time the lock is released, changes can be propagated to other copies. As long as only one CPU can lock a page at a given moment, this scheme preserves consistency.

Alternatively, when a potentially writable page is actually written for the first time, a clean copy is made and saved on the CPU doing the write. Locks on the page can then be acquired, the page updated, and the locks released. Later, when a process on a remote machine tries to acquire a lock on the page, the CPU that wrote it earlier compares the current state of the page to the clean copy and builds a message listing all the words that have changed. This list is then sent to the acquiring CPU so it can update its copy instead of invalidating it (Keleher et al., 1994).

8.2.6 Multicomputer Scheduling

On a multiprocessor, all processes reside in the same memory. When a CPU finishes its current task, it picks a process and runs it. In principle, all processes are potential candidates. On a multicomputer the situation is quite different. Each node has its own memory and its own set of processes. CPU 1 cannot suddenly decide to run a process located on node 4 without first doing a fair amount of work to go get it. This difference means that scheduling on multicomputers is easier but allocation of processes to nodes is more important. Below we will study these issues.

Multicomputer scheduling is somewhat similar to multiprocessor scheduling, but not all of the former's algorithms apply to the latter. The simplest multiprocessor algorithm—maintaining a single central list of ready processes—does not work however, since each process can only run on the CPU it is currently located

IBM Ex-1009
Page 408

on. However, when a new process is created, a choice can be made where to place it, for example to balance the load.

Given that each node has its own processes, any local scheduling algorithm can be used. However, it is also possible to use gang scheduling, the same way it is used on a multiprocessor, since that merely requires an initial agreement on which process to run in which time slot, and some way to coordinate the start of the time slots.

8.2.7 Load Balancing

There is relatively little to say about multicomputer scheduling because once a process has been assigned to a node, any local scheduling algorithm will do, unless gang scheduling is being used. However, precisely because there is so little control once a process has been assigned to a node, the decision about which process should go on which node is important. This is in contrast to multiprocessor systems, in which all processes live in the same memory and can be scheduled on any CPU at will. Consequently, it is worth looking at how processes can be assigned to nodes in an effective way. The algorithms and heuristics for doing this assignment are known as **processor allocation algorithms**.

A large number of processor (i.e., node) allocation algorithms have been proposed over the years. They differ in what they assume is known and what the goal is. Properties that might be known about a process include the CPU requirements, memory usage, and amount of communication with every other process. Possible goals include minimizing wasted CPU cycles due to lack of local work, minimizing total communication bandwidth, and ensuring fairness to users and processes. Below we will examine a few algorithms to give an idea of what is possible.

A Graph-Theoretic Deterministic Algorithm

A widely-studied class of algorithms is for systems consisting of processes with known CPU and memory requirements, and a known matrix giving the average amount of traffic between each pair of processes. If the number of processes is greater than the number of CPUs, k, several processes will have to be assigned to each CPU. The idea is to perform this assignment such as to minimize network traffic.

The system can be represented as a weighted graph, with each vertex being a process and each arc representing the flow of messages between two processes. Mathematically, the problem then reduces to finding a way to partition (i.e., cut) the graph into k disjoint subgraphs, subject to certain constraints (e.g., total CPU and memory requirements below some limits for each subgraph). For each solution that meets the constraints, arcs that are entirely within a single subgraph represent intramachine communication and can be ignored. Arcs that go from one

IBM Ex-1009
Page 409

subgraph to another represent network traffic. The goal is then to find the partitioning that minimizes the network traffic while meeting all the constraints. As an example, Fig. 8-25 shows a system of nine processes, *A* through *I*, with each arc labeled with the mean communication load between those two processes (e.g., in Mbps).

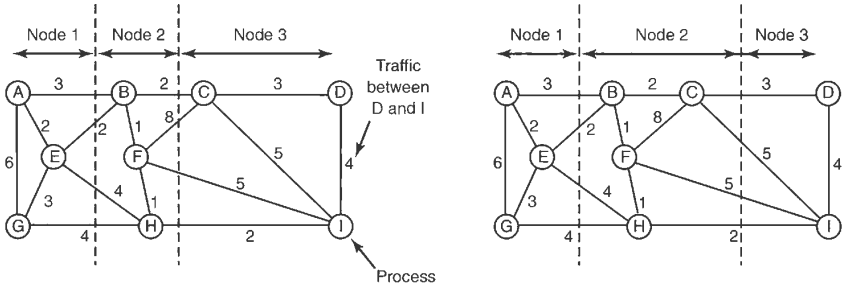


Figure 8-25. Two ways of allocating nine processes to three nodes.

In Fig. 8-25(a), we have partitioned the graph with processes *A*, *E*, and *G* on node 1, processes *B*, *F*, and *H* on node 2, and processes *C*, *D*, and *I* on node 3. The total network traffic is the sum of the arcs intersected by the cuts (the dashed lines), or 30 units. In Fig. 8-25(b) we have a different partitioning that has only 28 units of network traffic. Assuming that it meets all the memory and CPU constraints, this is a better choice because it requires less communication.

Intuitively, what we are doing is looking for clusters that are tightly coupled (high intracluster traffic flow) but which interact little with other clusters (low intercluster traffic flow). Some of the earliest papers discussing the problem are (Chow and Abraham, 1982; Lo, 1984; and Stone and Bokhari, 1978).

A Sender-Initiated Distributed Heuristic Algorithm

Now let us look at some distributed algorithms. One algorithm says that when a process is created, it runs on the node that created it unless that node is overloaded. The metric for overloaded might involve too many processes, too big a total working set, or some other metric. If it is overloaded, the node selects another node at random and asks it what its load is (using the same metric). If the probed node's load is below some threshold value, the new process is sent there (Eager et al., 1986). If not, another machine is chosen for probing. Probing does not go on forever. If no suitable host is found within *N* probes, the algorithm terminates and the process runs on the originating machine. The idea is for heavily loaded nodes to try to get rid of excess work, as shown in Fig. 8-26(a).

IBM Ex-1009
Page 410

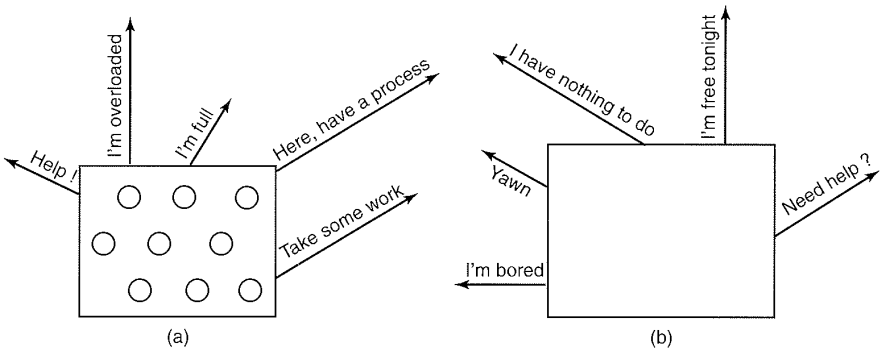


Figure 8-26. (a) An overloaded node looking for a lightly loaded node to hand off processes to. (b) An empty node looking for work to do.

Eager et al. (1986) constructed an analytical queueing model of this algorithm. Using this model, it was established that the algorithm behaves well and is stable under a wide range of parameters, including different threshold values, transfer costs, and probe limits.

Nevertheless, it should be observed that under conditions of heavy load, all machines will constantly send probes to other machines in a futile attempt to find one that is willing to accept more work. Few processes will be off loaded, but considerable overhead may be incurred in the attempt to do so.

A Receiver-Initiated Distributed Heuristic Algorithm

A complementary algorithm to the one given above, which is initiated by an overloaded sender, is one initiated by an underloaded receiver, as shown in Fig. 8-26(b). With this algorithm, whenever a process finishes, the system checks to see if it has enough work. If not, it picks some machine at random and asks it for work. If that machine has nothing to offer, a second, and then a third machine is asked. If no work is found with N probes, the node temporarily stops asking, does any work it has queued up, and tries again when the next process finishes. If no work is available, the machine goes idle. After some fixed time interval, it begins probing again.

An advantage of this algorithm is that it does not put extra load on the system at critical times. The sender-initiated algorithm makes large numbers of probes precisely when the system can least tolerate it—when it is heavily loaded. With the receiver-initiated algorithm, when the system is heavily loaded, the chance of a machine having insufficient work is small. However, when this does happen, it will be easy to find work to take over. Of course, when there is little work to do,

IBM Ex-1009
Page 411

the receiver-initiated algorithm creates considerable probe traffic as all the unemployed machines desperately hunt for work. However, it is far better to have the overhead go up when the system is underloaded than when it is overloaded.

It is also possible to combine both of these algorithms and have machines try to get rid of work when they have too much, and try to acquire work when they do not have enough. Furthermore, machines can perhaps improve on random polling by keeping a history of past probes to determine if any machines are chronically underloaded or overloaded. One of these can be tried first, depending on whether the initiator is trying to get rid of work or acquire it.

A Bidding Algorithm

Another class of algorithms tries to turn the computer system into a miniature economy, with buyers and sellers of services and prices set by supply and demand (Ferguson et al., 1988). The key players in the economy are the processes, which must buy CPU time to get their work done, and nodes, which auction their cycles off to the highest bidder.

Each node advertises its approximate price by putting it in a publicly readable file. This price is not guaranteed, but gives an indication of what the service is worth (actually, it is the price that the last customer paid). Different nodes may have different prices, depending on their speed, memory size, presence of fast floating-point hardware, and other features. An indication of the service provided, such as expected response time, can also be published.

When a process wants to start up a child process, it goes around and checks out who is currently offering the service that it needs. It then determines the set of nodes whose services it can afford. From this set, it computes the best candidate, where "best" may mean cheapest, fastest, or best price/performance, depending on the application. It then generates a bid and sends the bid to its first choice. The bid may be higher or lower than the advertised price.

Processors collect all the bids sent to them and make a choice, presumably by picking the highest one. The winners and losers are informed, and the winning process is executed. The published price of the server is then updated to reflect the new going rate.

Although Ferguson et al. do not go into the details, such an economic model raises all kinds of interesting questions, among them the following. Where do processes get money to bid? Do they get regular salaries? Does everyone get the same monthly salary, or do deans get more than professors, who in turn get more than students? If new users are introduced into the system without a corresponding increase in resources, do prices get bid up (inflation)? Can nodes form cartels to gouge users? Are users' unions allowed? Is disk space also chargeable? How about printer output? Does printing pictures cost more than printing text due to the additional ink or toner used? The list goes on and on.

IBM Ex-1009
Page 412

8.3 DISTRIBUTED SYSTEMS

Having now completed our study of multiprocessors and multicomputers, it is time to turn to the third type of multiple processor system, the **distributed system**. These systems are similar to multicomputers in that each node has its own private memory, with no shared physical memory in the system. However, distributed systems are even more loosely coupled than multicomputers.

To start with, the nodes of a multicomputer generally have a CPU, RAM, a network interface, and perhaps a hard disk for paging. In contrast, each node in a distributed system is a complete computer, with a full complement of peripherals. Next, the nodes of a multicomputer are normally in a single room so they can communicate by a dedicated high-speed network, whereas the nodes of a distributed system may be spread around the world. Finally, all the nodes of a multicomputer run the same operating system, share a single file system, and are under a common administration, whereas the nodes of a distributed systems may run different operating systems, each have their own file system, and be under different administrations. A typical example of a multicomputer is 512 nodes in a single room at a company or university working on, say, pharmaceutical modeling, whereas a typical distributed system consists of thousands of machines loosely cooperating over the Internet. Figure 8-27 compares multiprocessors, multicomputers, and distributed systems on the points mentioned above.

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

Figure 8-27. Comparison of three kinds of multiple CPU systems.

Multicomputers are clearly in the middle using these metrics. An interesting question is: "Are multicomputers more like multiprocessors or more like distributed systems?" Oddly enough, the answer depends strongly on your perspective. From a technical perspective, multiprocessors have shared memory and the other two do not. This difference leads to different programming models and different mind sets. However, from an applications perspective, multiprocessors and multicomputers are just big equipment racks in a machine room. Both are used for solving computationally intensive problems, whereas a distributed system connecting computers all over the Internet is typically much more involved in communication than in computation and is used in a different way.

IBM Ex-1009
Page 413

To some extent, loose coupling of the computers in a distributed system is both a strength and a weakness. It is a strength because the computers can be used for a wide variety of applications, but it is also a weakness, because programming these applications is difficult due to the lack of any common underlying model.

Typical Internet applications include access to remote computers (using *telnet* and *rlogin*), access to remote information (using the World Wide Web and FTP, the File Transfer Protocol), person-to-person communication (using email and chat programs), and many emerging applications (e.g., e-commerce, telemedicine, and distance learning). The trouble with all these applications is that each one has to reinvent the wheel. For example, email, FTP, and the World Wide Web all basically move files from point *A* to point *B*, but each one has its own way of doing it, complete with its own naming conventions, transfer protocols, replication techniques, and everything else. Although many Web browsers hide these differences from the average user, the underlying mechanisms are completely different. Hiding them at the user interface level is like having a person at a full-service travel agent Web site order a trip from New York to San Francisco, and only later discover whether she has purchased a plane, train, or bus ticket.

What distributed systems add to the underlying network is some common paradigm (model) that provides a uniform way of looking at the whole system. The intent of the distributed system is to turn a loosely-connected bunch of machines into a coherent system based on one concept. Sometimes the paradigm is simple and sometimes it is more elaborate, but the idea is always to provide something that unifies the system.

A simple example of a unifying paradigm in a slightly different context is found in UNIX, where all I/O devices are made to look like files. Having keyboards, printers, and serial lines all be operated on the same way, with the same primitives, makes it easier to deal with them than having them all be conceptually different.

One way a distributed system can achieve some measure of uniformity in the face of different underlying hardware and operating systems is to have a layer of software on top of the operating system. The layer, called **middleware**, is illustrated in Fig. 8-28. This layer provides certain data structures and operations that allow processes and users on far-flung machines to interoperate in a consistent way.

In a sense, middleware is like the operating system of a distributed system. That is why it is being discussed in a book on operating systems. On the other hand, it is *not* an operating system, so the discussion will not go into much detail. For a comprehensive, book-length treatment of distributed systems, see *Distributed Systems* (Tanenbaum and van Steen, 2002). In the remainder of this chapter, we will first look quickly at the hardware used in a distributed system (i.e., the underlying computer network), then its communication software (the network protocols). After that we will consider a variety of paradigms used in these systems.

IBM Ex-1009
Page 414

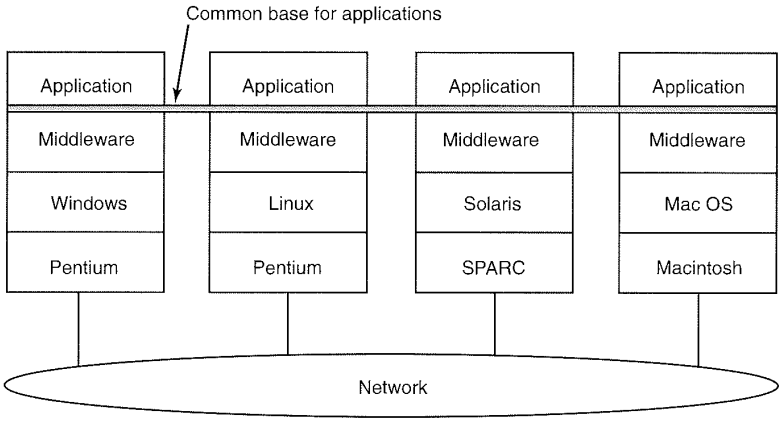


Figure 8-28. Positioning of middleware in a distributed system.

8.3.1 Network Hardware

Distributed systems are built on top of computer networks, so a brief introduction to the subject is in order. Networks come in two major varieties, **LANs** (**Local Area Networks**), which cover a building or a campus, and **WANs** (**Wide Area Networks**), that can be citywide, countrywide, or even worldwide. The most important kind of LAN is Ethernet, so we will examine that as an example LAN. As our example WAN, we will look at the Internet, even though technically, the Internet is not one network, but a federation of thousands of separate networks. However, for our purposes, it is sufficient to think of it as one WAN.

Ethernet

Classic Ethernet, which is described in IEEE Standard 802.3, consists of a coaxial cable to which a number of computers are attached. The cable is called the **Ethernet**, in reference to the *luminiferous ether*, through which electromagnetic radiation was once thought to propagate. (When the nineteenth-century British physicist James Clerk Maxwell discovered that electromagnetic radiation could be described by a wave equation, scientists assumed that space must be filled with some ethereal medium in which the radiation was propagating. Only after the famous Michelson-Morley experiment in 1887, which failed to detect the ether, did physicists realize that radiation could propagate in a vacuum.)

In the very first version of Ethernet, a computer was attached to the cable by literally drilling a hole halfway through the cable and screwing in a wire leading to the computer. This was called a **vampire tap**, and is shown symbolically in Fig. 8-29(a). The taps were hard to get right, so before long, proper connectors

IBM Ex-1009
Page 415

were used. Nevertheless, electrically, all the computers were connected as if the cables on their network interface cards were soldered together.

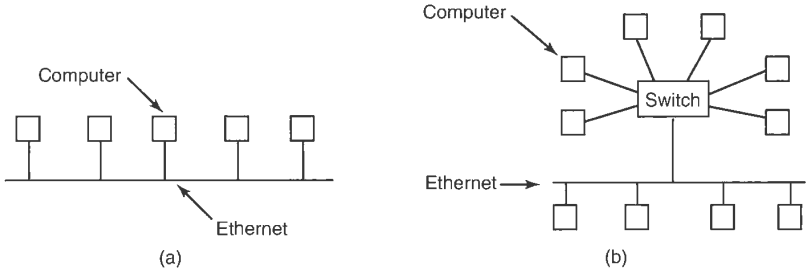


Figure 8-29. (a) Classic Ethernet. (b) Switched Ethernet.

To send a packet on an Ethernet, a computer first listens to the cable to see if any other computer is currently transmitting. If not, it just begins transmitting a packet, which consists of a short header followed by a 0- to 1500-byte payload. If the cable is in use, the computer simply waits until the current transmission finishes, then it begins sending.

If two computers start transmitting simultaneously, a collision results, which both of them detect. Both respond by terminating their transmissions, waiting a random amount of time between 0 and T μsec and then starting again. If another collision occurs, all colliding computers randomize the wait into the interval 0 to $2T$ μsec, and then try again. On each further collision, the maximum wait interval is doubled, reducing the chance of more collisions. This algorithm is called **binary exponential backoff**. We saw it earlier to reduce polling overhead on locks.

An Ethernet has a maximum cable length and also a maximum number of computers that can be connected to it. To exceed either of these limits, a large building or campus can be wired with multiple Ethernets, which are then connected by devices called **bridges**. A bridge allows traffic to pass from one Ethernet to another when the source is on one side and the destination is on the other.

To avoid the problem of collisions, modern Ethernets use switches, as shown in Fig. 8-29(b). Each switch has some number of ports, to which can be attached a computer, an Ethernet, or another switch. When a packet successfully avoids all collisions and makes it to the switch, it is buffered there and sent out on the port where the destination machine lives. By giving each computer its own port, all collisions can be eliminated, at the cost of bigger switches. Compromises, with just a few computers per port are also possible.

The Internet

The Internet evolved from the ARPANET, an experimental packet-switched network funded by the U.S. Dept. of Defense Advanced Research Projects Agency. It went live in Dec. 1969 with three computers in California and one in

IBM Ex-1009
Page 416

Utah. It was designed to a be a highly fault-tolerant network that would continue to relay military traffic even in the event of direct nuclear hits on multiple parts of the network by automatically rerouting traffic around the dead machines.

The ARPANET grew rapidly in the 1970s, eventually encompassing hundreds of computers. Then a packet radio network, a satellite network, and eventually thousands of Ethernets were attached to it, leading to the federation of networks we now know as the Internet.

The Internet consists of two kinds of computers, hosts and routers. **Hosts** are PCs, laptops, palmtops, servers, mainframes, and other computers owned by individuals or companies that want to connect to the Internet. **Routers** are specialized switching computers that accept incoming packets on one of many incoming lines and send them on their way along one of many outgoing lines. A router is similar to the switch of Fig. 8-29(b), but also differs from it in ways that will not concern us here. Routers are connected together in large networks, with each router having wires or fibers to many other routers and hosts. Large national or worldwide router networks are operated by telephone companies and ISPs (Internet Service Providers) for their customers.

Figure 8-30 shows a portion of the Internet. At the top we have one of the backbones, normally operated by a backbone operator. It consists of a number of routers connected by high-bandwidth fiber optics, with connections to backbones operated by other (competing) telephone companies. Usually, no hosts connect directly to the backbone, other than maintenance and test machines run by the telephone company.

Attached to the backbone routers by medium-speed fiber optic connections are regional networks and routers at ISPs. In turn, corporate Ethernets each have a router on them and these are connected to regional network routers. Routers at ISPs are connected to modem banks used by the ISP's customers. In this way, every host on the Internet has at least one path, and often many paths, to every other host.

All traffic on the Internet is sent in the form of packets. Each packet carries its destination address inside it, and this address is used for routing. When a packet comes into a router, the router extracts the destination address and looks (part of) it up in a table to find which outgoing line to send the packet on and thus to which router. This procedure is repeated until the packet reaches the destination host. The routing tables are highly dynamic and are updated continuously as routers and links go down and come back up and as traffic conditions change.

8.3.2 Network Services and Protocols

All computer networks provide certain services to their users (hosts and processes), which they implement using certain rules about legal message exchanges. Below we will give a brief introduction to these topics.

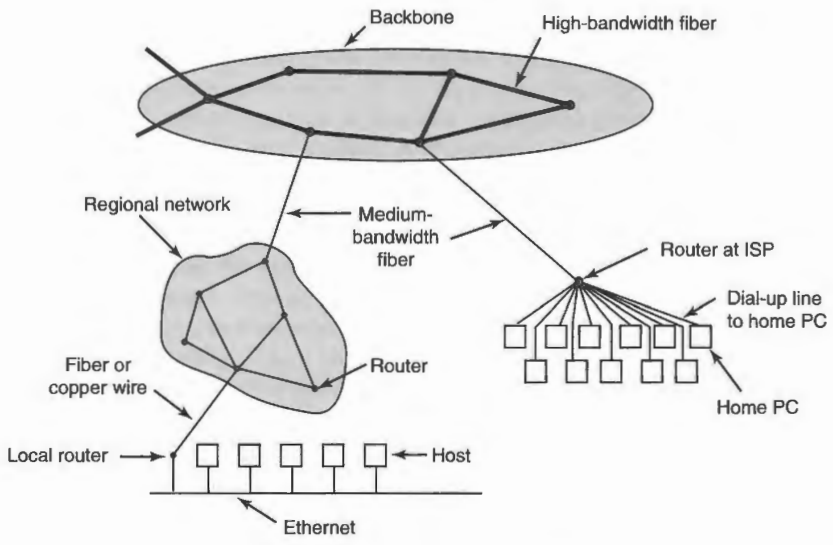


Figure 8-30. A portion of the Internet.

Network Services

Computer networks provide services to the hosts and processes using them. **Connection-oriented service** is modeled after the telephone system. To talk to someone, you pick up the phone, dial the number, talk, and then hang up. Similarly, to use a connection-oriented network service, the service user first establishes a connection, uses the connection, and then releases the connection. The essential aspect of a connection is that it acts like a tube: the sender pushes objects (bits) in at one end, and the receiver takes them out in the same order at the other end.

In contrast, **connectionless service** is modeled after the postal system. Each message (letter) carries the full destination address, and each one is routed through the system independent of all the others. Normally, when two messages are sent to the same destination, the first one sent will be the first one to arrive. However, it is possible that the first one sent can be delayed so that the second one arrives first. With a connection-oriented service this is impossible.

Each service can be characterized by a **quality of service**. Some services are reliable in the sense that they never lose data. Usually, a reliable service is implemented by having the receiver confirm the receipt of each message by sending back a special **acknowledgement packet** so the sender is sure that it arrived. The acknowledgement process introduces overhead and delays, which are necessary to detect packet loss, but which do slow things down.

A typical situation in which a reliable connection-oriented service is appropriate is file transfer. The owner of the file wants to be sure that all the bits arrive correctly and in the same order they were sent. Very few file transfer customers would prefer a service that occasionally scrambles or loses a few bits, even if it is much faster.

Reliable connection-oriented service has two minor variations: message sequences and byte streams. In the former, the message boundaries are preserved. When two 1-KB messages are sent, they arrive as two distinct 1-KB messages, never as one 2-KB message. In the latter, the connection is simply a stream of bytes, with no message boundaries. When 2K bytes arrive at the receiver, there is no way to tell if they were sent as one 2-KB message, two 1-KB messages, or 2048 1-byte messages. If the pages of a book are sent over a network to an imagesetter as separate messages, it might be important to preserve the message boundaries. On the other hand, with a terminal logging into a remote timesharing system, a byte stream from the terminal to the computer is all that is needed.

For some applications, the delays introduced by acknowledgements are unacceptable. One such application is digitized voice traffic. It is preferable for telephone users to hear a bit of noise on the line or a garbled word from time to time than to introduce a delay to wait for acknowledgements.

Not all applications require connections. For example, to test the network, all that is needed is a way to send a single packet that has a high probability of arrival, but no guarantee. Unreliable (meaning not acknowledged) connectionless service is often called **datagram service**, in analogy with telegram service, which also does not provide an acknowledgement back to the sender.

In other situations, the convenience of not having to establish a connection to send one short message is desired, but reliability is essential. The **acknowledged datagram service** can be provided for these applications. It is like sending a registered letter and requesting a return receipt. When the receipt comes back, the sender is absolutely sure that the letter was delivered to the intended party and not lost along the way.

Still another service is the **request-reply service**. In this service the sender transmits a single datagram containing a request; the reply contains the answer. For example, a query to the local library asking where Uighur is spoken falls into this category. Request-reply is commonly used to implement communication in the client-server model: the client issues a request and the server responds to it. Figure 8-31 summarizes the types of services discussed above.

Network Protocols

All networks have highly-specialized rules for what messages may be sent and what responses may be returned in response to these messages. For example, under certain circumstances (e.g., file transfer), when a message is sent from a source to a destination, the destination is required to send an acknowledgement

IBM Ex-1009
Page 419

	Service	Example
Connection-oriented	Reliable message stream	Sequence of pages of a book
	Reliable byte stream	Remote login
	Unreliable connection	Digitized voice
Connectionless	Unreliable datagram	Network test packets
	Acknowledged datagram	Registered mail
	Request-reply	Database query

Figure 8-31. Six different types of network service.

back indicating correct receipt of the message. Under other circumstances (e.g., digital telephony), no such acknowledgement is expected. The set of rules by which particular computers communicate is called a **protocol**. Many protocols exist, including router-router protocols, host-host protocols, and others. For a thorough treatment of computer networks and their protocols, see *Computer Networks* (Tanenbaum, 1996).

All modern networks use what is called a **protocol stack** to layer different protocols on top of one another. At each layer, different issues are dealt with. For example, at the bottom level protocols define how to tell where in the bit stream a packet begins and ends. At a higher level, protocols deal with how to route packets through complex networks from source to destination. And at a still higher level, they make sure that all the packets in a multipacket message have arrived correctly and in the proper order.

Since most distributed systems use the Internet as a base, the key protocols these systems use are the two major Internet protocols: IP and TCP. **IP (Internet Protocol)** is a datagram protocol in which a sender injects a datagram of up to 64 KB into the network and hopes that it arrives. No guarantees are given. The datagram may be fragmented into smaller packets as it passes through the Internet. These packets travel independently, possibly along different routes. When all the pieces get to the destination, they are assembled in the correct order and delivered.

Two versions of IP are currently in use, v4 and v6. At the moment, v4 still dominates, so we will describe that here, but v6 is up and coming. Each v4 packet starts with a 40-byte header that contains a 32-bit source address and a 32-bit destination address among other fields. These are called **IP addresses** and form the basis of routing in the Internet. They are conventionally written as four decimal numbers in the range 0–255 separated by dots, as in 192.31.231.65. When a packet arrives at router, the router extracts the IP destination address and uses that for routing the packet.

Since IP datagrams are not acknowledged, IP alone is not sufficient for reliable communication in the Internet. To provide reliable communication, another

IBM Ex-1009
Page 420

protocol, **TCP** (**Transmission Control Protocol**), is usually layered on top of IP. TCP uses IP to provide connection-oriented streams. To use TCP, a process first establishes a connection to a remote process. The process required is specified by the IP address of a machine and a port number on that machine, to which processes interested in receiving incoming connections listen. Once that has been done, it just pumps bytes into the connection and they are guaranteed to come out the other end undamaged and in the correct order. The TCP implementation achieves this guarantee by using sequence numbers, checksums, and retransmissions of incorrectly received packets. All of this is transparent to the sending and receiving processes. They just see reliable interprocess communication, just like a UNIX pipe .

To see how all these protocols interact, consider the simplest case of a very small message that does not need to be fragmented at any level. The host is on an Ethernet connected to the Internet. What happens exactly? The user process generates the message and makes a system call to send it on a previously established TCP connection. The kernel protocol stack adds a TCP header and then an IP header to the front. Then it goes to the Ethernet driver, which adds an Ethernet header directing the packet to the router on the Ethernet. This router then injects the packet into the Internet, as depicted in Fig. 8-32.

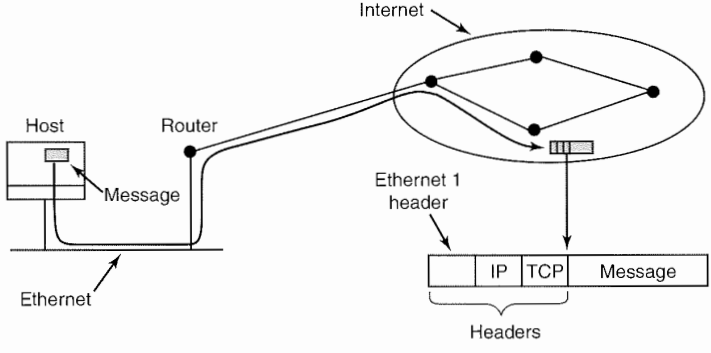


Figure 8-32. Accumulation of packet headers.

To establish a connection with a remote host (or even to send it a datagram), it is necessary to know its IP address. Since managing lists of 32-bit IP addresses is inconvenient for people, a scheme called **DNS** (**Domain Name System**) was invented as a database that maps ASCII names for hosts onto their IP addresses. Thus it is possible to use the DNS name *star.cs.vu.nl* instead of the corresponding IP address 130.37.24.6. DNS names are widely known because Internet email addresses are of the form *user-name@DNS-host-name*. This naming system allows the mail program on the sending host to look up the destination host's IP

IBM Ex-1009
Page 421

address in the DNS database, establish a TCP connection to the mail daemon process there, and send the message as a file. The *user-name* is sent along to identify which mailbox to put the message in.

8.3.3 Document-Based Middleware

Now that we have some background on networks and protocols, we can start looking at different middleware layers that can overlay the basic network to produce a consistent paradigm for applications and users. We will start with a simple, but well-known example: the World Wide Web. The Web was invented by Tim Berners-Lee at CERN, the European Nuclear Physics Research Center, in 1989 and has spread like wildfire all over the world since then.

The original paradigm behind the Web was very simple: every computer can hold one or more documents, called **Web pages**. Each Web page contains text, images, icons, sounds, movies, etc. as well as **hyperlinks** (pointers) to other Web pages. When a user requests a Web page using a program called a **Web browser**, the page is displayed on the screen. Clicking on a link causes the current page to be replaced on the screen by the page pointed to. Although many bells and whistles have been grafted onto the Web recently, the underlying paradigm is still clearly present: the Web is a great big directed graph of documents that can point to other documents, as shown in Fig. 8-33.

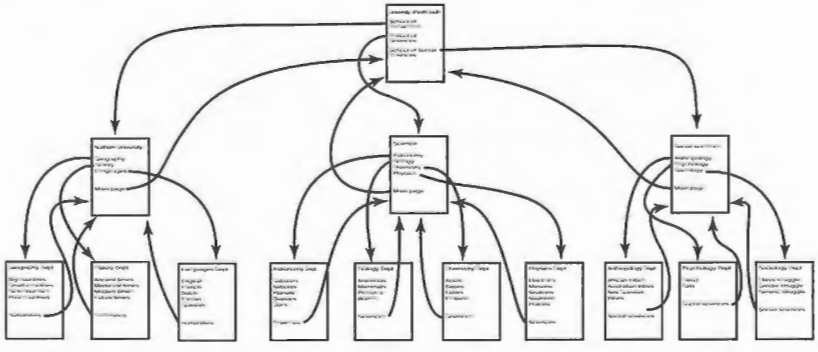


Figure 8-33. The Web is a big directed graph of documents.

Each Web page has a unique address, called a **URL (Uniform Resource Locator)**, of the form *protocol://DNS-name/file-name*. The protocol is most commonly *http* (HyperText Transfer Protocol), but *ftp* and others also exist. Then comes the DNS name of the host containing the file. Finally, there is a local file name telling which file is needed.

The way the whole system hangs together is as follows. The Web is fundamentally a client-server system, with the user being the client and the Web site

IBM Ex-1009
Page 422

being the server. When the user provides the browser with a URL, either by typing it in or clicking on a hyperlink on the current page, the browser takes certain steps to fetch the requested Web page. As an example, suppose the URL provided is *http://www.acm.org/dl/faq.html* The browser then takes the following steps to get the page.

1. The browser asks DNS for the IP address of *www.acm.org*.
2. DNS replies with 199.222.69.151.
3. The browser makes a TCP connection to port 80 on 199.222.69.151.
4. It then sends a request asking for the file *dl/faq.html*.
5. The *www.acm.org* server sends the file *dl/faq.html*.
6. The TCP connection is released.
7. The browser displays all the text in *dl/faq.html*.
8. The browser fetches and displays all images in *dl/faq.html*.

To a first approximation, that is the basis of the Web and how it works. Many other features have since been added to the basic Web, including style sheets, dynamic Web pages that are generated on-the-fly, Web pages that contain small programs or scripts that execute on the client machine, and more, but they are outside the scope of this discussion.

8.3.4 File System-Based Middleware

The basic idea behind the Web is to make a distributed system look like a giant collection of hyperlinked documents. A second approach is to make a distributed system look like a great big file system. In this section we will look at some of the issues involved in designing a worldwide file system.

Using a file system model for a distributed system means that there is a single global file system, with users all over the world able to read and write files for which they have authorization. Communication is achieved by having one process write data into a file and having other ones read them back. Many of the standard file system issues arise here, but also some new ones related to distribution.

Transfer Model

The first issue is the choice between the **upload/download model** and the **remote access model**. In the former, shown in Fig. 8-34(a), a process accesses a file by first copying it from the remote server where it lives. If the file is only to be read, the file is then read locally, for high performance. If the file is to be written, it is written locally. When the process is done with it, the updated file is put

back on the server. With the remote access model, the file stays on the server and the client sends commands there to get work done there, as shown in Fig. 8-34(b).

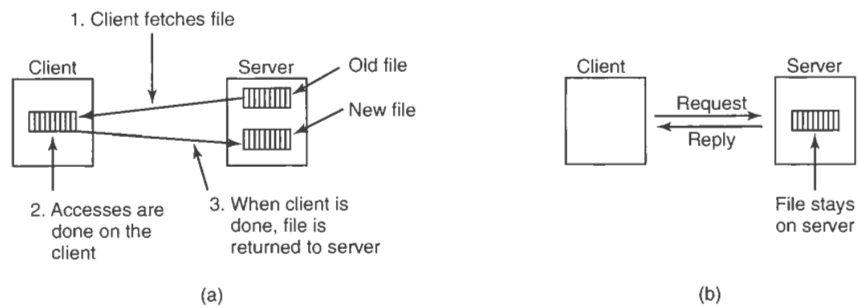


Figure 8-34. (a) The upload/download model. (b) The remote access model.

The advantages of the upload/download model are its simplicity, and the fact that transferring entire files at once is more efficient than transferring them in small pieces. The disadvantages are that there must be enough storage for the entire file locally, moving the entire file is wasteful if only parts of it are needed, and consistency problems arise if there are multiple concurrent users.

The Directory Hierarchy

Files are only part of the story. The other part is the directory system. All distributed file systems support directories containing multiple files. The next design issue is whether all clients have the same view of the directory hierarchy. As an example of what we mean by this remark, consider Fig. 8-35. In Fig. 8-35(a) we show two file servers, each holding three directories and some files. In Fig. 8-35(b) we have a system in which all clients (and other machines) have the same view of the distributed file system. If the path */D/E/x* is valid on one machine, it is valid on all of them.

In contrast, in Fig. 8-35(c), different machines can have different views of the file system. To repeat the preceding example, the path */D/E/x* might well be valid on client 1 but not on client 2. In systems that manage multiple file servers by remote mounting, Fig. 8-35(c) is the norm. It is flexible and straightforward to implement, but it has the disadvantage of not making the entire system behave like a single old-fashioned timesharing system. In a timesharing system, the file system looks the same to any process [i.e., the model of Fig. 8-35(b)]. This property makes a system easier to program and understand.

A closely related question is whether or not there is a global root directory, which all machines recognize as the root. One way to have a global root directory is to have the root contain one entry for each server and nothing else. Under these

IBM Ex-1009
Page 424

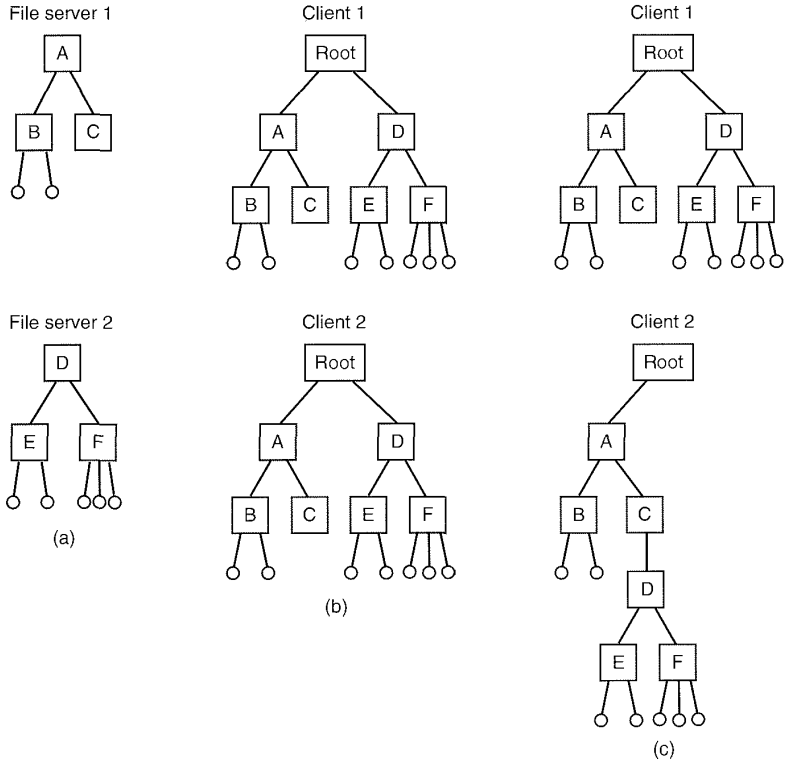


Figure 8-35. (a) Two file servers. The squares are directories and the circles are files. (b) A system in which all clients have the same view of the file system. (c) A system in which different clients may have different views of the file system.

circumstances, paths take the form */server/path*, which has its own disadvantages, but at least is the same everywhere in the system.

Naming Transparency

The principal problem with this form of naming is that it is not fully transparent. Two forms of transparency are relevant in this context and are worth distinguishing. The first one, **location transparency**, means that the path name gives no hint as to where the file is located. A path like */server1/dir1/dir2/x* tells everyone that *x* is located on server 1, but it does not tell where that server is located. The server is free to move anywhere it wants to in the network without the path name having to be changed. Thus this system has location transparency.

IBM Ex-1009
Page 425

However, suppose that file *x* is extremely large and space is tight on server 1. Furthermore, suppose that there is plenty of room on server 2. The system might well like to move *x* to server 2 automatically. Unfortunately, when the first component of all path names is the server, the system cannot move the file to the other server automatically, even if *dir1* and *dir2* exist on both servers. The problem is that moving the file automatically changes its path name from */server1/dir1/dir2/x* to */server2/dir1/dir2/x*. Programs that have the former string built into them will cease to work if the path changes. A system in which files can be moved without their names changing is said to have **location independence**. A distributed system that embeds machine or server names in path names clearly is not location independent. One based on remote mounting is not either, since it is not possible to move a file from one file group (the unit of mounting) to another and still be able to use the old path name. Location independence is not easy to achieve, but it is a desirable property to have in a distributed system.

To summarize what we have said earlier, there are three common approaches to file and directory naming in a distributed system:

1. Machine + path naming, such as */machine/path* or *machine:path*.
2. Mounting remote file systems onto the local file hierarchy.
3. A single name space that looks the same on all machines.

The first two are easy to implement, especially as a way to connect existing systems that were not designed for distributed use. The latter is difficult and requires careful design, but makes life easier for programmers and users.

Semantics of File Sharing

When two or more users share the same file, it is necessary to define the semantics of reading and writing precisely to avoid problems. In single-processor systems the semantics normally state that when a read system call follows a write system call, the read returns the value just written, as shown in Fig. 8-36(a). Similarly, when two writes happen in quick succession, followed by a read, the value read is the value stored by the last write. In effect, the system enforces an ordering on all system calls and all processors see the same ordering. We will refer to this model as **sequential consistency**.

In a distributed system, sequential consistency can be achieved easily as long as there is only one file server and clients do not cache files. All reads and writes go directly to the file server, which processes them strictly sequentially.

In practice, however, the performance of a distributed system in which all file requests must go to a single server is frequently poor. This problem is often solved by allowing clients to maintain local copies of heavily used files in their private caches. However, if client 1 modifies a cached file locally and shortly thereafter client 2 reads the file from the server, the second client will get an obsolete file, as illustrated in Fig. 8-36(b).

IBM Ex-1009
Page 426

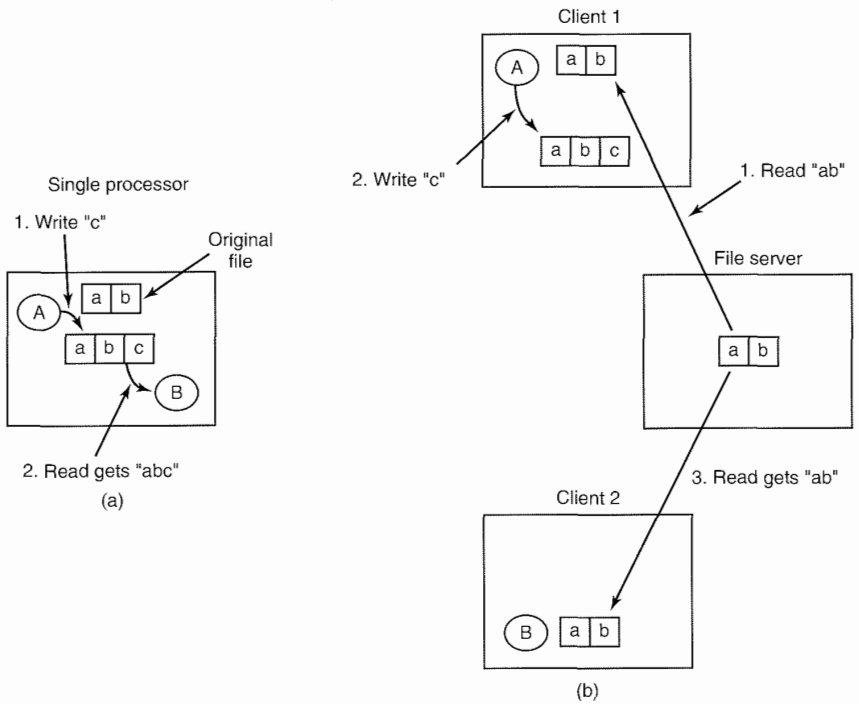


Figure 8-36. (a) Sequential consistency. (b) In a distributed system with caching, reading a file may return an obsolete value.

One way out of this difficulty is to propagate all changes to cached files back to the server immediately. Although conceptually simple, this approach is inefficient. An alternative solution is to relax the semantics of file sharing. Instead of requiring a read to see the effects of all previous writes, one can have a new rule that says: "Changes to an open file are initially visible only to the process that made them. Only when the file is closed are the changes visible to other processes." The adoption of such a rule does not change what happens in Fig. 8-36(b), but it does redefine the actual behavior (*B* getting the original value of the file) as being the correct one. When client 1 closes the file, it sends a copy back to the server, so that subsequent reads get the new value, as required. Effectively, this is the upload/download model of Fig. 8-34. This semantic rule is widely implemented and is known as **session semantics**.

Using session semantics raises the question of what happens if two or more clients are simultaneously caching and modifying the same file. One solution is to say that as each file is closed in turn, its value is sent back to the server, so the

IBM Ex-1009
Page 427

final result depends on who closes last. A less pleasant, but slightly easier to implement, alternative is to say that the final result is one of the candidates, but leave the choice of which one unspecified.

An alternative approach to session semantics is to use the upload/download model, but to automatically lock a file that has been downloaded. Attempts by other clients to download the file will be held up until the first client has returned it. If there is a heavy demand for a file, the server could send messages to the client holding the file, asking it to hurry up, but that may or may not help. All in all, getting the semantics of shared files right is a tricky business with no elegant and efficient solutions.

AFS

Several file-system based middleware systems have been built and deployed. Below we will briefly discuss one (AFS) based on the upload/download model of Fig. 8-34(a). In Chap. 10, we will discuss one (NFS) based on the remote access model of Fig. 8-34(b).

AFS was designed and implemented at Carnegie-Mellon University (Howard et al. 1988; Morris et al., 1986; and Satyanarayanan et al., 1985). It was originally called the **Andrew File System** in honor of the university's first benefactors, Andrew Carnegie and Andrew Mellon. The goal of the project, which started in the early 1980s, was to provide every student and faculty member at CMU with a powerful personal workstation running UNIX, but with a shared file system. Here the file system was being used as middleware to turn a collection of workstations into a coherent system.

Each AFS user has a private workstation running a slightly modified version of UNIX. The modifications consist of adding a piece of code called **venus** to the kernel and running a file server called **vice** in user space (originally, venus also ran in user space, but was later moved into the kernel for performance reasons). The positions of *venus* and *vice* are shown in Fig. 8-37(a). User workstations are grouped into **cells** for administrative purposes. A cell might be a LAN or a collection of interconnected LANs or even an entire academic department.

The name space visible to user programs looks like a traditional UNIX tree, with the addition of a directories */cmu* and */cache*, as depicted in Fig. 8-37(b). The */cache* directory contains cached remote files. The */cmu* directory contains the names of the shared remote cells, below which are their respective file systems. In effect, remote file systems are mounted in */cmu*. The other directories and files are strictly local and are not shared. Symbolic links from local file names to shared files are permitted, as indicated by *sh* in Fig. 8-37(b).

The basic idea behind AFS is for each user to do as much as possible locally and interact as little as possible with the rest of the system. When a file is opened, the *venus* code traps the open call and downloads the entire file (or if it is a huge file, a large chunk of it) to the local disk and inserts it into the */cache* directory.

IBM Ex-1009
Page 428

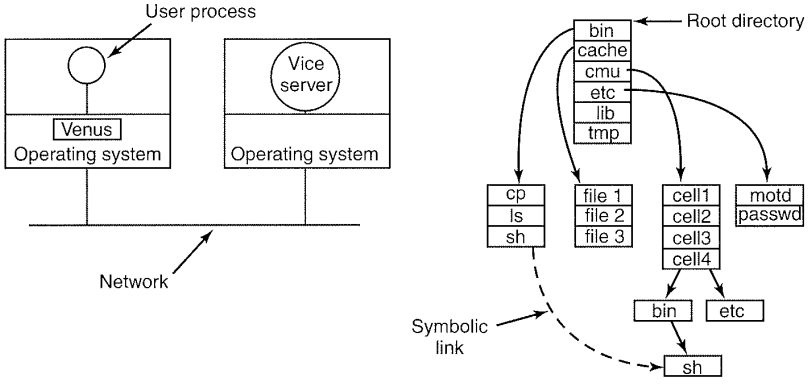


Figure 8-37. (a) The position of venus and vice in AFS. (b) A client's view of the file system.

The file descriptor returned by the open call refers to the file in */cache* so that subsequent read and write calls use the cached file.

The semantics offered by AFS are close to session semantics. When a file is opened, it is fetched from the appropriate server and placed in */cache* on the workstation's local disk. All reads and writes operate on the cached copy. When the file is closed, it is uploaded back to the server.

However, to prevent unsuspecting processes from using stale files in situations where it matters, when *venus* downloads a file into its cache, it tells *vice* whether or not it cares about subsequent opens by processes on other workstations. If it does, *vice* records the location of the cached file. If another process elsewhere in the system opens the file, *vice* sends a message to *venus* telling it to mark its cache entry as invalid and return the copy if it has been modified.

8.3.5 Shared Object-Based Middleware

Now let us take a look at a third paradigm. Instead of saying that everything is a document or everything is a file, we say that everything is an object. An **object** is a collection of variables that are bundled together with a set of access procedures, called **methods**. Processes are not permitted to access the variables directly. Instead, they are required to invoke the methods.

CORBA

Some programming languages, such as C++ and Java, are object oriented, but these are language-level objects rather than run-time objects. One well-known system based on run-time objects is **CORBA** (**Common Object Request Broker**

IBM Ex-1009
Page 429

Architecture) (Vinoski, 1997). CORBA is a client-server system, in which client processes on client machines can invoke operations on objects located on (possibly remote) server machines. CORBA was designed for a heterogeneous system running a variety of hardware platforms and operating systems and programmed in a variety of languages. To make it possible for a client on one platform to invoke a server on a different platform, **ORB**s (**Object Request Brokers**) are interposed between client and server to allow them to match up. The ORBs play an important role in CORBA, even providing the system with its name.

Each CORBA object is defined by an interface definition in a language called **IDL** (**Interface Definition Language**), which tells what methods the object exports and what parameter types each one expects. The IDL specification can be compiled into a client stub procedure and stored in a library. If a client process knows in advance that it will need to access a certain object, it is linked with the object's client stub code. The IDL specification can also be compiled into a **skeleton** procedure that is used on the server side. If it is not known in advance which CORBA objects a process needs to use, dynamic invocation is also possible, but how that works is beyond the scope of our treatment.

When a CORBA object is created, a reference to it is also created and returned to the creating process. This reference is how the process identifies the object for subsequent invocations of its methods. The reference can be passed to other processes or stored in an object directory.

To invoke a method on an object, a client process must first acquire a reference to the object. The reference can either come directly from the creating process, or more likely, by looking it up by name or by function in some kind of a directory. Once the object reference is available, the client process marshals the parameters to the method calls into a convenient structure and then contacts the client ORB. In turn, the client ORB sends a message to the server ORB, which actually invokes the method on the object. The whole mechanism is similar to RPC.

The function of the ORBs is to hide all the low-level distribution and communication details from the client and server code. In particular, the ORBs hide from the client the location of the server, whether the server is a binary program or a script, what hardware and operating system the server runs on, whether the object is currently active, and how the two ORBs communicate (e.g., TCP/IP, RPC, shared memory, etc.).

In the first version of CORBA, the protocol between the client ORB and the server ORB was not specified. As a result, every ORB vendor used a different protocol and no two of them could talk to each other. In version 2.0, the protocol was specified. For communication over the Internet, the protocol is called **IIOP** (**Internet InterOrb Protocol**).

To make it possible to use objects that were not written for CORBA with CORBA systems, every object can be equipped with an **object adapter**. This is a wrapper that handles chores such as registering the object, generating object ref-

IBM Ex-1009
Page 430

erences, and activating the object if it is invoked when it is not active. The arrangement of all these CORBA parts is shown in Fig. 8-38.

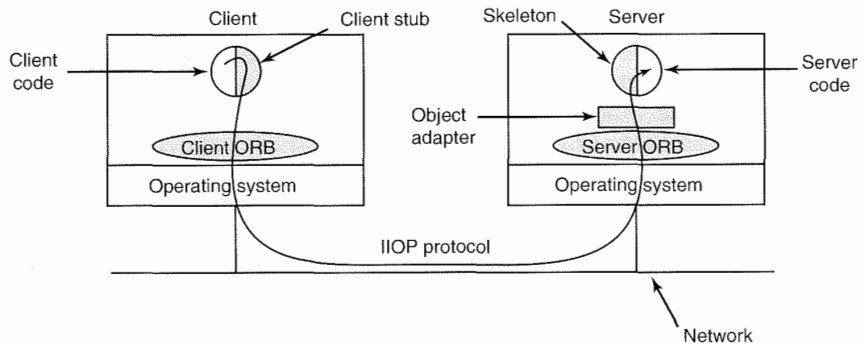


Figure 8-38. The main elements of a distributed system based on CORBA. The CORBA parts are shown in gray.

A serious problem with CORBA is that every object is located on only one server, which means the performance will be terrible for objects that are heavily used on client machines around the world. In practice, CORBA only functions acceptably in small-scale systems, such as to connect processes on one computer, one LAN or within a single company.

Globe

As an example of a distributed object system that was specifically designed to scale to a billion users and a trillion objects around the world, let us consider **Globe** (Van Steen et al., 1999a; Van Steen et al., 1999b). There are two key ideas to scaling to very large systems. The first is having replicated objects. If there is a single copy of a popular object that millions of users around the world want to access, the object will die under the weight of the requests. Think about an object that maintains stock prices or sports scores. Replicating this object allows the load to be spread over the replicas.

The second key idea is flexibility. In a worldwide system with a billion users, there is no way to get everyone to agree on one programming language, one replication strategy, one security model, or one anything else. The system has to allow different users and different objects to behave differently, while at the same time providing a coherent overall model. This is what Globe does.

Globe is also unusual in that, like DSM, it is based on the model of distributed shared memory, but now applied to a worldwide context. In principle, using normal page-based DSM on a worldwide system would work, only the performance

IBM Ex-1009
Page 431

would be horrible, so Globe takes a different approach. Conceptually, the basic idea is that the world is full of objects, each one containing some (hidden) internal state plus methods for accessing the internal state in controlled ways. The secret to making shared memory scalable worldwide is prohibiting direct LOADs and STOREs to an object's internal state and forcing all accesses to go through the methods. Because a Globe object can actively be shared by many processes at the same time, it is also called a **distributed shared object**. The positioning of systems like Globe is shown in Fig. 8-22(c).

Now let us see how scalability and flexibility are implemented. Every Globe object has a class object that contains the actual code for its methods. Every object also has one (or more) interfaces, each of which contains (method pointer, state pointer) pairs. Thus given an object interface, which is a table full of pointers present in memory at run time, a process can invoke the object's n-th method by making a call to the procedure pointed to by the n-th pair in the interface table and passing it the corresponding state pointer as a parameter. The state pointer is needed so that if there are, say, two objects of class mailbox in memory, each one has its own interface, with shared method pointers but private state pointers, as shown in Fig. 8-39. In this example, the process has two open mailboxes, each of which shares the code for the four mailbox methods, but each of which has its own private state (the messages stored in the mailbox instance). One mailbox might be for business mail and the other for personal mail, for example.

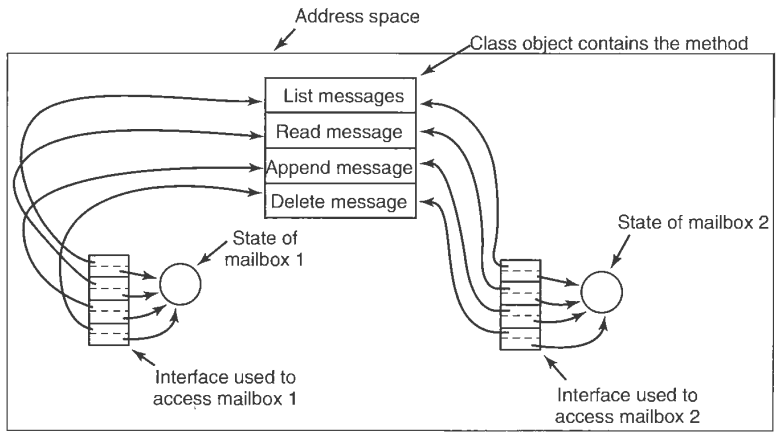


Figure 8-39. The structure of a Globe object.

The design of having interfaces be tables in memory at run time means that objects are not restricted to any particular language. This decision was made because a worldwide system will have many different people with many different

IBM Ex-1009
Page 432

favorite languages. An object's methods may be written in C, C++, Java, or even assembly language if the object's owner so desires. The interfaces are there to shield the process from what is behind the method pointers. This mix-and-match design is more flexible than a single-language design present in some systems (e.g., only Java or only C++).

To use a Globe object, a process must first bind to it by looking it up and finding at least one contact address (e.g., IP address and port). A security check is made at binding time, and if the process is authorized to bind to the object, the object's class object (i.e., its code) is loaded into the caller's address space, a copy of its state is instantiated and a pointer to its (standard) interface is returned. Using the interface pointer, the process can now invoke methods on this instance of the object. Depending on the object, the state may be the default state or a copy of the current state taken from one of the other live copies.

Imagine the simplest possible object. It has one integer as state and two methods: *read* and *write* that operate on the integer. If multiple processes in different countries are simultaneously bound to the object, all of them have an interface table pointing to the class object containing the two methods (which was loaded at bind time), as illustrated in Fig. 8-40. Each process (potentially) also has a copy of the integer comprising the state. Any *read* method is just invoked locally, but *write*s are more complicated. If the object wants to maintain sequential consistency, it must provide a mechanism for doing so.

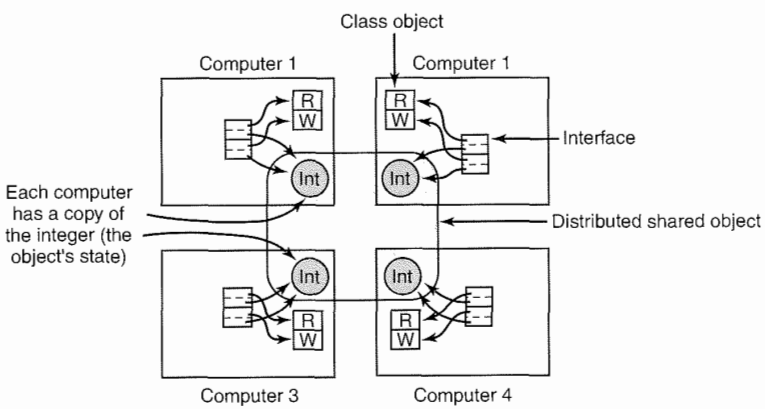


Figure 8-40. A distributed shared object can have its state copied on multiple computers at once.

One mechanism is to have a process called the **sequencer** whose job is to issue consecutive sequence numbers when requested to do so. To do a write, the *write* method might then first get a sequence number and then multicast a message

IBM Ex-1009
Page 433

containing the sequence number, operation name, and parameter to all the other processes bound to the object. If two processes invoked *write* simultaneously, they would be assigned different consecutive sequence numbers. All processes must apply incoming methods in sequence number order, not in message arrival order. If a process gets sequence number 26 and the previous one was 24, it must wait for 25 before applying 26. If 25 does not show up within a certain time, the process must take action to locate and get it. This scheme guarantees that all writes are done in the same order on all replicas of the object, ensuring sequential consistency.

Using this technique works reasonably well, but not all objects need sequential consistency. Consider, for example, an object maintaining stock prices. If the market maker for stock 1 issues an updated price for it concurrently with another market maker issuing an update for stock 2, it is not essential that all copies of the object apply those two updates in the same order because they are independent. It is probably sufficient that all processes apply the stream of updates from each market maker in the order they were sent, but this goal can be achieved by including a sequence number generated by the sending process. No object-wide sequencer is needed here.

The above replication scheme, namely a replicated object with all copies being equal and any copy being allowed to issue updates after first getting a sequence number is only one of many replication protocols. Another one has one master copy of each object, plus some number of slave copies. All updates are sent to the object's master copy, which then applies the update and sends out the new state to all the slave copies.

A third object replication strategy is having only one copy holding the object's state, with all the other copies being stateless proxies. When a *read* or *write* is done at a proxy (e.g., a client machine), the request is forwarded to the copy holding the state and executed there.

The strength of Globe is that each object can have its own replication policy. Some objects can use active replication at the same time other objects are using master-slave replication or any other strategy an object needs. Also, each object can have its own policy concerning consistency, replica creation and removal, security, etc. This is possible because all the policies are handled inside the object. Users of the object are not even aware of it, and neither are the system administrators. This approach is in contrast to CORBA, which does not hide any of these policies inside objects, making it difficult to have 1000 different objects with 1000 different policies.

A Globe object can be implemented as shown in Fig. 8-41. This figure illustrates the subobjects from which a Globe object is composed. The control object accepts incoming method invocations and uses the other subobjects to get them done. The semantics subobject actually does the work required by the object's interface. It is the only part of the object's code that must be written by the object's programmer; all the rest can be taken from standard libraries, unless the

IBM Ex-1009
Page 434

programmer wants a new strategy not currently available. The replication subobject's job is to manage replication. This module can be replaced to switch from active replication to master-slave replication or any other replication strategy without the rest of the object being affected. Similarly, the security subobject can be replaced to implement a new security policy (e.g., to switch from ACLs to capabilities) and the communication subobject can be replaced to change network protocols (e.g., from IP v4 to IP v6) without affecting the rest of the object.

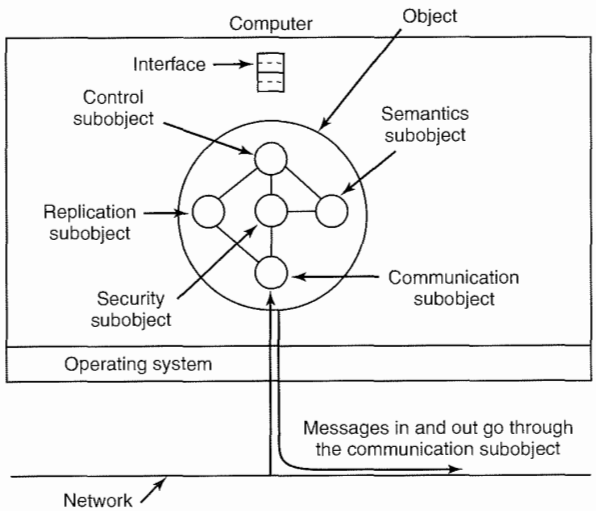


Figure 8-41. Structure of a Globe object.

To see how these subobjects interact, consider what happens when one of the object's methods is invoked. The code pointed to by the interface is in the control subobject, which then asks the replication subobject to do what it has to do. If the object is actively replicated, a sequence number is first acquired. Then the replication subobject tells all replicas (including its own) to actually do the work by invoking their semantics object. If the object is master-slave and the method invocation is on a slave, a message is sent to the master, and so on. At appropriate moments security checks are made by the security object (to see if the invocation is permitted, to see if outgoing data must be encrypted, etc.).

A key element of Globe is the **location service**, which allows objects to be looked up anywhere in the world. The location service is built as a tree, with object registrations being kept only in the node where the registration takes place. Pointers to this node are propagated up to the top of the tree so it is always possible to find the registration. Locality, partitioning of tree nodes, caching, and other techniques are used to make the scheme workable, even for mobile objects (Ballintijn et al., 2000; Van Steen et al., 1998a; and Van Steen et al., 1998b).

IBM Ex-1009
Page 435

8.3.6 Coordination-Based Middleware

Our last paradigm for a distributed system is called **coordination-based middleware**. We will start with the Linda system, an academic research project that started the whole field, and then look at two commercial examples heavily inspired by it: publish/subscribe and Jini.

Linda

Linda is a novel system for communication and synchronization developed at Yale University by David Gelernter and his student Nick Carriero (Carriero and Gelernter, 1986; Carriero and Gelernter, 1989; and Gelernter, 1985). In Linda, independent processes communicate via an abstract **tuple space**. The tuple space is global to the entire system, and processes on any machine can insert tuples into the tuple space or remove tuples from the tuple space without regard to how or where they are stored. To the user, the tuple space looks like a big, global shared memory, as we have seen in various forms before [and in Fig. 8-22(c)].

A **tuple** is like a structure in C or a record in Pascal. It consists of one or more fields, each of which is a value of some type supported by the base language (Linda is implemented by adding a library to an existing language, such as C). For C-Linda, field types include integers, long integers, and floating-point numbers, as well as composite types such as arrays (including strings) and structures (but not other tuples). Unlike objects, tuples are pure data; they do not have any associated methods. Figure 8-42 shows three tuples as examples.

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is-sister", "Stephany", "Roberta")
```

Figure 8-42. Three Linda tuples.

Four operations are provided on tuples. The first one, *out*, puts a tuple into the tuple space. For example,

```
out("abc", 2, 5);
```

puts the tuple ("abc", 2, 5) into the tuple space. The fields of *out* are normally constants, variables, or expressions, as in

```
out("matrix–1", i, j, 3.14);
```

which outputs a tuple with four fields, the second and third of which are determined by the current values of the variables i and j.

Tuples are retrieved from the tuple space by the *in* primitive. They are addressed by content rather than by name or address. The fields of *in* can be expressions or formal parameters. Consider, for example,

```
in("abc", 2, ?i);
```

IBM Ex-1009
Page 436

This operation "searches" the tuple space for a tuple consisting of the string "abc", the integer 2, and a third field containing any integer (assuming that *i* is an integer). If found, the tuple is removed from the tuple space and the variable *i* is assigned the value of the third field. The matching and removal are atomic, so if two processes execute the same *in* operation simultaneously, only one of them will succeed, unless two or more matching tuples are present. The tuple space may even contain multiple copies of the same tuple.

The matching algorithm used by *in* is straightforward. The fields of the *in* primitive, called the **template**, are (conceptually) compared to the corresponding fields of every tuple in the tuple space. A match occurs if the following three conditions are all met:

1. The template and the tuple have the same number of fields.
2. The types of the corresponding fields are equal.
3. Each constant or variable in the template matches its tuple field.

Formal parameters, indicated by a question mark followed by a variable name or type, do not participate in the matching (except for type checking), although those containing a variable name are assigned after a successful match.

If no matching tuple is present, the calling process is suspended until another process inserts the needed tuple, at which time the caller is automatically revived and given the new tuple. The fact that processes block and unblock automatically means that if one process is about to output a tuple and another is about to input it, it does not matter which goes first. The only difference is that if the *in* is done before the *out*, there will be a slight delay until the tuple is available for removal.

The fact that processes block when a needed tuple is not present can be put to many uses. For example, it can be used to implement semaphores. To create or do an up on semaphore *S*, a process can execute

```
out("semaphore S");
```

To do a down, it does

```
in("semaphore S");
```

The state of semaphore *S* is determined by the number of ("semaphore S") tuples in the tuple space. If none exist, any attempt to get one will block until some other process supplies one.

In addition to *out* and *in*, Linda also has a primitive *read*, which is the same as *in* except that it does not remove the tuple from the tuple space. There is also a primitive *eval*, which causes its parameters to be evaluated in parallel and the resulting tuple to be put in the tuple space. This mechanism can be used to perform an arbitrary computation. This is how parallel processes are created in Linda.

Publish/Subscribe

Our next example of a coordination-based model was inspired by Linda and is called **publish/subscribe** (Oki et al., 1993). It consists of a number of processes connected by a broadcast network. Each process can be a producer of information, a consumer of information, or both.

When an information producer has a new piece of information (e.g., a new stock price), it broadcasts the information as a tuple on the network. This action is called **publishing**. Each tuple contains a hierarchical subject line containing multiple fields separated by periods. Processes that are interested in certain information can **subscribe** to certain subjects, including the use of wildcards in the subject line. Subscription is done by telling a tuple daemon process on the same machine that monitors published tuples what subjects to look for.

Publish/subscribe is implemented as illustrated in Fig. 8-43. When a process has a tuple to publish, it broadcasts it out onto the local LAN. The tuple daemon on each machine copies all broadcasted tuples into its RAM. It then inspects the subject line to see which processes are interested in it, forwarding a copy to each one that is. Tuples can also be broadcast over a wide area network or the Internet by having one machine on each LAN act as an information router, collecting all published tuples and then forwarding them to other LANs for rebroadcasting. This forwarding can also be done intelligently, only forwarding a tuple to a remote LAN if that remote LAN has at least one subscriber who wants the tuple. Doing this requires having the information routers exchange information about subscribers.

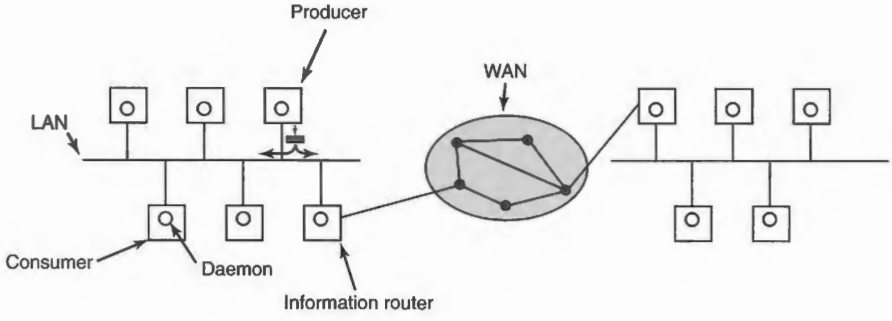


Figure 8-43. The publish/subscribe architecture.

Various kinds of semantics can be implemented, including reliable delivery and guaranteed delivery, even in the face of crashes. In the latter case, it is necessary to store old tuples in case they are needed later. One way to store them is to hook up a database system to the system and have it subscribe to all tuples. This can be done by wrapping the database system in an adapter, to allow an existing

IBM Ex-1009
Page 438

database to work with the publish/subscribe model. As tuples come by, the adapter captures all of them and puts them in the database.

The publish/subscribe model fully decouples producers from consumers, as does Linda. However, sometimes it is useful to know who else is out there. This information can be acquired by publishing a tuple that basically asks: "Who out there is interested in x?" Responses come back in the form of tuples that say: "I am interested in x."

Jini

For over 50 years, computing has been CPU-centric, with a computer being a freestanding device consisting of a CPU, some primary memory, and nearly always some mass storage such as a disk. Sun Microsystems' **Jini** (a variant spelling of genie) is an attempt to change that model to one that might be described as network-centric (Waldo, 1999).

The Jini world consists of a large number of self-contained Jini devices, each of which offers one or more services to the others. A Jini device can be plugged into a network and begin offering and using services instantly, with no complex installation procedure. Note that the devices are plugged into a *network*, not into a *computer* as is traditionally the case. A Jini device could be a traditional computer, but it could also be a printer, palmtop computer, cell phone, TV set, stereo, or other device with a CPU, some memory, and a (possibly wireless) network connection. A Jini system is a loose federation of Jini devices that may come and go at will, with no central administration.

When a Jini device wants to join the Jini federation, it broadcasts a packet on the local LAN or in the local wireless cell asking if there is a **lookup service** present. The protocol used to find a lookup service is the **discovery protocol** and is one of the few hardwired protocols in Jini. (Alternatively, the new Jini device can wait until one of the lookup service's periodic announcements comes by, but we will not treat this mechanism here.)

When the lookup service sees that a new device wants to register, it replies with a piece of code that can perform the registration. Since Jini is an all Java system, the code sent is in JVM (the Java Virtual Machine language), which all Jini devices must be capable of running, usually interpretively. The new device now runs the code, which contacts the lookup service and registers with it for some fixed period of time. Just before the time period expires, the device can reregister if it wishes. This mechanism means that a Jini device can just leave the system by shutting down and its previous existence will soon be forgotten, without the need for any central administration. The concept of registering for a fixed time interval is called acquiring a **lease**.

Note that since the code to register the device is downloaded into the device, it can be changed as the system evolves without affecting the hardware or software of the device. In fact, the device is not even aware of what the registration

IBM Ex-1009
Page 439

protocol is. A part of the registration process that the device is aware of consists of it providing some attributes and proxy code that other devices will later use to access it.

A device or user looking for a particular service can ask the lookup service if it knows about one. The request may involve some of the attributes that devices use when registering. If the request is successful, the proxy that the device provided at registration time is sent back to the requester and is run to contact the device. Thus a device or user can talk to another device without knowing where it is or even what protocol it speaks.

Jini clients and services (hardware or software devices) communicate and synchronize using **JavaSpaces**, which are modeled on the Linda tuple space but with some important differences. Each JavaSpace consists of some number of strongly typed entries. Entries are like Linda tuples, except that they are strongly typed, whereas Linda tuples are untyped. Each entry consists of some number of fields, each of which has a basic Java type. For example, an entry of type employee might consist of a string (for the name), an integer (for the department), a second integer (for the telephone extension), and a Boolean (for works-full-time).

Just four methods are defined on a JavaSpace (although two of them have a variant form):

1. **Write**: put a new entry into the JavaSpace.
2. **Read**: copy an entry that matches a template out of the JavaSpace.
3. **Take**: copy and remove an entry that matches a template.
4. **Notify**: notify the caller when a matching entry is written.

The *write* method provides the entry and specifies its lease time, that is when it should be discarded. In contrast, Linda tuples stay until removed. A JavaSpace may contain the same entry multiple times, so it is not a mathematical set (just as in Linda).

The *read* and *take* methods provide a template for the entry being sought. Each field in the template can contain a specific value that must be matched, or can contain a "don't care" wildcard that matches all values of the appropriate type. If a match is found, it is returned, and in the case of *take*, it is also removed from the JavaSpace. Each of these JavaSpace methods has two variants, which differ in the case that no entry matches. One variant returns with a failure indication immediately; the other one waits until a timeout (given as a parameter) has expired.

The *notify* method registers interest in a particular template. If a matching entry is later entered, the caller's *notify* method is invoked.

Unlike Linda's tuple space, JavaSpace supports atomic transactions. Using them, multiple methods can be grouped together. They will either all execute or

IBM Ex-1009
Page 440

none of them will execute. During the transaction, changes that are made to the JavaSpace are not visible outside the transaction. Only when the transaction commits, do they become visible to other callers.

JavaSpace can be used for synchronization between communicating processes. For example, in a producer-consumer situation, the producer puts items in a JavaSpace as it produces them. The consumer removes them with *take*, blocking if none are available. JavaSpace guarantees that each of the methods is executed atomically, so there is no danger of one process trying to read an entry that has only been half entered.

8.4 RESEARCH ON MULTIPLE PROCESSOR SYSTEMS

In this chapter we have looked at three kinds of multiple processor systems: multiprocessors, multicomputers, and distributed systems. Let us also look briefly at the research in these three areas. Most of the research on multiprocessors relates to the hardware, in particular, how to build the shared memory and keep it coherent. However, there has also been some research on using virtual machine monitors on multiprocessors (Bugnion et al., 1997) and on resource management on multiprocessors (Govil et al., 1999). Thread scheduling is also an issue in terms of the scheduling algorithm (Arora et al., 1998; and Philbin et al., 1996), and also in terms of contention for the run queue (Dandamudi, 1997).

Multicomputers are much easier to build than multiprocessors. All that is needed is a collection of PCs or workstations and a high-speed network. For this reason, they are a popular research topic at universities. A lot of the work relates to distributed shared memory in one form or another, sometimes page-based but sometimes entirely in software (Carter et al., 1995; Feeley et al., 1995; Johnson et al., 1995; Itzkovitz and Schuster, 1999; Scales and Gharachorloo, 1997; and Stets et al., 1997). Optimizing user-level communication is also a research topic (Von Eicken et al., 1995). So is load balancing (Harchol-Balter and Downey, 1996).

There are also many papers on distributed systems, for example on middleware (Bernstein, 1996), objects (Dogac et al., 1998), wireless systems (Liu et al., 1996), mobile agents (Chen et al., 2000), programming environments (Jo, 1999), distributed multimedia (Mourlas, 2000), theory (Buchs and Guelfi, 2000), and Web caching (Wolman et al., 1999), among others. Distributed file systems (Alexandrov et al., 1998; Hartman and Ousterhout, 1995; and Thekkath et al., 1997) and mobile file systems (Segarra and Andri, 1999) are also also popular.

8.5 SUMMARY

Computer systems can be made faster and more reliable by using multiple CPUs. Three organizations for multiCPU systems are multiprocessors, multicomputers, and distributed systems. Each of these has its own properties and issues.

IBM Ex-1009
Page 441

A multiprocessor consists of two or more CPUs that share a common RAM. The CPUs can be interconnected by a bus, a crossbar switch, or a multistage switching network. Various operating system configurations are possible, including giving each CPU its own operating system, having one master operating system with the rest being slaves, or having a symmetric multiprocessor, in which there is one copy of the operating system that any CPU can run. In the latter case, locks are needed to provide synchronization. When a lock is not available, a CPU can spin or do a context switch. Various scheduling algorithms are possible, including timesharing, space sharing, and gang scheduling.

Multicomputers also have two or more CPUs, but these CPUs each have their own private memory. They do not share any common RAM, so all communication uses message passing. In some cases, the network interface board has its own CPU, in which case the communication between the main CPU and the interface board CPU has to be carefully organized to avoid race conditions. User-level communication on multicomputers often uses remote procedure call, but distributed shared memory can also be used. Load balancing of processes is an issue here, and the various algorithms used for it include sender-initiated algorithms, receiver-initiated algorithms, and bidding algorithms.

Distributed systems are loosely coupled systems each of whose nodes is a complete computer with a complete set of peripherals and its own operating system. Often these systems are spread over a large geographical area. Middleware is often put on top of the operating system to provide a uniform layer for applications to interact with. The various kinds of middleware include document-based, file-based, object-based, and coordination-based middleware. Some examples are the World Wide Web, AFS, CORBA, Globe, Linda, and Jini.

PROBLEMS

1. Can the USENET newsgroup system or the SETI@home project be considered distributed systems? (SETI@home uses several million idle personal computers to analyze radiotelescope data to search for extraterrestrial intelligence). If so, how do they relate to the categories described in Fig. 8-1?
2. What happens if two CPUs in a multiprocessor attempt to access exactly the same word of memory at exactly the same instant?
3. If a CPU issues one memory request every instruction and the computer runs at 200 MIPS, about how many CPUs will it take to saturate a 400-MHz bus? Assume that a memory reference requires one bus cycle. Now repeat this problem for a system in which caching is used and the caches have a 90% hit rate. Finally, what cache hit rate would be needed to allow 32 CPUs to share the bus without overloading it?

IBM Ex-1009
Page 442

4. Suppose that the wire between switch 2A and switch 3B in the omega network of Fig. 8-5 breaks. Who is cut off from whom?

5. How is signal handling done in the model of Fig. 8-7?

6. When a system call is made in the model of Fig. 8-8, a problem has to be solved immediately after the trap that does not occur in the model of Fig. 8-7. What is the nature of this problem and how might it be solved?

7. Rewrite the enter_region code of Fig. 2-22 using the pure read to reduce thrashing induced by the TSL instruction.

8. Are critical regions on code sections really necessary in an SMP operating system to avoid race conditions or will mutexes on data structures do the job as well?

9. When the TSL instruction is used for multiprocessor synchronization, the cache block containing the mutex will get shuttled back and forth between the CPU holding the lock and the CPU requesting it if both of them keep touching the block. To reduce bus traffic, the requesting CPU executes one TSL every 50 bus cycles, but the CPU holding the lock always touches the cache block between TSL instructions. If a cache block consists of 16 32-bit words, each of which requires one bus cycle to transfer, and the bus runs at 400 MHz, what fraction of the bus bandwidth is eaten up by moving the cache block back and forth?

10. In the text, it was suggested that a binary exponential backoff algorithm be used between uses of TSL to poll a lock. It was also suggested to have a maximum delay between polls. Would the algorithm work correctly if there were no maximum delay?

11. Suppose that the TSL instruction was not available for synchronizing a multiprocessor. Instead, another instruction, SWP was provided that atomically swapped the contents of a register with a word in memory. Could that be used to provide multiprocessor synchronization? If so, how could it be used? If not, why does it not work?

12. In this problem you are to compute how much of a bus load a spin lock puts on the bus. Imagine that each instruction executed by a CPU takes 5 nsec. After an instruction has completed, any bus cycles needed, for example, for TSL are carried out. Each bus cycle takes an additional 10 nsec above and beyond the instruction execution time. If a process is attempting to enter a critical region using a TSL loop, what fraction of the bus bandwidth does it consume? Assume that normal caching is working so that fetching an instruction inside the loop consumes no bus cycles.

13. Fig. 8-12 was said to depict a timesharing environment. Why is only one process (A) shown in part (b)?

14. Affinity scheduling reduces cache misses. Does it also reduce TLB misses? What about page faults?

15. For each of the topologies of Fig. 8-16, what is the diameter of the interconnection network? Count all hops (host-router and router-router) equally for this problem.

16. Consider the double torus topology of Fig. 8-16(d) but expanded to size $k \times k$. What is the diameter of the network? *Hint*: Consider odd k and even k differently.

IBM Ex-1009
Page 443

17. The bisection bandwidth of an interconnection network is often used as a measure of its capacity. It is computed by removing a minimal number of links that splits the network into two equal-size units. The capacity of the removed links is then added up. If there are many ways to make the split, the one with the minimum bandwidth is the bisection bandwidth. For an interconnection network consisting of an $8 \times 8 \times 8$ cube, what is the bisection bandwidth if each link is 1 Gbps?

18. Consider a multicomputer in which the network interface is in user mode, so only three copies are needed from source RAM to destination RAM. Assume that moving a 32-bit word to or from the network interface board takes 20 nsec and that the network itself operates at 1 Gbps. What would the delay for a 64-byte packet being sent from source to destination be if we could ignore the copying time? What is it with the copying time? Now consider the case where two extra copies are needed, to the kernel on the sending side and from the kernel on the receiving side. What is the delay in this case?

19. Repeat the previous problem for both the three-copy case and the five-copy case but this time compute the bandwidth rather than the delay.

20. How must the implementation of send and receive differ between a shared memory multiprocessor system and a multicomputer, and how does this affect performance?

21. When transferring data from RAM to a network interface, pinning a page can be used, but suppose that system calls to pin and unpin pages each take 1 μsec. Copying takes 5 byte/nsec using DMA but 20 nsec per byte using programmed I/O. How big does a packet have to be before pinning the page and using DMA is worth it?

22. When a procedure is scooped up from one machine and placed on another to called by RPC, some problems can occur. In the text, we pointed out four of these: pointers, unknown array sizes, unknown parameter types, and global variables. An issue not discussed is what happens if the (remote) procedure executes a system call. What problems might that cause and what might be done to handle them?

23. In a DSM system, when a page fault occurs, the needed page has to be located. List two possible ways to find the page.

24. Consider the processor allocation of Fig. 8-25. Suppose that process *H* is moved from node 2 to node 3. What is the total weight of the external traffic now?

25. Some multicomputers allow running processes to be migrated from one node to another. Is it sufficient to stop a process, freeze its memory image, and just ship that off to a different node? Name two nontrivial problems that have to be solved to make this work.

26. Why is there a limit to cable length on an Ethernet network?

27. In Fig. 8-28, the third and fourth layers are labeled Middleware and Application on all four machines. In what sense are they all the same across platforms and in what sense are they different?

IBM Ex-1009
Page 444

28. Fig. 8-31 lists six different types of service. For each of the following applications, which service type is most appropriate?

(a) Video on demand over the Internet.
(b) Downloading a Web page.

29. DNS names have a hierarchical structure, such as *cs.uni.edu* or *sales.general-widget.com*. One way to maintain the DNS database would be as one centralized database, but that is not done because it would get too many requests/sec. Make a proposal how the DNS database could be maintained in practice.

30. In the discussion of how URLs are processed by a browser, it was stated that connections are made to port 80. Why?

31. Can the URLs used in the Web exhibit location transparency? Explain your answer.

32. When a browser fetches a Web page, it first makes a TCP connection to get the text on the page (in the HTML language). Then it closes the connection and examines the page. If there are figures or icons, it then makes a separate TCP connection to fetch each one. Suggest two alternative designs to improve performance here.

33. When session semantics are used, it is always true that changes to a file are immediately visible to the process making the change and never visible to processes on other machines. However, it is an open question as to whether or not they should be immediately visible to other processes on the same machine. Give an argument each way.

34. In AFS, whole files are cached on the client machines. Suppose that there is only so much disk space allocated for cached files and the allocation is full. When a new file is requested, what should be done? Give an algorithm for doing it.

35. When multiple processes need access to data, in what way is object-based access better than shared memory?

36. When a Linda *in* operation is done to locate a tuple, searching the entire tuple space linearly is very inefficient. Design a way to organize the tuple space that will speed up searches on all *in* operations.

37. Copying buffers takes time. Write a C program to find out how much time it takes on a system to which you have access. Use the *clock* or *times* functions to determine how long it takes to copy a large array. Test with different array sizes to separate copying time from overhead time.

38. Write C functions that could be used as client and server stubs to make an RPC call to the standard *printf* function, and a main program to test the functions. The client and server should communicate by means of a data structure that could be transmitted over a network. You may impose reasonable limits on the length of the format string and the number, types, and sizes of variables your client stub will accept.

39. Write two programs to simulate load balancing on a multicomputer. The first program should set up m processes distributed across n machines according to an initialization file. Each process should have running time chosen at random from a Gaussian

distribuion whose mean and standard deviation are parameters of the simulation. At the end of each run, the process creates some number of new processes, chosen from a Poisson distribution. When a process exits, the CPU must decide whether to give away processes or try to find new processes. The first program should use the sender-initiated algorithm to give away work if it has more than k processes total on its machine. The second program should use the receiver-initiated algorithm to fetch work when needed. Make any other reasonable assumptions you need, but state them clearly.

IBM Ex-1009
Page 446

11

CASE STUDY 2: WINDOWS 2000

Windows 2000 is a modern operating system that runs on high-end desktop PCs and servers. In this chapter we will examine various aspects of it, starting with a brief history, then moving on to its architecture. After this we will look at processes, memory management, I/O, the file system, and finally, security. One thing we will not look at is networking as that could easily fill a chapter (or an entire book) all by itself.

11.1 HISTORY OF WINDOWS 2000

Microsoft operating systems for desktop and laptop PCs can be divided into three families: MS-DOS, Consumer Windows (Windows 95/98/Me), and Windows NT. Below we will briefly sketch each of these families.

11.1.1 MS-DOS

In 1981, IBM, at the time the biggest and most powerful computer company in the world, produced the 8088-based IBM PC. The PC came equipped with a 16-bit real-mode, single-user, command-line oriented operating system called MS-DOS 1.0. The operating system was provided by Microsoft, a tiny startup, mostly known at that time for its BASIC interpreter used on 8080 and Z-80 systems. This operating system consisted of 8 KB of memory resident code and was

IBM Ex-1009
Page 447

closely modeled on CP/M, a tiny operating system for the 8-bit 8080 and Z80 CPUs. Two years later, a much more powerful 24-KB operating system, MS-DOS 2.0, was released. It contained a command line processor (shell), with a number of features borrowed from UNIX.

When Intel came out with the 286 chip, IBM built a new computer around it, the PC/AT, released in 1986. AT stood for Advanced Technology, because the 286 ran at a then-impressive 8 MHz and could address—with great difficulty—all of 16 MB of RAM. In practice, most systems had at most 1 MB or 2 MB, due to the great expense of so much memory. The PC/AT came equipped with Microsoft's MS-DOS 3.0, by now 36 KB. Over the years, MS-DOS continued to acquire new features, but it was still a command-line oriented system.

11.1.2 Windows 95/98/Me

Inspired by the user interface of the Apple Lisa, the forerunner to the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface (shell) that it called **Windows**. Windows 1.0, released in 1985, was something of a dud. Windows 2.0, designed for the PC-AT and released in 1987, was not much better. Finally, Windows 3.0 for the 386 (released in 1990), and especially its successors 3.1 and 3.11, caught on and were huge commercial successes. None of these early versions of Windows were true operating systems, but more like graphical user interfaces on top of MS-DOS, which was still in control of the machine and the file system. All programs ran in the same address space and a bug in any one of them could bring the whole system to a grinding halt.

The release of Windows 95 in August 1995 still did not completely eliminate MS-DOS, although it transferred nearly all the features from the MS-DOS part to the Windows part. Together, Windows 95 and the new MS-DOS 7.0 contained most of the features of a full-blown operating system, including virtual memory, process management, and multiprogramming. However, Windows 95 was not a full 32-bit program. It contained large chunks of old 16-bit assembly code (as well as some 32-bit code) and still used the MS-DOS file system, with nearly all its limitations. The only major change to the file system was the addition of long file names in place of the 8 + 3 character file names allowed in MS-DOS.

Even with the release of Windows 98 in June 1998, MS-DOS was still there (now called version 7.1) and running 16-bit code. Although yet more functionality migrated from the MS-DOS part to the Windows part, and a disk layout suitable for larger disks was now standard, under the hood, Windows 98 was not very different from Windows 95. The main difference was the user interface, which integrated the desktop and the Internet more closely. It was precisely this integration that attracted the attention of the U.S. Dept. of Justice, which then sued Microsoft claiming that it was an illegal monopoly, an accusation Microsoft vigorously denied. In April 2000, a U.S. Federal court agreed with the government.

IBM Ex-1009
Page 448

In addition to containing a large lump of old 16-bit assembly code in the kernel, Windows 98 had two other serious problems. First, although it was a multiprogramming system, the kernel itself was not reentrant. If a process was busy manipulating some kernel data structure and then suddenly its quantum ran out and another process started running, the new process might find the data structure in an inconsistent state. To prevent this type of problem, after entering the kernel, most processes first acquired a giant mutex covering the whole system before doing anything. While this approach eliminated potential inconsistencies, it also eliminated much of the value of multiprogramming since processes were frequently forced to wait for unrelated processes to leave the kernel before they could enter it.

Second, each Windows 98 process had a 4-GB virtual address space. Of this, 2 GB was completely private to the process. However, the next 1 GB was shared (writably) among all other processes in the system. The bottom 1 MB was also shared among all processes to allow all of them to access the MS-DOS interrupt vectors. This sharing facility was heavily used by most Windows 98 applications. As a consequence, a bug in one program could wipe out key data structures used by unrelated processes, leading to them all crashing. Worse yet, the last 1 GB was shared (writably) with the kernel and contained some critical kernel data structures. Any rogue program that overwrote these data structures with garbage could bring down the system. The obvious solution of not putting kernel data structures in user space was not possible because this feature was essential to making old MS-DOS programs work under Windows 98.

In the millennium year, 2000, Microsoft brought out a minor revision to Windows 98 called **Windows Me (Windows Millennium Edition)**. Although it fixed a few bugs and added a few features, under the covers it is essentially Windows 98. The new features included better ways to catalog and share images, music, and movies, more support for home networking and multiuser games, and more Internet-related features, such as support for instant messaging and broadband connections (cable modems and ADSL). One interesting new feature was the ability to restore the computer to its previous settings after a misconfiguration. If a user reconfigures the system (e.g., changing the screen from 640×480 to 1024×768) and it no longer works, this feature makes it possible to revert to the last known working configuration.

11.1.3 Windows NT

By the late 1980s, Microsoft realized that building a modern 32-bit operating system on top of the leaky 16-bit MS-DOS probably was not the best way to go. It recruited David Cutler, one of the key designers of DEC's VMS operating system, to work for Microsoft and gave him the job of leading a team to produce a brand-new 32-bit Windows compatible operating system from the ground up. This new

system, later called **Windows NT (Windows New Technology)**, was intended for mission-critical business applications as well as for home users. At the time, mainframes still ruled the (business) world, so designing an operating system on the assumption that companies would use personal computers for anything important was a visionary goal, but one that history has shown to be a very good one. Features such as security and high reliability, clearly lacking on the MS-DOS-based versions of Windows, were high on the agenda for (Windows) NT. Cutler's background with VMS clearly shows in various places, with there being more than a passing similarity between the design of NT and that of VMS.

The project succeeded and the first version, called Windows NT 3.1, was released in 1993. This initial release number was chosen to match the number of Microsoft's then-popular 16-bit Windows 3.1 system. Microsoft expected that NT would rapidly replace Windows 3.1 because it was technically a far superior system.

Much to its surprise, nearly all users preferred to stick with the old 16-bit system they knew, rather than upgrade to an unknown 32-bit system they did not know, however better it might be. Furthermore, NT required far more memory than Windows 3.1 and there were no 32-bit programs for it to run, so why bother? The failure of NT 3.1 to catch on in the marketplace was the reason Microsoft decided to build a 32-bit-ish version of Windows 3.1, namely Windows 95. The continued user resistance to NT then caused Microsoft to produce Windows 98 and finally Windows Me; each one claimed to be the very last release of the MS-DOS-based systems.

Despite the fact that nearly all consumers and most businesses ignored NT 3.1 for desktop systems, it did acquire a small following in the server market. A few new 3.x releases with small changes occurred in 1994 and 1995. These slowly began to acquire more following among desktop users as well.

The first major upgrade to NT came with NT 4.0 in 1996. This system had the power, security, and reliability of the new operating system, but also sported the same user interface as the by-then very popular Windows 95. This compatibility made it much easier for users to migrate from Windows 95 to NT, and many of them did so. Some of the differences between Windows 95/98 and Windows NT are summarized in Fig. 11-1.

From the beginning, NT was designed to be portable, so it was written almost entirely in C, with only a tiny bit of assembly code for low-level functions such as interrupt handling. The initial release consisted of 3.1 million lines of C for the operating system, libraries, and the environment subsystems (discussed below). When NT 4.0 came out, the code base had grown to 16 million lines of code, still mostly C, but with a small amount of C++ in the user interface part. By this time the system was highly portable, with versions running on the Pentium, Alpha, MIPS, and PowerPC, among other CPUs. Some of these have been dropped since then. The story of how NT was developed is given in the book *Showstopper* (Zachary, 1994). The book also tells a lot about the key people involved.

IBM Ex-1009
Page 450

Item	Windows 95/98	Windows NT
Full 32-bit system?	No	Yes
Security?	No	Yes
Protected file mappings?	No	Yes
Private addr space for each MS-DOS prog?	No	Yes
Unicode?	No	Yes
Runs on	Intel 80x86	80x86, Alpha, MIPS, ...
Multiprocessor support?	No	Yes
Re-entrant code inside OS?	No	Yes
Plug and play?	Yes	No
Power management?	Yes	No
FAT-32 file system?	Yes	Optional
NTFS file system	No	Yes
Win32 API?	Yes	Yes
Run all old MS-DOS programs?	Yes	No
Some critical OS data writable by user?	Yes	No

Figure 11-1. Some differences between Windows 98 and Windows NT.

11.1.4 Windows 2000

The release of NT following NT 4.0 was originally going to be called NT 5.0. However, in 1999, Microsoft changed the name to Windows 2000, mostly in an attempt to have a neutral name that both Windows 98 users and NT users could see as a logical next step for them. To the extent that this approach succeeds, Microsoft will have a single main operating system built on reliable 32-bit technology but using the popular Windows 98 user interface.

Since Windows 2000 really is NT 5.0, it inherits many properties from NT 4.0. It is a true 32-bit (soon to be 64-bit) multiprogramming system with individually protected processes. Each process has a private 32-bit (soon 64-bit) demand-paged virtual address space. The operating system runs in kernel mode, whereas user processes run in user mode, providing complete protection (with none of the protection flaws of Windows 98). Processes can have one or more threads, which are visible to, and scheduled by, the operating system. It has Dept. of Defense C2 security for all files, directories, processes, and other shareable objects (at least, if the floppy disk is removed and the network is unplugged). Finally, it also has full support for running on symmetric multiprocessors with up to 32 CPUs.

The fact that Windows 2000 really is NT 5.0 is visible in many places. For example, the system directory is called *\winnt* and the operating system binary (in *\winnt\system32*) is called *ntoskrnl.exe*. Right clicking on this file to examine its

properties shows that its version number is *5.xxx.yyy.zzz*, where the *5* stands for NT 5, *xxx* is the release number, *yyy* is the build (compilation) number, and *zzz* the minor variant. Also, many of the files in *\winnt* and its subdirectories have *nt* in their names, such as *ntvdm*, NT's virtual MS-DOS emulator.

Windows 2000 is more than just a better NT 4.0 with the Windows 98 user interface. To start with, it contains a number of other features previously found only in Windows 98. These include complete support for plug-and-play devices, the USB bus, IEEE 1394 (FireWire), IrDA (the infrared link between portable computers and printers), and power management, among others. In addition, a number of new features not present in any other Microsoft operating system have been added, including active directory directory service, security using Kerberos, support for smart cards, system monitoring tools, better integration of laptop computers with desktop computers, a system management infrastructure, and job objects. Also, the main file system, NTFS, has been extended to support encrypted files, quotas, linked files, mounted volumes, and content indexing, for example. Another novel NTFS feature is the single instance store, which is a kind of copy-on-write link in which two users can share a linked file until one of them writes on it, at which time a copy is made automatically.

One other major improvement is internationalization. NT 4.0 came in separate versions for different languages with the text strings embedded in the code. Installing an English software package on a Dutch computer often caused parts of the operating system to stop using Dutch and start using English because certain files containing code and text strings were overwritten. This problem has been eliminated. Windows 2000 has a single binary that runs everywhere in the world. An installation, or even an individual user, can choose the language to use at run time because all the menu items, dialog strings, error reports, and other text strings have been removed from the operating system and put in separate directories, one per installed language. Like all previous versions of NT, Windows 2000 uses Unicode throughout the system to support languages not using the Latin alphabet, such as Russian, Greek, Hebrew, and Japanese.

One thing that Windows 2000 does not have is MS-DOS. It is simply not there in any form (nor was it there in NT). There is a command line interface, but this is a new 32-bit program that includes the old MS-DOS functionality and considerably new functionality as well.

Despite many portability features with regard to the code, hardware, language, etc., in one respect Windows 2000 is less portable than NT 4.0: it runs on only two platforms, the Pentium and the Intel IA-64. Originally NT supported additional platforms, including the PowerPC, MIPS, and Alpha, but over the years, Microsoft dropped one after another for commercial reasons.

Like previous versions of NT, Windows 2000 comes in several product levels, this time: Professional, Server, Advanced server, and Datacenter server. The differences between all these versions are minor however, with the same executable binary used for all versions. When the system is installed, the product type is

IBM Ex-1009
Page 452

recorded in an internal database (the registry). At boot time, the operating system checks the registry to see which version it is. The differences are shown in Fig. 11-2.

Version	Max RAM	CPUs	Max clients	Cluster size	Optimized for
Professional	4 GB	2	10	0	Response time
Server	4 GB	4	Unlimited	0	Throughput
Advanced server	8 GB	8	Unlimited	2	Throughput
Datacenter server	64 GB	32	Unlimited	4	Throughput

Figure 11-2. The different versions of Windows 2000.

As can be seen from the figure, the differences include the maximum memory supported, the maximum number of CPUs (for a multiprocessor configuration), and the maximum number of clients that can be served. The cluster size relates to the ability of Windows 2000 to make two or four machines look like a single server to the outside world, a useful feature for Web servers, for example. Finally, the default parameters are tuned differently on Professional, to favor interactive programs over batch work, although these can easily be changed if desired. One last difference is that some extra software is provided on the servers and some extra tools are provided on Datacenter server for managing large jobs.

The reason for having multiple versions is simply marketing: this allows Microsoft to charge big companies more than they charge individuals for what is essentially the same product. This idea is not new, however, and hardly unique to Microsoft. For years, airlines having been charging business passengers much more, not only for Business Class, but also for Cattle Class if they want the luxury of buying the ticket a day before the flight instead of a month before the flight.

Technically, the way the version differences are maintained is that in a few places in the code, two variables are read from the registry, *ProductType* and *ProductSuite*. Depending on their values, slightly different code is executed. Changing these variables is in violation of the license. In addition, the system traps any attempt to change them and records the attempt at tampering in an indelible way so it can be detected later.

In addition to the basic operating system, Microsoft has also developed several tool kits for advanced users. These include the Support Tools, the Software Development Kit, the Driver Development Kit, and the Resource Kit. These include a large number of utilities and tools for tweaking and monitoring the system. The support tools are on the Windows 2000 CD-ROM in the directory *\support\tools*. The standard installation procedure does not install them, but there is a file *setup.exe* in that directory that does. The SDK and DDK are available to developers at *msdn.microsoft.com*. The Resource Kit is a Microsoft product in a box. There are also various third-party tools available for snooping on the Windows 2000 internals, including a nice set available for free at the Web site

IBM Ex-1009
Page 453

www.sysinternals.com. Some of these even provide more information than the corresponding Microsoft tools.

Windows 2000 is an immensely complex system, now consisting of over 29 million lines of C code. If printed 50 lines per page and 1000 pages per bound book, the full code would occupy 580 volumes. This opus would occupy 23 running meters of shelf space (for the paperback version). If arranged in bookcases 1 m wide with 6 shelves per bookcase, the set would occupy a wall 4 m wide.

Just for fun, a comparison of a few operating system source code sizes is given in Fig. 11-3. However, this table should be taken with a grain (or better yet, a metric ton) of salt because what constitutes the operating system is different for different systems. For example, the entire window system and GUI is part of the kernel in Windows, but not in any UNIX version. It is simply a user process there. Counting X Windows adds another 1.5 million lines of code to all the UNIX versions, and that does not even count the GUI code (Motif, GNOME, etc.), which is also not part of the operating system in the UNIX world. Additionally, some systems include code for multiple architectures (e.g., five for 4.4 BSD and nine for Linux), with each architecture adding 10,000 to 50,000 lines of code. The reason Free BSD 1.0 has only 235,000 lines of code whereas 4BSD Lite, from which it is derived, has 743,000 lines is that support for all the obsolete architectures (e.g., the VAX) was dropped in Free BSD.

Also, the number of file systems, devices drivers, and libraries supplied varies greatly from system to system. In addition, Windows contains large amounts of test code that UNIX does not contain as well as some utilities and support for numerous languages besides English. Finally, the measurements were made by different people, which introduces considerable variance (e.g., did makefiles, headers, configuration files and documentation count and how much was there?). This is not like comparing apples with oranges; it is like comparing apples with telephones. However, all the counts within a single family came from the same source, so intrafamily counts are somewhat meaningful.

Despite all these disclaimers, two conclusions are fairly clear:

1. System bloat seems to be as inevitable as death and taxes.
2. Windows is much bigger than UNIX

Whether small is beautiful or big is beautiful is a matter of heated controversy. The argument for the former is that small size and a lean-and-mean mentality produces a manageable, reliable system that users can understand. The argument for the latter is that many users want lots of features. In any event, it should also be clear that any students planning to write a full-blown, state-of-the-art operating system from scratch have their work cut out for them.

Although Windows 2000 is already the world heavyweight champion in terms of pure mass, it is still growing, with bugs being fixed and new features being added. The way Microsoft manages its development is worth noting. Hundreds

IBM Ex-1009
Page 454

Year	AT&T	BSD	MINIX	Linux	Solaris	Win NT
1976	V6 9K					
1979	V7 21K					
1980		4.1 38K				
1982	Sys III 58K					
1984		4.2 98K				
1986		4.3 179K				
1987	SVR3 92K		1.0 13K			
1989	SVR4 280K					
1991				0.01 10K		
1993		Free 1.0 235K			5.3 850K	3.1 6M
1994		4.4 Lite 743K		1.0 165K		3.5 10M
1996				2.0 470K		4.0 16M
1997			2.0 62K		5.6 1.4M	
1999				2.2 1M		
2000		Free 4.0 1.4M			5.8 2.0M	2000 29M

Figure 11-3. A comparison of some operating system sizes. The first string in each box is the version; the second is the size measured in lines of source code, where K = 1000 and M = 1,000,000. Comparisons within a column have real meaning; comparisons across columns do not, as discussed in the text.

of programmers work on various aspects of Windows 2000 all day. Whenever a piece of code is finished, the programmer submits it electronically to the build team. At 6 P.M. every day, the door is closed and the system is rebuilt (i.e., recompiled and linked). Each build gets a unique sequence number, which can be seen by examining the version number of *ntoskrnl.exe* (the first public release of Windows 2000 was build 2195).

The new operating system is electronically distributed to thousands of machines around the Microsoft campus in Redmond, WA, where it is subjected to intense stress tests all night. Early the next morning, the results of all the tests are sent to the relevant groups, so they can see if their new code works. Each team then decides which code they want to work on that day. During the day, the programmers work on their code and at 6 P.M. the build-and-test-cycle begins anew.

11.2 PROGRAMMING WINDOWS 2000

It is now time to start our technical study of Windows 2000. However, before getting into the details of the internal structure, we will first take a look at the programming interface and the registry, a small in-memory data base.

11.2.1 The Win32 Application Programming Interface

Like all other operating systems, Windows 2000 has a set of system calls it can perform. However, Microsoft has never made the list of Windows system calls public, and it also changes them from release to release. Instead, what Microsoft has done is define a set of function calls called the **Win32 API** (**Win32 Application Programming Interface**) that are publicly known and fully documented. These are library procedures that either make system calls to get the work done, or, in some cases, do the work right in user space. The existing Win32 API calls do not change with new releases of Windows, although new API calls are added frequently.

Binary programs for the Intel x86 that adhere exactly to the Win32 API interface will run unmodified on all versions of Windows since Windows 95. As shown in Fig. 11-4, an extra library is needed for Windows 3.x to match a subset of the 32-bit API calls to the 16-bit operating system, but for the other systems no adaptation is needed. It should be noted that Windows 2000 adds substantial new functionality to Win32, so it has additional API calls not included on older versions of Win32 and which will not work on older versions of Windows.

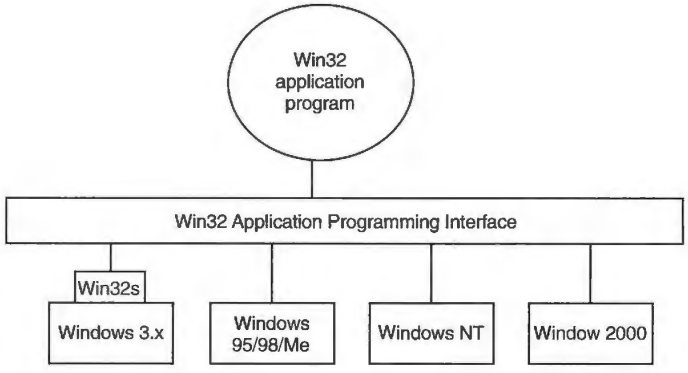


Figure 11-4. The Win32 API allows programs to run on almost all versions of Windows.

The Win32 API philosophy is completely different from the UNIX philosophy. In the latter, the system calls are all publicly known and form a minimal interface: removing even one of them would reduce the functionality of the operating system. The Win32 philosophy is to provide a very comprehensive interface, often with three or four ways of doing the same thing, and including many functions (i.e., procedures) that clearly should not be (and are not) system calls, such as an API call to copy an entire file.

Many Win32 API calls create kernel objects of one kind or another, including files, processes, threads, pipes, and so on. Every call creating an object returns a

IBM Ex-1009
Page 456

result called a **handle** to the caller. This handle can subsequently be used to perform operations on the object. Handles are specific to the process that created the object referred to by the handle. They cannot be passed directly to another process and used there (just as UNIX file descriptors cannot be passed to other processes and used there). However, under certain circumstances, it is possible to duplicate a handle and pass it to other processes in a protected way, allowing them controlled access to objects belonging to other processes. Every object also has a security descriptor associated with it, telling in detail who may and may not perform what kinds of operations on the object.

Not all system-created data structures are objects and not all objects are kernel objects. The only ones that are true kernel objects are those that need to be named, protected, or shared in some way. Every kernel object has a system-defined type, has well-defined operations on it, and occupies storage in kernel memory. Although users can perform the operations (by making Win32 calls), they cannot get at the data directly.

The operating system itself can also create and use objects and does so heavily. Most of these objects are created to allow one component of the system to store some information for a substantial period of time or to pass some data structure to another component. For example, when a device driver is loaded, an object is created for it holding its properties and pointers to the functions it contains. Within the operating system, the driver is then referred to by using its object.

Windows 2000 is sometimes said to be object-oriented because the only way to manipulate objects is by invoking operations on their handles by making Win32 API calls. On the other hand, it lacks some of the most basic properties of object-oriented systems such as inheritance and polymorphism.

The Win32 API calls cover every conceivable area an operating system could deal with, and quite a few it arguably should not deal with. Naturally, there are calls for creating and managing processes and threads. There are also many calls that relate to interprocess (actually, interthread) communication, such as creating, destroying, and using mutexes, semaphores, events, and other IPC objects.

Although much of the memory management system is invisible to programmers (fundamentally, it is just demand paging), one important feature is visible: namely the ability of a process to map a file onto a region of its virtual memory. This allows the process the ability to read and write parts of the file as though they were memory words.

An important area for many programs is file I/O. In the Win32 view, a file is just a linear sequence of bytes. Win32 provides over 60 calls for creating and destroying files and directories, opening and closing files, reading and writing them, requesting and setting file attributes, and much more.

Another area for which Win32 provides calls is security. Every process has an ID telling who it is and every object can have an access control list telling in great detail precisely which users may access it and which operations they may

perform on it. This approach provides for a fine-grained security in which specific individuals can be allowed or denied specific access to every object.

Processes, threads, synchronization, memory management, file I/O, and security system calls are nothing new. Other operating systems have them too, although generally not hundreds of them, as Win32 does. But what really distinguishes Win32 are the thousands upon thousands of calls for the graphical interface. There are calls for creating, destroying, managing and using windows, menus, tool bars, status bars, scroll bars, dialog boxes, icons, and many more items that appear on the screen. There are calls for drawing geometric figures, filling them in, managing the color palettes they use, dealing with fonts, and placing icons on the screen. Finally, there are calls for dealing with the keyboard, mouse and other input devices as well as audio, printing, and other output devices. In short, the Win32 API (especially the GUI part) is immense and we could not even begin to describe it in any detail in this chapter, so we will not try. Interested readers should consult one of the many books on Win32 (e.g., Petzold, 1999; Simon, 1997; and Rector and Newcomer, 1997).

Although the Win32 API is available on Windows 98 (as well as on the consumer electronics operating system, Windows CE), not every version of Windows implements every call and sometimes there are minor differences as well. For example, Windows 98 does not have any security, so those API calls that relate to security just return error codes on Windows 98. Also, Windows 2000 file names use the Unicode character set, which is not available on Windows 98 and Windows 98 file names are not case sensitive, whereas Windows 2000 file names are case sensitive (although some kinds of searches on file names are not case sensitive). There are also differences in parameters to some API function calls. On Windows 2000, for example, all the screen coordinates given in the graphics functions are true 32-bit numbers; on Windows 98, only the low-order 16 bits are used because much of the graphics subsystem is still 16-bit code. The existence of the Win32 API on several different operating systems makes it easier to port programs between them, but since these minor variations exist, some care must be taken to achieve portability.

11.2.2 The Registry

Windows needs to keep track of a great deal of information about hardware, software, and users. In Window 3.x, this information was stored in hundreds of *.ini* (initialization) files spread all over the disk. Starting with Windows 95, nearly all the information needed for booting and configuring the system and tailoring it to the current user was gathered in a big central database called the **registry**. In this section we will give an overview of the Windows 2000 registry.

To start with, it is worth noting that although many parts of Windows 2000 are complicated and messy, the registry is one of the worst, and the cryptic nomenclature does not make it any better. Fortunately, entire books have been

IBM Ex-1009
Page 458

written describing it (Born, 1998; Hipson, 2000; and Ivens, 1998). That said, the *idea* behind the registry is very simple. It consists of a collection of directories, each of which contains either subdirectories or entries. In this respect it is a kind of file system for very small files. It has directories and entries (the files).

The confusion starts with the fact that Microsoft calls a directory a **key**, which it definitely is not. Furthermore, all the top-level directories start with the string *HKEY*, which means handle to key. Subdirectories tend to have somewhat better chosen names, although not always.

At the bottom of the hierarchy are the entries, called **values**, which contain the information. Each value has three parts: a name, a type, and the data. The name is just a Unicode string, often default if the directory contains only one value. The type is one of 11 standard types. The most common ones are Unicode string, a list of Unicode strings, a 32-bit integer, an arbitrary length binary number, and a symbolic link to a directory or entry elsewhere in the registry. Symbolic names are completely analogous to symbolic links in file systems or shortcuts on the Windows desktop: they allow one entry to point to another entry or directory. A symbolic link can also be used as a key, meaning that something that appears to be a directory is just a pointer to a different directory.

At the top level, the Windows 2000 registry has six keys, called **root keys**, as listed in Fig. 11-5. Some interesting **subkeys** (subdirectories) are also shown here. To see this list on your system, use one of the registry editors, either *regedit* or *regedt32*, which unfortunately display different information and use different formats. They can also change registry values. Amateurs should not change keys or values on any system they plan to boot again. Just looking is safe, though. You have been warned.

The first key (i.e., directory), HKEY_LOCAL_MACHINE, is probably the most important as it contains all the information about the local system. It has five subkeys (i.e., subdirectories). The HARDWARE subkey contains many subkeys telling all about the hardware and which driver controls which piece of hardware. This information is constructed on the fly by the plug-and-play manager as the system boots. Unlike the other subkeys, it is not stored on disk.

The SAM (Security Account Manager) subkey contains the user names, groups, passwords, and other account and security information needed for logging in. The SECURITY subkey contains general security policy information, such as minimum length for passwords, how many failed login attempts are tolerated before the police are called, etc.

The SOFTWARE subkey is where software manufacturers store preferences etc. For example, if a user has Adobe *Acrobat*, *Photoshop* and *Premiere* installed, there will be a subkey Adobe here, and below that further subkeys for Acrobat, Photoshop, Premiere, and any other Adobe products. The entries in these subdirectories can store anything the Adobe programmers want to put there, generally system-wide properties such as the version and build number, how to uninstall the package, drivers to use, and so forth. The registry saves them the trouble of hav-

Key	Description
HKEY_LOCAL_MACHINE	Properties of the hardware and software
HARDWARE	Hardware description and mapping of hardware to drivers
SAM	Security and account information for users
SECURITY	System-wide security policies
SOFTWARE	Generic information about installed application programs
SYSTEM	Information for booting the system
HKEY_USERS	Information about the users; one subkey per user
USER-AST-ID	User AST's profile
AppEvents	Which sound to make when (incoming email/fax, error, etc.)
Console	Command prompt settings (colors, fonts, history, etc.)
Control Panel	Desktop appearance, screensaver, mouse sensitivity, etc.
Environment	Environment variables
Keyboard Layout	Which keyboard: 102-key US, AZERTY, Dvorak, etc.
Printers	Information about installed printers
Software	User preferences for Microsoft and third party software
HKEY_PERFORMANCE_DATA	Hundreds of counters monitoring system performance
HKEY_CLASSES_ROOT	Link to HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES
HKEY_CURRENT_CONFIG	Link to the current hardware profile
HKEY_CURRENT_USER	Link to the current user profile

Figure 11-5. The root keys registry keys and selected subkeys. The capitalization has no meaning but follows the Microsoft practice here.

ing to invent their own method for storing this information. User-specific information also goes in the registry, but under HKEY_USERS.

The SYSTEM subkey holds mostly information about booting the system, for example, the list of drivers that must be loaded. It also holds the list of services (daemons) that must be started after booting up and the configuration information for all of them.

The next top-level key is HKEY_USERS, which contains the profiles of all the users. All the user-specific preferences in a number of areas are stored here. When a user changes a preference using the control panel, for example, the desktop color scheme, the new settings are recorded here. In fact, many of the programs on the control panel do little more than collect user information and change the registry accordingly. Some of the subkeys under HKEY_USERS are shown in Fig. 11-5 and should need little additional comment. Some of the subkeys, such as Software, contain surprisingly large numbers of subkeys, even if no software packages are installed.

The next top-level key, HKEY_PERFORMANCE_DATA contains neither data read in from the disk nor data collected by the plug-and-play manager. Instead, it offers a window into the operating system. The system itself contains hundreds of counters for monitoring system performance. These counters are accessible via

this registry key. When a subkey is queried, a specified procedure is run to collect and return the information (possibly by reading one or more counters and combining them in some way). This key is not visible using *regedit* or *regedt32*. Instead one has to use the performance tools, such as *pfmon*, *perfmon*, and *pview*. There are many such tools, some on the Windows 2000 CD-ROM, some in the resource kits, and some from third parties.

The next three top-level keys do not actually exist. Each one is a symbolic link to some place elsewhere in the registry. The HKEY_CLASSES_ROOT key is the most interesting. It points to the directory that handles COM (Component Object Model) objects and also the associations between file extensions and programs. When a user double clicks on a file ending in, say, *.doc*, the program catching the mouse click looks here to see which program to run (probably *Microsoft Word*). The complete database of recognized extensions and which program each one is owned by is under this key.

The HKEY_CURRENT_CONFIG key links to the current hardware configuration. A user can construct multiple hardware configurations, for example by disabling various devices to see if they were the cause of strange system behavior. This key points to the current configuration. Similarly, HKEY_CURRENT_USER points to the current user so that user's preferences can be found quickly.

None of the last three keys really adds anything, since the information was available elsewhere anyway (although less conveniently accessible). Thus despite the fact that *regedit* and *regedt32* list five top-level keys, there are really only three top-level directories and one of those is not shown among the five displayed.

The registry is fully available to the Win32 programmer. There are calls to create and delete keys, look up values within keys, and more. Some of the more useful ones are listed in Fig. 11-6.

Win32 API function	Description
RegCreateKeyEx	Create a new registry key
RegDeleteKey	Delete a registry key
RegOpenKeyEx	Open a key to get a handle to it
RegEnumKeyEx	Enumerate the subkeys subordinate to the key of the handle
RegQueryValueEx	Look up the data for a value within a key

Figure 11-6. Some of the Win32 API calls for using the registry

When the system is turned off, most of the registry information (but not all, as discussed above) is stored on the disk in files called **hives**. Most of them are in *\winnt\system32\config*. Because their integrity is so critical to correct system functioning, when they are updated, backups are made automatically and writes are done using atomic transactions to prevent corruption in the event of system crash during the write. Loss of the registry requires reinstalling all software.

11.3 SYSTEM STRUCTURE

In the previous sections we examined Windows 2000 as seen by the programmer. Now we are going to look under the hood to see how the system is organized internally, what the various components do, and how they interact with each other and with user programs. Although there are many books on how to use Windows 2000, there are many fewer on how it works. Far and away the best place to look for additional information on this topic is *Inside Windows 2000*, 3rd ed. by Solomon and Russinovich (2000). Some of the material in this chapter is based on information in that book and information from the authors. Microsoft was also a key source.

11.3.1 Operating System Structure

Windows 2000 consists of two major pieces: the operating system itself, which runs in kernel mode, and the environment subsystems, which run in user mode. The kernel is a traditional kernel in the sense that it handles process management, memory management, file systems, and so on. The environment subsystems are somewhat unusual in that they are separate processes that help user programs carry out certain system functions. In the following sections we will examine each of these parts in turn.

One of NT's many improvements over Windows 3.x was its modular structure. It consisted of a moderately small kernel that ran in kernel mode, plus some server processes that ran in user mode. User processes interacted with the server processes using the client-server model: a client sent a request message to a server, and the server did the work and returned the result to the client via a second message. This modular structure made it easier to port it to several computers besides the Intel line, including the DEC Alpha, IBM PowerPC, and SGI MIPS. It also protected the kernel from bugs in the server code. However, for performance reasons, starting with NT 4.0, pretty much all of the operating system (e.g., system call handling and all of the screen graphics) was put back into kernel mode. This design was carried over to Windows 2000.

Nevertheless, there is still some structure in Windows 2000. It is divided into several layers, each one using the services of the ones beneath it. The structure is illustrated in Fig. 11-7. One of the layers is divided horizontally into many modules. Each module has some particular function and a well-defined interface to the other modules.

The lowest two software layers, the HAL and the kernel, are written in C and in assembly language and are partly machine dependent. The upper ones are written entirely in C and are almost entirely machine independent. The drivers are written in C, or in a few cases C++. Below we will first examine the various components of the system starting at the bottom and working our way up.

IBM Ex-1009
Page 462

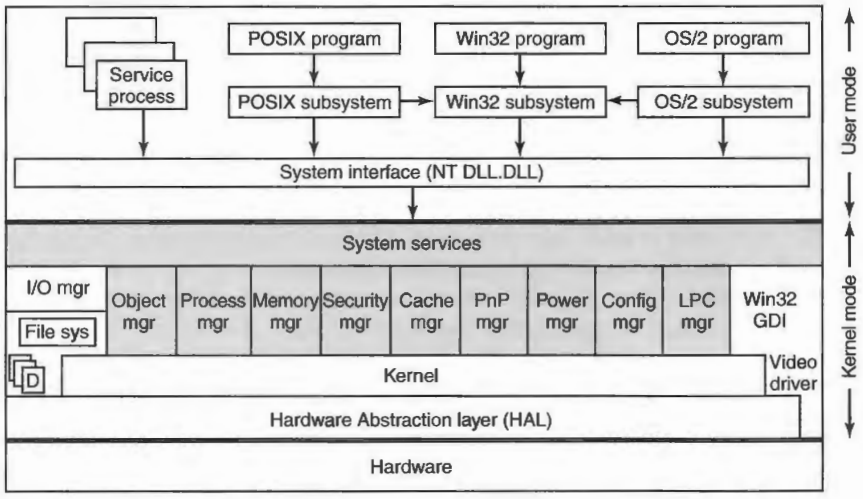


Figure 11-7. The structure of Windows 2000 (slightly simplified). The shaded area is the executive. The boxes indicated by D are device drivers. The service processes are system daemons.

The Hardware Abstraction Layer

One of the goals of Windows 2000 (and Windows NT before it) was to make the operating system portable across platforms. Ideally, when a new machine comes along, it should be possible to just recompile the operating system with the new machine's compiler and have it run the first time. Unfortunately, life is not like that. While the upper layers of the operating system can be made completely portable (because they mostly deal with internal data structures), the lower layers deal with device registers, interrupts, DMA, and other hardware features that differ appreciably from machine to machine. Even though most of the low-level code is written in C, it cannot just be scooped up from a Pentium, plopped down on, say, an Alpha, recompiled, and rebooted due to the many small hardware differences between the Pentium and the Alpha that have nothing to do with the different instruction sets and which cannot be hidden by the compiler.

Fully aware of this problem, Microsoft made a serious attempt to hide many of the machine dependencies in a thin layer at the bottom called the **HAL** (**Hardware Abstraction Layer**). (The name HAL was no doubt inspired by the computer HAL in the late Stanley Kubrick's movie *2001: A Space Odyssey*. Rumor has it that Kubrick chose the name "HAL" by taking the name of the then-dominant computer company—IBM— and subtracting 1 from each letter.)

The job of the HAL is to present the rest of the operating system with abstract hardware devices, in particular, devoid of the warts and idiosyncracies with which

real hardware is so richly endowed. These devices are presented in the form of machine-independent services (procedure calls and macros) that the rest of the operating system and the drivers can use. By using the HAL services (which are identical on all Windows 2000 systems, no matter what the hardware is) and not addressing the hardware directly, drivers and the kernel require fewer changes when being ported to new hardware. Porting the HAL itself is straightforward because all the machine-dependent code is concentrated in one place and the goals of the port are well defined, namely, implement all of the HAL services.

The services chosen for inclusion in the HAL are those that relate to the chip set on the parentboard and which vary from machine to machine within reasonably predictable limits. In other words, it is designed to hide the differences between one vendor's parentboard and another one's, but not the differences between a Pentium and an Alpha. The HAL services include access to the device registers, bus-independent device addressing, interrupt handling and resetting, DMA transfers, control of the timers and real-time clock, low-level spin locks and multiprocessor synchronization, interfacing with the BIOS and its CMOS configuration memory. The HAL does not provide abstractions or services for specific I/O devices such as keyboards, mice, or disks or for the memory management unit.

As an example of what the hardware abstraction layer does, consider the issue of memory-mapped I/O versus I/O ports. Some machines have one and some have the other. How should a driver be programmed: to use memory-mapped I/O or not? Rather than forcing a choice, which would make the driver not portable to a machine that did it the other way, the hardware abstraction layer offers three procedures for driver writers to use for reading the device registers and another three for writing them:

```
uc = READ_PORT_UCHAR(port);      WRITE_PORT_UCHAR(port, uc);
us = READ_PORT_USHORT(port);     WRITE_PORT_USHORT(port, us);
ul = READ_PORT_ULONG(port);      WRITE_PORT_LONG(port, ul);
```

These procedures read and write unsigned 8-, 16-, and 32-bit integers, respectively, to the specified port. It is up to the hardware abstraction layer to decide whether memory-mapped I/O is needed here. In this way, a driver can be moved without modification between machines that differ in the way the device registers are implemented.

Drivers often need to access specific I/O devices for various purposes. At the hardware level, a device has one or more addresses on a certain bus. Since modern computers often have multiple buses (ISA, PCI, SCSI, USB, 1394, etc.), it can happen that two or more devices have the same bus address, so some way is needed to distinguish them. The HAL provides a service for identifying devices by mapping bus-relative device addresses onto system-wide logical addresses. In this way, drivers are not required to keep track of which device is on which bus.

These logical addresses are analogous to the handles the operating system gives user programs to refer to files and other system resources. This mechanism also shields higher layers from properties of alternative bus structures and addressing conventions.

Interrupts have a similar problem—they are also bus dependent. Here, too, the HAL provides services to name interrupts in a system-wide way and also provides services to allow drivers to attach interrupt service routines to interrupts in a portable way, without having to know anything about which interrupt vector is for which bus. Interrupt request level management is also handled in the HAL.

Another HAL service is setting up and managing DMA transfers in a device-independent way. Both the system-wide DMA engine and DMA engines on specific I/O cards can be handled. Devices are referred to by their logical addresses. The HAL also implements software scatter/gather (writing or reading from noncontiguous blocks of physical memory).

The HAL also manages clocks and timers in a portable way. Time is kept track of in units of 100 nsec starting at 1 January 1601, which is far more precise than MS-DOS's keeping track of time in units of 2 sec since 1 January 1980 and provides support for the many computer-related activities in the 17th, 18th, and 19th centuries. The time services decouple the drivers from the actual frequencies at which the clocks run.

Kernel components sometimes need to synchronize at a very low level, especially to prevent race conditions in multiprocessor systems. The HAL provides some primitives to manage this synchronization, such as spin locks, in which one CPU simply waits for a resource held by another CPU to be released, particularly in situations where the resource is typically only held for a few machine instructions.

Finally, after the system has been booted, the HAL talks to the BIOS and inspects the CMOS configuration memory, if any, to find out which buses and I/O devices the system contains and how they have been configured. This information is then put into the registry so other system components can look it up without having to understand how the BIOS or configuration memory work. A summary of some of the things the HAL does is given in Fig. 11-8.

Since the HAL is highly-machine dependent, it must match the system it is installed on perfectly, so a variety of HALs are provided on the Windows 2000 CD-ROM. At system installation time, the appropriate one is selected and copied to the system directory *\winnt\system32* on the hard disk as *hal.dll*. All subsequent boots use this version of the HAL. Removing this file will make the system unbootable.

Although the HAL is reasonably efficient, for multimedia applications, it may not be fast enough. For this reason, Microsoft also produced a software package called **DirectX**, which augments the HAL with additional procedures and allows user processes much more direct access to the hardware. DirectX is somewhat specialized, so we will not discuss it further in this chapter.

IBM Ex-1009
Page 465

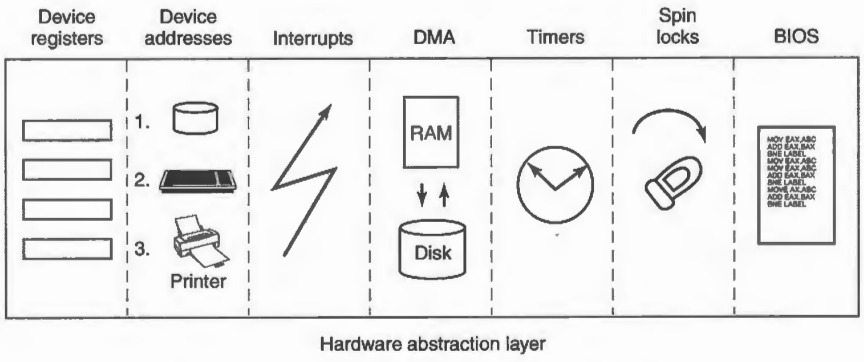


Figure 11-8. Some of the hardware functions the HAL manages.

The Kernel Layer

Above the hardware abstraction layer is a layer that contains what Microsoft calls the **kernel**, as well as the device drivers. Some early documentation refers to the kernel as the "microkernel," which it really never was because the memory manager, file system, and other major components resided in kernel space and ran in kernel mode from day one. The kernel is certainly not a microkernel now since virtually the entire operating system was put in kernel space starting with NT 4.0.

In the chapter on UNIX, we used the term "kernel" to mean everything running in kernel mode. In this chapter we will (very grudgingly) reserve the term "kernel" for the part labeled as such in Fig. 11-7 and call the totality of the code running in kernel mode the "operating system." Part of kernel (and much of the HAL) is permanently resident in main memory (i.e., is not paged). By adjusting its priority, it can control whether it can tolerate being preempted by I/O interrupts or not. Although a substantial fraction of the kernel is machine specific, most of it is nevertheless written in C, except where top performance overshadows all other concerns.

The purpose of the kernel is to make the rest of the operating system completely independent of the hardware, and thus highly portable. It picks up where the HAL leaves off. It accesses the hardware via the HAL and builds upon the extremely low-level HAL services to construct higher-level abstractions. For example, the HAL has calls to associate interrupt service procedures with interrupts, and set their priorities, but does little else in this area. The kernel, in contrast, provides a complete mechanism for doing context switches. It properly saves all the CPU registers, changes the page tables, flushes the CPU cache, and so on, so that when it is done, the previously running thread has been saved in tables in memory. It then sets up the new thread's memory map and loads its registers so the new thread can start running.

The code for thread scheduling is also in the kernel. When it is time to see if a new thread can run, for example, after a quantum runs out or after an I/O interrupt completes, the kernel chooses the thread and does the context switch necessary to run it. From the point of view of the rest of the operating system, thread switching is automatically handled by lower layers without any work on their part and in a portable way. The scheduling algorithm itself will be discussed later in this chapter when we come to processes and threads.

In addition to providing a higher-level abstraction of the hardware and handling thread switches, the kernel also has another key function: providing low-level support for two classes of objects: control objects and dispatcher objects. These objects are not the objects that user processes get handles to, but are internal objects upon which the executive builds the user objects.

Control objects are those objects that control the system, including primitive process objects, interrupt objects, and two somewhat strange objects called DPC and APC. A **DPC (Deferred Procedure Call)** object is used to split off the non-time-critical part of an interrupt service procedure from the time critical part. Generally, an interrupt service procedure saves a few volatile hardware registers associated with the interrupting I/O device so they do not get overwritten and reenables the hardware, but saves the bulk of the processing for later.

For example, after a key is struck, the keyboard interrupt service procedure reads the key code from a register and reenables the keyboard interrupt, but does not need to process the key immediately, especially if something more important (i.e., higher priority) is currently going on. As long as the key is processed within about 100 msec, the user will be none the wiser. DPCs are also used for timer expirations and other activities whose actual processing need not be instantaneous. The DPC queue is the mechanism for remembering that there is more work to do later.

Another kernel control object is the **APC (Asynchronous Procedure Call)**. APCs are like DPCs except that they execute in the context of a specific process. When processing a key press, it does not matter whose context the DPC runs in because all that is going to happen is that the key code will be inspected and probably put in a kernel buffer. However, if an interrupt requires copying a buffer from kernel space to a buffer in some user process' address space (e.g., as it may on completion of a read from the modem), then the copying procedure needs to run in the receiver's context. The receiver's context is needed so the page table will contain both the kernel buffer and the user buffer (all processes contain the entire kernel in their address spaces, as we will see later). For this reason, the kernel distinguishes between DPCs and APCs.

The other kind of kernel objects are **dispatcher objects**. These include semaphores, mutexes, events, waitable timers, and other objects that threads can wait on. The reason that these have to be handled (in part) in the kernel is that they are intimately intertwined with thread scheduling, which is a kernel task. As a little aside, mutexes are called "mutants" in the code because they were required to

implement the OS/2 semantics of not automatically unlocking themselves when a thread holding one exited, something the Windows 2000 designers considered bizarre. (The OS/2 semantics are relevant because NT was originally conceived of as a replacement for OS/2, the operating system shipped on IBM's PC/2.)

The Executive

Above the kernel and device drivers is the upper portion of the operating system, called the **executive**, shown as the shaded area in Fig. 11-7. The executive is written in C, is architecture independent, and can be ported to new machines with relatively little effort. It consists of 10 components, each of which is just a collection of procedures that work together to accomplish some goal. There are no hard boundaries between the pieces and different authors describing the executive might even group the procedures differently into components. It should be noted that components on the same level can (and do) call each other extensively.

The **object manager** manages all objects known to the operating system. These include processes, threads, files, directories, semaphores, I/O devices, timers, and many others. The object manager allocates a block of virtual memory from kernel address space when an object is created and returns it to the free list when the object is deallocated. Its job is to keep track of all the objects.

To avoid any confusion, most of the executive components labeled "manager" in Fig. 11-7 are not processes or threads, but merely collections of procedures that other threads can execute when in kernel mode. A few of them, such as the power manager and plug-and-play manager, really are independent threads though.

The object manager also manages a name space in which newly created objects may be placed so they can be referred to later. All other components of the executive use objects heavily to do their work. Objects are so central to the functioning of Windows 2000 that they will be discussed in detail in the next section.

The **I/O manager** provides a framework for managing I/O devices and provides generic I/O services. It provides the rest of the system with device-independent I/O, calling the appropriate driver to perform physical I/O. It is also home to all the device drivers (indicated by *D* in Fig. 11-7). The file systems are technically device drivers under control of the I/O manager. Two different ones are present for the FAT and NTFS file systems, each one independent of the others and controlling different disk partitions. All the FAT file systems are managed by a single driver. We will study I/O further in Sec. 11.6 and one of the file systems, NTFS, in Sec. 11.7.

The **process manager** handles processes and threads, including their creation and termination. It deals with the mechanisms used to manage them, rather than policies about how they are used. It builds upon the kernel process and thread objects and adds extra functionality to them. It is the key to multiprogramming in Windows 2000. We will study process and thread management in Sec. 11.4.

IBM Ex-1009
Page 468

The **memory manager** implements Windows 2000's demand-paged virtual memory architecture. It manages the mapping of virtual pages onto physical page frames. It thereby enforces the protection rules that restrict each process to only access those pages belonging to its address space and not to other processes' address spaces (except under special circumstances). It also handles certain system calls that relate to virtual memory. We will study memory management in Sec. 11.5.

The **security manager** enforces Windows 2000's elaborate security mechanism, which meets the U.S. Dept. of Defense's Orange Book C2 requirements. The Orange Book specifies a large number of rules that a conforming system must meet, starting with authenticated login through how access control is handled, to the fact that virtual pages must be zeroed out before being reused. We will study the security manager in Sec. 11.8.

The **cache manager** keeps the most recently used disk blocks in memory to speed up access to them in the (likely) event that they are needed again. Its job is to figure out which blocks are probably going to be needed again and which ones are not. It is possible to configure Windows 2000 with multiple file systems, in which case the cache manager works for all of them, so each one does not have to do its own cache management. When a block is needed, the cache manager is asked to supply it. If it does not have the block, the cache manager calls upon the appropriate file system to get it. Since files can be mapped into processes' address spaces, the cache manager must interact with the virtual memory manager to provide the necessary consistency. The amount of space devoted to caching is dynamic and can increase or decrease as demands on it change. We will study the cache manager in Sec. 11.9.

The **plug-and-play manager** is sent all notifications of newly attached devices. For some devices, a check is made at boot time and not thereafter. Other devices, for example, USB devices, can be attached at any time and their attachment triggers a message to the plug-and-play manager, which then locates and loads the appropriate driver.

The **power manager** rides herd on power usage. This consists of turning off the monitor and disks after they have been idle for a while. On laptops, the power manager monitors battery usage and takes action when the battery is about to run dry. Such action typically tells programs to save their files and prepare for a graceful shutdown. And pronto.

The **configuration manager** is in charge of the registry. It adds new entries and looks up keys when asked to.

The **local procedure call manager** provides for a highly-efficient interprocess communication used between processes and their subsystems. Since this path is needed to carry out some system calls, efficiency is critical here, which is why the standard interprocess communication mechanisms are not used.

The Win32 GDI executive module handles certain system calls (but not all of them). It was originally in user space but was moved to kernel space in NT 4.0 to

IBM Ex-1009
Page 469

improve performance. The **GDI (Graphics Device Interface)** handles image management for the monitor and printers. It provides system calls to allow user programs to write on the monitor and printers in a device-independent way. It also contains the window manager and display driver. Prior to NT 4.0, it, too, was in user space but the performance was disappointing, so Microsoft moved it into the kernel to speed it up. It is worth mentioning that Fig. 11-7 is not at all to scale. For example, the Win32 and graphics device interface module is larger than the rest of the executive combined.

At the top of the executive is a thin layer called **system services**. Its function is to provide an interface to the executive. It accepts the true Windows 2000 system calls and calls other parts of the executive to have them executed.

At boot time, Windows 2000 is loaded into memory as a collection of files. The main part of the operating system, consisting of the kernel and executive, is located in the file *ntoskrnl.exe*. The HAL is a shared library located in a separate file, *hal.dll*. The Win32 and graphics device interface are together in a third file, *win32k.sys*. Finally, many device drivers are also loaded. Most of these have extension *.sys*.

Actually, things are not quite that simple. The *ntoskrnl.exe* file comes in uniprocessor and multiprocessor versions. Also, there are versions for the Xeon processor, which can have more than 4 GB of physical memory and the Pentium, which cannot. Finally, versions can consist of a free build (sold in stores and preinstalled by computer manufacturers) or a checked build (for debugging purposes). Together there could be eight combinations, although two pairs were combined leaving only six. One of these is copied to *ntoskrnl.exe* when the system is installed.

The checked builds are worth a few words. When a new I/O device is installed on a PC, there is invariably a manufacturer-supplied driver that has to be installed to make it work. Suppose that an IEEE 1394 card is installed on a computer and appears to work fine. Two weeks later the system suddenly crashes. Who does the owner blame? Microsoft.

The bug may indeed be Microsoft's, but some bugs are actually due to flakey drivers, over which Microsoft has no control and which are installed in kernel memory and have full access to all kernel tables as well as the entire hardware. In an attempt to reduce the number of irate customers on the phone, Microsoft tries to help driver writers debug their code by putting statements of the form

```
ASSERT(some condition)
```

throughout the code. These statements make sanity checks on all parameters to internal kernel procedures (which may be freely called by drivers) and make many other checks as well. The free builds have *ASSERT* defined as a macro that does nothing, removing all the checks. The checked builds have it defined as

```
#define ASSERT(a) if (! (a)) error( ... )
```

causing all the checks to appear in the *ntoskrnl.exe* executable code and be carried out at run time. While this slows down the system enormously, it helps driver writers debug their drivers before they ship them to customers. The checked builds also have numerous other debugging features turned on.

The Device Drivers

The last part of Fig. 11-7 consists of the **device drivers**. Each device driver can control one or more I/O devices, but a device driver can also do things not related to a specific device, such as encrypting a data stream or even just providing access to kernel data structures. Device drivers are not part of the *ntoskrnl.exe* binary. The advantage of this approach is that once a driver has been installed on a system, it is added to a list in the registry and is loaded dynamically when the system boots. In this way, *ntoskrnl.exe* is the same for everyone, but every system is configured precisely for those devices it contains.

There are device drivers for macroscopically visible I/O devices such as disks and printers, but also for many internal devices and chips that practically no one has ever heard of. In addition, the file systems are also present as device drivers, as mentioned above. The largest device driver, the one for Win32, GDI, and video, is shown on the far right of Fig. 11-7. It handles many system calls and most of the graphics. Since customers can install new device drivers, they have the power to affect the kernel and corrupt the system. For this reason, drivers must be written with great care.

11.3.2 Implementation of Objects

Objects are probably the single most important concept in Windows 2000. They provide a uniform and consistent interface to all system resources and data structures such as processes, threads, semaphores, etc. This uniformity has various facets. First, all objects are named and accessed the same way, using object handles. Second, because all accesses to objects go through the object manager, it is possible to put all the security checks in one place and ensure that no process can make an end run around them. Third, sharing of objects among processes can be handled in a uniform way. Fourth, since all object opens and closes go through the object manager, it is easy to keep track of which objects are still in use and which can be safely deleted. Fifth, this uniform model for object management makes it easy to manage resource quotas in a straightforward way.

A key to understanding objects is to realize that an (executive) object is just some number of consecutive words in memory (i.e., in kernel virtual address space). An object is a data structure in RAM, no more and no less. A file on disk is not an object, although an object (i.e., a data structure in kernel virtual address space) is created for a file when it is opened. A consequence of the fact that objects are just kernel data structures is that when the system is rebooted (or

IBM Ex-1009
Page 471

crashes) all objects are lost. In fact, when the system boots, there are no objects present at all (except for the idle and system processes, whose objects are hardwired into the *ntoskrnl.exe* file). All other objects are created on the fly as the system boots up and various initialization (and later user) programs run.

Objects have a structure, as shown in Fig. 11-9. Each object contains a header with certain information common to all objects of all types. The fields in this header include the object's name, the object directory in which it lives in object space, security information (so a check can be made when an object is opened), and a list of processes with open handles to the object (if a certain debugging flag is enabled).

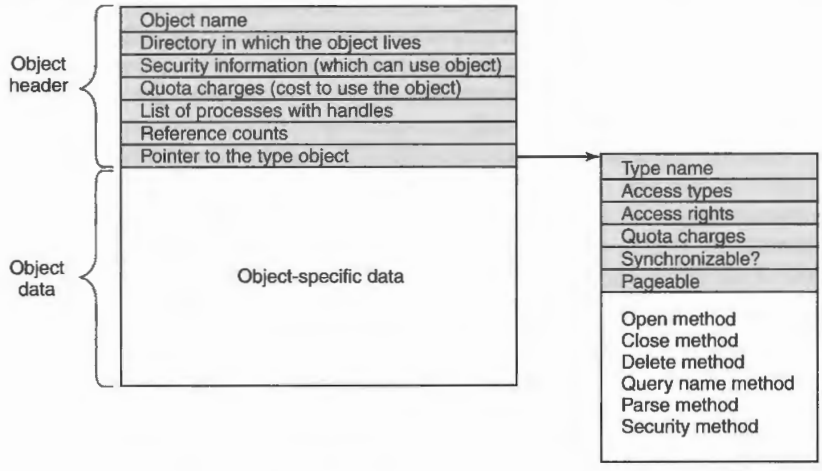


Figure 11-9. The structure of an object.

Each object header also contains a quota charge field, which is the charge levied against a process for opening the object. If a file object costs 1 point and a process belongs to a job that has 10 file points worth of quota, the processes in that job can only open 10 files in total. In this way resource limits can be enforced for each object type separately.

Objects occupy valuable real estate—pieces of kernel virtual address space—so when an object is no longer needed it should be removed and its address space reclaimed. The mechanism for reclamation is to have a reference counter in each object's header. It counts the number of open handles held by processes. This counter is incremented every time the object is opened and decremented every time it is closed. When it hits 0, no more users hold handles to the object. When an object is acquired or released by an executive component, a second counter is incremented or decremented, even though no actual handle is issued. When both counters hit 0, no user process is using the object and no executive process is using the object, so the object can be removed and its memory freed.

IBM Ex-1009
Page 472

The object manager needs to maintain dynamic data structures (its objects), but it is not the only part of the executive with this need. Other pieces also need to allocate and release chunks of kernel memory dynamically. To meet these needs, the executive maintains two page pools in kernel address space: for objects and for other dynamic data structures. Such pools operate as heaps, similar to the C language calls *malloc* and *free* for managing dynamic data. One pool is paged and the other is nonpaged (pinned in memory). Objects that are needed often are kept in the nonpaged pool; objects that are rarely accessed, such as registry keys and some security information, are kept in the paged pool. When memory is tight, the latter can be paged out and faulted back on demand. In fact, substantial portions of the operating system code and data structures are also pageable, to reduce memory consumption. Objects that may be needed when the system is running critical code (and when paging is not permitted) must go in the nonpaged pool. When a small amount of storage is needed, a page can be taken from either pool and then broken up into units as small as 8 bytes.

Objects are typed, which means each one has certain properties common to all objects of its type. The type is indicated by a pointer in the header to a type object, as shown in Fig. 11-9. The type object information includes items such as the type name, whether a thread can wait on the object (yes for mutexes, no for open files), and whether new objects of this type go on the paged or nonpaged pool. Each object points to its type object.

The last thing a type object has is also the most important: pointers to the code for certain standard operations such as open, close, and delete. Whenever one of these operations is invoked on an object, the pointer to the type object is followed and the relevant code located and executed. This mechanism gives the system the opportunity to initialize new objects, and recover storage when they are deleted.

Executive components can create new types dynamically. There is no definitive list of object types, but some of the more common ones are listed in Fig. 11-10. Let us briefly go over the object types in Fig. 11-10. Process and thread are obvious. There is one object for every process and every thread, which holds the main properties needed to manage the process or thread. The next three objects, semaphore, mutex, and event, all deal with interprocess synchronization. Semaphores and mutexes work as expected, but with various extra bells and whistles (e.g., maximum values and timeouts). Events can be in one of two states: signaled or nonsignaled. If a thread waits on an event that is in signaled state, the thread is released immediately. If the event is in nonsignaled state, it blocks until some other thread signals the event, which releases all blocked threads. An event can also be set up, so after a signal has been successfully waited for, it automatically reverts to nonsignaled state, rather than staying in signaled state.

Port, timer, and queue objects also relate to communication and synchronization. Ports are channels between processes for exchanging messages. Timers provide a way to block for a specific time interval. Queues are used to notify threads that a previously started asynchronous I/O operation has completed.

Type	Description
Process	User process
Thread	Thread within a process
Semaphore	Counting semaphore used for interprocess synchronization
Mutex	Binary semaphore used to enter a critical region
Event	Synchronization object with persistent state (signaled/not)
Port	Mechanism for interprocess message passing
Timer	Object allowing a thread to sleep for a fixed time interval
Queue	Object used for completion notification on asynchronous I/O
Open file	Object associated with an open file
Access token	Security descriptor for some object
Profile	Data structure used for profiling CPU usage
Section	Structure used for mapping files onto virtual address space
Key	Registry key
Object directory	Directory for grouping objects within the object manager
Symbolic link	Pointer to another object by name
Device	I/O device object
Device driver	Each loaded device driver has its own object

Figure 11-10. Some common executive object types managed by object manager.

Open file objects are created when a file is opened. Files that are not opened do not have objects managed by the object manager. Access tokens are security objects; they identify a user and tell what special privileges the user has, if any. Profiles are structures used for storing periodic samples of the program counter of a running thread to see where the program is spending its time.

Sections are objects used by the memory system for handling memory-mapped files. They record which file (or part thereof) is mapped onto which memory addresses. Keys are registry keys and are used to relate names to values. Object directories are entirely local to the object manager. They provide a way to collect related objects together in exactly the same way directories work in the file system. Symbolic links are also similar to their file system counterparts: they allow a name in one part of the object name space to refer to an object in a different part of the object name space. Each known device has a device object that contains information about it and is used to refer to the device within the system. Finally, each device driver that has been loaded has an object in the object space.

Users can create new objects or open existing objects by making Win32 calls such as CreateSemaphore or OpenSemaphore. These are calls to library procedures that ultimately result in the appropriate system calls being made. The result of any successful call that creates or opens an object is a 64-bit handle table

entry that is stored in the process' private handle table in kernel memory. The 32-bit index of the handle's position in the table is returned to the user to use on subsequent calls.

The 64-bit handle table entry in the kernel contains two 32-bit words. One word contains a 29-bit pointer to the object's header. The low-order 3 bits are used as flags (e.g., whether the handle is inherited by child processes). These bits are masked off before the pointer is followed. The other word contains a 32-bit rights mask. It is needed because permissions checking is done only at the time the object is created or opened. If a process has only read permission to an object, all the other rights bits in the mask will be 0s, giving the operating system the ability to reject any operation on the object other than reading it.

The handle tables for two processes and their relationships to some objects are illustrated in Fig. 11-11. In this example, process *A* has access to threads 1 and 2 and access to mutexes 1 and 2. Process *B* has access to thread 3 and mutexes 2 and 3. The corresponding entries in the handle tables hold the rights to each of these objects. For example, process *A* might have the rights to lock and unlock its mutexes, but not the right to destroy them. Note that mutex 2 is shared by both processes allowing threads in them to synchronize. The other mutexes are not shared, which might mean that the threads within process *A* use mutex 1 for their internal synchronization and the threads within process *B* use mutex 3 for their internal synchronization.

The Object Name Space

As objects are created and deleted during execution, the object manager needs a way to keep track of them. To do this job, it maintains a name space, in which all objects in the system are located. The name space can be used by a process to locate and open a handle for some other process' object, provided it has been granted permission to do so. The object name space is one of three name spaces maintained by Windows 2000. The other ones are the file system name space and the registry name space. All three are hierarchical name spaces with multiple levels of directories for organizing entries. The directory objects listed in Fig. 11-10 provide the means to implement this hierarchical name space for objects.

Since executive objects are volatile (i.e., vanish when the computer is shut down, unlike file system and registry entries), when the system boots up, there are no objects in memory and the object name space is empty. During booting, various parts of the executive create directories and then fill them with objects. For example, as the plug-and-play manager discovers devices out there, it creates a device object for each one and enters this object into the name space. When the system is fully booted, all I/O devices, disk partitions, and other interesting discoveries are in the object name space.

Not all objects get entered by the Columbus method—just go look and see what you find. Some executive components look in the registry to see what to do. A key example here is device drivers. During bootup, the system looks in the

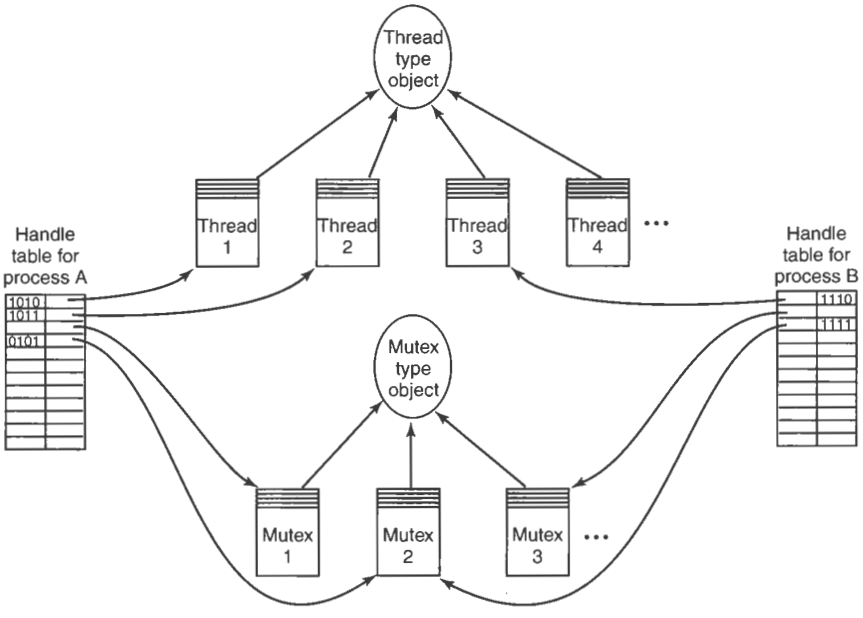


Figure 11-11. The relationship between handle tables, objects, and type objects.

registry to see which device drivers are needed. As they are loaded one by one, an object is created for each one and its name is inserted into the object space. Within the system, the driver is referred to by a pointer to its object.

Although the object name space is crucial to the entire operation of the system, few people know that it even exists because it is not visible to users without special viewing tools. One such viewing tool is *winobj*, available for free at *www.sysinternals.com*. When run, this tool depicts an object name space that typically contains the object directories listed in Fig. 11-12 as well as a few others.

The somewhat strangely named directory *\??* contains the names of all the MS-DOS-style device names, such as *A:* for the floppy disk and *C:* for the first hard disk. These names are actually symbolic links to the directory *\Device* where the device objects live. The name *\??* was chosen to make it alphabetically first to speed up lookup of all path names beginning with a drive letter. The contents of the other object directories should be self explanatory.

11.3.3 Environment Subsystems

Going back to Fig. 11-7, we see that Windows 2000 consists of components in kernel mode and components in user mode. We have now completed our examination of the kernel mode components; now it is time to look at the user mode

IBM Ex-1009
Page 476

Directory	Contents
??	Starting place for looking up MS-DOS devices like C:
Device	All discovered I/O devices
Driver	Objects corresponding to each loaded device driver
ObjectTypes	The type objects shown in Fig. 11-11
Windows	Objects for sending messages to all the windows
BaseNamedObjs	User-created objects such as semaphores, mutexes, etc.
Arcname	Partition names discovered by the boot loader
NLS	National language support objects
FileSystem	File system driver objects and file system recognizer objects
Security	Objects belonging to the security system
KnownDLLs	Key shared libraries that are opened early and held open

Figure 11-12. Some typical directories in the object name space.

components of which there are three kinds: DLLs, environment subsystems, and service processes. These components work together to provide each user process with an interface that is distinct from the Windows 2000 system call interface.

Windows 2000 supports three different documented APIs: Win32, POSIX, and OS/2. Each of these interfaces has a published list of library calls that programmers can use. The job of the DLLs (Dynamic Link Libraries) and environment subsystems is to implement the functionality of the published interface, thereby hiding the true system call interface from application programs. In particular, the Win32 interface is the official interface for Windows 2000, Windows NT, Windows 95/98/Me, and to a limited extent, Windows CE. By using the DLLs and Win32 environment subsystem, a program can be written to the Win32 specification and run unmodified on all these versions of Windows, even though the system calls are not the same on the various systems.

To see how these interfaces are implemented, let us look at Win32. A Win32 program normally contains many calls to Win32 API functions, for example, CreateWindow, DrawMenuBar, and OpenSemaphore. There are thousands of such calls, and most programs use a substantial number of them. One possible implementation would be to statically link every Win32 program with all the library procedures that it uses. If this were done, each binary program would contain one copy of each procedure it used in its executable binary.

The trouble with this approach is that it wastes memory if the user has multiple programs open at once and they use many of the same library procedures. For example, *Word*, *Excel*, and *Powerpoint* all use exactly the same procedures for opening dialog boxes, drawing windows, displaying menus, managing the clipboard, etc., so if a user had all of them open and active at once, there would be three (identical) copies of each of the libraries in memory.

IBM Ex-1009
Page 477

To avoid this problem, all versions of Windows support shared libraries, called **DLLs (Dynamic Link Libraries)**. Each DLL collects together a set of closely related library procedures and their data structures into a single file, usually (but not always) with extension *.dll*. When an application is linked, the linker sees that some of the library procedures belong to DLLs and records this information in the executable's header. Calls to procedures in DLLs are made indirectly through a transfer vector in the caller's address space. Initially this vector is filled with 0s, since the addresses of the procedures to be called are not yet known.

When the application process is started, the DLLs that are needed are located (on disk or in memory) and mapped into the process' virtual address space. The transfer vector is then filled in with the correct addresses so that the procedures can then be called via the transfer vector with only a negligible loss of efficiency. The win here is that even though multiple application programs have the same DLL mapped in, only one copy of the DLL text is needed in physical memory (but each process gets its own copy of the private static data in the DLL). Windows 2000 uses DLLs extremely heavily for all aspects of the system.

Now we have enough background to see how the Win32 and other process interfaces are implemented. Each user process generally links with a number of DLLs that together implement the Win32 interface. To make an API call, one of the procedures in a DLL is called, shown as step 1 in Fig. 11-13. What happens next depends on the Win32 API call. Different ones are implemented in different ways.

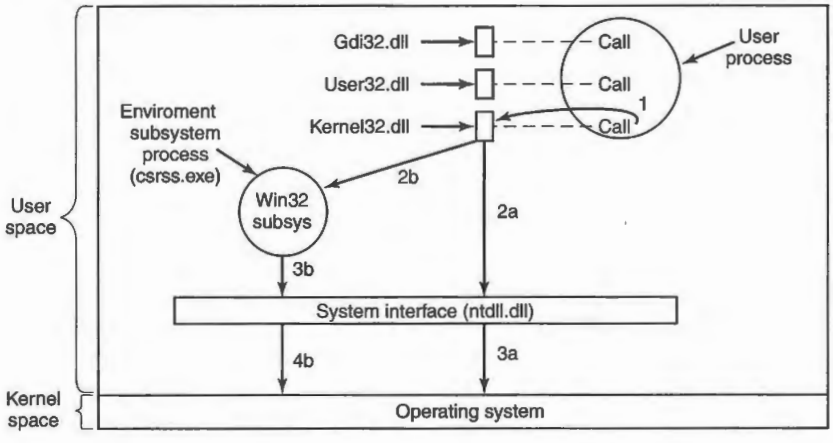


Figure 11-13. Various routes taken to implement Win32 API function calls.

In some cases, the DLL calls another DLL (*ntdll.dll*) that actually traps to the operating system. This path is shown as steps 2a and 3a in Fig. 11-13. It is also possible that the DLL does all the work itself without making a system call at all.

IBM Ex-1009
Page 478

For other Win32 API calls a different route is taken, namely, first a message is sent to the Win32 subsystem process (*csrss.exe*), which then does some work and then makes a system call (steps 2b, 3b, and 4b). Here, too, in some cases the environment subsystem does all the work in user space and just returns immediately. The message passing between the application process and the Win32 subsystem process has been carefully optimized for performance using a special local procedure call mechanism implemented by the executive and shown as LPC in Fig. 11-7.

In the first version of Windows NT, virtually all the Win32 API calls took route 2b, 3b, 4b, putting a large chunk of the operating system in user space (e.g., the graphics). However, starting with NT 4.0, much of the code was put into kernel mode (in the Win32/GDI driver in Fig. 11-7) for performance reasons. In Windows 2000, only a small number of Win32 API calls, for example process and thread creation, take the long route. The other ones take the direct route, bypassing the Win32 environment subsystem.

As an aside, the three most important DLLs are shown in Fig. 11-13, but they are not the only ones. There are over 800 separate DLLs in the *\winnt\system32* directory totalling 130 MB. To avoid any confusion, the number of DLL *files* is over 800; the number of API calls contained in them exceeds 13,000. (The 29 million lines of code had to compile into something, after all.) A few of the more important DLLs are listed in Fig. 11-14. The number of exported functions (i.e., those visible outside the file) in each one is given, but these tend to change (meaning increase) over time. The number of exported functions in the first public release of *ntdll.dll* in Windows 2000 is 1179. These are the real system calls. The 1209 calls exported by *ntoskrnl.exe* are the functions available to device drivers and other code linked with the kernel. The list of exported functions in any *.exe* or *.dll* file can be viewed using the *depends* program in the platform SDK Kit.

File	Mode	Fcns	Contents
hal.dll	Kernel	95	Low-level hardware management, e.g., port I/O
ntoskrnl.exe	Kernel	1209	Windows 2000 operating system (kernel + executive)
win32k.sys	Kernel	-	Many system calls including most of the graphics
ntdll.dll	User	1179	Dispatcher from user mode to kernel mode
csrss.exe	User	0	Win32 environment subsystem process
kernel32.dll	User	823	Most of the core (nongraphics) system calls
gdi32.dll	User	543	Font, text, color, brush, pen, bitmap, palette, drawing, etc. calls
user32.dll	User	695	Window, icon, menu, cursor, dialog, clipboard, etc. calls
advapi32.dll	User	557	Security, cryptography, registry, management calls

Figure 11-14. Some key Windows 2000 files, the mode they run in, the number of exported function calls, and the main contents of each file. The calls in *win32k.sys* are not formally exported since *win32k.sys* is not called directly.

IBM Ex-1009
Page 479

Although the Win32 process interface is the most important one, there are also two other ones: POSIX and OS/2. The POSIX environment provides minimal support for UNIX applications. It supports only the P1003.1 functionality and little else. It does not have threads, windowing, or networking, for example. In practice, porting any real UNIX program to Windows 2000 using this subsystem is close to impossible. It was included only because parts of the U.S. government require operating systems for government computers to be P1003.1 compliant. This subsystem is not self-contained and uses the Win32 subsystem for much of its work, but without exporting the full Win32 interface to its user programs (which would have made it usable, at no extra cost to Microsoft).

To allow UNIX users to migrate to Windows 2000, Microsoft has a product called Interix that provides a better degree of UNIX compatibility than the POSIX subsystem.

The OS/2 subsystem is similarly limited in functionality and does not support any graphical applications. In practice, it, too, is completely useless. Thus the original idea of having multiple operating system interfaces implemented by different processes in user space is essentially gone. What is left is a full Win32 implementation in kernel mode and little else.

11.4 PROCESSES AND THREADS IN WINDOWS 2000

Windows 2000 has a number of concepts for managing the CPU and grouping resources together. In the following sections we will examine these, discussing some of the relevant Win32 API calls, and show how these concepts are implemented.

11.4.1 Fundamental Concepts

Windows 2000 supports traditional processes, which can communicate and synchronize with one another, just as they can in UNIX. Each process contains at least one thread, which in turn contains at least one fiber (lightweight thread). Furthermore, processes can be collected into jobs for certain resource management purposes. Together, jobs, processes, threads, and fibers provide a very general set of tools for managing parallelism and resources, both on uniprocessors (single-CPU machines) and on multiprocessors (multiCPU machines). A brief summary of these four concepts is given in Fig. 11-15.

Let us examine these concepts from the largest to the smallest. A **job** in Windows 2000 is a collection of one or more processes that are to be managed as a unit. In particular, there are quotas and resource limits associated with each job, stored in the corresponding job object. The quotas include items such as the maximum number of processes (prevents any process from generating an unbounded number of children), the total CPU time available to each process individually and

IBM Ex-1009
Page 480

Name	Description
Job	Collection of processes that share quotas and limits
Process	Container for holding resources
Thread	Entity scheduled by the kernel
Fiber	Lightweight thread managed entirely in user space

Figure 11-15. Basic concepts used for CPU and resource management.

to all the processes combined, and the maximum memory usage, again, per process and total. Jobs can also impose security restrictions on the processes in the job, such as not being able to acquire administrator (superuser) power, even with the proper password.

Processes are more interesting than jobs and also more important. As in UNIX, processes are containers for resources. Every process has a 4-GB address space, with the user occupying the bottom 2 GB (optionally 3 GB on Advanced Server and Datacenter server) and the operating system occupying the rest. Thus the operating system is present in every process' address, although protected from tampering by the memory management unit hardware. A process has a process ID, one or more threads, a list of handles (managed in kernel mode), and an access token holding its security information. Processes are created using a Win32 call that takes as its input the name of an executable file, which defines the initial contents of the address space and creates the first thread.

Every process starts out with one thread, but new ones can be created dynamically. Threads forms the basis of CPU scheduling as the operating system always selects a thread to run, not a process. Consequently, every thread has a state (ready, running, blocked, etc.), whereas processes do not have states. Threads can be created dynamically by a Win32 call that specifies the address within the enclosing process' address space it is to start running at. Every thread has a thread ID, which is taken from the same space as the process IDs, so an ID can never be in use for both a process and a thread at the same time. Process and thread IDs are multiples of four so they can be used as byte indices into kernel tables, the same as other objects.

A thread normally runs in user mode, but when it makes a system call it switches to kernel mode and continues to run as the same thread with the same properties and limits it had in user mode. Each thread has two stacks, one for use when it is in user mode and one for use when it is in kernel mode. In addition to a state, an ID, and two stacks, every thread has a context (in which to save its registers when it is not running), a private area for its own local variables, and possibly its own access token. If it has its own access token, this one overrides the process access token in order to let client threads pass their access rights to server threads who are doing work for them. When a thread is finished executing, it can exit. When the last thread still active in a process exits, the process terminates.

It is important to realize that threads are a scheduling concept, not a resource ownership concept. Any thread is able to access all the objects that belong to its process. All it has to do is grab the handle and make the appropriate Win32 call. There is no restriction on a thread that it cannot access an object because a different thread created or opened it. The system does not even keep track of which thread created which object. Once an object handle has been put in a process' handle table, any thread in the process can use it.

In addition to the normal threads that run within user processes, Windows 2000 has a number of daemon threads that run only in kernel space and are not associated with any user process (they are associated with the special system or idle processes). Some perform administrative tasks, such as writing dirty pages to the disk, while others form a pool that can be assigned to a component of the executive or a driver that needs to get some work done asynchronously in the background. We will study some of these threads later when we come to memory management.

Switching threads in Windows 2000 is relatively expensive because doing a thread switch requires entering and later leaving kernel mode. To provide very lightweight pseudoparallelism, Windows 2000 provides **fibers**, which are like threads, but are scheduled in user space by the program that created them (or its run-time system). Each thread can have multiple fibers, the same way a process can have multiple threads, except that when a fiber logically blocks, it puts itself on the queue of blocked fibers and selects another fiber to run in the context of its thread.

The operating system is not aware of this transition because the thread keeps running, even though it may be first running one fiber, then another. In fact, the operating system knows nothing at all about fibers, so there are no executive objects relating to fibers, as there are for jobs, processes, and threads. There are also no true system calls for managing fibers. However, there are Win32 API calls. These are among the Win32 API calls that do not make system calls, which we mentioned during the discussion of Fig. 11-13. The relationship between jobs, processes, and threads is illustrated in Fig. 11-16.

Although we will not discuss it in much detail, Windows 2000 is capable of running on a symmetric multiprocessor system. This requirement means that the operating system code must be fully reentrant, that is, every procedure must be written in such a way that two or more CPUs may be changing its variables at once, without causing problems. In many cases this means that code sections have to be protected by spin locks or mutexes to keep additional CPUs at bay until the first one is done (i.e., serialize access to critical regions). The number of CPUs the system can handle is governed by the licensing restrictions and listed in Fig. 11-2. There is no technical reason why Windows Professional cannot run on a 32-node multiprocessor—it is the same binary as Datacenter Server, after all.

The upper limit of 32 CPUs is a hard limit because word-length bitmaps are used to keep track of CPU usage in various ways. For example, one word-length

IBM Ex-1009
Page 482

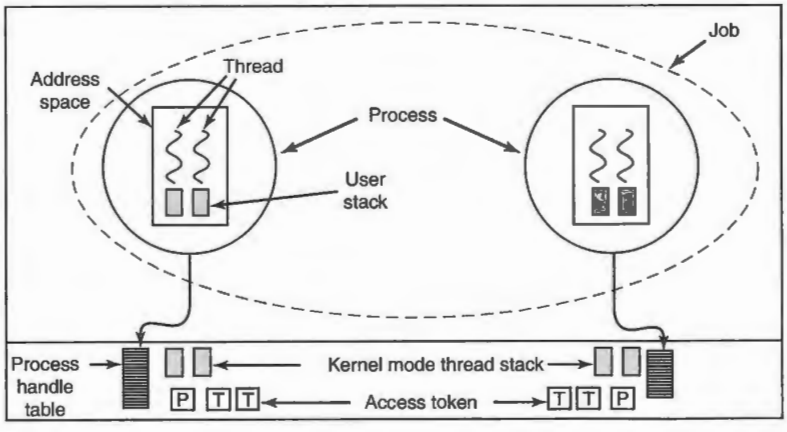


Figure 11-16. The relationship between jobs, processes, and threads. Several fibers can also be multiplexed on one thread (not shown).

bitmap keeps track of which of the (up to) 32 CPUs are currently idle, and another bitmap is used per process to list the CPUs this process is permitted to run on. The 64-bit version of Windows 2000 should be able to effortlessly support up to 64 CPUs; beyond that requires actually changing the code substantially (to use multiple words for the bitmaps).

11.4.2 Job, Process, Thread and Fiber Management API Calls

New processes are created using the Win32 API function CreateProcess. This function has 10 parameters, each of which has many options. This design is clearly a lot more complicated than the UNIX scheme, in which fork has no parameters, and exec has just three: pointers to the name of the file to execute, the (parsed) command line parameter array, and the environment strings. Roughly speaking, the 10 parameters to CreateProcess are as follows:

1. A pointer to the name of the executable file.
2. The command line itself (unparsed).
3. A pointer to a security descriptor for the process.
4. A pointer to a security descriptor for the initial thread.
5. A bit telling whether the new process inherits the creator's handles.
6. Miscellaneous flags (e.g., error mode, priority, debugging, consoles).
7. A pointer to the environment strings.

8. A pointer to the name of the new process' current working directory.
9. A pointer to a structure describing the initial window on the screen.
10. A pointer to a structure that returns 18 values to the caller.

Windows 2000 does not enforce any kind of parent-child or other hierarchy. All processes are created equal (no processes are created more equal). However, since 1 of the 18 parameters returned to the creating process is a handle to the new process (allowing considerable control over the new process), there is an implicit hierarchy in terms of who has a handle to whom. Although these handles cannot just be passed directly to other processes, there is a way for a process to make a duplicate handle suitable for another process and then give it the handle, so the implicit process hierarchy may not last long.

Each process in Windows 2000 is created with a single thread, but a process can create more threads later on. Thread creation is simpler than process creation: CreateThread has only six parameters instead of 10:

1. The optional security descriptor.
2. The initial stack size.
3. The starting address.
4. A user-defined parameter.
5. The initial state of the thread (ready or blocked).
6. The thread's ID.

The kernel does the thread creation, so it is clearly aware of threads (i.e., they are not implemented purely in user space as is the case in some other systems).

Interprocess Communication

Threads can communicate in a wide variety of ways, including pipes, named pipes, mailslots, sockets, remote procedure calls, and shared files. Pipes have two modes: byte and message, selected at creation time. Byte-mode pipes work the same way as in UNIX. Message-mode pipes are somewhat similar but preserve message boundaries, so that four writes of 128 bytes will be read as four 128-byte messages, and not as one 512-byte message, as might happen with byte-mode pipes. Named pipes also exist and have the same two modes as regular pipes. Named pipes can also be used over a network; regular pipes cannot.

Mailslots are a feature of Windows 2000 not present in UNIX. They are similar to pipes in some ways, but not all. For one thing, they are one-way, whereas pipes are two-way. They can also be used over a network but do not provide guaranteed delivery. Finally, they allow the sending process to broadcast a message to many receivers, instead of to just one receiver.

IBM Ex-1009
Page 484

Sockets are like pipes, except that they normally connect processes on different machines. For example, one process writes to a socket and another one on a remote machine reads from it. Sockets can also be used to connect processes on the same machine, but since they entail more overhead than pipes, they are generally only used in a networking context.

Remote procedure calls are a way for process *A* to have process *B* call a procedure in *B*'s address space on *A*'s behalf and return the result to *A*. Various restrictions on the parameters exist. For example, it makes no sense to pass a pointer to a different process.

Finally, processes can share memory by mapping onto the same file at the same time. All writes done by one process then appear in the address spaces of the other processes. Using this mechanism, the shared buffer used in producer-consumer problems can easily be implemented.

Just as Windows 2000 provides numerous interprocess communication mechanisms, it also provides numerous synchronization mechanisms, including semaphores, mutexes, critical regions, and events. All of these mechanisms work on threads, not processes, so that when a thread blocks on a semaphore, other threads in that process (if any) are not affected and can continue to run.

A semaphore is created using the CreateSemaphore API function, which can initialize it to a given value and define a maximum value as well. Semaphores are kernel objects and thus have security descriptors and handles. The handle for a semaphore can be duplicated using DuplicateHandle and passed to another process so that multiple processes can synchronize on the same semaphore. Calls for up and down are present, although they have the somewhat peculiar names of ReleaseSemaphore (up) and WaitForSingleObject (down). It is also possible to give WaitForSingleObject a timeout, so the calling thread can be released eventually, even if the semaphore remains at 0 (although timers reintroduce races).

Mutexes are also kernel objects used for synchronization, but simpler than semaphores because they do not have counters. They are essentially locks, with API functions for locking (WaitForSingleObject) and unlocking (ReleaseMutex). Like semaphore handles, mutex handles can be duplicated and passed between processes so that threads in different processes can access the same mutex.

The third synchronization mechanism is based on **critical sections**, (which we have called critical regions elsewhere in this book) which are similar to mutexes, except local to the address space of the creating thread. Because critical sections are not kernel objects, they do not have handles or security descriptors and cannot be passed between processes. Locking and unlocking is done with EnterCriticalSection and LeaveCriticalSection, respectively. Because these API functions are performed initially in user space and only make kernel calls when blocking is needed, they are faster than mutexes.

The last synchronization mechanism uses kernel objects called **events**, of which there are two kinds: **manual-reset events** and **auto-reset events**. Any event can be in one of two states: set and cleared. A thread can wait for an event

IBM Ex-1009
Page 485

to occur with WaitForSingleObject. If another thread signals an event with SetEvent, what happens depends on the type of event. With a manual-reset event, all waiting threads are released and the event stays set until manually cleared with ResetEvent. With an auto-reset event, if one or more threads are waiting, exactly one thread is released and the event is cleared. An alternative operation is PulseEvent, which is like SetEvent except that if nobody is waiting, the pulse is lost and the event is cleared. In contrast, a SetEvent that occurs with no waiting threads is remembered by leaving the event in set state so a subsequent thread waiting on it is released immediately.

Events, mutexes, and semaphores can all be named and stored in the file system, like named pipes. Two or more processes can synchronize by opening the same event, mutex, or semaphore, rather than having one of them create the object and then make duplicate handles for the others, although the latter approach is certainly an option as well.

The number of Win32 API calls dealing with processes, threads, and fibers is nearly 100, a substantial number of which deal with IPC in one form or another. A summary of the ones discussed above as well as some other important ones are given in Fig. 11-17.

Most of the calls in Fig. 11-17 were either discussed above or should be self-explanatory. Again note that not all of these are system calls. As we mentioned earlier, Windows 2000 knows nothing about fibers. They are entirely implemented in user space. As a consequence, the CreateFiber call does its work entirely in user space without making any system calls (unless it has to allocate some memory). Many other Win32 calls have this property as well, including EnterCriticalSection and LeaveCriticalSection as we noted above.

11.4.3 Implementation of Processes and Threads

Processes and threads are more important and more elaborate than jobs and fibers, so we will concentrate on them here. A process is created when another process makes the Win32 CreateProcess call. This call invokes a (user-mode) procedure in *kernel32.dll* that creates the process in several steps using multiple system calls and other work.

1. The executable file given as a parameter is examined and opened. If it is a valid POSIX, OS/2, 16-bit Windows, or MS-DOS file, a special environment is set up for it. If it is a valid 32-bit Win32 *.exe* file, the registry is checked to see if it is special in some way (e.g., to be run under supervision of a debugger). All of this is done in user mode inside *kernel32.dll*.
2. A system call, NtCreateProcess, is made to create the empty process object and enter it into the object manager's name space. Both the kernel object and the executive object are created. In addition, the

IBM Ex-1009
Page 486

Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semapahore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitForMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled then to nonsignaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section

Figure 11-17. Some of the Win32 calls for managing processes, threads, and fibers.

process manager creates a process control block for the object and initializes it with the process ID, quotas, access token, and various other fields. A section object is also created to keep track of the process' address space.

3. When *kernel32.dll* gets control back, it makes another system call, NtCreateThread, to create the initial thread. The thread's user and kernel stacks are also created. The stack size is given in the header of the executable file.

4. *Kernel32.dll* now sends a message to the Win32 environment subsystem telling it about the new process and passing it the process and thread handles. The process and threads are entered into the subsystems tables so it has a complete list of all processes and threads. The subsystem then displays a cursor containing a pointer with an hourglass to tell the user that something is going on but that the cursor can be used in the meanwhile. When the process makes its first GUI

IBM Ex-1009
Page 487

call, usually to create a window, the cursor is removed (it times out after 2 seconds if no call is forthcoming).

5. At this point, the thread is able to run. It starts out by running a run-time system procedure to complete the initialization.
6. The run-time procedure sets the thread's priority, tells loaded DLLs that a new thread is present, and does other housekeeping chores. Finally, it begins running the code of the process' main program.

Thread creation also consists of a number of steps, but we will not go into them in much detail. It starts when the running process executes CreateThread, which calls a procedure inside *kernel32.dll*. This procedure allocates a user stack within the calling process and then makes the NtCreateThread call to create an executive thread object, initialize it, and also create and initialize a thread control block. Again, the Win32 subsystem is notified and enters the new thread in its tables. Then the thread starts running and completes its own initialization.

When a process or thread is created, a handle is returned for it. This handle can be used to start, stop, kill, and inspect the process or thread. It is possible for the owner of a handle to pass the handle to another process in a controlled and secure way. This technique is used to allow debuggers to have control over the processes they are debugging.

Scheduling

Windows 2000 does not have a central scheduling thread. Instead, when a thread cannot run any more, the thread enters kernel mode and runs the scheduler itself to see which thread to switch to. The following conditions cause the currently running thread to execute the scheduler code:

1. The thread blocks on a semaphore, mutex, event, I/O, etc.
2. It signals an object (e.g., does an up on a semaphore).
3. The running thread's quantum expires.

In case 1, the thread is already running in kernel mode to carry out the operation on the dispatcher or I/O object. It cannot possibly continue, so it must save its own context, run the scheduler code to pick its successor, and load that thread's context to start it.

In case 2, the running thread is in the kernel, too. However, after signaling some object, it can definitely continue because signaling an object never blocks. Still, the thread is required to run the scheduler to see if the result of its action has released a higher priority thread that is now free to run. If so, a thread switch occurs because Windows 2000 is fully preemptive (i.e., thread switches can occur at any moment, not just at the end of the current thread's quantum).

In case 3, a trap to kernel mode occurs, at which time the thread executes the scheduler code to see who runs next. Depending on what other threads are waiting, the same thread may be selected, in which case it gets a new quantum and continues running. Otherwise a thread switch happens.

The scheduler is also called under two other conditions:

1. An I/O operation completes.
2. A timed wait expires.

In the first case, a thread may have been waiting on this I/O and is now released to run. A check has to be made to see if it should preempt the running thread since there is no guaranteed minimum run time. The scheduler is not run in the interrupt handler itself (since that may keep interrupts turned off too long). Instead a DPC is queued for slightly later, after the interrupt handler is done. In the second case, a thread has done a down on a semaphore or blocked on some other object, but with a timeout that has now expired. Again it is necessary for the interrupt handler to queue a DPC to avoid having it run during the clock interrupt handler. If a thread has been made ready by this timeout, the scheduler will be run and if nothing more important is available, the DPC will run next.

Now we come to the actual scheduling algorithm. The Win32 API provides two hooks for processes to influence thread scheduling. These hooks largely determine the algorithm. First, there is a call SetPriorityClass that sets the priority class of all the threads in the caller's process. The allowed values are: realtime, high, above normal, normal, below normal, and idle.

Second, there is a call SetThreadPriority that sets the relative priority of some thread (possibly, but not necessarily, the calling thread) compared to the other threads in its process. The allowed values are: time critical, highest, above normal, normal, below normal, lowest, and idle. With six process classes and seven thread classes, a thread can have any one of 42 combinations. This is the input to the scheduling algorithm.

The scheduler works as follows. The system has 32 priorities, numbered from 0 to 31. The 42 combinations are mapped onto the 32 priority classes according to the table of Fig. 11-18. The number in the table determines the thread's **base priority**. In addition, every thread has a **current priority**, which may be higher (but not lower) than the base priority and which we will discuss shortly.

To use these priorities for scheduling, the system maintains an array with 32 entries, corresponding to priorities 0 through 31 derived from the table of Fig. 11-18. Each array entry points to the head of a list of ready threads at the corresponding priority. The basic scheduling algorithm consists of searching the array from priority 31 down to priority 0. As soon as a nonempty slot is found, the thread at the head of the queue is selected and run for one quantum. If the quantum expires, the thread goes to the end of the queue at its priority level and the thread at the front is chosen next. In other words, when there are multiple

		Win32 process class priorities					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Figure 11-18. Mapping of Win32 priorities to Windows 2000 priorities.

threads ready at the highest priority level, they run round robin for one quantum each. If no thread is ready, the idle thread is run.

It should be noted that scheduling is done by picking a thread without regard to which process that thread belongs. Thus the scheduler does *not* first pick a process and then pick a thread in that process. It only looks at the threads. It does not even know which thread belongs to which process. On a multiprocessor, each CPU schedules itself using the priority array. A spin lock is used to make sure that only one CPU at a time is inspecting the array.

The array of queue headers is shown in Fig. 11-19. The figure shows that there are actually four categories of priorities: realtime, user, zero, and idle, which is effectively –1. These deserve some comment. Priorities 16–31 are called real time, but they are not. There are no guarantees given and no deadlines are met. They are simply higher priority than 0–15. However, priorities 16 through 31 are reserved for the system itself and for threads explicitly assigned those priorities by the system administrator. Ordinary users may not run there for a good reason. If a user thread were to run at a higher priority than, say, the keyboard or mouse thread and get into a loop, the keyboard or mouse thread would never run, effectively hanging the system.

User threads run at priorities 1–15. By setting the process and thread priorities, a user can determine which threads get preference. The zero thread runs in the background and eats up whatever CPU time nobody else wants. Its job is to zero pages for the memory manager. We will discuss its role later. If there is absolutely nothing to do, not even zero pages, the idle thread runs. It is not really a full-blown thread though.

Over the course of time, some patches were made to the basic scheduling algorithm to improve system performance. Under certain specific conditions, the current priority of a user thread can be raised above the base priority (by the operating system), but never above priority 15. Since the array of Fig. 11-19 is

IBM Ex-1009
Page 490

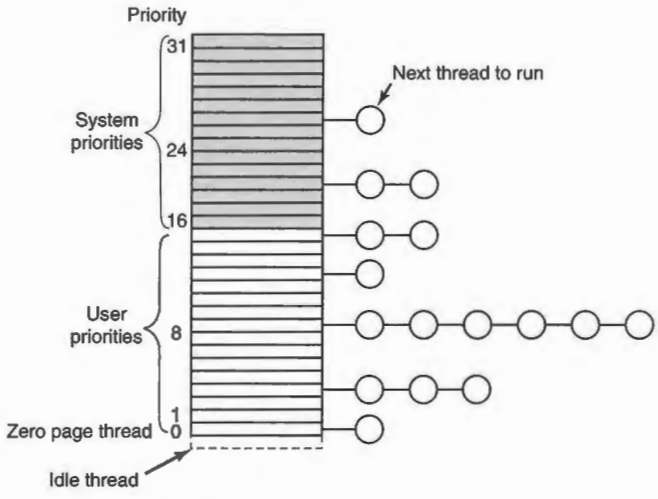


Figure 11-19. Windows 2000 supports 32 priorities for threads.

based on the current priority, changing this priority affects scheduling. No adjustments are ever made to threads running at priority 15 or higher.

Let us now see when a thread's priority is raised. First, when an I/O operation completes and releases a waiting thread, the priority is boosted to give it a chance to run again quickly and start more I/O. The idea here is to keep the I/O devices busy. The amount of boost depends on the I/O device, typically 1 for a disk, 2 for a serial line, 6 for the keyboard, and 8 for the sound card.

Second, if a thread was waiting on a semaphore, mutex, or other event, when it is released, it gets boosted by 2 units if it is in the foreground process (the process controlling the window to which keyboard input is sent) and 1 unit otherwise. This fix tends to raise interactive processes above the big crowd at level 8. Finally, if a GUI thread wakes up because window input is now available, it gets a boost for the same reason.

These boosts are not forever. They take effect immediately, but if a thread uses all of its next quantum, it loses one point and moves down one queue in the priority array. If it uses up another full quantum, it moves down another level, and so on until it hits its base level, where it remains until it is boosted again. Clearly, if a thread wants good service, it should play a lot of music.

There is one other case in which the system fiddles with the priorities. Imagine that two threads are working together on a producer-consumer type problem. The producer's work is harder, so it gets a high priority, say 12, compared to the consumer's 4. At a certain point, the producer has filled up a shared buffer and blocks on a semaphore, as illustrated in Fig. 11-20(a).

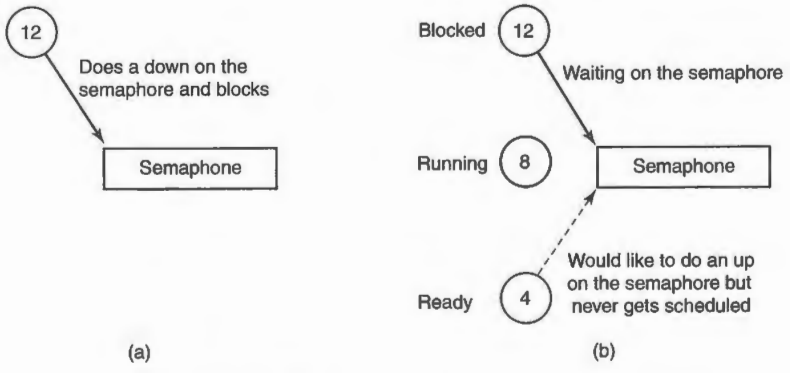


Figure 11-20. An example of priority inversion.

Before the consumer gets a chance to run again, an unrelated thread at priority 8 becomes ready and starts running, as shown in Fig. 11-20(b). As long as this thread wants to run, it will be able to, since it outguns the consumer and the producer, while higher, is blocked. Under these circumstances, the producer will never get to run again until the priority 8 thread gives up.

Windows 2000 solves this problem through what might be charitably called a big hack. The system keeps track of how long it has been since a ready thread ran last. If it exceeds a certain threshold, it is moved to priority 15 for two quanta. This may give it the opportunity to unblock the producer. After the two quanta are up, the boost is abruptly removed rather than decaying gradually. Probably a better solution would be to penalize threads that use up their quantum over and over by lowering their priority. After all, the problem was not caused by the starved thread, but by the greedy thread. This problem is well known under the name **priority inversion**.

An analogous problem happens if a priority 16 thread grabs a mutex and does not get a chance to run for a long time, starving more important system threads that are waiting for the mutex. This problem can be prevented within the operating system by having a thread that needs a mutex for a short time just disable scheduling while it is busy. On a multiprocessor, a spin lock should be used.

Before leaving the subject of scheduling, it is worth saying a couple of words about the quantum. On Windows 2000 Professional the default is 20 msec; on uniprocessor servers it is 120 msec; on multiprocessors various other values are used, depending on the clock frequency. The shorter quantum favors interactive users whereas the longer quantum reduces context switches and thus provides better efficiency. This is the meaning of the last column in Fig. 11-2. These defaults can be increased manually by 2x, 4x, or 6x if desired. As an aside, the size of the quantum was chosen a decade ago and not changed since although machines are now more than an order of magnitude faster. The numbers probably

IBM Ex-1009
Page 492

could be reduced by a factor of 5 to 10 with no harm and possibly better response time for interactive threads in a heavily loaded system.

One last patch to the scheduling algorithm says that when a new window becomes the foreground window, all of its threads get a longer quantum by an amount taken from the registry. This change gives them more CPU time, which usually translates to better service for the window that just moved to the foreground.

11.4.4 MS-DOS Emulation

One of the design goals of Windows 2000 was inherited from NT: try to run as many reasonable MS-DOS programs as possible. This goal is quite different from Windows 98's stated goal: run all old MS-DOS programs (to which we add: no matter how ill-behaved they may be).

The way Windows 2000 deals with ancient programs is to run them in a fully protected environment. When an MS-DOS program is started, a normal Win32 process is started and loaded with an MS-DOS emulation program, *ntvdm* (NT Virtual DOS Machine) that will monitor the MS-DOS program and carry out its system calls. Since MS-DOS only recognized memory up to 1 MB on the 8088 and only up to 16 MB with bank switching and other tricks on the 286, it is safe to put *ntvdm* high in the process' virtual address space where the program has no way to address it. This situation is shown in Fig. 11-21.

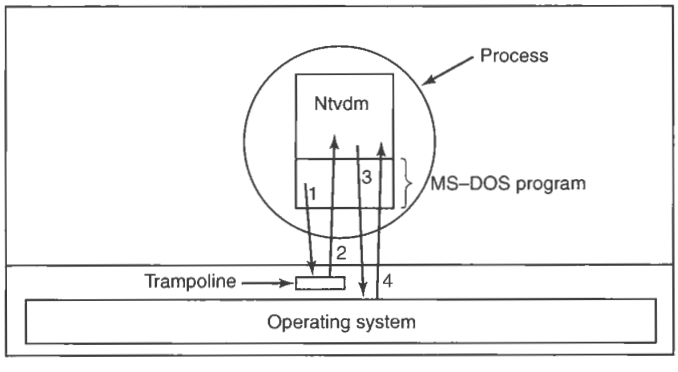


Figure 11-21. How old MS-DOS programs are run under Windows 2000.

When the MS-DOS program is just executing normal instructions, it can run on the bare hardware since the Pentium includes all the 8088 and 286 instructions as subsets. The interesting part is what happens when the MS-DOS program wants to do I/O or interact with the system. A well-behaved program just makes a system call. In expectation of this, *ntvdm* instructs Windows 2000 to reflect all MS-DOS system calls back to it. In effect, the system call just bounces off the operating

system and is caught by the emulator, as shown in steps 1 and 2 in Fig. 11-21. Sometimes this technique is referred to as using a **trampoline**.

Once it gets control, the emulator figures out what the program was trying to do and issues its own Win32 calls to get the work done (step 3 and 4 in Fig. 11-21). As long as the program is well behaved and just makes legal MS-DOS system calls, this technique works fine. The trouble is that some old MS-DOS programs bypassed the operating system and wrote directly to the video RAM, read directly from the keyboard, and so on, things that are impossible in a protected environment. To the extent that the illegal behavior causes a trap, there is some hope that the emulator can figure out what the program was trying to do and emulate it. If it does not know what the program wants, the program is just killed because 100 percent emulation was not a Windows 2000 requirement.

11.4.5 Booting Windows 2000

Before Windows 2000 can start up, it must be booted. The boot process creates the initial processes that bring up the system. In this section we will briefly discuss how the boot process works for Windows 2000. The hardware boot process consists of reading in the first sector of the first disk (the master boot record) and jumping to it, as we described in Sec. 5.4.2. This short assembly language program reads the partition table to see which partition contains the bootable operating system. When it finds the operating system partition, it reads in the first sector of that partition, called the **boot sector**, and jumps to it. The program in the boot sector reads its partition's root directory, searching for a file called *ntldr* (another piece of archaeological evidence that Windows 2000 is really NT). If it finds that file, it reads the file into memory and executes it. *Ntldr* loads Windows 2000. As an aside, there are several versions of the boot sector, depending on whether the partition is formatted as FAT-16, FAT-32, or NTFS. When Windows 2000 is installed, the correct version of the master boot record and boot sector are written to disk.

Ntldr now reads a file called *Boot.ini*, which is the only configuration information not in the registry. It lists all the versions of *hal.dll* and *ntoskrnl.exe* available for booting in this partition. The file also provides many parameters, such as how many CPUs and how much RAM to use, whether to give user processes 2 GB or 3 GB, and what rate to set the real-time clock to. *Ntldr* then selects and loads *hal.dll* and *ntoskrnl.exe* files as well as *bootvid.dll*, the default video driver for writing on the display during the boot process. *Ntldr* next reads the registry to find out which drivers are needed to complete the boot (e.g., the keyboard and mouse drivers, but also dozens more for controlling various chips on the parentboard). Finally, it reads in all these drivers and passes control to *ntoskrnl.exe*.

Once started, the operating system does some general initialization and then calls the executive components to do their own initialization. For example, the object manager prepares its name space to allow other components call it to insert

their objects into the name space. Many components also do specific things related to their function, such as the memory manager setting up the initial page tables and the plug-and-play manager finding out which I/O devices are present and loading their drivers. All in all, dozens of steps are involved, during which time the progress bar displayed on the screen is growing in length as steps are completed. The last step is creating the first true user process, the **session manager**, *smss.exe*. Once this process is up and running, booting is completed.

The session manager is a native Windows 2000 process. It makes true system calls and does not use the Win32 environment subsystem, which is not even running yet. In fact, one of its first duties is to start it (*csrss.exe*). It also reads the registry hives from disk and learns what else it is supposed to do. Typically its work includes entering many objects in the object manager's name space, creating any extra paging files needed, and opening important DLLs to have them around all the time. After it has done most of this work, it creates the login daemon, *winlogon.exe*.

At this point, the operating system is up and running. Now it is time to get the service processes (user space daemons) going and allow users to log in. *Winlogon.exe* first creates the authentication manager (*lsass.exe*), and then the parent process of all the services (*services.exe*). The latter looks in the registry to find out which user space daemon processes are needed and what files they are in. It then starts creating them. They are shown in Fig. 11-7. The fact that the disk is generally being heavily used after the first user has logged in (but has done nothing) is not the user's fault. The culprit is *services.exe* creating all the services. In addition, it also loads any remaining device drivers that have not yet been loaded. The hierarchy of initial processes and some typical services are shown in Fig. 11-22.

Winlogon.exe is also responsible for all user logins. The actual login dialog is handled by a separate program in *msgina.dll* to make it possible for third parties to replace the standard login with faceprint identification or something else other than name and password. After a successful login, *winlogon.exe* gets the user's profile from the registry and from it determines which shell to run. Many people do not realize it, but the standard Windows desktop is just *explorer.exe* with some options set. If desired, a user can select any other program as the shell, including the command prompt or even *Word*, by editing the registry. However, editing the registry is not for the faint of heart; a mistake here can make the system unusable.

11.5 MEMORY MANAGEMENT

Windows 2000 has an extremely sophisticated virtual memory system. It has a number of Win32 functions for using it and part of the executive plus six dedicated kernel threads for managing it. In the following sections we will look at the fundamental concepts, the Win32 API calls, and finally the implementation.

Process	Description
idle	Not really a process, but home to the idle thread
system	Creates smss.exe & paging files; reads registry; opens DLLs
smss.exe	First real proc; much initialization; creates csrss & winlogon
csrss.exe	Win32 subsystem process
winlogon.exe	Login daemon
lsass.exe	Authentication manager
services.exe	Looks in registry and starts services
Printer server	Allows remote jobs to use the printer
File server	Serves requests for local files
Telnet daemon	Allows remote logins
Incoming email handler	Accepts and stores inbound email
Incoming fax handler	Accepts and prints inbound faxes
DNS resolver	Internet domain name system server
Event logger	Logs various system events
Plug-and-play manager	Monitors hardware to see what is out there

Figure 11-22. The processes starting up during the boot phase. The ones above the line are always started. The ones below it are examples of services that could be started.

11.5.1 Fundamental Concepts

In Windows 2000, every user process has its own virtual address space. Virtual addresses are 32 bits long, so each process has 4 GB of virtual address space. The lower 2 GB minus about 256 MB are available for the process' code and data; the upper 2 GB map onto kernel memory in a protected way. The virtual address space is demand paged, with a fixed page size (4 KB on the Pentium).

The virtual address space layout for three user processes is shown in Fig. 11-23 in slightly simplified form. The bottom and top 64 KB of each process' virtual address space is normally unmapped This choice was made intentionally to help catch programming errors. Invalid pointers are often 0 or −1, so attempts to use them on Windows 2000 will cause an immediate trap instead of reading garbage or, worse yet, writing to an incorrect memory location. However, when old MS-DOS programs are being run in emulation mode, they can be mapped in.

Starting at 64 KB comes the user's private code and data. This extends up to almost 2 GB. The last piece of the bottom 2 GB contains some system counters and timers that are shared among all users read only. Making them visible here allows processes to access them without the overhead of a system call.

The upper 2 GB contains the operating system, including the code, data, and the paged and nonpaged pools (used for objects, etc.). The upper 2 GB is shared among all user processes, except for the page tables, which are each process' own page tables. The upper 2 GB of memory is not writable and mostly not even

IBM Ex-1009
Page 496

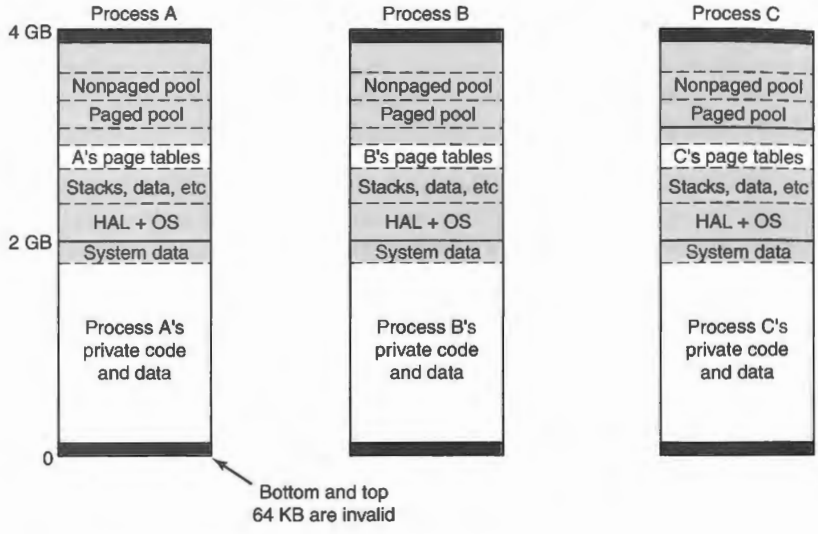


Figure 11-23. Virtual address space layout for three user processes. The white areas are private per process. The shaded areas are shared among all processes.

readable for user-mode processes. The reason for putting it here is that when a thread makes a system call, it traps into kernel mode and just keeps on running in the same thread. By making the whole operating system and all of its data structures (as well as the whole user process) visible within a thread's address space when it enters kernel mode, there is no need to change the memory map or flush the cache upon kernel entry. All that has to be done is switch to the thread's kernel stack. The trade-off here is less private address space per process in return for faster system calls. Large database servers already feel cramped, which is why the 3-GB user space option is available on Advanced Server and Datacenter Server.

Each virtual page can be in one of three states: free, reserved, or committed. A **free page** is not currently in use and a reference to it causes a page fault. When a process is started, all of its pages are in free state until the program and initial data are mapped into its address space. Once code or data is mapped onto a page, the page is said to be **committed**. A reference to a committed page is mapped using the virtual memory hardware and succeeds if the page is in main memory. If the page is not in main memory, a page fault occurs and the operating system finds and brings in the page from disk.

A virtual page can also be in **reserved** state, meaning it is not available for being mapped until the reservation is explicitly removed. For example, when a

new thread is created, 1 MB of stack space is reserved in the process' virtual address space, but only one page is committed. This technique means that the stack can eventually grow to 1 MB without fear that some other thread will allocate the needed contiguous piece of virtual address space out from under it. In addition to the free, reserved, and committed attributes, pages also have other attributes, such as being readable, writable, and executable.

An interesting trade-off occurs with assignment of backing store to committed pages. A simple strategy would be to assign a page in one of the paging files to back up each committed page at the time the page was committed. This would guarantee that there was always a known place to write out each committed page should it be necessary to evict it from memory. The downside of this strategy is that the paging file might have to be as large as the union of all processes' virtual memory. On a large system that rarely ran out of memory and thus rarely paged, this approach would waste disk space.

To avoid wasting disk space, Windows 2000 committed pages that have no natural home on the disk (e.g., stack pages) are not assigned a disk page until the moment that they have to be paged out. This design makes the system more complex because the paging files maps may have to be fetched during a page fault, and fetching them may cause one or more additional page faults inside the page fault handler. On the other hand, no disk space need be allocated for pages that are never paged out.

Trade-offs like this (system complexity versus better performance or more features) tend to get resolved in favor of the latter because the value of better performance or more features is clear but the downside of complexity (a bigger maintenance headache and more crashes per year) is hard to quantify. Free and reserved pages never have shadow pages on disk and references to them always cause page faults.

The shadow pages on the disk are arranged into one or more paging files. There may be up to 16 paging files, possibly spread over 16 separate disks, for higher I/O bandwidth. Each one has an initial size and a maximum size it can grow to later if needed. These files can be created at the maximum size at system installation time in order to reduce the chances that they are highly fragmented, but new ones can be created using the control panel later on. The operating system keeps track of which virtual page maps onto which part of which paging file. For (execute only) program text, the executable binary file (i.e., *.exe* or *.dll* file) contains the shadow pages; for data pages, the paging files are used.

Windows 2000, like many versions of UNIX, allows files to be mapped directly onto regions of the virtual address spaces (i.e., runs of consecutive pages). Once a file has been mapped onto the address space, it can be read or written using ordinary memory references. Memory-mapped files are implemented in the same way as other committed pages, only the shadow pages are in the user's file instead of in the paging file. As a result, while a file is mapped in, the version in memory may not be identical to the disk version (due to recent writes to the vir-

IBM Ex-1009
Page 498

tual address space). However, when the file is unmapped or is explicitly flushed, the disk version is brought up to date.

Windows 2000 explicitly allows two or more processes to map onto the same part of the same file at the same time, possibly at different virtual addresses, as shown in Fig. 11-24. By reading and writing memory words, the processes can now communicate with each other and pass data back and forth at very high bandwidth, since no copying is required. Different processes may have different access permissions. Since all the processes using a mapped file share the same pages, changes made by one of them are immediately visible to all the others, even if the disk file has not yet been updated. Care is also taken that if another process opens the file for normal reading, it sees the current pages in RAM, not stale pages from the disk.

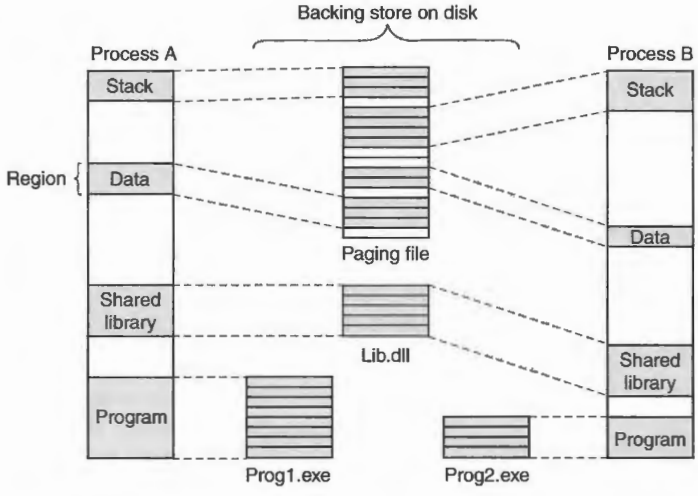


Figure 11-24. Mapped regions with their shadow pages on disk. The *lib.dll* file is mapped into two address spaces at the same time.

It is worth noting that there is a problem if two programs share a DLL file and one of them changes the file's static data. If no special action is taken, the other one will see the changed data, which is probably not what is desired. The problem is solved by mapping all pages in as read only by secretly noting that some are really writable. When a write happens to a page that is mapped read only but is really writable, a private copy of the page is made and mapped in. Now it can be written safely without affecting other users or the original copy on disk. This technique is called **copy-on-write**.

Also it is worth noting that if program text is mapped into two address spaces at different addresses, a certain problem arises with addressing. What happens if the first instruction is JMP 300? If process one maps the program in at address

IBM Ex-1009
Page 499

65,536, the code can easily be patched to read JMP 65836. But what happens if a second process now maps it in at 131,072? The JMP 65836 will go to address 65,836 instead of 131,372 and the program will fail. The solution is to use only relative offsets, not absolute virtual addresses in code that is to be shared. Fortunately, most machines have instructions using relative offsets as well as instructions using absolute addresses. Compilers can use the relative offset instructions, but they have to know in advance whether to use them or the absolute ones. The relative ones are not used all the time because the resulting code is usually less efficient. Usually, a compiler flag tells them which to use. The technique of making it possible to place a piece of code at any virtual address without relocation is called **position independent code**.

Years ago, when 16-bit (or 20-bit) virtual address spaces were standard, but machines had megabytes of physical memory, all kinds of tricks were thought of to allow programs to use more physical memory than fit in the address space. Often these tricks went under the name of **bank switching**, in which a program could substitute some block of memory above the 16-bit or 20-bit limit for a block of its own memory. When 32-bit machines were introduced, people thought they would have enough address space forever. They were wrong. The problem is back. Large programs often need more than the 2 GB or 3 GB of user address space Windows 2000 allocates to them, so bank switching is back, now called **address windowing extensions**. This facility allows programs to map into shuffle chunks of memory in and out of the user portion of the address space (and especially above the dreaded 4-GB boundary). Since it is only used on servers with more than 2 GB of physical memory, we will defer the discussion until the next edition of this book (by which time even entry-level desktop machines will be feeling the 32-bit pinch).

11.5.2 Memory Management System Calls

The Win32 API contains a number of functions that allow a process to manage its virtual memory explicitly. The most important of these functions are listed in Fig. 11-25. All of them operate on a region consisting either of a single page or a sequence of two or more pages that are consecutive in the virtual address space.

The first four API functions are used to allocate, free, protect and query regions of virtual address space. Allocated regions always begin on 64-KB boundaries to minimize porting problems to future architectures with pages larger than current ones (up to 64 KB). The actual amount of address space allocated can be less than 64 KB, but must be a multiple of the page size. The next two give a process the ability to hardwire pages in memory so they will not be paged out and to undo this property. A real-time program might need this ability, for example. A limit is enforced by the operating system to prevent processes from getting too greedy. Actually, the pages can be removed from memory, but only if the entire

Win32 API function	Description
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file mapping object

Figure 11-25. The principal Win32 API functions for managing virtual memory in Windows 2000.

process is swapped out. When it is brought back, all the locked pages are reloaded before any thread can start running again. Although not shown in Fig. 11-25, Windows 2000 also has API functions to allow a process to access the virtual memory of a different process over which it has been given control (i.e., for which it has a handle).

The last four API functions listed are for managing memory-mapped files. To map a file, a file mapping object (see Fig. 11-10) must first be created, with CreateFileMapping. This function returns a handle to the file mapping object and optionally enters a name for it into the file system so another process can use it. The next two functions map and unmap files, respectively. The last one can be used by a process to map in a file currently also mapped in by a different process. In this way, two or more processes can share regions of their address spaces. This technique allows them to write in limited regions of each other's memory.

11.5.3 Implementation of Memory Management

Windows 2000 supports a single linear 4-GB demand-paged address space per process. Segmentation is not supported in any form. Theoretically, page sizes can be any power of two up to 64 KB. On the Pentium they are fixed at 4 KB; on the Itanium they can be 8 KB or 16 KB. In addition, the operating system itself can use 4-MB pages to reduce page table space consumed.

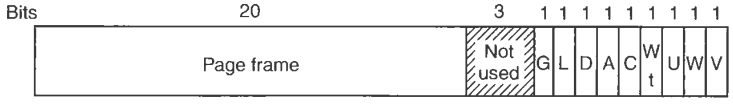
Unlike the scheduler, which selects individual threads to run and does not care much about processes, the memory manager deals entirely with processes and does not care much about threads. After all, processes, not threads, own the address space and that is what the memory manager deals with. When a region of virtual address space is allocated, as four of them have been for process *A* in

Fig. 11-24, the memory manager creates a **VAD (Virtual Address Descriptor)** for it, listing the range of addresses mapped, the backing store file and offset where it is mapped, and the protection code. When the first page is touched, the directory of page tables is created and a pointer to it inserted is in the VAD. In fact, an address space is completely defined by the list of its VADs. This scheme supports sparse address spaces because unused areas between the mapped regions use no resources.

Page Fault Handling

Windows 2000 does not use any form of prepaging. When a process starts, none of its pages are in memory. All of them are brought in dynamically as page faults occurs. On each page fault, a trap to the kernel (in the sense of Fig. 11-7) occurs. The kernel builds a machine-independent descriptor telling what happened and passes this to the memory manager part of the executive. The memory manager then checks it for validity. If the faulted page falls within a committed or reserved region, it looks up the address in the list of VADs, finds (or creates) the page table, and looks up the relevant entry.

The page table entries are different for different architectures. For the Pentium, the entry for a mapped page is shown in Fig. 11-26. Unmapped pages also have entries, but their format is somewhat different. For example, for an unmapped page that must be zeroed before it may be used, that fact is noted in the page table.



G: Page is global to all processes
L: Large (4-MB) page
D: Page is dirty
A: Page has been accessed
C: Caching enabled/disabled
Wt: Write through (no caching)
U: Page is accessible in user mode
W: Writing to the page permitted
V: Valid page table entry

Figure 11-26. A page table entry for a mapped page on the Pentium.

The most important bits in the page table entry for purposes of the paging algorithm are the *A* and *D* bits. They are fed by the hardware and keep track of whether the page has been referenced or written on, respectively, since the last time they were cleared.

Page faults come in five categories:

1. The page referenced is not committed.
2. A protection violation occurred.
3. A shared page has been written.

IBM Ex-1009
Page 502

4. The stack needs to grow.
5. The page referenced is committed but not currently mapped in.

The first and second cases are fatal errors from which there is no recovery for the faulting process. The third case has the same symptoms as the second one (an attempt to write to a read-only page), but the treatment is different. The solution is to copy the page to a new physical page frame and map that one in read/write. This is how copy-on-write works. (If a shared page is marked writable in all processes using it, it is not copy-on-write and no fault occurs when writing to it.) The fourth case requires allocating a new page frame and mapping it in. However, the security rules require that the page contain only 0s, to prevent the process from snooping on the previous owner of the page. Thus a page of 0s must be found, or if one is not available, another page frame must be allocated and zeroed on the spot. Finally, the fifth case is a normal page fault. The page is located and mapped in.

The actual mechanics of getting and mapping pages is fairly standard, so we will not discuss this issue. The only noteworthy feature is that Windows 2000 does not read in isolated pages from the disk. Instead, it reads in runs of consecutive pages, usually about 1–8 pages, in an attempt to minimize the number of disk transfers. The run size is larger for code pages than for data pages.

The Page Replacement Algorithm

Page replacement works like this. The system makes a serious attempt to maintain a substantial number of free pages in memory so that when a page fault occurs, a free page can be claimed on the spot, without the need to first write some other page to disk. As a consequence of this strategy, most page faults can be satisfied with at most one disk operation (reading in the page), rather than sometimes two (writing back a dirty page and then reading in the needed page).

Of course, the pages on the free list have to come from somewhere, so the real page replacement algorithm is how pages get taken away from processes and put on the free list (actually, there are four free lists, but for the moment it is simplest to think of there being just one; we will come to the details later). Let us now take a look at how Windows 2000 frees pages. To start with, the entire paging system makes heavy use of the working set concept. Each process (*not* each thread) has a working set. This set consists of the mapped-in pages that are in memory and can be thus referenced without a page fault. The size and composition of the working set fluctuates as the process' threads run, of course.

Each process' working set is described by two parameters: the minimum size and the maximum size. These are not hard bounds, so a process may have fewer pages in memory than its minimum or (under certain circumstances) more than its maximum. Every process starts with the same minimum and maximum, but these

bounds can change over time. The default initial minimum is in the range 20–50 and the default initial maximum is in the range 45–345, depending on the total amount of RAM. The system administrator can change these defaults, however.

If a page fault occurs and the working set is smaller than the minimum, the page is added. On the other hand, if a page fault occurs and the working set is larger than the maximum, a page is evicted from the working set (but not from memory) to make room for the new page. This algorithm means that Windows 2000 uses a local algorithm, to prevent one process from hurting others by hogging memory. However, the system does try to tune itself to some extent. For example, if it observes that one process is paging like crazy (and the others are not), the system may increase the size of its maximum working set, so that over time, the algorithm is a mix of local and global. There is an absolute limit on the working set size, however: even if there is only one process running, it may not take the last 512 pages, to leave some slack for new processes.

So far, so good, but the story is not over yet. Once a second, a dedicated kernel daemon thread, the **balance set manager**, checks to see if there are enough free pages. If there are not enough, it starts the **working set manager** thread to examine the working sets and recover more pages. The working set manager first determines the order to examine the processes in. Large processes that have been idle for a long time are considered before small active processes and the foreground process is considered last.

The working set manager then starts inspecting processes in the chosen order. If a process' working set is currently less than its minimum or it has incurred more than a certain number of page faults since the last inspection, it is passed over. Otherwise, one or more pages are removed. The target number of pages to remove is a complicated function of the total RAM size, how tight memory is, how the current working set size compares to the process' minimum and maximum, and other parameters. All the pages are examined in turn.

On a uniprocessor, if a page's reference bit is clear, a counter associated with the page is incremented. If the reference bit is set, the counter is set to zero. After the scan, the pages with the highest counters are removed from the working set. The thread continues examining processes until it has recovered enough pages, then it stops. If a complete pass through all processes still has not recovered enough pages, it makes another pass, trimming more aggressively, even reducing working sets below their minimum if necessary.

On a multiprocessor, looking at the reference bit does not work because although the current CPU may not have touched the page recently, some other one may have. Examining another CPU's reference bits is too expensive to do. Consequently, the reference bit is not examined and the oldest pages are removed.

It should be noted that for page replacement purposes, the operating system itself is regarded as a process. It owns pages and also has a working set. This working set can be trimmed. However, parts of the code and the nonpaged pool are locked in memory and cannot be paged out under any circumstances.

IBM Ex-1009
Page 504

Physical Memory Management

Above we mentioned that there were actually four free lists. Now it is time to see what all of them are for. Every page in memory is either in one or more working sets or on exactly one of these four lists, which are illustrated in Fig. 11-27. The standby (clean) and modified (dirty) lists hold pages that have recently been evicted from a working set, are still in memory, and are still associated with the process that was using them. The difference between them is that clean pages have a valid copy on disk and can thus be abandoned at will, whereas dirty pages do not have an up-to-date copy on disk. The free list consists of clean pages that are no longer associated with any process. The pages on the zeroed page list are not associated with any process and are also filled with zeros. A fifth list holds any physically defective RAM pages that may exist to make sure that they are not used for anything.

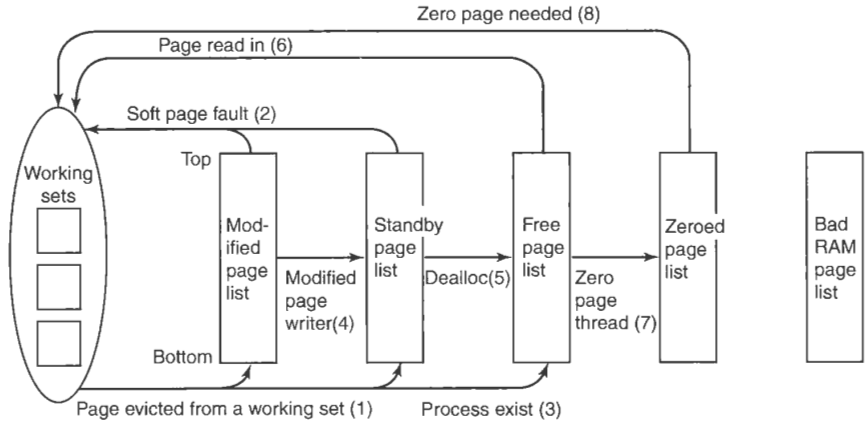


Figure 11-27. The various page lists and the transitions between them.

Pages are moved between the working sets and the various lists by the working set manager and other kernel daemon threads. Let us examine the transitions. When the working set manager removes a page from a working set, the page goes on the bottom of the standby or modified list, depending on its state of cleanliness. This transition is shown as (1). Pages on both lists are still valid pages, so if a page fault occurs and one of these pages is needed, it is removed from the list and faulted back into the working set without any disk I/O (2). When a process exits, its nonshared pages cannot be faulted back to it, so they go on the free list (3). These pages are no longer associated with any process.

Other transitions are caused by other daemon threads. Every 4 seconds the **swapper thread** runs and looks for processes all of whose threads have been idle

IBM Ex-1009
Page 505

for a certain number of seconds. If it finds any such processes, their kernel stacks are unpinned and their pages are moved to the standby or modified lists, also shown as (1).

Two other daemon threads, the **mapped page writer** and the **modified page writer**, wake up periodically to see if there are enough clean pages. If there are not, they take pages from the top of the modified list, write them back to disk, and then move them to the standby list (4). The former handles writes to mapped files and the latter handles writes to the paging files. The result of these writes is to transform dirty pages into clean pages.

The reason for having two threads is that a mapped file might have to grow as a result of the write, and growing it requires access to on-disk data structures to allocate a free disk block. If there is no room in memory to bring them in when a page has to be written, a deadlock could result. The other thread can solve the problem by writing out pages to a paging file, which never grows. Nobody ever said Windows 2000 was simple.

The other transitions in Fig. 11-27 are as follows. If a process unmaps a page, the page is no longer associated with a process and can go on the free list (5), except for the case that it is shared. When a page fault requires a page frame to hold the page about to be read in, the page frame is taken from the free list (6), if possible. It does not matter that the page may still contain confidential information because it is about to be overwritten in its entirety. The situation is different when a stack grows.

In that case, an empty page frame is needed and the security rules require the page to contain all zeros. For this reason, another kernel daemon thread, the **zero page thread**, runs at the lowest priority (see Fig. 11-19), erasing pages that are on the free list and putting them on the zeroed page list (7). Whenever the CPU is idle and there are free pages, they might as well be zeroed since a zeroed page is potentially more useful than a free page.

The existence of all these lists leads to some subtle policy choices. For example, suppose that a page has to be brought in from disk and the free list is empty. The system is now forced to choose between taking a clean page from the standby list (which might otherwise have been faulted back in later) or an empty page from the zeroed page list (throwing away the work done in zeroing it). Which is better? If the CPU is idle a lot and the zero page thread gets to run often, taking a zeroed page is better because there is no shortage of them. However, if the CPU is always busy and the disk is mostly idle, it is better to take a page from the standby list to avoid the CPU cost of having to zero another page later if a stack grows.

Another puzzle. How aggressively should the daemons move pages from the modified list to the standby list? Having clean pages around is better than having dirty pages around (since they can be reused instantly), but an aggressive cleaning policy means more disk I/O and there is some chance that a newly-cleaned page may be faulted back into a working set and dirtied again.

IBM Ex-1009
Page 506

In general, Windows 2000 resolves these kinds of conflicts through complex heuristics, guesswork, historical precedent, rules of thumb, and administrator-controlled parameter settings. Furthermore, the code is so complex that the designers are loathe to touch parts of it for fear of breaking something somewhere else in the system that nobody really understands any more.

To keep track of all the pages and all the lists, Windows maintains a page frame database with as many entries as there are RAM pages, as shown in Fig. 11-28. This table is indexed by physical page frame number. The entries are fixed length, but different formats are used for different kinds of entries (e.g., valid versus invalid). Valid entries maintain the page's state and a count of how many page tables point to the page, so the system can tell when the page is no longer in use. Pages that are in a working set tell which one. There is also a pointer to the page table pointing to the page, if any (shared pages are handled specially), a link to the next page on the list (if any), and various other fields and flags, such as read in progress, write in progress, etc.

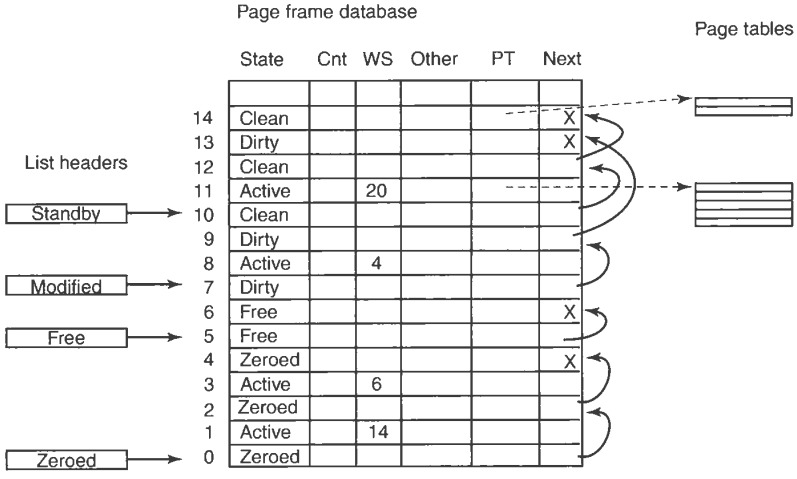


Figure 11-28. Some of the major fields in the page frame database for a valid page.

All in all, memory management is a highly complex subsystem with many data structures, algorithms, and heuristics. It attempts to be largely self tuning, but there are also many knobs that administrators can tweak to affect system performance. A number of these knobs and the associated counters can be viewed using tools in the various tool kits mentioned earlier. Probably the most important thing to remember here is that memory management in real systems is a lot more than just one simple paging algorithm like clock or aging.

IBM Ex-1009
Page 507

11.6 INPUT/OUTPUT IN WINDOWS 2000

The goal of the Windows 2000 I/O system is to provide a framework for efficiently handling a very wide variety of I/O devices. Current input devices include various kinds of keyboards, mice, touch pads, joysticks, scanners, still cameras, television cameras, bar code readers, microphones, and laboratory rats. Current output devices include monitors, printers, plotters, beamers, CD-recorders, and sound cards. Storage devices include floppy disks, IDE and SCSI hard disks, CD-ROMs, DVDs, Zip drives, and tape drives. Finally, other devices include clocks, networks, telephones, and camcorders. No doubt many new I/O devices will be invented in the years to come, so Window 2000 has been designed with a general framework to which new devices can easily be attached. In the following sections we will examine some of the issues relating to I/O.

11.6.1 Fundamental Concepts

The I/O manager is on intimate terms with the plug-and-play manager. The basic idea behind plug and play is that of an enumerable bus. Many buses, including PC Card, PCI, USB, IEEE 1394, and SCSI, have been designed so that the plug-and-play manager can send a request to each slot and ask the device there to identify itself. Having discovered what is out there, the plug-and-play manager allocates hardware resources, such as interrupt levels, locates the appropriate drivers, and loads them into memory. As each one is loaded, a **driver object** is created for it. For some buses, such as SCSI, enumeration happens only at boot time, but for other buses, such as USB and IEEE 1394, it can happen at any moment, requiring close contact between the plug-and-play manager, the bus driver (which actually does the enumeration), and the I/O manager.

The I/O manager is also closely connected with the power manager. The power manager can put the computer into any of six states, roughly described as:

1. Fully operational.
2. Sleep-1: CPU power reduced, RAM and cache on; instant wake-up.
3. Sleep-2: CPU and RAM on; CPU cache off; continue from current PC.
4. Sleep-3: CPU and cache off; RAM on; restart from fixed address.
5. Hibernate: CPU, cache, and RAM off; restart from saved disk file.
6. Off: Everything off; full reboot required.

I/O devices can also be in various power states. Turning them on and off is handled by the power manager and I/O manager together. Note that states 2 through 6 are only used when the CPU has been idle for a shorter or longer time interval.

IBM Ex-1009
Page 508

Somewhat surprisingly, all the file systems are technically I/O drivers. Requests for data blocks from user processes are initially sent to the cache manager. If the cache manager cannot satisfy the request from the cache, it has the I/O manager call the proper file system driver to go get the block it needs from disk.

An interesting feature of Windows 2000 is its support for **dynamic disks**. These disks may span multiple partitions and even multiple disks and may be reconfigured on the fly, without even having to reboot. In this way, logical volumes are no longer constrained to a single partition or even a single disk so that a single file system may span multiple drives in a transparent way.

Another interesting aspect of Windows 2000 is its support for asynchronous I/O. It is possible for a thread to start an I/O operation and then continue executing in parallel with the I/O. This feature is especially important on servers. There are various ways the thread can find out that the I/O has completed. One is to specify an event object at the time the call is made and then wait on it eventually. Another is to specify a queue to which a completion event will be posted by the system when the I/O is done. A third is to provide a callback procedure that the system calls when the I/O has completed.

11.6.2 Input/Output API Calls

Windows 2000 has over 100 separate APIs for a wide variety of I/O devices, including mice, sound cards, telephones, tape drives, etc. Probably the most important is the graphics system, for which there are thousands of Win32 API calls. In Sec. 5.7.3 we began our discussion of the Window graphical system. Here we will continue, mentioning a few of the Win32 API categories, each of which has many calls. A brief summary of the categories are given in Fig. 11-29. As we mentioned in Chap. 5, multiple 1500-page books have been written on the graphics portion of the Win32 API.

Win32 calls exist to create, destroy, and manage windows. Windows have a vast number of styles and options that can be specified, including titles, borders, colors, sizes, and scroll bars. Windows can be fixed or movable, of constant size or resizable. Their properties can be queried and messages can be sent to them.

Many windows contain menus, so there are Win32 calls for creating and deleting menus and menu bars. Dynamic menus can be popped up and removed. Menu items can be highlighted, dimmed out, or cascaded.

Dialog boxes are popped up to inform the user of some event or ask a question. They may contain buttons, sliders, or text fields to be filled in. Sounds can also be associated with dialog boxes, for example for warning messages.

There are hundreds of drawing and painting functions available, ranging from setting a single pixel to doing complex region clipping operations. Many calls are provided for drawing lines and closed geometric figures of various kinds, with detailed control over textures, colors, widths, and many other attributes.

IBM Ex-1009
Page 509

API group	Description
Window management	Create, destroy, and manage windows,
Menus	Create, destroy, and append to menus and menu bars
Dialog boxes	Pop up a dialog box and collect information
Painting and drawing	Display points, lines, and geometric figures
Text	Display text in some font, size, and color
Bitmaps and icons	Placement of bitmaps and icons on the screen
Colors and palettes	Manage the set of colors available
The clipboard	Pass information from one application to another
Input	Get information from the mouse and keyboard

Figure 11-29. Some categories of Win32 API calls.

Another group of calls relates to displaying text. Actually, the text display call, TextOut, is straightforward. It is the management of the color, point sizes, typefaces, character widths, glyphs, kerning, and other typesetting details where the complexity comes in. Fortunately, the rasterization of text (conversion to bitmaps) is generally done automatically.

Bitmaps are small rectangular blocks of pixels that can be placed on the screen using the BitBlt Win32 call. They are used for icons and occasionally text. Various calls are provided for creating, destroying, and managing icon objects.

Many displays use a color mode with only 256 or 65,536 of the 2^{24} possible colors in order to represent each pixel with only 1 or 2 bytes, respectively. In these cases a color palette is needed to determine which 256 or 65,536 colors are available. The calls in this group create, destroy, and manage palettes, select the nearest available color to a given color, and try to make colors on the screen match colors on color printers.

Many Windows 2000 programs allow the user to select some data (e.g., a block of text, part of a drawing, a set of cells in a spreadsheet), put it on the clipboard, and allow it to be pasted into another application. The clipboard is generally used for this transmission. Many clipboard formats are defined, including text, bitmaps, objects, and metafiles. The latter are sets of Win32 calls that when executed draw something, allowing arbitrary drawings to be cut and pasted. This group of calls puts things on the clipboard, takes things off the clipboard, and generally manages it.

Finally, we come to input. There are no Win32 calls for GUI applications for reading input from the keyboard because GUI applications are event driven. The main program consists of a big loop getting input messages. When the user types something interesting, a message is sent to the program telling it what just came in. On the other hand, there are calls relating to the mouse, such as reading its (x, y) position and the state of its buttons. Some of the input calls are actually output

calls, though, such as selecting a mouse cursor icon and moving it around the screen (basically, this is output to the screen). For nonGUI applications, it is possible to read from the keyboard.

11.6.3 Implementation of I/O

We could go on more-or-less indefinitely about the Win32 graphics calls, but now it is time to look at how the I/O manager implements graphics and other I/O functions. The main function of the I/O manager is to create a framework in which different I/O devices can operate. The basic structure of the framework is a set of device-independent procedures for certain aspects of I/O plus a set of loaded device drivers for communicating with the devices.

11.6.4 Device Drivers

To make sure that device drivers work well with the rest of Windows 2000, Microsoft has defined a **Windows Driver Model** that device drivers are expected to conform with. Furthermore, it also has provided a tool kit that is designed to help driver writers produce conformant drivers. In this section we will briefly examine this model. Conformant drivers must meet all of the following requirements as well as some others:

1. Handle incoming I/O requests, which arrive in a standard format.
2. Be as object based as the rest of Windows 2000.
3. Allow plug-and-play devices to be dynamically added or removed.
4. Permit power management, where applicable.
5. Be configurable in terms of resource usage.
6. Be reentrant for use on multiprocessors.
7. Be portable across Windows 98 and Windows 2000

I/O Requests are passed to drivers in the form of a standardized packet called an **IRP** (**I/O Request Packet**). Conformant drivers must be able to handle them. Drivers must be object based in the sense of supporting a specific list of methods that the rest of the system can call. They must also correctly deal with other Windows 2000 objects when given an object handle to deal with.

Conformant drivers must fully support plug and play, which means that if a device managed by the driver is suddenly added or removed from the system, the driver must be prepared to accept this information and act on it, even in the case that the device currently being accessed is suddenly removed. Power management must also be supported for devices for which this is relevant. For example, if the system decides it is now time to go into a low-power hibernation mode, all

devices that are capable of doing this must do so to save energy. They must also wake up when told to do so.

Drivers must be configurable, which means not having any built-in assumptions about which interrupt lines or I/O ports certain devices use. For example, the printer port on the IBM PC and its successors has been at address 0x378 for more than 20 years and it is unlikely to change now. But a printer driver that has this address hard coded into it is not conformant.

Being multiprocessor safe is also a requirement as Windows 2000 was designed for use on multiprocessors. Concretely, this requirement means while a driver is actively running and processing one request on behalf of one CPU, a second request may come in on behalf of a different CPU. The second CPU may begin executing the driver code simultaneously with the first one. The driver must function correctly even when being executed concurrently by two or more CPUs, which implies that all sensitive data structures may only be accessed from inside critical regions. Just assuming that there will not be any other calls until the current one is finished is not permitted.

Finally, conformant drivers must work not only on Windows 2000 but also on Windows 98. It may be necessary to recompile the driver on each system however, and use of C preprocessor commands to isolate platform dependencies is permitted.

In UNIX, drivers are located by using their major device numbers. Windows 2000 uses a different scheme. At boot time, or when a new hot pluggable plug-and-play device is attached to the computer, Windows 2000 automatically detects it and calls the plug-and-play manager. The manager queries the device to find out what the manufacturer and model number are. Equipped with this knowledge, it looks on the hard disk in a certain directory to see if it has the driver. If it does not, it displays a dialog box asking the user to insert a floppy disk or CD-ROM with the driver. Once the driver is located, it is loaded into memory.

Each driver must supply a set of procedures that can be called to get its services. The first one, called *DriverEntry*, initializes the driver. It is called just after the driver is loaded. It may create tables and data structures, but must not touch the device yet. It also fills in some of the fields of the driver object created by the I/O manager when the driver was loaded. The fields in the driver object include pointers to all the other procedures that drivers must supply. In addition, for each device controlled by the driver (e.g., each IDE disk controlled by the IDE disk driver), a **device object** is created and initialized to point to the driver object. These driver objects are entered into a special directory, *\??*. Given a device object, the driver object can be located easily, and hence its methods can be called.

A second required procedure is *AddDevice*, which is called once (by the plug-and-play manager) for each device to be added. Once this has been accomplished, the driver is called with the first IRP, which sets up the interrupt vector and actually initializes the hardware. Other procedures that drivers must contain

are the interrupt service procedure, various timer management procedures, a fast I/O path, DMA control, a way to cancel currently executing requests, and many more. All in all, Windows 2000 drivers are so complex that multiple books have been written about them (Cant, 1999; Oney, 1999; and Viscarola and Mason, 1999).

A driver in Windows 2000 may do all the work by itself, as the printer driver does in Fig. 11-30 (just as an example). On the other hand, drivers may also be stacked, which means that a request may pass through a sequence of drivers, each doing part of the work. Two stacked drivers are also illustrated in Fig. 11-30.

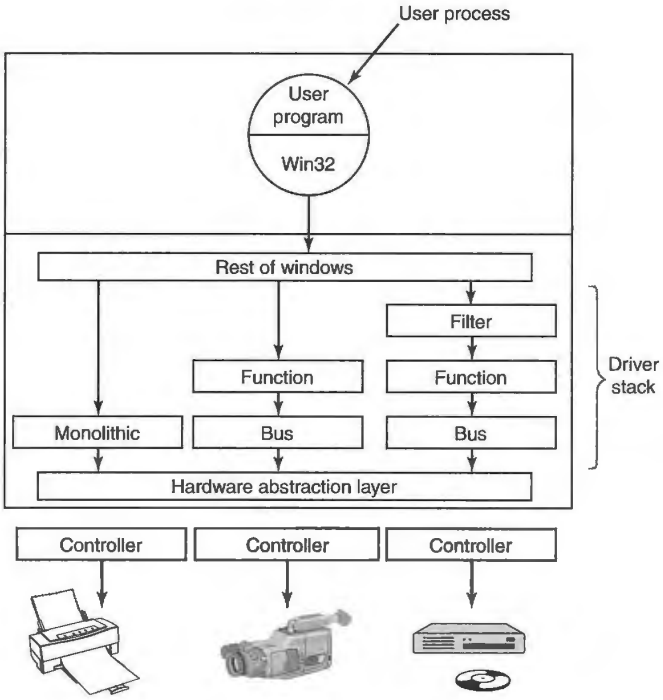


Figure 11-30. Windows 2000 allows drivers to be stacked.

One common use for stacked drivers is to separate out the bus management from the functional work of actually controlling the device. Bus management on the PCI bus is quite complicated on account of many kinds of modes and bus transactions, and by separating this work from the device-specific part, driver writers are freed from learning how to control the bus. They can just use the standard bus driver in their stack. Similarly, USB and SCSI drivers have a device-specific part and a generic part, with common drivers used for the generic part.

Another use of stacking drivers is to be able to insert **filter drivers** into the stack. A filter driver performs some transformation on the data on the way up or down. For example, a filter driver could compress data on the way to the disk or encrypt data on the way to the network. Putting the filter here means that neither the application program nor the true device driver have to be aware of it and it works automatically for all data going to (or coming from) the device.

11.7 THE WINDOWS 2000 FILE SYSTEM

Windows 2000 supports several file systems, the most important of which are **FAT-16**, **FAT-32**, and **NTFS (NT File System)**. FAT-16 is the old MS-DOS file system. It uses 16-bit disk addresses, which limits it to disk partitions no larger than 2 GB. FAT-32 uses 32-bit disk addresses and supports disk partitions up to 2 TB. NTFS is a new file system developed specifically for Windows NT and carried over to Windows 2000. It uses 64-bit disk addresses and can (theoretically) support disk partitions up to 2^{64} bytes, although other considerations limit it to smaller sizes. Windows 2000 also supports read-only file systems for CD-ROMs and DVDs. It is possible (even common) to have the same running system have access to multiple file system types available at the same time.

In this chapter we will treat the NTFS file system because it is a modern file system unencumbered by the need to be fully compatible with the MS-DOS file system, which was based on the CP/M file system designed for 8-inch floppy disks more than 20 years ago. Times have changed and 8-inch floppy disks are not quite state of the art any more. Neither are their file systems. Also, NTFS differs both in user interface and implementation in a number of ways from the UNIX file system, which makes it a good second example to study. NTFS is a large and complex system and space limitations prevent us from covering all of its features, but the material presented below should give a reasonable impression of it.

11.7.1 Fundamental Concepts

Individual file names in NTFS are limited to 255 characters; full paths are limited to 32,767 characters. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Greece, Japan, India, Russia, and Israel) to write file names in their native language. For example, φιλε is a perfectly legal file name. NTFS fully supports case sensitive names (so *foo* is different from *Foo* and *FOO*). Unfortunately, the Win32 API does not fully support case-sensitivity for file names and not at all for directory names, so this advantage is lost to programs restricted to using Win32 (e.g., for Windows 98 compatibility).

IBM Ex-1009
Page 514

An NTFS file is not just a linear sequence of bytes, as FAT-32 and UNIX files are. Instead, a file consists of multiple attributes, each of which is represented by a stream of bytes. Most files have a few short streams, such as the name of the file and its 64-bit object ID, plus one long (unnamed) stream with the data. However, a file can also have two or more (long) data streams as well. Each stream has a name consisting of the file name, a colon, and the stream name, as in *foo:stream1*. Each stream has its own size and is lockable independently of all the other streams. The idea of multiple streams in a file was borrowed from the Apple Macintosh, in which files have two streams, the data fork and the resource fork. This concept was incorporated into NTFS to allow an NTFS server be able to serve Macintosh clients.

File streams can be used for purposes other than Macintosh compatibility. For example, a photo editing program could use the unnamed stream for the main image and a named stream for a small thumbnail version. This scheme is simpler than the traditional way of putting them in the same file one after another. Another use of streams is in word processing. These programs often make two versions of a document, a temporary one for use during editing and a final one when the user is done. By making the temporary one a named stream and the final one the unnamed stream, both versions automatically share a file name, security information, timestamps, etc. with no extra work.

The maximum stream length is 2^{64} bytes. To get some idea of how big a 2^{64}-byte stream is, imagine that the stream were written out in binary, with each of the 0s and 1s in each byte occupying 1 mm of space. The 2^{67}-mm listing would be 15 light-years long, reaching far beyond the solar system, to Alpha Centauri and back. File pointers are used to keep track of where a process is in each stream, and these are 64 bits wide to handle the maximum length stream, which is about 18.4 exabytes.

The Win32 API function calls for file and directory manipulation are roughly similar to their UNIX counterparts, except most have more parameters and the security model is different. Opening a file returns a handle, which is then used for reading and writing the file. For graphical applications, no file handles are predefined. Standard input, standard output, and standard error have to be acquired explicitly if needed; in console mode they are preopened, however. Win32 also has a number of additional calls not present in UNIX.

11.7.2 File System API Calls in Windows 2000

The principal Win32 API functions for file management are listed in Fig. 11-31. There are actually many more, but these give a reasonable first impression of the basic ones. Let us now examine these calls briefly. CreateFile can be used to create a new file and return a handle to it. This API function must also be used to open existing files as there is no FileOpen API function. We have not listed the

Win32 API function	UNIX	Description
CreateFile	open	Create a file or open an existing file; return a handle
DeleteFile	unlink	Destroy an existing file
CloseHandle	close	Close a file
ReadFile	read	Read data from a file
WriteFile	write	Write data to a file
SetFilePointer	lseek	Set the file pointer to a specific place in the file
GetFileAttributes	stat	Return the file properties
LockFile	fcntl	Lock a region of the file to provide mutual exclusion
UnlockFile	fcntl	Unlock a previously locked region of the file

Figure 11-31. The principal Win32 API functions for file I/O. The second column gives the nearest UNIX equivalent.

parameters for the API functions because they are so voluminous. As an example, CreateFile has seven parameters, which are roughly summarized as follows:

1. A pointer to the name of the file to create or open.
2. Flags telling whether the file can be read, written, or both.
3. Flags telling whether multiple processes can open the file at once.
4. A pointer to the security descriptor, telling who can access the file.
5. Flags telling what to do if the file exists/does not exist.
6. Flags dealing with attributes such as archiving, compression, etc.
7. The handle of a file whose attributes should be cloned for the new file.

The next six API functions in Fig. 11-31 are fairly similar to the corresponding UNIX system calls. The last two allow a region of a file to be locked and unlocked to permit a process to get guaranteed mutual exclusion to it.

Using these API functions, it is possible to write a procedure to copy a file, analogous to the UNIX version of Fig. 6-5. Such a code fragment (without any error checking) is shown in Fig. 11-32. It has been designed to mimic our UNIX version. In practice, one would not have to program a copy file program since CopyFile is an API function (which executes something close to this program as a library procedure).

Windows 2000 NTFS is a hierarchical file system, similar to the UNIX file system. The separator between component names is \ however, instead of /, a fossil inherited from MS-DOS. There is a concept of a current working directory and path names can be relative or absolute. Hard and symbolic links are supported,

IBM Ex-1009
Page 516

```
/* Open files for input and output. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);

/* Copy the file. */
do {
    s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
    if (s && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Close the files. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Figure 11-32. A program fragment for copying a file using the Windows 2000 API functions.

the former implemented by having multiple directory entries, as in UNIX, and the latter implemented using reparse points (discussed later in this chapter). In addition, compression, encryption, and fault tolerance are also supported. These features and their implementations will be discussed later in this chapter.

The major directory management API functions are given in Fig. 11-33, again along with their nearest UNIX equivalents. The functions should be self explanatory.

Win32 API function	UNIX	Description
CreateDirectory	mkdir	Create a new directory
RemoveDirectory	rmdir	Remove an empty directory
FindFirstFile	opendir	Initialize to start reading the entries in a directory
FindNextFile	readdir	Read the next directory entry
MoveFile	rename	Move a file from one directory to another
SetCurrentDirectory	chdir	Change the current working directory

Figure 11-33. The principal Win32 API functions for directory management. The second column gives the nearest UNIX equivalent, when one exists.

11.7.3 Implementation of the Windows 2000 File System

NTFS is a highly complex and sophisticated file system. It was designed from scratch, rather than being an attempt to improve the old MS-DOS file system. Below we will examine a number of its features, starting with its structure, then moving on to file name lookup, file compression, and file encryption.

IBM Ex-1009
Page 517

File System Structure

Each NTFS volume (e.g., disk partition) contains files, directories, bitmaps, and other data structures. Each volume is organized as a linear sequence of blocks (clusters in Microsoft's terminology), with the block size being fixed for each volume and ranging from 512 bytes to 64 KB, depending on the volume size. Most NTFS disks use 4-KB blocks as a compromise between large blocks (for efficient transfers) and small blocks (for low internal fragmentation). Blocks are referred to by their offset from the start of the volume using 64-bit numbers.

The main data structure in each volume is the **MFT (Master File Table)**, which is a linear sequence of fixed-size 1-KB records. Each MFT record describes one file or directory. It contains the file's attributes, such as its name and timestamps, and the list of disk addresses where its blocks are located. If a file is extremely large, it is sometimes necessary to use two or more MFT records to contain the list of all the blocks, in which case the first MFT record, called the **base record**, points to the other MFT records. This overflow scheme dates back to CP/M, where each directory entry was called an extent. A bitmap keeps track of which MFT entries are free.

The MFT is itself a file and as such can be placed anywhere within the volume, thus eliminating the problem with defective sectors in the first track. Furthermore, the file can grow as needed, up to a maximum size of 2^{48} records.

The MFT is shown in Fig. 11-34. Each MFT record consists of a sequence of (attribute header, value) pairs. Each attribute begins with a header telling which attribute this is and how long the value is because some attribute values are variable length, such as the file name and the data. If the attribute value is short enough to fit in the MFT record, it is placed there. If it is too long, it is placed elsewhere on the disk and a pointer to it is placed in the MFT record.

The first 16 MFT records are reserved for NTFS metadata files, as shown in Fig. 11-34. Each of the records describes a normal file that has attributes and data blocks, just like any other file. Each of these files has a name that begins with a dollar sign to indicate that it is a metadata file. The first record describes the MFT file itself. In particular, it tells where the blocks of the MFT file are located so the system can find the MFT file. Clearly, Windows 2000 needs a way to find the first block of the MFT file in order to find the rest of the file system information. The way it finds the first block of the MFT file is to look in the boot block, where its address is installed at system installation time.

Record 1 is a duplicate of the early part of the MFT file. This information is so precious that having a second copy can be critical in the event one of the first blocks of the MFT ever goes bad. Record 2 is the log file. When structural changes are made to the file system, such as adding a new directory or removing an existing one, the action is logged here before it is performed, in order to increase the chance of correct recovery in the event of a failure during the operation. Changes to file attributes are also logged here. In fact, the only changes not

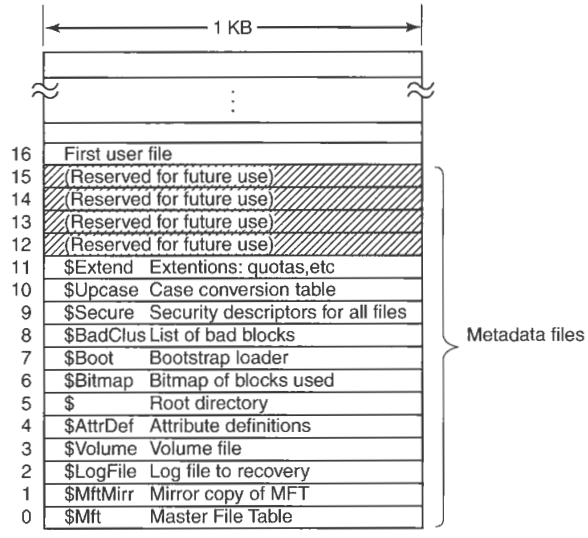


Figure 11-34. The NTFS master file table.

logged here are changes to user data. Record 3 contains information about the volume, such as its size, label, and version.

As mentioned above, each MFT record contains a sequence of (attribute header, value) pairs. The *$AttrDef* file is where the attributes are defined. Information about this file is in MFT record 4. Next comes the root directory, which itself is a file and can grow to arbitrary length. It is described by MFT record 5.

Free space on the volume is kept track of with a bitmap. The bitmap is itself a file and its attributes and disk addresses are given in MFT record 6. The next MFT record points to the bootstrap loader file. Record 8 is used to link all the bad blocks together to make sure they never occur in a file. Record 9 contains the security information. Record 10 is used for case mapping. For the Latin letters A-Z case mapping is obvious (at least for people who speak Latin). Case mapping for other languages, such as Greek, Armenian, or Georgian (the country, not the state), is less obvious to Latin speakers, so this file tells how to do it. Finally, record 11 is a directory containing miscellaneous files for things like disk quotas, object identifiers, reparse points, and so on. The last 4 MFT records are reserved for future use.

Each MFT record consists of a record header followed by a sequence of (attribute header, value) pairs. The record header contains a magic number used for validity checking, a sequence number updated each time the record is reused for a new file, a count of references to the file, the actual number of bytes in the record

IBM Ex-1009
Page 519

used, the identifier (index, sequence number) of the base record (used only for extension records), and some other miscellaneous fields. Following the record header comes the header of the first attribute, then the first attribute value, the second attribute header, the second attribute value, and so on.

NTFS defines 13 attributes that can appear in MFT records. These are listed in Fig. 11-35. Each MFT record consists of a sequence of attribute headers, each of which identifies the attribute it is heading and gives the length and location of the value field along with a variety of flags and other information. Usually, attribute values follow their attribute headers directly, but if a value is too long to fit in the MFT record, it may be put in a separate disk block. Such an attribute is said to be a **nonresident attribute**. The data attribute is an obvious candidate. Some attributes, such as the name, may be repeated, but all attributes must appear in a fixed order in the MFT record. The headers for resident attributes are 24 bytes long; those for nonresident attributes are longer because they contain information about where to find the attribute on disk.

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

Figure 11-35. The attributes used in MFT records.

The standard information field contains the file owner, security information, the timestamps needed by POSIX, the hard link count, the read-only and archive bits, etc. It is a fixed-length field and is always present. The file name is variable length in Unicode. In order to make files with nonMS-DOS names accessible to old 16-bit programs, files can also have an 8 + 3 MS-DOS name. If the actual file name conforms to the MS-DOS 8 + 3 naming rule, a secondary MS-DOS name is not used.

In NT 4.0, security information could be put in an attribute, but in Windows 2000 it all goes into a single file so that multiple files can share the same security

descriptions. The attribute list is needed in case the attributes do not fit in the MFT record. This attribute then tells where to find the extension records. Each entry in the list contains a 48-bit index into the MFT telling where the extension record is and a 16-bit sequence number to allow verification that the extension record and base records match up.

The object ID attribute gives the file a unique name. This is sometimes needed internally. The reparse point tells the procedure parsing the file name to do something special. This mechanism is used for mounting and symbolic links. The two volume attributes are only used for volume identification. The next three attributes deal with how directories are implemented. Small ones are just lists of files but large ones are implemented using B+ trees. The logged utility stream attribute is used by the encrypting file system.

Finally, we come to the attribute that everyone has been waiting for: the data. The stream name, if present, goes in this attribute header. Following the header is either a list of disk addresses telling which blocks the file contains, or for files of only a few hundred bytes (and there are many of these), the file itself. Putting the actual file data in the MFT record is called an **immediate file** (Mullender and Tanenbaum, 1987).

Of course, most of the time the data does not fit in the MFT record, so this attribute is usually nonresident. Let us now take a look at how NTFS keeps track of the location of nonresident attributes, in particular data.

The model for keeping track of disk blocks is that they are assigned in runs of consecutive blocks, where possible, for efficiency reasons. For example, if the first logical block of a file is placed in block 20 on the disk, then the system will try hard to place the second logical block in block 21, the third logical block in 22, and so on. One way to achieve these runs is to allocate disk storage several blocks at a time, if possible.

The blocks in a file are described by a sequence of records, each one describing a sequence of logically contiguous blocks. For a file with no holes in it, there will be only one such record. Files that are written in order from beginning to end all belong in this category. For a file with one hole in it (e.g., only blocks 0–49 and blocks 60–79 are defined), there will be two records. Such a file could be produced by writing the first 50 blocks, then seeking forward to logical block 60 and writing another 20 blocks. When a hole is read back, all the missing bytes are zeros.

Each record begins with a header giving the offset of the first block within the file. Next comes the offset of the first block not covered by the record. In the example above, the first record would have a header of (0, 50) and would provide the disk addresses for these 50 blocks. The second one would have a header of (60,80) and would provide the disk addresses for these 20 blocks.

Each record header is followed by one or more pairs, each giving a disk address and run length. The disk address is the offset of the disk block from the start of its partition; the run length is the number of blocks in the run. As many

pairs as needed can be in the run record. Use of this scheme for a three-run, nine-block file is illustrated in Fig. 11-36.

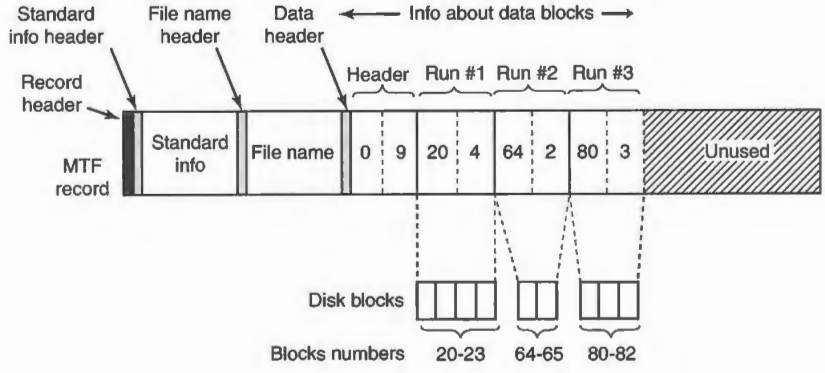


Figure 11-36. An MFT record for a three-run, nine-block file.

In this figure we have an MFT record for a short file (short here means that all the information about the file blocks fits in one MFT record). It consists of the three runs of consecutive blocks on the disk. The first run is blocks 20-23, the second is blocks 64-65, and the third is blocks 80-82. Each of these runs is recorded in the MFT record as a (disk address, block count) pair. How many runs there are depends on how good a job the disk block allocator did in finding runs of consecutive blocks when the file was created. For a n-block file, the number of runs can be anything from 1 up to and including n.

Several comments are worth making here. First, there is no upper limit to the size of files that can be represented this way. In the absence of address compression, each pair requires two 64-bit numbers in the pair for a total of 16 bytes. However, a pair could represent 1 million or more consecutive disk blocks. In fact, a 20-MB file consisting of 20 separate runs of 1 million 1-KB blocks each fits easily in one MFT record, whereas a 60-KB file scattered into 60 isolated blocks does not.

Second, while the straightforward way of representing each pair takes 2×8 bytes, a compression method is available to reduce the size of the pairs below 16. Many disk addresses have multiple high-order zero-bytes. These can be omitted. The data header tells how many are omitted, that is, how many bytes are actually used per address. Other kinds of compression are also used. In practice, the pairs are often only 4 bytes.

Our first example was easy: all the file information fit in one MFT record. What happens if the file is so large or highly fragmented that the block information does not fit in one MFT record? The answer is simple: use two or more MFT records. In Fig. 11-37 we see a file whose base record is in MFT record 102. It

IBM Ex-1009
Page 522

has too many runs for one MFT record, so it computes how many extension records it needs, say, two, and puts their indices in the base record. The rest of the record is used for the first k data runs.

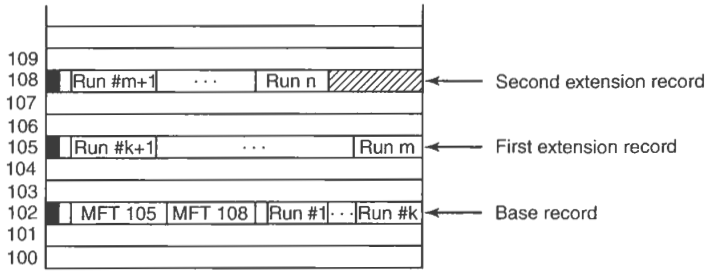


Figure 11-37. A file that requires three MFT records to store all its runs.

Note that Fig. 11-37 contains some redundancy. In theory, it should not be necessary to specify the end of a sequence of runs because this information can be calculated from the run pairs. The reason for "overspecifying" this information is to make seeking more efficient: to find the block at a given file offset, it is only necessary to examine the record headers, not the run pairs.

When all the space in record 102 has been used up, storage of the runs continues with MFT record 105. As many runs are packed in this record as fit. When this record is also full, the rest of the runs go in MFT record 108. In this way many MFT records can be used to handle large fragmented files.

A problem arises if so many MFT records are needed that there is no room in the base MFT to list all their indices. There is also a solution to this problem: the list of extension MFT records is made nonresident (i.e., stored on disk instead of in the base MFT record). Then it can grow as large as needed.

An MFT entry for a small directory is shown in Fig. 11-38. The record contains a number of directory entries, each of which describes one file or directory. Each entry has a fixed-length structure followed by a variable-length file name. The fixed part contains the index of the MFT entry for the file, the length of the file name, and a variety of other fields and flags. Looking for an entry in a directory consists of examining all the file names in turn.

Large directories use a different format. Instead of listing the files linearly, a B+ tree is used to make alphabetical lookup possible and to make it easy to insert new names in the directory in the proper place.

File Name Lookup

We now have enough information to see how file name lookup occurs. When a user program wants to open a file, it typically makes a call like

```
CreateFile("C:\maria\web.htm", ...)
```

IBM Ex-1009
Page 523

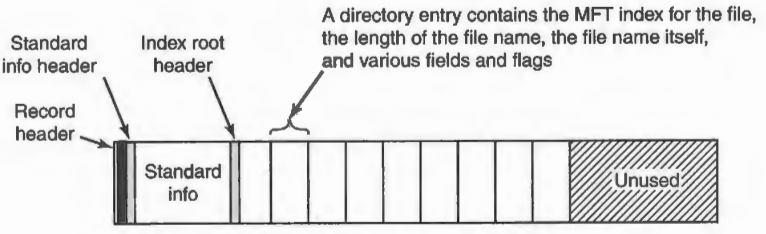


Figure 11-38. The MFT record for a small directory.

This call goes to the user-level shared library, *kernel32.dll*, where *\??* is prepended to the file name giving

```
\??\C:\maria\web.htm
```

It is this name that is passed as a parameter to the system call NtFileCreate.

Then the operating system starts the search at the root of the object manager's name space (see Fig. 11-12). It then looks in the directory *\??* to find *C:*, which it will find. This file is a symbolic link to another part of the object manager's name space, the directory *\Device*. The link typically ends at an object whose name is something like *\Device\HarddiskVolume1*. This object corresponds to the first partition of the first hard disk. From this object it is possible to determine which MFT to use, namely the one on this partition. These steps are shown in Fig. 11-39.

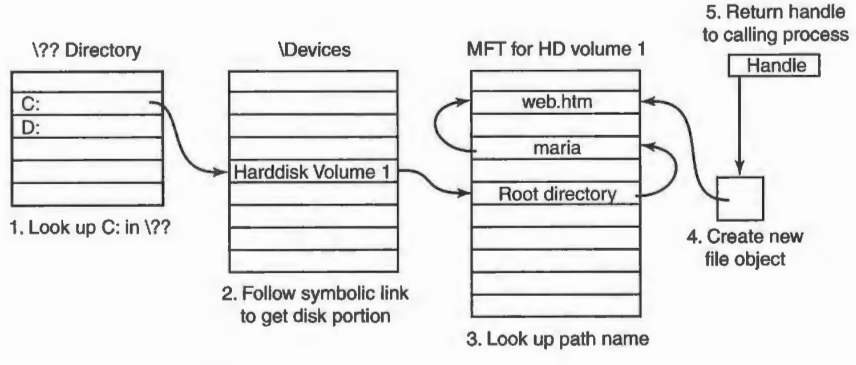


Figure 11-39. Steps in looking up the file *C:\maria\web.htm*.

The parsing of the file name continues now at the root directory, whose blocks can be found from entry 5 in the MFT (see Fig. 11-34). The string "maria" is now looked up in the root directory, which returns the index into the MFT for the directory *maria*. This directory is then searched for the string "web.htm". If

successful, the result is a new object created by the object manager. The object, which is unnamed, contains the index of the MFT record for the file. A handle to this object is returned to the calling process. On subsequent ReadFile calls, the handle is provided, which allows the object manager to find the index and then the contents of the MFT record for the file. If a thread in a second process opens the file again, it gets a handle to a new file object.

In addition to regular files and directories, NTFS supports hard links in the UNIX sense, and also symbolic links using a mechanism called **reparse points**. It is possible to tag a file or directory as a reparse point and associate a block of data with it. When the file or directory is encountered during a file name parse, exception processing is triggered and the block of data is interpreted. It can do various things, including redirecting the search to a different part of the directory hierarchy or even to a different partition. This mechanism is used to support both symbolic links and mounted file systems.

File Compression

NTFS supports transparent file compression. A file can be created in compressed mode, which means that NTFS automatically tries to compress the blocks as they are written to disk and automatically uncompresses them when they are read back. Processes that read or write compressed files are completely unaware of the fact that compression and decompression are going on.

Compression works as follows. When NTFS writes a file marked for compression to disk, it examines the first 16 (logical) blocks in the file, irrespective of how many runs they occupy. It then runs a compression algorithm on them. If the resulting data can be stored in 15 or fewer blocks, the compressed data are written to the disk, preferably in one run, if possible. If the compressed data still take 16 blocks, the 16 blocks are written in uncompressed form. Then blocks 16-31 are examined to see if they can be compressed to 15 blocks or less, and so on.

Figure 11-40(a) shows a file in which the first 16 blocks have successfully compressed to eight blocks, the second 16 blocks failed to compress, and the third 16 blocks have also compressed by 50%. The three parts have been written as three runs and stored in the MFT record. The "missing" blocks are stored in the MFT entry with disk address 0 as shown in Fig. 11-40(b). Here the header (0, 48) is followed by five pairs, two for the first (compressed) run, one for the uncompressed run, and two for the final (compressed) run.

When the file is read back, NTFS has to know which runs are compressed and which are not. It sees that based on the disk addresses. A disk address of 0 indicates that it is the final part of 16 compressed blocks. Disk block 0 may not be used for storing data, to avoid ambiguity. Since it contains the boot sector, using it for data is impossible anyway.

IBM Ex-1009
Page 525

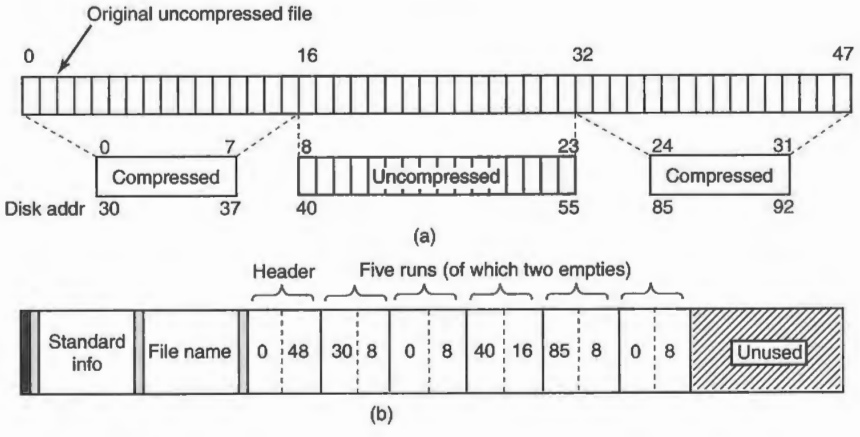


Figure 11-40. (a) An example of a 48-block file being compressed to 32 blocks. (b) The MFT record for the file after compression.

Random access to compressed files is possible, but tricky. Suppose that a process does a seek to block 35 in Fig. 11-40. How does NTFS locate block 35 in a compressed file? The answer is that it has to read and decompress the entire run first. Then it knows where block 35 is and can pass it to any process that reads it. The choice of 16 blocks for the compression unit was a compromise. Making it shorter would have made the compression less effective. Making it longer would have made random access more expensive.

File Encryption

Computers are used nowadays to store all kinds of sensitive data, including plans for corporate takeovers, tax information, and love letters (love email?), which the owners do not especially want revealed to anyone. Information loss can happen when a laptop computer is lost or stolen, a desktop system is rebooted using an MS-DOS floppy disk to bypass Windows 2000 security, or a hard disk is physically removed from one computer and installed on another one with an insecure operating system. Even the simple act of going to the bathroom and leaving the computer unattended and logged in can be a huge security breach.

Windows 2000 addresses these problem by having an option to encrypt files, so even in the event the computer is stolen or rebooted using MS-DOS, the files will be unreadable. The normal way to use Windows 2000 encryption is to mark certain directories as encrypted, which causes all the files in them to be encrypted, and new files moved to them or created in them to be encrypted as well. The actual encryption and decryption is not done by NTFS itself, but by a driver called **EFS (Encrypting File System)**, which is positioned between NTFS and the user

IBM Ex-1009
Page 526

process. In this way, application programs are unaware of encryption and NTFS itself is only partially involved in it.

To understand how the encrypting file system works, it is necessary to understand how modern cryptography works. For this purpose, a brief review was given in Sec. 9.2. Readers not familiar with the basics of cryptography should read that section before continuing.

Now let us see how Windows 2000 encrypts files. When the user asks a file to be encrypted, a random 128-bit file key is generated and used to encrypt the file block by block using a symmetric algorithm parametrized by this key. Each new file encrypted gets a different 128-bit random file key, so no two files use the same encryption key, which increases security in case one key is compromised. The current encryption algorithm is a variant of **DES (Data Encryption Standard)**, but the EFS architecture supports the addition of new algorithms in the future. Encrypting each block independently of all the others is necessary to make random access still possible.

The file key has to be stored somewhere so the file can be decrypted later. If it were just stored on the disk in plaintext, then someone who stole or found the computer could easily decrypt the file, defeating the purpose of encrypting the files. For this reason, the file keys must all be encrypted before they are stored on the disk. Public-key cryptography is used for this purpose.

After the file is encrypted, the location of the user's public key is looked up using information in the registry. There is no danger of storing the public key's location in the registry because if a thief steals the computer and finds the public key, there is no way to deduce the private key from it. The 128-bit random file key is now encrypted with the public key and the result stored on disk along with the file, as shown in Fig. 11-41.

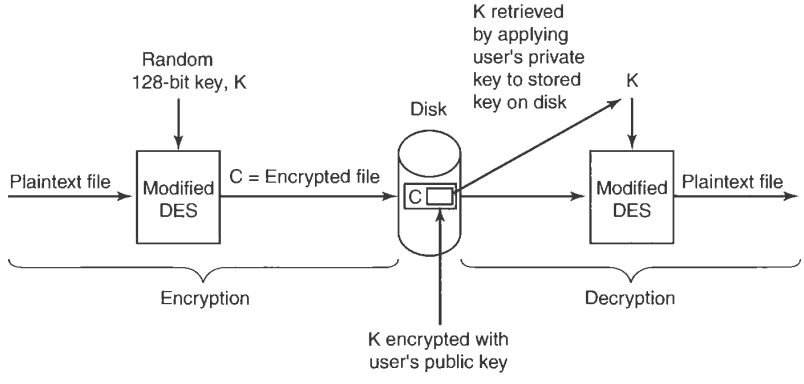


Figure 11-41. Operating of the encrypting file system.

To decrypt a file, the encrypted 128-bit random file key is fetched from disk. However, to decrypt it and retrieve the file key, the user must present the private

key. Ideally, this should be stored on a smart card, external to the computer, and only inserted in a reader when a file has to be decrypted. Although Windows 2000 supports smart cards, it does not store private keys on them.

Instead, the first time a user encrypts a file using EFS, Windows 2000 generates a (private key, public key) pair and stores the private key on disk encrypted using a symmetric encryption algorithm. The key used for the symmetric algorithm is derived either from the user's login password or from a key stored on the smart card, if smart card login is enabled. In this way, EFS can decrypt the private key at login time and keep it within its own virtual address space during normal operation so it can decrypt the 128-bit file keys as needed without further disk accesses. When the computer is shut down, the private key is erased from EFS' virtual address space so anyone stealing the computer will not have access to the private key.

A complication occurs when multiple users need access to the same encrypted file. Currently the shared use of encrypted files by multiple users is not supported. However, the EFS architecture could support sharing in the future by encrypting each file's key multiple times, once with the public key of each authorized user. All of these encrypted versions of the file key could be attached to the file.

The potential need to share encrypted files is one reason why this two-key system is used. If all files were encrypted by their owner's key, there would be no way to share any files. By using a different key to encrypt each file, this problem can be solved.

Having a random file key per file but encrypting it with the owner's symmetric key does not work because having the symmetric encryption key just lying around in plain view would ruin the security— generating the decryption key from the encryption key is too easy. Thus (slow) public-key cryptography is needed to encrypt the file keys. Because the encryption key is public anyway, having it lying around is not dangerous.

The other reason the two-key system is used is performance. Using public-key cryptography to encrypt each file would be too slow. It is much more efficient to use symmetric-key cryptography to encrypt the data and public-key cryptography to encrypt the symmetric file key.

11.8 SECURITY IN WINDOWS 2000

Having just looked at encryption in the file system, this is a good time to examine security in general. NT was designed to meet the U.S. Department of Defense's C2 security requirements (DoD 5200.28-STD), the Orange Book, which we studied in Chap. 9. This standard requires operating systems to have certain properties in order to be classified as secure enough for certain kinds of

military work. Although Windows 2000 was not specifically designed for C2 compliance, it inherits many security properties from NT, including the following:

1. Secure login with antispoofing measures.
2. Discretionary access controls.
3. Privileged access controls.
4. Address space protection per process.
5. New pages must be zeroed before being mapped in.
6. Security auditing.

Let us review these items briefly (none of which are met by Windows 98, incidentally).

Secure login means that the system administrator can require all users to have a password in order to log in. Spoofing is when a malicious user writes a program that displays the login prompt or screen and then walks away from the computer in the hope that an innocent user will sit down and enter a name and password. The name and password are then written to disk and the user is told that login has failed. Windows 2000 prevents this attack by instructing users to hit CTRL-ALT-DEL to log in. This key sequence is always captured by the keyboard driver, which then invokes a system program that puts up the genuine login screen. This procedure works because there is no way for user processes to disable CTRL-ALT-DEL processing in the keyboard driver.

Discretionary access controls allow the owner of a file or other object to say who can use it and in what way. Privileged access controls allow the system administrator (superuser) to override them when needed. Address space protection simply means that each process has its own protected virtual address space not accessible by any unauthorized process. The next item means that when a stack grows, the pages mapped in are initialized to zero so processes cannot find any old information put there by the previous owner (hence the zeroed page list in Fig. 11-27, which provides a supply of zeroed pages for this purpose). Finally, security auditing allows the administrator to produce a log of certain security-related events.

In the next section we will describe the basic concepts behind Windows 2000 security. After that we will look at the security system calls. Finally, we will conclude by seeing how security is implemented.

11.8.1 Fundamental Concepts

Every Windows 2000 user (and group) is identified by a **SID** (**Security ID**). SIDs are binary numbers with a short header followed by a long random component. Each SID is intended to be unique worldwide. When a user starts up a

process, the process and its threads run under the user's SID. Most of the security system is designed to make sure that each object can be accessed only by threads with authorized SIDs.

Each process has an **access token** that specifies its SID and other properties. It is normally assigned at login time by *winlogon* and is shown in Fig. 11-42, although processes should call *GetTokenInformation* to acquire this information since it may change in the future. The header contains some administrative information. The expiration time field could tell when the token ceases to be valid, but it is currently not used. The *Groups* fields specify the groups to which the process belongs; this is needed for POSIX conformance. The default **DACL (Discretionary ACL)** is the access control list assigned to objects created by the process if no other ACL is specified. The user SID tells who owns the process. The restricted SIDS are to allow untrustworthy processes to take part in jobs with trustworthy processes but with less power to do damage.

Finally, the privileges listed, if any, give the process special powers, such as the right to shut the machine down or access files to which access would otherwise be denied. In effect, the privileges split up the power of the superuser into several rights that can be assigned to processes individually. In this way, a user can be given some superuser power, but not all of it. In summary, the access token tells who owns the process and which defaults and powers are associated with it.

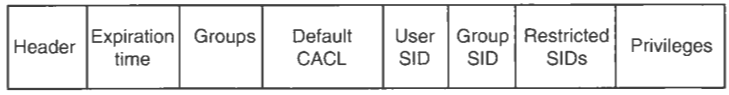


Figure 11-42. Structure of an access token .

When a user logs in, *winlogon* gives the initial process an access token. Subsequent processes normally inherit this token on down the line. A process' access token initially applies to all the threads in the process. However, a thread can acquire a different access token during execution, in which case case the thread's access token overrides the process' access token. In particular, a client thread can pass its access token to a server thread to allow the server to access the client's protected files and other objects. This mechanism is called **impersonation**.

Another basic concept is the **security descriptor**. Every object has a security descriptor associated with it that tells who can perform which operations on it. A security descriptor consists of a header followed by a DACL with one or more **ACEs (Access Control Elements)**. The two main kinds of elements are Allow and Deny. An allow element specifies a SID and a bitmap that specifies which operations processes with that SID may perform on the object. A deny element works the same way, except a match means the caller may not perform the operation. For example, Ida has a file whose security descriptor specifies that everyone

IBM Ex-1009
Page 530

has read access, Elvis has no access. Cathy has read/write access, and Ida herself has full access. This simple example is illustrated in Fig. 11-43. The SID Everyone refers to the set of all users, but it is overridden by any explicit ACEs that follow.

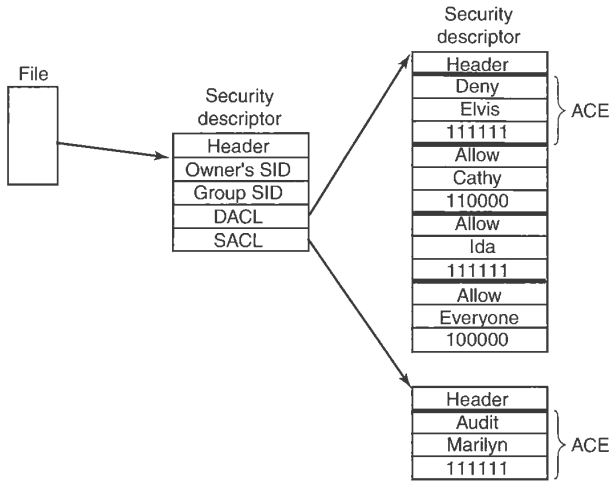


Figure 11-43. An example security descriptor for a file.

In addition to the DACL, a security descriptor also has a **SACL** (**System Access Control list**), which is like a DACL except that it specifies not who may use the object, but which operations on the object are recorded in the system-wide security event log. In Fig. 11-43, every operation that Marilyn performs on the file will be logged. Windows 2000 provides additional auditing features to log sensitive accesses.

11.8.2 Security API Calls

Most of the Windows 2000 access control mechanism is based on security descriptors. The usual pattern is that when a process creates an object, it provides a security descriptor as one of the parameters to the CreateProcess, CreateFile, or other object creation call. This security descriptor then becomes the security descriptor attached to the object, as we saw in Fig. 11-43. If no security descriptor is provided in the object creation call, the default security in the caller's access token (see Fig. 11-42) is used instead.

Many of the Win32 API security calls relate to the management of security descriptors, so we will focus on those here. The most important calls are listed in Fig. 11-44. To create a security descriptor, storage for it is first allocated and then initialized using InitializeSecurityDescriptor. This call fills in the header. If the

IBM Ex-1009
Page 531

owner SID is not known, it can be looked up by name using LookupAccountSid. It can then be inserted into the security descriptor. The same holds for the group SID, if any. Normally, these will be the caller's own SID and one of the caller's groups, but the system administrator can fill in any SIDs.

Win32 API function	Description
InitializeSecurityDescriptor	Prepare a new security descriptor for use
LookupAccountSid	Look up the SID for a given user name
SetSecurityDescriptorOwner	Enter the owner SID in the security descriptor
SetSecurityDescriptorGroup	Enter a group SID in the security descriptor
InitializeAcl	Initialize a DACL or SACL
AddAccessAllowedAce	Add a new ACE to a DACL or SACL allowing access
AddAccessDeniedAce	Add a new ACE to a DACL or SACL denying access
DeleteAce	Remove an ACE from a DACL or SACL
SetSecurityDescriptorDacl	Attach a DACL to a security descriptor

Figure 11-44. The principal Win32 API functions for security.

At this point the security descriptor's DACL (or SACL) can be initialized with InitializeAcl. ACL entries can be added using AddAccessAllowedAce, and AddAccessDeniedAce. These calls can be repeated multiple times to add as many ACE entries as are needed. DeleteAce can be used to remove an entry, more like on an existing ACL than on one being constructed for the first time. When the ACL is ready, SetSecurityDescriptorDacl can be used to attach it to the security descriptor. Finally, when the object is created, the newly minted security descriptor can be passed as a parameter to have it attached to the object.

11.8.3 Implementation of Security

Security in a standalone Windows 2000 system is implemented by a number of components, most of which we have already seen (networking is a whole other story and beyond the scope of this book). Logging in is handled by *winlogon* and authentication is handled by *lsass* and *msgina.dll* as discussed in Sec. 11.4.5. The result of a successful login is a new shell with its associated access token. This process uses the SECURITY and SAM keys in the registry. The former sets the general security policy and the latter contains the security information for the individual users, as discussed in Sec. 11.2.3.

Once a user is logged in, security operations happen when an object is opened for access. Every OpenXXX call requires the name of the object being opened and the set of rights needed. During processing of the open, the security manager (see Fig. 11-7) checks to see if the caller has all the rights required. It performs this check by looking at the caller's access token and the DACL associated with the

IBM Ex-1009
Page 532

object. It goes down the list of entries in the ACL in order. As soon as it finds an entry that matches the caller's SID or one of the caller's groups, the access found there is taken as definitive. If all the rights the caller needs are available, the open succeeds; otherwise it fails.

DACLs can have Deny entries as well as Allow entries, as we have seen. For this reason, it is usual to put entries denying access ahead of entries granting access in the ACL, so that a user who is specifically denied access cannot get in via a back door by being a member of a group that has legitimate access.

After an object has been opened, a handle to it is returned to the caller. On subsequent calls, the only check that is made is whether the operation now being tried was in the set of operations requested at open time, to prevent a caller from opening a file for reading and then trying to write on it. Any log entries required by the SACL are made.

11.9 CACHING IN WINDOWS 2000

The Windows 2000 cache manager does caching for performance reasons, conceptually similar to caches in other operating systems. However, its design has some unusual properties that are worth looking at briefly.

The cache manager's job is to keep file system blocks that have been used recently in memory to reduce access time on any subsequent reference. Windows 2000 has a single integrated cache that works for all the file systems in use, including NTFS, FAT-32, FAT-16, and even CD-ROM file systems. This means that the file systems do not need to maintain their own caches.

As a consequence of the design goal to have a single integrated cache despite the presence of multiple file systems, the cache manager is located in an unusual position in the system, as we saw in Fig. 11-7. It is not part of the file system because there are multiple independent file systems that may have nothing in common. Instead it operates at a higher level than the file systems, which are technically drivers under control of the I/O manager.

The Windows 2000 cache is organized by virtual block, not physical block. To see what this means, remember that traditional file caches keep track of blocks by two-part addresses of the form (partition, block), where the first member denotes a device and partition and the second member is a block number within that partition. The Windows 2000 cache manager does not do that. Instead, it uses (file, offset) to refer to a block. (Technically, streams are cached rather than files, but we will ignore that detail below.)

The reason for this unorthodox arrangement is that when a request comes in to the cache manager, it is specified as (file, offset) because that is all the calling process knows. If cache blocks had been labeled by a (partition, block) tag, the cache manager would have no way of knowing which (file, offset) block cor-

IBM Ex-1009
Page 533

responds to which (partition, block) block since it is the file systems that maintain those mappings.

Let us now examine how the cache manager works. When a file is referenced, the cache manager maps a 256-KB chunk of kernel virtual address space onto the file. If the file is larger than 256 KB, only a portion of the file is mapped. The total amount of virtual address space the cache manager can use is determined at boot time and depends on the amount of RAM present. If the cache manager runs out of 256-KB chunks of virtual address space, it must unmap an old file before mapping in a new one.

Once a file is mapped, the cache manager can satisfy requests for its blocks by just copying from kernel virtual address space to the user buffer. If the block copied is not in physical memory, a page fault will occur and the memory manager will satisfy the fault in the usual away. The cache manager is not even aware of whether the block was in the cache or not. The copy always succeeds.

The operation of the cache manager is shown in Fig. 11-45 in the case of an NTFS file system on a SCSI disk and a FAT-32 file system on an IDE disk. When a process does a read on a file, the request is directed to the cache manager. If the block needed is in the cache, it is copied to the user immediately. If it is not in the cache, the cache manager gets a page fault when trying to copy it. When the page fault has been handled, the block is copied to the calling process.

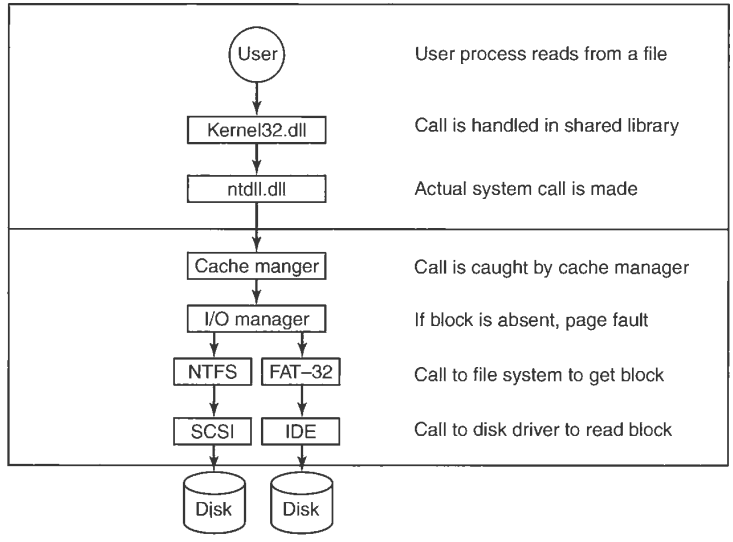


Figure 11-45. The path through the cache to the hardware.

As a consequence of this design, the cache manager does not know how many of its mapped pages are in physical memory or even how large its cache is. Only

the memory manager knows for sure. This approach allows the memory manager to dynamically trade off the size of the cache against memory for user pages. If there is little file activity but there are many processes active, the memory manager can use most of physical memory for process pages. On the other hand, if there is a lot of file activity and few processes, more physical memory can be devoted to the cache.

Another property the cache manager has is that coherence is maintained between memory-mapped files and files that are open for reading and writing. Consider, for example, a situation in which one process opens some file for reading and writing and a second process maps that file onto its address space. What happens if the second process writes on the file directly and immediately thereafter the first one reads in a block that was just changed? Does it get stale data?

The answer is no. In both cases—open files and mapped files— the cache manager maps a 256-KB piece of its virtual address space onto the file. The file is only mapped once, no matter how many processes have it open or mapped. In the case of a mapped file, both the cache manager and the user process share pages in memory. When a read request has to be satisfied, the cache manager just copies the page from memory to the user buffer, which always reflects the exact current state of the file because the cache manager is using the same pages as the process that has the file mapped in.

11.10 SUMMARY

Windows 2000 is structured in the HAL, the kernel, the executive, and a thin system services layer that catches incoming system calls. In addition, there are a variety of device drivers, including the file systems and GDI. The HAL hides certain differences in hardware from the upper layers. The kernel attempts to hide the remaining differences from the executive to make it almost completely machine independent.

The executive is based on memory objects. User processes create them and get back handles to manipulate them later. Executive components can also create objects. The object manager maintains a name space into which objects can be inserted for subsequent lookup.

Windows 2000 supports processes, jobs, threads, and fibers. Processes have virtual address spaces and are containers for resources. Threads are the unit of execution and are scheduled by the operating system. Fibers are lightweight threads that are scheduled entirely in user space. Jobs are collections of processes and are used for assigning resource quotas. Scheduling is done using a priority algorithm in which the highest priority ready thread runs next.

Windows 2000 supports a demand-paged virtual memory. The paging algorithm is based on the working set concept. The system maintains several lists of

free pages, so that when a page fault occurs, there is generally a free page available. The free page lists are fed by trimming the working sets using complex formulas that try to throw out pages that have not been used in a long time.

I/O is done by device drivers, which follow the Windows Device Model. Each driver starts out by initializing a driver object that contains the addresses of the procedures that the system can call to add devices or perform I/O. Drivers can be stacked to act as filters.

The NTFS file system is based on a master file table, which has one record per file or directory. Each file has multiple attributes, which can either be in the MFT record or nonresident, on the disk. NTFS supports compression and encryption, among other features.

Security is based on access control lists. Each process has an access control token that tells who it is and what special privileges it has, if any. Each object has a security descriptor associated with it. The security descriptor points to a discretionary access control list that contains access control entries that can allow or deny access to individuals or groups.

Finally, Windows 2000 maintains a single system-wide cache for all the file systems. It is a virtual cache rather than a physical one. Requests for disk blocks go first to the cache. If they cannot be satisfied, then the appropriate file system is called to fetch the needed blocks.

PROBLEMS

1. Give one advantage and one disadvantage of the registry versus having individual *.ini* files?
2. A mouse can have 1, 2, or 3 buttons. All three types are in use. Does the HAL hide this difference from the rest of the operating system? Why or why not?
3. The HAL keeps track of time starting in the year 1601. Give an example of an application where this feature is useful.
4. The POSIX subsystem needs to implement UNIX-style signals. If a user hits the key for the Quit signal, is this scheduled as a DPC or an APC?
5. Many components of the executive (Fig. 11-7) call other components of the executive. Give three examples of one component calling another one, but use (six) different components in all.
6. Win32 does not have signals. If they were to be introduced, they could be per process, per thread, both, or neither. Make a proposal and explain why it is a good idea.
7. An alternative to using DLLs is to statically link each program with precisely those library procedures it actually calls, no more and no less. If this scheme were to be introduced, would it make more sense on client machines or on server machines?

IBM Ex-1009
Page 536

8. The file *ntdll.dll* exports 1179 function calls while *ntoskrnl.exe* exports 1209 function calls. Is this a bug? What might be the cause of this discrepancy?

9. Objects managed by the object manager are variable-sized, with different objects having different sizes. Can an object start at an arbitrary byte in the nonpaged pool? *Hint*: You do not need any information about Windows 2000 other than what was given in the text.

10. Is there any limit on the number of different operations that can be defined on an executive object? If so, where does this limit come from? If not, why not?

11. The Win32 API call WaitForMultipleObjects allows a thread to block on a set of synchronization objects whose handles are passed as parameters. As soon as any one of them is signaled, the calling thread is released. Is it possible to have the set of synchronization objects include two semaphores, one mutex, and one critical section? Why or why not? *Hint*: This is not a trick question but it does require some careful thought.

12. Name three reasons why a process might be terminated.

13. Consider the situation of Fig. 11-19 in which the system is about to schedule a thread. Assuming that each thread is compute bound, how long does it take before some thread at priority 3 gets to run on Windows 2000 Professional?

14. Suppose that the quantum is set to 20 msec and the current thread, at priority 24, has just started a quantum. Suddenly an I/O operation completes and a priority 28 thread is made ready. About how long does it have to wait to get serviced?

15. In Windows 2000, the current priority is always greater than or equal to the base priority. Are there any circumstances in which it would make sense to have the current priority be lower than the base priority? If so, give an example. If not, why not?

16. Some MS-DOS programs were written in assembly language using instructions such as IN port and OUT port. Can such programs be run under Windows 2000? If not, can you think of a way they could be supported?

17. Name two ways to give better response time to important processes.

18. Position-independent code was briefly discussed in the text. It is a technique for allowing two processes to share the same procedure at different virtual addresses. Can this problem be solved by just setting the page tables of the two processes appropriately? Explain your answer.

19. Shared libraries in Windows 2000 are contained in *.dll* files that multiple processes may map in at the same time. If two processes need to map in the same shared library at different virtual addresses, a problem will occur. How could this problem be solved on the Pentium using a property of its memory architecture? If the solution requires changes to Windows 2000 to implement, state what changes are needed.

20. If a region of virtual address space is reserved but not committed, do you think a VAD is created for it? Defend your answer.

21. Which of the transitions shown in Fig. 11-27 are policy decisions, as opposed to required moves forced by system events (e.g., a process exiting and freeing its pages)?

22. Suppose that a page is shared and in two working sets at once. If it is evicted from one of the working sets, where does it go in Fig. 11-27?

23. When a process unmaps a clean page, it makes the transition (5) in Fig. 11-27. Why is there no transition to the modified list when a dirty page is unmapped?

24. Assuming that both RAM and video RAM each can read or write a 32-bit word in 10 nsec, how long does it take to paint the background on an XGA screen in the best case?

25. A file has the following mapping. Give the MFT run entries.

Offset	0	1	2	3	4	5	6	7	8	9	10
Disk address	50	51	52	22	24	25	26	53	54	-	60

26. Consider the MFT record of Fig. 11-36. Suppose that the file grew and a 10th block was assigned to the end of the file. The number of this block is 66. What would the MFT record look like now?

27. In Fig. 11-40(b), the first two runs are each of length 8 blocks. Is it just an accident they are equal or does this have to do with the way compression works? Explain your answer.

28. Suppose that you wanted to build Windows 2000 Lite. Which of the fields of Fig. 11-42 could be removed without weakening the security of the system?

29. The *regedit* command can be used to export part or all of the registry to a text file under all current versions of Windows. Save the registry several times during a work session and see what changes. If you have access to a Windows computer on which you can install software or hardware, find out what changes when a program or device are added or removed.

30. Write a UNIX program that simulates writing an NTFS file with multiple streams. It should accept a list of one or more files as arguments and write an output file that contains one stream with the attributes of all arguments and additional streams with the contents of each of the arguments. Now write a second program for reporting on the attributes and streams and extracting all the components.

31. Write a program that generates an MS-DOS 8 + 3 filename given any file name. Use the Windows 2000 algorithm.

IBM Ex-1009
Page 538

12

OPERATING SYSTEM DESIGN

In the past 11 chapters, we have covered a lot of ground and taken a look at many concepts and examples relating to operating systems. But studying existing operating systems is different from designing a new one. In this chapter we are going to take a quick look at some of the issues and trade-offs that operating systems designers have to take into account when designing and implementing a new system.

There is a certain amount of folklore about what is good and what is bad floating around in the operating systems community, but surprisingly little has been written down. Probably the most important book is Fred Brooks' classic (1975) book *The Mythical Man Month* in which he relates his experiences in designing and implementing IBM's OS/360. The 20th anniversary edition revises some of that material and adds four new chapters (Brooks, 1995). Probably the only operating systems textbook dealing with design in a serious way is *Operating Systems: A Design-Oriented Approach* (Crowley, 1997).

Three classic papers on operating system design are "Hints for Computer System Design" (Lampson, 1984), "On Building Systems that Will Fail" (Corbató, 1991), and "End-to-End Arguments in System Design" (Saltzer et al., 1984). Like Brooks' book, all three papers have survived the years extremely well; most of their insights are still as valid now as when they were first published.

This chapter draws upon these sources, plus the author's personal experience as designer or co-designer of three systems: Amoeba (Tanenbaum et al., 1990), MINIX (Tanenbaum and Woodhull, 1997), and Globe (Van Steen et al., 1999a).

IBM Ex-1009
Page 539

Since no consensus exists among operating system designers about the best way to design an operating system, this chapter will thus be more personal, speculative, and undoubtedly more controversial than the previous ones.

12.1 THE NATURE OF THE DESIGN PROBLEM

Operating system design is more of an engineering project than an exact science. It is much harder to set clear goals and to meet them. Let us look at start with these points.

12.1.1 Goals

In order to design a successful operating system, the designers have to have a clear idea of what they want. Lack of a goal makes it very hard to make subsequent decisions. To make this point clearer, it is instructive to take a look at two programming languages PL/I and C. PL/I was designed by IBM in the 1960s because it was a nuisance to have to support both FORTRAN and COBOL, and embarrassing to have academics yapping in the background that Algol was better than both of them. So a committee was set up to produce a language that would be all things to all people: PL/I. It had a little bit of FORTRAN, a little bit of COBOL, and a little bit of Algol. It failed because it lacked any unifying vision. It was simply a collection of features at war with one another, and too cumbersome to be compiled efficiently, to boot.

Now consider C. It was designed by one person (Dennis Ritchie) for one purpose (system programming). It was a huge success, in no small part because Ritchie knew what he wanted and did not want. As a result, it is still in widespread use decades after its appearance. Having a clear vision of what you want is crucial.

What do operating system designers want? It obviously varies from system to system, being different for embedded systems than for server systems. However, for general-purpose operating systems four main items come to mind:

1. Define abstractions.
2. Provide primitive operations.
3. Ensure isolation.
4. Manage the hardware.

Each of these items will be discussed below.

The most important, but probably hardest task of an operating system is to define the right abstractions. Some of them, such as processes and files, have been around so long that they may seem obvious. Other ones, such as threads, are

newer, and are less mature. For example, if a multithread process that has one thread blocked waiting for keyboard input forks, is there a thread in the new process also waiting for keyboard input? Other abstractions relate to synchronization, signals, the memory model, modeling of I/O and many other areas.

Each of the abstractions can be instantiated in the form of concrete data structures. Users can create processes, files, semaphores, etc. The primitive operations manipulate these data structures. For example, users can read and write files. The primitive operations are implemented in the form of system calls. From the user's point of view, the heart of the operating system is formed by the abstractions and the operations on them available via the system calls.

Since multiple users can be logged into a computer at the same time, the operating system needs to provide mechanisms to keep them separated. One user may not interfere with another. The process concept is widely used to group resources together for protection purposes. Files and other data structures generally are protected as well. Making sure each user can perform only authorized operations on authorized data is a key goal of system design. However, users also want to share data and resources, so the isolation has to be selective and under user control. This makes it much harder.

Closely related to this point is the need to isolate failures. If some part of the system goes down, most commonly a user process, it should not be able to take the rest of the system down with it. The system design should make sure that the various parts are well isolated from one another. Ideally, parts of the operating system should also be isolated from one another to allow independent failures.

Finally, the operating system has to manage the hardware. In particular, it has to take care of all the low-level chips, such as interrupt controllers and bus controllers. It also has to provide a framework for allowing device drivers to manage the larger I/O devices, such as disks, printers, and the display.

12.1.2 Why is it Hard to Design an Operating System?

Moore's law says that computer hardware improves by a factor of 100 every decade. Nobody has a law saying that operating systems improve by a factor of 100 every decade. Or even get better at all. In fact, a case can be made that some of them are worse in key respects (such as reliability) than UNIX Version 7 was back in the 1970s.

Why? Inertia and the desire for backward compatibility often get much of the blame, and the failure to adhere to good design principles is also a culprit. But there is more to it. Operating systems are fundamentally different in certain ways from small application programs sold in computer stores for $49. Let us look at eight of the issues that make designing an operating system much harder than designing an application program.

First, operating systems have become extremely large programs. No one person can sit down at a PC and dash off a serious operating system in a few months.

All current versions of UNIX exceed 1 million lines of code; Windows 2000 is 29 million lines of code. No one person can understand even 1 million lines of code, let alone 29 million lines of code. When you have a product that none of the designers can hope to fully understand, it should be no surprise that the results are often far from optimal.

Operating systems are not the most complex systems around. Aircraft carriers are far more complicated, for example, but they partition into isolated subsystems better. The people designing the toilets on a aircraft carrier do not have to worry about the radar system. The two subsystems do not interact much. In an operating system, the file system often interacts with the memory system in unexpected and unforeseen ways.

Second, operating systems have to deal with concurrency. There are multiple users and multiple I/O devices all active at once. Managing concurrency is inherently much harder than managing a single sequential activity. Race conditions and deadlocks are just two of the problems that come up.

Third, operating systems have to deal with potentially hostile users—users who want to interfere with system operation or do things that are forbidden from doing, such as stealing another user's files. The operating system needs to take measures to prevent these users from behaving improperly. Word processing programs and photo editors do not have this problem.

Fourth, despite the fact that not all users trust each other, many users do want to share some of their information and resources with selected other users. The operating system has to make this possible, but in such a way that malicious users cannot interfere. Again, application programs do not face anything like this challenge.

Fifth, operating systems live for a very long time. UNIX has been around for a quarter of a century; Windows has been around for over a decade and shows no signs of vanishing. Consequently, the designers have to think about how hardware and applications may change in the distant future and how they should prepare for it. Systems that are locked too closely into one particular vision of the world usually die off.

Sixth, operating system designers really do not have a good idea of how their systems will be used, so they need to provide for considerable generality. Neither UNIX nor Windows was designed with email or Web browsers in mind, yet many computers running these systems do little else. Nobody tells a ship designer to build a ship without specifying whether they want a fishing vessel, a cruise ship, or a battleship. And even fewer change their minds after the product has arrived.

Seventh, modern operating systems are generally designed to be portable, meaning they have to run on multiple hardware platforms. They also have to support hundreds, even thousands of I/O devices, all of which are independently designed with no regard to one another. An example of where this diversity causes problems is the need for an operating system to run on both little-endian and big-endian machines. A second example was seen constantly under MS-DOS

when users attempted to install, say, a sound card and a modem that used the same I/O ports or interrupt request lines. Few programs other than operating systems have to deal with sorting out problems caused by conflicting pieces of hardware.

Eighth, and last in our list, is the frequent need to be backward compatible with some previous operating system. That system may have restrictions on word lengths, file names, or other aspects that the designers now regard as obsolete, but are stuck with. It is like converting a factory to produce next year's cars instead of this year's cars, but while continuing to produce this year's cars at full capacity.

12.2 INTERFACE DESIGN

It should be clear by now that writing a modern operating system is not easy. But where does one begin? Probably the best place to begin is to think about the interfaces it provides. An operating system provides a set of services, mostly data types (e.g., files) and operations on them (e.g., read). Together, these form the interface to its users. Note that in this context the users of the operating system are programmers who write code that use system calls, not people running application programs.

In addition to the main system call interface, most operating systems have additional interfaces. For example, some programmers need to write device drivers to insert into the operating system. These drivers see certain features and can make certain procedure calls. These features and calls also define and interface, but a very different one from one application programmers see. All of these interfaces must be carefully designed if the system is to succeed.

12.2.1 Guiding Principles

Are their any principles that can guide interface design? We believe there are. Briefly summarized, they are simplicity, completeness, and the ability to be implemented efficiently.

Principle 1: Simplicity

A simple interface is easier to understand and implement in a bug-free way. All system designers should memorize this famous quote from the pioneer French aviator and writer, Antoine de St. Exupéry:

> *Perfection is reached not when there is no longer anything to add, but when there is no longer anything to take away.*

This principle says that less is better than more, at least in the operating system itself. Another way to say this is the KISS principle: Keep It Simple, Stupid.

Principle 2: Completeness

Of course, the interface must make it possible to do everything that the users need to do, that is, it must be complete. This brings us to another famous quote, this one from Albert Einstein:

Everything should be as simple as possible, but no simpler.

In other words, the operating system should do exactly what is needed of it and no more. If users need to store data, it must provide some mechanism for storing data. If users need to communicate with each other, the operating system has to provide a communication mechanism, and so on. In his 1991 Turing Award lecture, Fernando Corbató, one of the designers of CTSS and MULTICS, combined the concepts of simplicity and completeness and said:.SP 0.25

> *First, it is important to emphasize the value of simplicity and elegance, for complexity has a way of compounding difficulties and as we have seen, creating mistakes. My definition of elegance is the achievement of a given functionality with a minimum of mechanism and a maximum of clarity.*

The key idea here is *minimum of mechanism*. In other words, every feature, function, and system call should carry its own weight. It should do one thing and do it well. When a member of the design team proposes extending a system call or adding some new feature, the others should ask the question: "Would something awful happen if we left it out?" If the answer is: "No, but somebody might find this feature useful some day" put it in a user-level library, not in the operating system, even if it is slower that way. Not every feature has to be faster than a speeding bullet. The goal is to preserve what Corbató called minimum of mechanism.

Let us briefly consider two examples from my own experience: MINIX and Amoeba. For all intents and purposes, MINIX has three system calls: send, receive, and sendrec. The system is structured as a collection of processes, with the memory manager, the file system, and each device driver being a separate schedulable process. To a first approximation, all the kernel does is schedule processes and handle message passing between them. Consequently, only two system calls are needed: send, to send a message, and receive to receive one. The third call, sendrec is simply an optimization for efficiency reasons to allow a message to be sent and the reply to be requested with only one kernel trap. Everything else is done by requesting some other process (for example, the file system process or the disk driver) to do the work.

Amoeba is even simpler. It has only one system call: perform remote procedure call. This call sends a message and waits for a reply. It is essentially the same as MINIX' sendrec. Everything else is built on this one call.

IBM Ex-1009
Page 544

Principle 3: Efficiency

The third guideline is efficiency of implementation. If a feature or system call cannot be implemented efficiently, it is probably not worth having. It should also be intuitively obviously to the programmer about how much a system call costs. For example, UNIX programmers expect the lseek system call to be cheaper than the read system call because the former just changes a pointer in memory while the latter performs disk I/O. If the intuitive costs are wrong, programmers will write inefficient programs.

12.2.2 Paradigms

Once the goals have been established, the design can begin. A good starting place is thinking about how the customers will view the system. One of the most important issues is how to make all the features of the system hang together well and present what is often called **architectural coherence**. In this regard, it is important to distinguish two kinds of operating system "customers." On the one hand, there are the *users*, who interact with application programs; on the other are the *programmers*, who write them. The former mostly deal with the GUI; the latter mostly deal with the system call interface. If the intention is to have a single GUI that pervades the complete system, such as in the Macintosh, the design should start there. If, on the other hand, the intention is to support many possible GUIs, such as in UNIX, the system call interface should be designed first. Doing the GUI first is essentially a top-down design. The issues are what features will it have, how will the user interact with it, and how should the system be designed to support it? For example, if most programs display icons on the screen and then wait for the user to click on one of them, this suggests an event-driven model for the GUI and probably also for the operating system. On the other hand, if the screen is mostly full of text windows, then a model in which processes read from the keyboard is probably better.

Doing the system call interface first is a bottom-up design. Here the issues are what kinds of features programmers in general need. Actually, not many special features are needed to support a GUI. For example, the UNIX windowing system, X, is just a big C program that does reads and writes on the keyboard, mouse, and screen. X was developed long after UNIX and did not require many changes to the operating system to get it to work. This experience validated that fact that UNIX was sufficiently complete.

User Interface Paradigms

For both the GUI-level interface and the system-call interface, the most important aspect is having a good paradigm (sometimes called a metaphor) to provide a way of looking at the interface. Many GUIs for desktop machines use the

WIMP paradigm that we discussed in Chap. 5. This paradigm uses point-and-click, point-and-double-click, dragging, and other idioms throughout the interface to provide an architectural coherence to the whole. Often there are additional requirements for programs, such as having a menu bar with FILE, EDIT, and other entries, each of which has certain well-known menu items. In this way, users who know one program can quickly learn another one.

However, the WIMP user interface is not the only possible one. Some palmtop computers use a stylized handwriting interface. Dedicated multimedia devices may use a VCR-like interface. And of course, voice input has a completely different paradigm. What is important is not so much the paradigm chosen, but the fact that there is a single overriding paradigm that unifies the entire user interface.

Whatever paradigm is chosen, it is important that all application programs use it. Consequently, the system designers need to provide libraries and tool kits to application developers that give them access to procedures that produce the uniform look-and-feel. User interface design is very important, but it is not the subject of this book, so we will now drop back down to the subject of the operating system interface.

Execution Paradigms

Architectural coherence is important at the user level, but equally important at the system call interface level. Here it is frequently useful to distinguish between the execution paradigm and the data paradigm, so we will do both, starting with the former.

Two execution paradigms are widespread: algorithmic and event driven. The **algorithmic paradigm** is based on the idea that a program is started to perform some function that it knows in advance or gets from its parameters. That function might be to compile a program, do the payroll, or fly an airplane to San Francisco. The basic logic is hardwired into the code with the program making system calls from time to time to get user input, obtain operating system services, etc. This approach is outlined in Fig. 12-1(a).

The other execution paradigm is the **event-driven paradigm** of Fig. 12-1(b). Here the program performs some kind of initialization, for example by displaying a certain screen, and then waiting for the operating system to tell it about the first event. The event is often a key being struck or a mouse movement. This design is useful for highly interactive programs.

Each of these ways of doing business engenders its own programming style. In the algorithmic paradigm, algorithms are central and the operating system is regarded as a service provider. In the event-driven paradigm, the operating system also provides services, but this role is overshadowed by its role as a coordinator of user activities and a generator of events that are consumed by processes.

IBM Ex-1009
Page 546

```
main( )
{
     int ... ;

     init( );
     do_something( );
     read(...);
     do_something_else( );
     write(...);
     keep_going( );
     exit(0);
}
```

(a)

```
main( )
{
     mess_t msg;

     init( );
     while (get_message(&msg)) {
          switch (msg.type) {
               case 1: ... ;
               case 2: ... ;
               case 3: ... ;
          }
     }
}
```

(b)

Figure 12-1. (a) Algorithmic code. (b) Event-driven code.

Data Paradigms

The execution paradigm is not the only one exported by the operating system. An equally important one is the data paradigm. The key question here is how are system structures and devices presented to the programmer. In early FORTRAN batch systems, everything was modeled as a sequential magnetic tape. Card decks read in were treated as input tapes, card decks to be punched were treated as output tapes, and output for the printer was treated as an output tape. Disk files were also treated as tapes. Random access to a file was possible only by rewinding the tape corresponding to the file and reading it again.

The mapping was done using job control cards like these:

```
MOUNT(TAPE08, REEL781)
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

The first card instructed the operator to go get tape reel 781 from the tape rack and mount it on tape drive 8. The second card instructed the operating system to run the just compiled FORTRAN program, mapping *INPUT* (meaning the card reader) to logical tape 1, disk file *MYDATA* to logical tape 2, the printer (called *OUTPUT*) to logical tape 3, the card punch (called *PUNCH*) to logical tape 4. and physical tape drive 8 to logical tape 5.

FORTRAN had a syntax for reading and writing logical tapes. By reading from logical tape 1, the program got card input. By writing to logical tape 3, output would later appear on the printer. By reading from logical tape 5, tape reel 781 could be read in, and so on. Note that the tape idea was just a paradigm to integrate the card reader, printer, punch, disk files, and tapes. In this example, only logical tape 5 was a physical tape; the rest were ordinary (spooled) disk files. It was a primitive paradigm, but it was a start in the right direction.

IBM Ex-1009
Page 547

Later came UNIX, which goes much further using the model of "everything is a file." Using this paradigm, all I/O devices are treated as files and can be opened and manipulated as ordinary files. The C statements

```
fd1 = open("file1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR)'
```

open a true disk file and the user's terminal. Subsequent statements can use *fd1* and *fd2* to read and write them, respectively. From that point on, there is no difference between accessing the file and accessing the terminal, except that seeks on the terminal are not allowed.

Not only does UNIX unify files and I/O devices, but it also allows other processes to be accessed over pipes as files. Furthermore, when mapped files are supported, a process can get at its own virtual memory as though it were a file. Finally, in versions of UNIX that support the */proc* file system, the C statement

```
fd3 = open("/proc/501", O_RDWR);
```

allows the process to (try to) access process 501's memory for reading and writing using file descriptor *fd3*, something useful for, say, a debugger.

Windows 2000 goes further still and tries to make everything look like an object. Once a process has acquired a valid handle to a file, process, semaphore, mailbox, or other kernel object, it can perform operations on it. This paradigm is even more general than that of UNIX and much more general than that of FORTRAN.

Unifying paradigms occur in other contexts as well. One of them is worth mentioning here: the Web. The paradigm behind the Web is that cyberspace is full of documents, each of which has a URL. By typing in a URL or clicking on an entry backed by a URL, you get the document. In reality, many "documents" are not documents at all, but are generated by a program or shell script when a request comes in. For example, when a user asks an online store for a list of CDs by a particular artist, the document is generated on-the-fly by a program; it certainly did not exist before the query was made.

We have now seen four cases: namely, everything is a tape, file, object, or document. In all four cases, the intention is to unify data, devices, and other resources to make them easier to deal with. Every operating system should have such a unifying data paradigm.

12.2.3 The System Call Interface

If one believes in Corbató's dictum of minimal mechanism, then the operating system should provide as few system calls as it can get away with, and each one should be as simple as possible (but no simpler). A unifying data paradigm can play a major role in helping here. For example, if files, processes, I/O devices, and much more all look like files or objects, then they can all be read with a single

IBM Ex-1009
Page 548

read system call. Otherwise it may be necessary to have separate calls for read_file, read_proc, and read_tty, among others.

In some cases, system calls may appear to need several variants, but it is often better practice to have one system call that handles the general case, with different library procedures to hide this fact from the programmers. For example, UNIX has a system call for overlaying a process virtual address space, exec. The most general call is

```
exec(name, argp, envp);
```

which loads the executable file *name* and gives it arguments pointed to by *argp* and environment variables pointed to by *envp*. Sometimes it is convenient to list the arguments explicitly, so the library contains procedures that are called as follows:

```
execl(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

All these procedures do is stick the arguments in an array and then call exec to do the work. This arrangement is the best of both worlds: a single straightforward system call keeps the operating system simple, yet the programmer gets the convenience of various ways to call exec.

Of course, trying to have one call to handle every possible case can easily get out of hand. In UNIX creating a process requires two calls: fork followed by exec. The former has no parameters; the latter has three parameters. In contrast, the Win32 API call for creating a process, CreateProcess, has 10 parameters, one of which is a pointer to a structure with an additional 18 parameters.

A long time ago, someone should have asked the question: "Would something awful happen if we left some of these out?" The truthful answer would have been: "In some cases programmers might have to do more work to achieve a particular effect, but the net result would have been a simpler, smaller, and more reliable operating system." Of course, the person proposing the 10+ 18 parameter version might have added: "But users like all these features." The rejoinder to that might have been: "They like systems that use little memory and never crash even more." Trade-offs between more functionality at the cost of more memory are at least visible and can be given a price tag (since the price of memory is known). However, it is hard to estimate the additional crashes per year some feature will add and whether the users would make the same choice if they knew the hidden price. This effect can be summarized in Tanenbaum's first law of software:

Adding more code adds more bugs

Adding more features adds more code and thus adds more bugs. Programmers who believe adding new features does not add new bugs are either new to computers or believe the tooth fairy is out there watching over them.

Simplicity is not the only issue that comes out when designing system calls. An important consideration is Lampson's (1984) slogan:

Don't hide power.

If the hardware has an extremely efficient way of doing something, it should be exposed to the programmers in a simple way and not buried inside some other abstraction. The purpose of abstractions is to hide undesirable properties, not hide desirable ones. For example, suppose that the hardware has a special way to move large bitmaps around the screen (i.e., the video RAM) at high speed. It would be justified to have a new system call to get at this mechanism, rather than just providing ways to read video RAM into main memory and write it back again. The new call should just move bits and nothing else. If a system call is fast, users can always build more convenient interfaces on top of it. If it is slow, nobody will use it.

Another design issue is connection-oriented versus connectionless calls. The standard UNIX and Win32 system calls for reading a file are connection-oriented. First you open a file, then you read it, finally you close it. Some remote file access protocols are also connection-oriented. For example, to use FTP, the user first logs in to the remote machine, reads the files, and then logs out.

On the other hand, some remote file access protocols are connectionless. NFS is connectionless, as we saw in Chap. 10. Each NFS call stands on its own, so files are not opened before reading or writing. And, of course, they do not have to be closed afterward, either. The Web is also connectionless: to read a Web page you just ask for it; there is no advance setup required (a TCP connection *is* required, but this is at a lower level of protocol; the HTTP protocol for accessing the Web itself is connectionless).

The trade-off between any connection-oriented mechanism and a connectionless one is the additional work required to set up the mechanism (e.g., open the file), and the gain from not having to do it on (possibly many) subsequent calls. For file I/O on a single machine, where the setup cost is low, probably the standard way (first open, then use) is the best way. For remote file systems, a case can be made both ways.

Another issue relating to the system call interface is its visibility. The list of POSIX-mandated system calls is easy to find. All UNIX systems support these, as well as a small number of other calls, but the complete list is always public. In contrast, Microsoft has never made the list of Windows 2000 system calls public. Instead the Win32 API and other APIs have been made public, but these contain vast numbers of library calls (over 13,000 for Windows 2000), but only a small number are true system calls. The argument for making all the system calls public is that it lets programmers know what is cheap (functions performed in user space) and what is expensive (kernel calls). The argument for not making them public is that it gives the implementers the flexibility of changing the actual underlying system calls to make them better without breaking user programs.

IBM Ex-1009
Page 550

12.3 IMPLEMENTATION

Turning away from the user and system call interfaces, let us now take a look at how to implement an operating system. In the next eight sections we will examine some general conceptual issues relating to implementation strategies. After that we will look at some low-level techniques that are often helpful.

12.3.1 System Structure

Probably the first decision the implementers have to make is what the system structure should be. We examined the main possibilities in Sec. 1.7, but will review them here. An unstructured monolithic design is really not a good idea, except maybe for a tiny operating system in, say, a refrigerator, but even there it is arguable.

Layered Systems

A reasonable approach that has been well established over the years is a layered system. Dijkstra's THE system (Fig. 1-25) was the first layered operating system. UNIX (Fig. 10-3) and Windows 2000 (Fig. 11-7) also have a layered structure, but the layering in both of them is more a way of trying to describe the system than a real guiding principle that was used in building the system.

For a new system, designers choosing to go this route should *first* very carefully choose the layers and define the functionality of each one. The bottom layer should always try to hide the worst idiosyncracies of the hardware, as the HAL does in Fig. 11-7. Probably the next layer should handle interrupts, context switching, and the MMU, so above this level, the code is mostly machine independent. Above this, different designers will have different tastes (and biases). One possibility is to have layer 3 manage threads, including scheduling and interthread synchronization, as shown in Fig. 12-2. The idea here is that starting at layer 4 we have proper threads that are scheduled normally and synchronize using a standard mechanism (e.g., mutexes).

In layer 4 we might find the device drivers, each one running as a separate thread, with its own state, program counter, registers, etc., possibly (but not necessarily) within the kernel address space. Such a design can greatly simplify the I/O structure because when an interrupt occurs, it can be converted into an unlock on a mutex and a call to the scheduler to (potentially) schedule the newly readied thread that was blocked on the mutex. MINIX uses this approach, but in UNIX, Linux, and Windows 2000, the interrupt handlers run in a kind of no-man's land, rather than as proper threads that can be scheduled, suspended, etc. Since a huge amount of the complexity of any operating system is in the I/O, any technique for making it more tractable and encapsulated is worth considering.

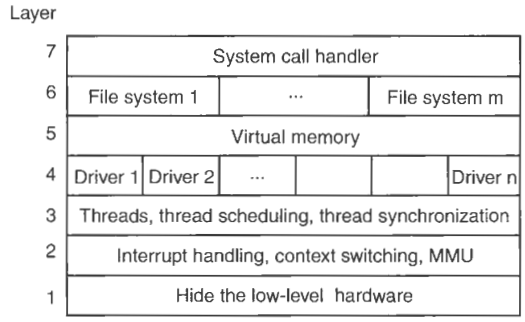


Figure 12-2. One possible design for a modern layered operating system.

Above layer 4, we would expect to find virtual memory, one or more file systems, and the system call handlers. If the virtual memory is at a lower level than the file systems, then the block cache can be paged out, allowing the virtual memory manager to dynamically determine how the real memory should be divided among user pages and kernel pages, including the cache. Windows 2000 works this way.

Exokernels

While layering has its supporters among system designers, there is also another camp that has precisely the opposite view (Engler et al., 1995). Their view is based on the **end-to-end argument** (Saltzer et al., 1984). This concept says that if something has to be done by the user program itself, it is wasteful to do it in a lower layer as well.

Consider an application of that principle to remote file access. If a system is worried about data being corrupted in transit, it should arrange for each file to be checksummed at the time it is written and the checksum stored along with the file. When a file is transferred over a network from the source disk to the destination process, the checksum is transferred, too, and also recomputed at the receiving end. If the two disagree, the file is discarded and transferred again.

This check is more accurate than using a reliable network protocol since it also catches disk errors, memory errors, software errors in the routers, and other errors besides bit transmission errors. The end-to-end argument says that using a reliable network protocol is then not necessary, since the end point (the receiving process) has enough information to verify the correctness of the file itself. The only reason for using a reliable network protocol in this view is for efficiency, that is, catching and repairing transmission errors earlier.

The end-to-end argument can be extended to almost all of the operating system. It argues for not having the operating system do anything that the user

program can do itself. For example, why have a file system? Just let the user read and write a portion of the raw disk in a protected way. Of course, most users like having files, but the end-to-end argument says that the file system should be a library procedure linked with any program that needs to use files. This approach allows different programs to have different file systems. This line of reasoning says that all the operating system should do is securely allocate resources (e.g., the CPU and the disks) among the competing users. The Exokernel is an operating system built according to the end-to-end argument (Engler et al., 1995).

Client-Server Systems

A compromise between having the operating system do everything and the operating system do nothing is to have the operating system do a little bit. This design leads to a microkernel with much of the operating system running as user-level server processes as illustrated in Fig. 1-27. This is the most modular and flexible of all the designs. The ultimate in flexibility is to have each device driver also run as a user process, fully protected against the kernel and other drivers. Getting the drivers out of the kernel would eliminate the largest source of instability in any operating system—buggy third-party drivers—and would be a tremendous win in terms of reliability.

Of course, device drivers need to access the hardware device registers, so some mechanism is needed to provide this. If the hardware permits, each driver process could be given access to only those I/O devices it needs. For example, with memory-mapped I/O, each driver process could have the page for its device mapped in, but no other device pages. If the I/O port space can be partially protected, the correct portion of it could be made available to each driver.

Even if no hardware assistance is available, the idea can still be made to work. What is then needed is a new system call, available only to device driver processes, supplying a list of (port, value) pairs. What the kernel does is first check to see if the process owns all the ports in the list. If so, it then copies the corresponding values to the ports to initiate device I/O. A similar call can be used to read I/O ports in a protected way.

This approach keeps device drivers from examining (and damaging) kernel data structures, which is (for the most part) a good thing. An analogous set of calls could be made available to allow driver processes to read and write kernel tables, but only in a controlled way and with the approval of the kernel.

The main problem with this approach, and microkernels in general, is the performance hit all the extra context switches cause. However, virtually all work on microkernels was done many years ago when CPUs were much slower. Nowadays, applications that use every drop of CPU power and cannot tolerate a small loss of performance, are few and far between. After all, when running a word processor or Web browser, the CPU is probably idle 90% of the time. If a microkernel-based operating system turned an unreliable 900-MHz system into a

reliable 800-MHz system, probably few users would complain. After all, most of them were quite happy only a few years ago when they got their previous computer, at the then-stupendous speed of 100 MHz.

Extensible Systems

With the client-server systems discussed above, the idea was to get as much out of the kernel as possible. The opposite approach is to put more modules into the kernel, but in a protected way. The key word here is *protected*, of course. We studied some protection mechanisms in Sec. 9.5.6 that were initially intended for importing applets over the Internet, but are equally applicable to inserting foreign code into kernel. The most important ones are sandboxing and code signing as interpretation is not really practical for kernel code.

Of course, an extensible system by itself is not a way to structure an operating system. However, by starting with a minimal system consisting of little more than a protection mechanism and then adding protected modules to the kernel one at a time until reaching the functionality desired, a minimal system can be built for the application at hand. In this view, a new operating system can be tailored to each application by including only the parts it requires. Paramecium is an example of such a system (Van Doorn, 2001).

Kernel Threads

Another issue relevant here is that of system threads, no matter which structuring model is chosen. It is sometimes convenient to allow kernel threads to exist, separate from any user process. These threads can run in the background, writing dirty pages to disk, swapping processes between main memory and disk, and so on. In fact, the kernel itself can be structured entirely of such threads, so that when a user does a system call, instead of the user's thread executing in kernel mode, the user's thread blocks and passes control to a kernel thread that takes over to do the work.

In addition to kernel threads running in the background, most operating systems start up many daemon processes in the background, too. While these are not part of the operating system, they often perform "system" type activities. These might including getting and sending email and serving various kinds of requests for remote users, such as FTP and Web pages.

12.3.2 Mechanism versus Policy

Another principle that helps architectural coherence, along with keeping things small and well structured, is that of separating mechanism from policy. By putting the mechanism in the operating system and leaving the policy to user processes, the system itself can be left unmodified, even if there is a need to change policy. Even if the policy module has to be kept in the kernel, it should be

IBM Ex-1009
Page 554

isolated from the mechanism, if possible, so that changes in the policy module do not affect the mechanism module.

To make the split between policy and mechanism clearer, let us consider two real-world examples. As a first example, consider a large company that has a payroll department, which is in charge of paying the employees' salaries. It has computers, software, blank checks, agreements with banks, and more mechanism for actually paying out the salaries. However, the policy—determining who gets paid how much—is completely separate and is decided by management. The payroll department just does what it is told to do.

As the second example, consider a restaurant. It has the mechanism for serving diners, including tables, plates, waiters, a kitchen full of equipment, agreements with credit card companies, and so on. The policy is set by the chef, namely, what is on the menu. If the chef decides that tofu is out and big steaks are in, this new policy can be handled by the existing mechanism.

Now let us consider some operating system examples. First, consider thread scheduling. The kernel could have a priority scheduler, with k priority levels. The mechanism is an array, indexed by priority level, as shown in Fig. 10-11 or Fig. 11-19. Each entry is the head of a list of ready threads at that priority level. The scheduler just searches the array from highest priority to lowest priority, selecting the first threads it hits. The policy is setting the priorities. The system may have different classes of users, each with a different priority, for example. It might also allow user processes to set the relative priority of its threads. Priorities might be increased after completing I/O or decreased after using up a quantum. There are numerous other policies that could be followed, but the idea here is the separation between setting policy and carrying it out.

A second example is paging. The mechanism involves MMU management, keeping lists of occupied pages and free pages, and code for shuttling pages to and from disk. The policy is deciding what to do when a page fault occurs. It could be local or global, LRU-based or FIFO-based, or something else, but this algorithm can (and should) be completely separate from the mechanics of actually managing the pages.

A third example is allowing modules to be loaded into the kernel. The mechanism concerns how they are inserted, how they are linked, what calls they can make, and what calls can be made on them. The policy is determining who is allowed to load a module into the kernel and which modules. Maybe only the superuser can load modules, but maybe any user can load a module that has been digitally signed by the appropriate authority.

12.3.3 Orthogonality

Good system design consists of separate concepts that can be combined independently. For example, in C, there are primitive data types including integers, characters, and floating-point numbers. There are also mechanisms for com-

bining data types, including arrays, structures, and unions. These ideas combine independently, allowing arrays of integers, arrays of characters, structure and union members that are floating-point numbers, etc. In fact, once a new data type has been defined, such as an array of integers, it can be used as if it were a primitive data type, for example as a member of a structure or a union. The ability to combine separate concepts independently is called **orthogonality**. It is direct consequence of the simplicity and completeness principles.

The concept of orthogonality also occurs in operating systems in various disguises. One example is the Linux clone system call, which creates a new thread. The call has a bitmap as a parameter, which allows the address space, working directory, file descriptors, and signals to shared or copied individually. If everything is copied, we have a new process, the same as fork. If nothing is copied, a new thread is created in the current process. However, it is also possible to create intermediate forms of sharing not possible in traditional UNIX systems. By separating out the various features and making them orthogonal, a finer degree of control is possible.

Another use of orthogonality is the separation of the process concept from the thread concept in Windows 2000. A process is a container for resources, nothing more and nothing less. A thread is a schedulable entity. When one process is given a handle for another process, it does not matter how many threads it has. When a thread is scheduled, it does not matter which process it belongs to. These concepts are orthogonal.

Our last example of orthogonality comes from UNIX. Process creation there is done in two steps: fork plus exec. Creating the new address space and loading it with a new memory image are separate, allowing things to be done in between (such as manipulating file descriptors). In Windows 2000, these two steps cannot be separated, that is, the concepts of making a new address space and filling it in are not orthogonal there. The Linux sequence of clone plus exec is yet more orthogonal, since there are even more fine-grained building blocks available. As a general rule, having a small number of orthogonal elements that can be combined in many ways leads to a small, simple, and elegant system.

12.3.4 Naming

Most long-lived data structures used by an operating system have some kind of name or identifier by which they can be referred. Obvious examples are login names, file names, device names, process IDs, and so on. How these names are constructed and managed is an important issue in system design and implementation.

Names designed for people to use are character-string names in ASCII or Unicode and are usually hierarchical. Directory paths, for example one such as */usr/ast/books/mos2/chap-12*, are clearly hierarchical, indicating a series of directories to search starting at the root. URLs are also hierarchical. For example,

IBM Ex-1009
Page 556

www.cs.vu.nl/~ast/ indicates a specific machine (*www*) in a specific department (*cs*) at specific university (*vu*) in a specific country (*nl*). The part after the slash indicates a specific file on the designated machine, in this case, by convention, *www/index.html* in ast's home directory. Note that URLs (and DNS addresses in general, including email addresses) are "backward," starting at the bottom of the tree and going up, unlike file names, which start at the top of the tree and go down. Another way of looking at this is whether the tree is written from the top starting at the left and going right or starting at the right and going left.

Often naming is done at two levels: external and internal. For example, files always have a character-string name for people to use. In addition, there is almost always an internal name that the system uses. In UNIX, the real name of a file is its i-node number; the ASCI name is not used at all internally. In fact, it is not even unique since a file may have multiple links to it. The analogous internal name in Windows 2000 is the file's index in the MFT. The job of the directory is to provide the mapping between the external name and the internal name, as shown in Fig. 12-3.

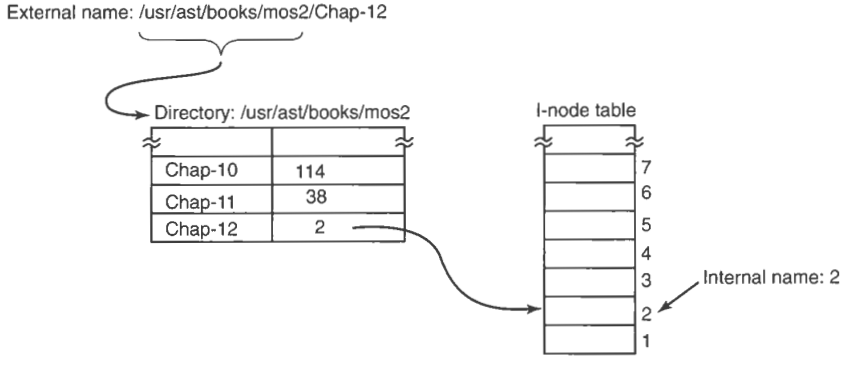


Figure 12-3. Directories are used to map external names onto internal names.

In many cases (such as the file name example given above), the internal name is an unsigned integer that serves as an index into a kernel table. Other examples of table-index names are file descriptors in UNIX and object handles in Windows 2000. Note that neither of these has any external representation. They are strictly for use by the system and running processes. In general, using table indices for transient names that are lost when the system is rebooted is a good idea.

Operating systems often support multiple name spaces, both external and internal. For example, in Chap. 11 we looked at three external name spaces supported by Windows 2000: file names, object names, and registry names (and there is also the Active Directory name space, which we did not look at). In addition, there are innumerable internal name spaces using unsigned integers, for example, object handles, MFT entries, etc. Although the names in the external name spaces

IBM Ex-1009
Page 557

are all Unicode strings, looking up a file name in the registry will not work, just as using an MFT index in the object table will not work. In a good design, considerable thought is given to how many names spaces are needed, what the syntax of names is in each one, how they can be told apart, whether absolute and relative names exist, and so on.

12.3.5 Binding Time

As we have just seen, operating systems use various kinds of names to refer to objects. Sometimes the mapping between a name and an object is fixed, but sometimes it is not. In the latter case, it may matter when the name is bound to the object. In general, **early binding** is simple, but is not flexible, whereas **late binding** is more complicated but often more flexible.

To clarify the concept of binding time, let us look at some real-world examples. An example of early binding is the practice of some colleges to allow parents to enroll a baby at birth and prepay the current tuition. When the student shows up 18 years later, the tuition is fully paid up, no matter how high it may be at that moment.

In manufacturing, ordering parts in advance and maintaining an inventory of them is early binding. In contrast, just-in-time manufacturing requires suppliers to be able to provide parts on the spot, with no advance notice required. This is late binding.

Programming languages often support multiple binding times for variables. Global variables are bound to a particular virtual address by the compiler. This exemplifies early binding. Variables local to a procedure are assigned a virtual address (on the stack) at the time the procedure is invoked. This is intermediate binding. Variables stored on the heap (those allocated by *malloc* in C or *new* in Java) are assigned virtual addresses only at the time they are actually used. Here we have late binding.

Operating systems often use early binding for most data structures, but occasionally use late binding for flexibility. Memory allocation is a case in point. Early multiprogramming systems on machines lacking address relocation hardware had to load a program at some memory address and relocate it to run there. If it was ever swapped out, it had to be brought back at the same memory address or it would fail. In contrast, paged virtual memory is a form of late binding. The actual physical address corresponding to a given virtual address is not known until the page is touched and actually brought into memory.

Another example of late binding is window placement in a GUI. In contrast to the early graphical systems, in which the programmer had to specify the absolute screen coordinates for all images on the screen, in modern GUIs, the software uses coordinates relative to the window's origin, but that is not determined until the window is put on the screen, and it may even be changed later.

IBM Ex-1009
Page 558

12.3.6 Static versus Dynamic Structures

Operating system designers are constantly forced to choose between static and dynamic data structures. Static ones are always simpler to understand, easier to program, and faster in use; dynamic ones are more flexible. An obvious example is the process table. Early systems simply allocated a fixed array of per-process structures. If the process table consisted of 256 entries, then only 256 processes could exist at any one instant. An attempt to create a 257th one would fail for lack of table space. Similar considerations held for the table of open files (both per user and system wide), and many other kernel tables.

An alternative strategy is to build the process table as a linked list of minitables, initially just one. If this table fills up, another one is allocated from a global storage pool and linked to the first one. In this way, the process table cannot fill up until all of kernel memory is exhausted.

On the other hand, the code for searching the table becomes more complicated. For example, the code for searching a static process table to see if a given PID, *pid*, is given in Fig. 12-4. It is simple and efficient. Doing the same thing for a linked list of minitables is more work.

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
     if (p->proc_pid == pid) {
          found = 1;
          break;
     }
}
```

Figure 12-4. Code for searching the process table for a given PID.

Static tables are best when there is plenty of memory or table utilizations can be guessed fairly accurately. For example, in a single-user system, it is unlikely that the user will start up more than 32 processes at once and it is not a total disaster if an attempt to start a 33rd one fails.

Yet another alternative is to use a fixed-size table, but if it fills up, allocate a new fixed-size table, say, twice as big. The current entries are then copied over to the new table and the old table is returned to the free storage pool. In this way, the table is always contiguous rather than linked. The disadvantage here is that some storage management is needed and the address of the table is now a variable instead of a constant.

A similar issue holds for kernel stacks. When a thread switches to kernel mode, or a kernel-mode thread is run, it needs a stack in kernel space. For user threads, the stack can be initialized to run down from the top of the virtual address space, so the size need not be specified in advance. For kernel threads, the size must be specified in advance because the stack takes up some kernel virtual

IBM Ex-1009
Page 559

address space and there may be many stacks. The question is: how much space should each one get? The trade-offs here are similar to those for the process table.

Another static-dynamic trade-off is process scheduling. In some systems, especially real-time ones, the scheduling can be done statically in advance. For example, an airline knows what time its flights will leave weeks before their departure. Similarly, multimedia systems know when to schedule audio, video, and other processes in advance. For general-purpose use, these considerations do not hold and scheduling must be dynamic.

Yet another static-dynamic issue is kernel structure. It is much simpler if the kernel is built as a single binary program and loaded into memory to run. The consequence of this design, however, is that adding a new I/O device requires a relinking of the kernel with the new device driver. Early versions of UNIX worked this way, and it was quite satisfactory in a minicomputer environment when adding new I/O devices was a rare occurrence. Nowadays, most operating systems allow code to be added to the kernel dynamically, with all the additional complexity that entails.

12.3.7 Top-Down versus Bottom-Up Implementation

While it is best to design the system top down, in theory it can be implemented top down or bottom up. In a top-down implementation, the implementers start with the system call handlers and see what mechanisms and data structures are needed to support them. These procedures are written and so on until the hardware is reached.

The problem with this approach is that it is hard to test anything with only the top-level procedures available. For this reason, many developers find it more practical to actually build the system bottom up. This approach entails first writing code that hides the low-level hardware, essentially the HAL in Fig. 11-7. Interrupt handling and the clock driver are also needed early on.

Then multiprogramming can be tackled, along with a simple scheduler (e.g., round-robin scheduling). At this point it should be possible to test the system to see if it can run multiple processes correctly. If that works, it is now time to begin the careful definition of the various tables and data structures needed throughout the system, especially those for process and thread management and later memory management. I/O and the file system can wait initially, except for a primitive way to read the keyboard and write to the screen for testing and debugging. In some cases, the key low-level data structures should be protected by allowing access only through specific access procedures—in effect, object-oriented programming, no matter what the programming language is. As lower layers are completed, they can be tested thoroughly. In this way, the system advances from the bottom up, much the way contractors build tall office buildings.

If a large team is available, an alternative approach is to first make a detailed design of the whole system, and then assign different groups to write different

IBM Ex-1009
Page 560

modules. Each one tests its own work in isolation. When all the pieces are ready, they are integrated and tested. The problem with this line of attack is that if nothing works initially, it may be hard to isolate whether one or more modules are malfunctioning, or one group misunderstood what some other module was supposed to do. Nevertheless, with large teams, this approach is often used to maximize the amount of parallelism in the programming effort.

12.3.8 Useful Techniques

We have just looked at some abstract ideas for system design and implementation. Now we will examine a number of useful concrete techniques for system implementation. There are numerous other ones, of course, but space limitations restrict us to just a few of them.

Hiding the Hardware

A lot of hardware is ugly. It has to be hidden early on (unless it exposes power, which most hardware does not). Some of the very low-level details can be hidden by a HAL-type layer of the type shown in Fig. 12-2. However, many hardware details cannot be hidden this way.

One thing that deserves early attention is how to deal with interrupts. They make programming unpleasant, but operating systems have to deal with them. One approach is to turn them into something else immediately. For example, every interrupt could be turned into a pop-up thread instantly. At that point we are dealing with threads, rather than interrupts.

A second approach is to convert each interrupt into an unlock operation on a mutex that the corresponding driver is waiting on. Then the only effect of an interrupt is to cause some thread to become ready.

A third approach is convert an interrupt into a message to some thread. The low-level code just builds a message telling where the interrupt came from, enqueues it, and calls the scheduler to (potentially) run the handler, which was probably blocked waiting for the message. All these techniques, and other ones like them, all try to convert interrupts into thread synchronization operations. Having each interrupt handled by a proper thread in a proper context is easier to manage than running a handler in the arbitrary context that it happened to occur in. Of course, this must be done efficiently, but deep within the operating system, everything must be done efficiently.

Most operating systems are designed to run on multiple hardware platforms. These platforms can differ in terms of the CPU chip, MMU, word length, RAM size, and other features that cannot easily be masked by the HAL or equivalent. Nevertheless, it is highly desirable to have a single set of source files that are used to generate all versions; otherwise each bug that later turns up must be fixed multiple times in multiple sources, with the danger that the sources drift apart.

IBM Ex-1009
Page 561

Some hardware differences, such as RAM size, can be dealt with by having the operating system determine the value at boot time and keep it in a variable. Memory allocators, for example, can use the RAM size variable to determine how big to make the block cache, page tables, etc. Even static tables such as the process table can be sized based on the total memory available.

However, other differences, such as different CPU chips, cannot be solved by having a single binary that determines at run time which CPU it is running on. One way to tackle the problem of one source and multiple targets is to use conditional compilation. In the source files, certain compile-time flags are defined for the different configurations and these are used to bracket code that is dependent on the CPU, word length, MMU, etc. For example, imagine an operating system that is to run on the Pentium and UltraSPARC chips, which need different initialization code. The *init* procedure could be written as illustrated in Fig. 12-5(a). Depending on the value of *CPU*, which is defined in the header file *config.h*, one kind of initialization or other is done. Because the actual binary contains only the code needed for the target machine, there is no loss of efficiency this way.

```
#include "config.h"

init( )
{
#if (CPU == PENTIUM)
/* Pentium initialization here. */
#endif

#if (CPU == ULTRASPARC)
/* UltraSPARC initialization here. */
#endif
```

(a)

```
}
```

```
#include "config.h"

#if (WORD_LENGTH == 32)
typedef int Register;
#endif

#if (WORD_LENGTH == 64)
typedef long Register;
#endif

Register R0, R1, R2, R3;
```

(b)

Figure 12-5. (a) CPU-dependent conditional compilation. (b) Word-length dependent conditional compilation.

As a second example, suppose there is a need for a data type *Register*, which should be 32 bits on the Pentium and 64 bits on the UltraSPARC. This could be handled by the conditional code of Fig. 12-5(b) (assuming that the compiler produces 32-bit ints and 64-bit longs). Once this definition has been made (probably in a header file included everywhere), the programmer can just declare variables to be of type *Register* and know they will be the right length.

The header file, *config.h*, has to be defined correctly, of course. For the Pentium it might be something like this:

```
#define CPU PENTIUM
#define WORD_LENGTH 32
```

IBM Ex-1009
Page 562

To compile the system for the UltraSPARC, a different *config.h* would be used, with the correct values for the UltraSPARC, probably something like

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

Some readers may be wondering why *CPU* and *WORD_LENGTH* are handled by different macros. We could easily have bracketed the definition of *Register* with a test on *CPU*, setting it to 32 bits for the Pentium and 64 bits for the UltraSPARC. However, this is not a good idea. Consider what happens when we later port the system to the 64-bit Intel Itanium. We would have to add a third conditional to Fig. 12-5(b) for the Itanium. By doing it as we have, all we have to do is include the line

```
#define WORD_LENGTH 64
```

to the *config.h* file for the Itanium.

This example illustrates the orthogonality principle we discussed earlier. Those items that are CPU-dependent should be conditionally compiled based on the *CPU* macro and those things that are word-length dependent should use the *WORD_LENGTH* macro. Similar considerations hold for many other parameters.

Indirection

It is sometimes said that there is no problem in computer science that cannot be solved with another level of indirection. While something of an exaggeration, there is definitely a grain of truth here. Let us consider some examples. On Pentium-based systems, when a key is depressed, the hardware generates an interrupt and puts the key number, rather than an ASCII character code, in a device register. Furthermore, when the key is released later, a second interrupt is generated, also with the key number. This indirection allows the operating system the possibility of using the key number to index into a table to get the ASCII character, which makes it easy to handle the many keyboards used around the world in different countries. Getting both the depress and release information makes it possible to use any key as a shift key since the operating system knows the exact sequence the keys were depressed and released.

Indirection is also used on output. Programs can write ASCII characters to the screen, but these are interpreted as indices into a table for the current output font. The table entry contains the bitmap for the character. This indirection makes it possible to separate characters from fonts.

Another example of indirection is the use of major device numbers in UNIX. Within the kernel there is a table indexed by major device number for the block devices and another one for the character devices. When a process opens a special file such as */dev/hd0*, the system extracts the type (block or character) and

major and minor device numbers from the i-node and indexes into the appropriate driver table to find the driver. This indirection makes it easy to reconfigure the system, because programs deal with symbolic device names, not actual driver names.

Yet another example of indirection occurs in message-passing systems that name a mailbox rather than a process as the message destination. By indirecting through mailboxes (as opposed to naming a process as the destination), considerable flexibility can be achieved (e.g., having a secretary handle her boss' messages).

In a sense, the use of macros, such as

```
#define PROC_TABLE_SIZE 256
```

is also a form of indirection, since the programmer can write code without having to know how big the table really is. It is good practice to give symbolic names to all constants (except sometimes −1, 0, and 1), and put these in headers with comments explaining what they are for.

Reusability

It is frequently possible to reuse the same code in slightly different contexts. Doing so is a good idea as it reduces the size of the binary and means that the code has to be debugged only once. For example, suppose that bitmaps are used to keep track of free blocks on the disk. Disk block management can be handled by having procedures *alloc* and *free* that manage the bitmaps.

As a bare minimum, these procedures should work for any disk. But we can go further than that. The same procedures can also work for managing memory blocks, blocks in the file system's block cache, and i-nodes. In fact, they can be used to allocate and deallocate any resources that can be numbered linearly.

Reentrancy

Reentrancy refers for the ability of code to be executed two or more times simultaneously. On a multiprocessor, there is always the danger than while one CPU is executing some procedure, another CPU will start executing it as well, before the first one has finished. In this case, two (or more) threads on different CPUs might be executing the same code at the same time. This situation must be protected against by using mutexes or some other means to protect critical regions.

However, the problem also exists on a uniprocessor. In particular, most of any operating system runs with interrupts enabled. To do otherwise, would lose many interrupts and make the system unreliable. While the operating system is busy executing some procedure, *P*, it is entirely possible that an interrupt occurs and that the interrupt handler also calls *P*. If the data structures of *P* were in an

inconsistent state at the time of the interrupt, the handler will see them in an inconsistent state and fail.

An obvious example where this can happen is if P is the scheduler. Suppose that some process used up its quantum and the operating system was moving it to the end of its queue. Part way through the list manipulation, the interrupt occurs, makes some process ready, and runs the scheduler. With the queues in an inconsistent state, the system will probably crash. As a consequence even on a uniprocessor, it is best that most of the operating system is reentrant, critical data structures are protected by mutexes, and interrupts are disabled at moments when they cannot be tolerated.

Brute Force

Using brute force to solve a problem has acquired a bad name over the years, but it is often the way to go in the name of simplicity. Every operating system has many procedures that are rarely called or operate with so little data that optimizing them is not worthwhile. For example, it is frequently necessary to search various tables and arrays within the system. The brute force algorithm is just leave the table in the order the entries are made and search it linearly when something has to be looked up. If the number of entries is small (say, under 100), the gain from sorting the table or hashing it is small, but the code is far more complicated and more likely to have bugs in it.

Of course, for functions that are on the critical path, say, context switching, everything should be done to make them fast, possibly even writing them in (heaven forbid) assembly language. But large parts of the system are not on the critical path. For example, many system calls are rarely called. If there is one fork every 10 sec, and it takes 10 msec to carry out, then even optimizing it to 0 wins only 0.1%. If the optimized code is bigger and buggier, a case can be made not to bother with the optimization.

Check for Errors First

Many system calls can potentially fail for a variety of reasons: the file to be opened belongs to someone else; process creation fails because the process table is full; or a signal cannot be sent because the target process does not exist. The operating system must painstakingly check for every possible error before carrying out the call.

Many system calls also require acquiring resources such as process table slots, i-node table slots, or file descriptors. A general piece of advice that can save a lot of grief is to first check to see if the system call can actually be carried out before acquiring any resources. This means putting all the tests at the beginning of the procedure that executes the system call. Each test should be of the form

```
if (error_condition) return(ERROR_CODE);
```

IBM Ex-1009
Page 565

If the call gets all the way through the gauntlet of tests, then it is certain that it will succeed. At that point resources can be acquired.

Interspersing the tests with resource acquisition means that if some test fails along the way, all the resources acquired up to that point must be returned. If an error is made here and some resource is not returned, no damage is done immediately. For example, one process table entry may just become permanently unavailable. However, over a period of time, this bug may be triggered multiple times. Eventually, most or all the process table entries may become unavailable, leading to a system crash in an extremely unpredictable and difficult to debug way.

Many systems suffer from this problem in the form of memory leaks. Typically, the program calls *malloc* to allocate space but forgets to call *free* later to release it. Ever so gradually, all of memory disappears until the system is rebooted.

Engler et al. (2000) have proposed an interesting way to check for some of these errors at compile time. They observed that the programmer knows many invariants that the compiler does not know, such as when you lock a mutex, all paths starting at the lock must contain an unlock and no more locks of the same mutex. They have devised a way for the programmer to tell the compiler this fact and instruct it to check all the paths at compile time for violations of the invariant. The programmer can also specify that allocated memory must be released on all paths and many other conditions as well.

12.4 PERFORMANCE

All things being equal, a fast operating system is better than a slow one. However, a fast unreliable operating system is not as good as a reliable slow one. Since complex optimizations often lead to bugs, it is important to use them sparingly. This notwithstanding, there are places where performance is critical and optimizations are well worth the effort. In the following sections, we will look at some general techniques that can be used to improve performance in places where that is called for.

12.4.1 Why Are Operating Systems Slow?

Before talking about optimization techniques, it is worth pointing out that the slowness of many operating systems is to a large extent self-inflicted. For example, older operating systems, such as MS-DOS and UNIX Version 7, booted within a few seconds. Modern UNIX systems and Windows 2000 can take minutes to boot, despite running on hardware that is 100 times as fast. The reason is that they are doing much more, wanted or not. A case in point. Plug and play makes it somewhat easier to install a new hardware device, but the price paid is that on

every boot, the operating system has to go out and inspect all the hardware to see if there is anything new out there. This bus scan takes time.

An alternative (and in the author's opinion, better) approach would be to scrap plug-and-play altogether and have an icon on the screen labeled "Install new hardware." Upon installing a new hardware device, the user would click on this icon to start the bus scan, instead of doing it on every boot. The designers of current systems were well aware of this option, of course. They rejected it, basically, because they assumed that the users were too stupid to be able to do this correctly (although they would word it more kindly). This is only one example, but there are many more where the desire to make the system "user-friendly" (or "idiot-proof," depending on your viewpoint) slows the system down all the time for everyone.

Probably the biggest single thing system designers can do to improve performance is to be much more selective about adding new features. The question to ask is not: "Will the users like this?" but "Is this feature worth the inevitable price in code size, speed, complexity, and reliability?" Only if the advantages clearly outweigh the drawbacks should it be included. Programmers have a tendency to assume that code size and bug count will be 0 and speed will be infinite. Experience shows this view to be a bit optimistic.

Another factor that plays a role is product marketing. By the time version 4 or 5 of some product has hit the market, probably all the features that are actually useful have been included and most of the people who need the product already have it. To keep sales going, many manufacturers nevertheless continue to produce new versions, with more features, just so they can sell their existing customers upgrades. Adding new features just for the sake of adding new features may help sales but rarely helps performance.

12.4.2 What Should Be Optimized?

As a general rule, the first version of the system should be as straightforward as possible. The only optimizations should be things that are so obviously going to be a problem that they are unavoidable. Having a block cache for the file system is such an example. Once the system is up and running, careful measurements should be made to see where the time is *really* going. Based on these numbers, optimizations should be made where they will help most.

Here is a true story of where an optimization did more harm than good. One of the author's students (who shall here remain nameless) wrote the MINIX *mkfs* program. This program lays down a fresh file system on a newly formatted disk. The student spent about 6 months optimizing it, including putting in disk caching. When he turned it in, it did not work and it required several additional months of debugging. This program typically runs once on the hard disk during the life of the computer, when the system is installed. It also runs once for each floppy disk that is formatted. Each run takes about 2 sec. Even if the unoptimized version

IBM Ex-1009
Page 567

had taken 1 minute, it was a poor use of resources to spend so much time optimizing a program that is used so infrequently.

A slogan that has considerable applicability to performance optimization is

Good enough is good enough.

By this we mean that once performance has achieved a reasonable level, it is probably not worth the effort and complexity to squeeze out the last few percent. If the scheduling algorithm is reasonably fair and keeps the CPU busy 90% of the time, it is doing its job. Devising a far more complex one that is 5% better is probably a bad idea. Similarly, if the page rate is low enough that it is not a bottleneck, jumping through hoops to get optimal performance is usually not worth it. Avoiding disaster is far more important than getting optimal performance, especially since what is optimal with one load may not be optimal with another.

12.4.3 Space-Time Trade-offs

One general approach to improving performance is to trade off time versus space. It frequently occurs in computer science that there is a choice between an algorithm that uses little memory but is slow and an algorithm that uses much more memory but is faster. When making an important optimization, it is worth looking for algorithms that gain speed by using more memory or conversely save precious memory by doing more computation.

One technique that is often helpful is to replace small procedures by macros. Using a macro eliminates the overhead normally associated with a procedure call. The gain is especially significant if the call occurs inside a loop. As an example, suppose we use bitmaps to keep track of resources and frequently need to know how many units are free in some portion of the bitmap. For this purpose we need a procedure, *bit_count*, that counts the number of 1 bits in a byte. The straightforward procedure is given in Fig. 12-6(a). It loops over the bits in a byte counting them one at a time.

This procedure has two sources of inefficiency. First, it must be called, stack space must be allocated for it, and it must return. Every procedure call has this overhead. Second, it contains a loop, and there is always some overhead associated with a loop.

A completely different approach is to use the macro of Fig. 12-6(b). It is an inline expression that computes the sum of the bits by successively shifting the argument, masking out everything but the low-order bit, and adding up the eight terms. The macro is hardly a work of art, but it appears in the code only once. When the macro is called, for example, by

```
sum = bit_count(table[i]);
```

the macro call looks identical to the call of the procedure. Thus other than one somewhat messy definition, the code does not look any worse in the macro case

IBM Ex-1009
Page 568

```
#define BYTE_SIZE 8                          /* A byte contains 8 bits */

int bit_count(int byte)
{                                            /* Count the bits in a byte. */
     int i, count = 0;

     for (i = 0; i < BYTE_SIZE; i++)         /* loop over the bits in a byte */
          if ((byte >> i) & 1) count++;      /* if this bit is a 1, add to count */
     return(count);                          /* return sum */
}
```

(a)

```
/*Macro to add up the bits in a byte and return the sum. */
#define bit_count(b) (b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
                     ((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1)
```

(b)

```
/*Macro to look up the bit count in a table. */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, ...};
#define bit_count(b) (int) bits[b]
```

(c)

Figure 12-6. (a) A procedure for counting bits in a byte. (b) A macro to count the bits.

than in the procedure case, but it is much more efficient since it eliminates both the procedure call overhead and the loop overhead.

We can take this example one step further. Why compute the bit count at all? Why not look it up in a table? After all, there are only 256 different bytes, each with a unique value between 0 and 8. We can declare a 256-entry table, *bits*, with each entry initialized (at compile time) to the bit count corresponding to that byte value. With this approach no computation at all is needed at run time, just one indexing operation. A macro to do the job is given in Fig. 12-6(c).

This is a clear example of trading computation time against memory. However, we could go still further. If the bit counts for whole 32-bit words are needed, using our *bit_count* macro, we need to perform four lookups per word. If we expand the table to 65,536 entries, we can suffice with two lookups per word, at the price of a much bigger table.

Looking answers up in tables can be used in other ways. For example, in Chap. 7 we saw how JPEG image compression works, with fairly complex discrete cosine transformations. An alternative compression technique, GIF, uses table lookup to encode 24-bit RGB pixels. However, GIF only works on images with 256 or fewer colors. For each image to be compressed, a palette of 256 entries is constructed, each entry containing one 24-bit RGB value. The compressed

IBM Ex-1009
Page 569

image then consists of an 8-bit index for each pixel instead of a 24-bit color value, a gain of a factor of three. This idea is illustrated for a 4×4 section of an image in Fig. 12-7. The original compressed image is shown in Fig. 12-7(a). Each value here is a 24-bit value, with 8 bits each giving the intensity for red, green, and blue. The GIF image is shown in Fig. 12-7(b). Here each value is an 8-bit index into the color palette. The color palette is stored as part of the image file, and is shown in Fig. 12-7(c). Actually, there is more to GIF, but the core of the idea is table lookup.

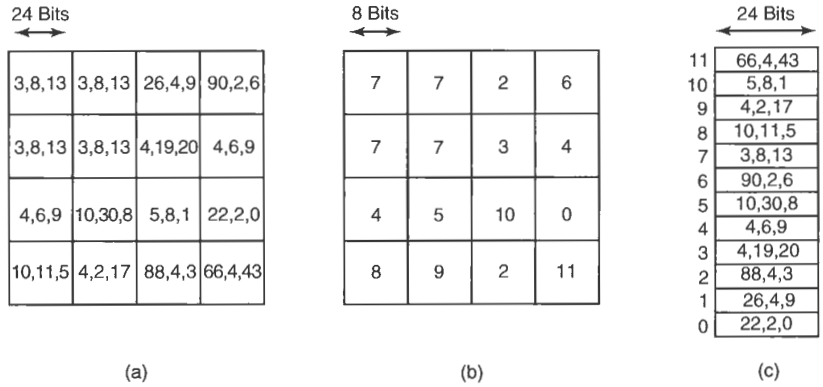


Figure 12-7. (a) Part of an uncompressed image with 24 bits per pixel. (b) The same part compressed with GIF, with 8 bits per pixel. (c) The color palette.

There is another way to reduce image size, and it illustrates a different trade-off. PostScript is a programming language that can be used to describe images. (Actually, any programming language can describe images, but PostScript is tuned for this purpose.) Many printers have a PostScript interpreter built into them to be able to run PostScript programs sent to them.

For example, if there is a rectangular block of pixels all the same color in an image, a PostScript program for the image would carry instructions to place a rectangle at a certain location and fill it with a certain color. Only a handful of bits are needed to issue this command. When the image is received at the printer, an interpreter there must run the program to construct the image. Thus PostScript achieves data compression at the expense of more computation, a different trade-off than table lookup, but a valuable one when memory or bandwidth is scarce.

Other trade-offs often involve data structures. Doubly-linked lists take up more memory than singly-linked lists, but often allow faster access to items. Hash tables are even more wasteful of space, but faster still. In short, one of the main things to consider when optimizing a piece of code is whether using different data structures would make the best time-space tradeoff.

12.4.4 Caching

A well-known technique for improving performance is caching. It is applicable any time it is likely the same result will be needed multiple times. The general approach is to d[Ho the full work the first time, and then save the result in a cache. On subsequent attempts, the cache is first checked. If the result is there, it is used. Otherwise, the full work is done again.

We have already seen the use of caching within the file system to hold some number of recently used disk blocks, thus saving a disk read on each hit. However, caching can be used for many other purposes as well. For example, parsing path names is surprising expensive. Consider the UNIX example of Fig. 6-39 again. To look up */usr/ast/mbox* requires the following disk accesses:

1. Read the i-node for the root directory (i-node 1).
2. Read the root directory (block 1).
3. Read the i-node for */usr* (i-node 6).
4. Read the */usr* directory (block 132).
5. Read the i-node for */usr/ast* (i-node 26).
6. Read the */usr/ast* directory (block 406).

It takes six disk accesses just to discover the i-node number of the file. Then the i-node itself has to be read to discover the disk block numbers. If the file is smaller than the block size (e.g., 1024 bytes), it takes eight disk accesses to read the data.

Some systems optimize path name parsing by caching (path, i-node) combinations. For the example of Fig. 6-39, the cache will certainly hold the first three entries of Fig. 12-8 after parsing */usr/ast/mbox*. The last three entries come from parsing other paths.

Path	I-node number
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/paper.ps	85

Figure 12-8. Part of the i-node cache for Fig. 6-39.

When a path has to be looked up, the name parser first consults the cache and searches it for the longest substring present in the cache. For example, if the path

IBM Ex-1009
Page 571

/usr/ast/grants/stw is presented, the cache returns the fact that */usr/ast* is i-node 26, so the search can start there, eliminating four disk accesses.

A problem with caching paths is that the mapping between file name and i-node number is not fixed for all time. Suppose that the file */usr/ast/mbox* is removed from the system and its i-node reused for a different file owned by a different user. Later, the file */usr/ast/mbox* is created again, and this time it gets i-node 106. If nothing is done to prevent it, the cache entry will now be wrong and subsequent lookups will return the wrong i-node number. For this reason, when a file or directory is deleted, its cache entry and (if it is a directory) all the entries below it must be purged from the cache.

Disk blocks and path names are not the only items that can be cached. I-nodes can be cached, too. If pop-up threads are used to handle interrupts, each one of them requires a stack and some additional machinery. These previously-used threads can also be cached, since refurbishing a used one is easier than creating a new one from scratch (to avoid having to allocate memory). Just about anything that is hard to produce can be cached.

12.4.5 Hints

Cache entries are always correct. A cache search may fail, but if it finds an entry, that entry is guaranteed to be correct and can be used without further ado. In some systems, it is convenient to have a table of **hints**. These are suggestions about the solution, but they are not guaranteed to be correct. The caller must verify the result itself.

A well-known example of hints are the URLs embedded on Web pages. Clicking on a link does not guarantee that the Web page pointed to is there. In fact, the page pointed to may have been removed years ago. Thus the information on the pointing page is really only a hint.

Hints are also used in connection with remote files. The information is the hint tells something about the remote file, such as where it is located. However, the file may have moved or been deleted since the hint was recorded, so a check is always needed to see if it is correct.

12.4.6 Exploiting Locality

Processes and programs do not act at random. They exhibit a fair amount of locality in time and space, and this information can be exploited in various ways to improve performance. One well-known example of spatial locality is the fact that processes do not jump around at random within their address spaces. They tend to use a relatively small number of pages during a given time interval. The pages that a process is actively using can be noted as its working set and the operating system can make sure that when the process is allowed to run, its working set is in memory, thus reducing the number of page faults.

IBM Ex-1009
Page 572

The locality principle also holds for files. When a process has selected a particular working directory, it is likely that many of its future file references will be to files in that directory. By putting all the i-nodes and files for each directory close together on the disk, performance improvements can be obtained. This principle is what underlies the Berkeley Fast File System (McKusick et al., 1984).

Another area in which locality plays a role is in thread scheduling in multiprocessors. As we saw in Chap. 8, one way to schedule threads on a multiprocessor is to try to run each thread on the CPU it last used, in hopes that some of its memory blocks will still be in the memory cache.

12.4.7 Optimize the Common Case

It is frequently a good idea to distinguish between the most common case and the worst possible case and treat them differently. Often the code for the two is quite different. It is important to make the common case fast. For the worst case, if it occurs rarely, it is sufficient to make it correct.

As a first example, consider entering a critical region. Most of the time, the entry will succeed, especially if processes do not spend a lot of time inside critical regions. Windows 2000 takes advantage of this expectation by providing a Win32 API call EnterCriticalSection that atomically tests a flag in user mode (using TSL or equivalent). If the test succeeds, the process just enters the critical region and no kernel call is needed. If the test fails, the library procedure does a down on a semaphore to block the process. Thus in the normal case, no kernel call is needed.

As a second example, consider setting an alarm (using signals in UNIX). If no alarm is currently pending, it is straightforward to make an entry and put it on the timer queue. However, if an alarm is already pending, it has to be found and removed from the timer queue. Since the alarm call does not specify whether there is already an alarm set, the system has to assume worst case, that there is. However, since most of the time there is no alarm pending, and since removing an existing alarm is expensive, it is a good idea to distinguish these two cases.

One way to do this is to keep a bit in the process table that tells whether an alarm is pending. If the bit is off, the easy path is followed (just add a new timer queue entry without checking). If the bit is on, the timer queue must be checked.

12.5 PROJECT MANAGEMENT

Programmers are perpetual optimists. Most of them think that the way to write a program is to run to the keyboard and start typing. Shortly thereafter the fully debugged program is finished. For very large programs, it does not quite work like that. In the following sections we have a bit to say about managing large software projects, especially large operating system projects.

12.5.1 The Mythical Man Month

In his classic book, Fred Brooks, one of the designers of OS/360, who later moved to academia, addresses the question of why it is so hard to build big operating systems (Brooks, 1975, 1995). When most programmers see his claim that programmers can produce only 1000 lines of debugged code per *year* on large projects, they wonder whether Prof. Brooks is living in outer space, perhaps on Planet Bug. After all, most of them can remember an all nighter when they produced a 1000-line program in one night. How could this be the annual output of anybody with an IQ > 50?

What Brooks pointed out is that large projects, with hundreds of programmers, are completely different than small projects and that the results obtained from small projects do not scale to large ones. In a large project, a huge amount of time is consumed planning how to divide the work into modules, carefully specifying the modules and their interfaces, and trying to imagine how the modules will interact, even before coding begins. Then the modules have to be coded and debugged in isolation. Finally, the modules have to be integrated and system as a whole has to be tested. The normal case is that each module works perfectly when tested by itself, but the system crashes instantly when all the pieces are put together. Brooks estimated the work as being

1/3 Planning
1/6 Coding
1/4 Module testing
1/4 System testing

In other words, writing the code is the easy part. The hard part is figuring out what the modules should be and making module *A* correctly talk to module *B*. In a small program written by a single programmer, all that is left over is the easy part.

The title of Brooks' book comes from his assertion that people and time are not interchangeable. There is no such unit as a man-month (or a person-month). If a project takes 15 people 2 years to build, it is inconceivable that 360 people could do it in one month and probably not possible to have 60 people do it in 6 months.

There are three reasons for this effect. First, the work cannot be fully parallelized. Until the planning is done and it has been determined what modules are needed and what their interfaces will be, no coding can be even started. On a two-year project, the planning alone may take 8 months.

Second, to fully utilize a large number of programmers, the work must be partitioned into large numbers of modules so everyone has something to do. Since every module may potentially interact with every other module, the number of module-module interactions that need to be considered grows as the square of the number of modules, that is, as the square of the number of programmers. This

complexity quickly gets out of hand. Careful measurements of 63 software projects have confirmed that the trade-off between people and months is far from linear on large projects (Boehm, 1981).

Third, debugging is highly sequential. Setting ten debuggers on a problem does not find the bug ten times as fast. In fact, ten debuggers are probably slower than one because they will waste so much time talking to each other.

Brooks sums up his experience with trading-off people and time in Brooks' law:

Adding manpower to a late software project makes it later.

The problem with adding people is that they have to be trained in the project, the modules have to be redivided to match the larger number of programmers now available, many meetings will be needed to coordinate all the efforts and so on. Abdel-Hamid and Madnick (1991) confirmed this law experimentally. A slightly irreverent way of restating Brooks law is

It takes 9 months to bear a child, no matter how many women you assign to the job.

12.5.2 Team Structure

Commercial operating systems are large software projects and invariably require large teams of people. The quality of the people matters immensely. It has been known for decades that top programmers are 10× more productive than bad programmers (Sackman et al., 1968). The trouble is, when you need 200 programmers, it is hard to find 200 top programmers; you have to settle for a wide spectrum of qualities.

What is also important in any large design project, software or otherwise, is the need for architectural coherence. There should be one mind controlling the design. Brooks cites the Reims cathedral as an example of a large project that took decades to build, and in which the architects who came later subordinated their desire to put their stamp on the project to carry out the initial architect's plans. The result is an architectural coherence unmatched in other European cathedrals.

In the 1970s, Harlan Mills combined the observation that some programmers are much better than others with the need for architectural coherence to propose the **chief programmer team** paradigm (Baker, 1972). His idea was to organize a programming team like a surgical team rather than like a hog-butchering team. Instead of everyone hacking away like mad, one person wields the scalpel. Everyone else is there to provide support. For a 10-person project, Mills suggested the team structure of Fig. 12-9.

Three decades have gone by since this was proposed and put into production. Some things have changed (such as the need for a language lawyer—C is simpler

Title	Duties
Chief programmer	Performs the architectural design and writes the code
Copilot	Helps the chief programmer and serves as a sounding board
Administrator	Manages the people, budget, space, equipment, reporting, etc.
Editor	Edits the documentation, which must be written by the chief programmer
Secretaries	The administrator and editor each need a secretary
Program clerk	Maintains the code and documentation archives
Toolsmith	Provides any tools the chief programmer needs
Tester	Tests the chief programmer's code
Language lawyer	Part timer who can advise the chief programmer on the language

Figure 12-9. Mills' proposal for populating a 10-person chief programmer team.

than PL/I), but the need to have only one mind controlling the design is still true. And that one mind should be able to work 100% on designing and programming, hence the need for the support staff, although with help from the computer, a smaller staff will suffice now. But in its essence, the idea is still valid.

Any large project needs to be organized as a hierarchy. At the bottom level are many small teams, each headed by a chief programmer. At the next level, groups of teams must be coordinated by a manager. Experience shows that each person you manage costs you 10% of your time, so a full-time manager is needed for each group of 10 teams. These managers must be managed, and so on up the tree.

Brooks observed that bad news does not travel up the tree well. Jerry Saltzer of M.I.T. called this effect the **bad-news diode**. No chief programmer or manager wants to tell his boss that the project is 4 months late and has no chance whatsoever of meeting the deadline because there is a 1000-year old tradition of beheading the messenger who brings bad news. As a consequence, top management is generally in the dark about the state of the project. When it becomes obvious that the deadline cannot be met, top management responds by adding people, at which time Brooks' law kicks in.

In practice, large companies, which have had long experience producing software and know what happens if it is produced haphazardly, have a tendency to at least try to do it right. In contrast, smaller, newer companies, which are in a huge rush to get to market, do not always take the care to produce their software carefully. This haste often leads to far from optimal results.

Neither Brooks nor Mills foresaw the growth of the open source movement. Although it has had some successes, it remains to be seen if this is a viable model for producing large amounts of quality software once the novelty wears off. Recall that in its early days, radio was dominated by ham radio operators, but that

soon gave way to commercial radio and later to commercial television. What is noticeable is that the open source software projects that have been most successful have clearly used the chief programmer model of having one mind control the architectural design (e.g., Linus Torvalds for the Linux kernel and Richard Stallman for the GNU C compiler).

12.5.3 The Role of Experience

Having experienced designers is critical to an operating systems project. Brooks points out that most of the errors are not in the code, but in the design. The programmers correctly did what they were told to do. What they were told to do was wrong. No amount of test software will catch bad specifications.

Brooks' solution is to abandon the classical development model of Fig. 12-10(a) and use the model of Fig. 12-10(b). Here the idea is to first write a main program that merely calls the top-level procedures, which are initially dummies. Starting on day 1 of the project, the system will compile and run, although it does nothing. As time goes on, modules are inserted into the full system. The result of this approach is that system integration testing is performed continuously, so errors in the design show up much earlier. In effect, the learning process caused by bad design decisions starts much earlier in the cycle.

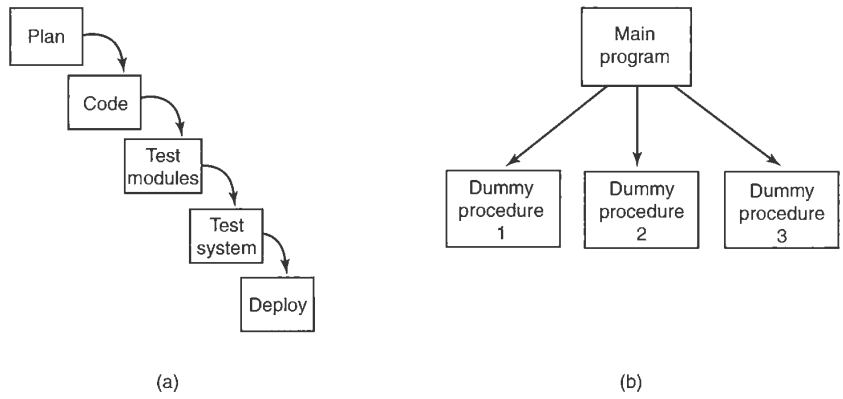


Figure 12-10. (a) Traditional software design progresses in stages. (b) Alternative design produces a working system (that does nothing) starting on day 1.

A little knowledge is a dangerous thing. Brooks observed what he called the **second system effect**. Often the first product produced by a design team is minimal because the designers are afraid it may not work at all. As a result, they are hesitant to put in many features. It the project succeeds, they build a follow-up system. Impressed by their own success, the second time the designers include all the bells and whistles that were intentionally left out the first time. As a result,

IBM Ex-1009
Page 577

the second system is bloated and performs poorly. The third time around they are sobered by the failure of the second system and are cautious again.

The CTSS-MULTICS pair is a clear case in point. CTSS was the first general-purpose timesharing system and was a huge success despite having minimal functionality. Its successor, MULTICS was too ambitious and suffered badly for it. The ideas were good, but there were too many new things so the system performed poorly for years and was never a big commercial success. The third system in this line of development, UNIX, was much more cautious and much more successful.

12.5.4 No Silver Bullet

In addition to *The Mythical Man Month*, Brooks also wrote an influential paper called "No Silver Bullet" (Brooks, 1987). In it, he argued that none of the many nostrums being hawked by various people at the time was going to generate an order-of-magnitude improvement in software productivity within a decade. Experience shows that he was right.

Among the silver bullets that were proposed were better high-level languages, object-oriented programming, artificial intelligence, expert systems, automatic programming, graphical programming, program verification, and programming environments. Perhaps the next decade will see a silver bullet, but maybe we will have to settle for gradual, incremental improvements.

12.6 TRENDS IN OPERATING SYSTEM DESIGN

Making predictions is always difficult—especially about the future. For example, in 1899, the head of the U.S. Patent Office, Charles H. Duell, asked then-President McKinley to abolish the Patent Office (and his job!), because, as he put it: "Everything that can be invented, has been invented" (Cerf and Navasky, 1984). Nevertheless, Thomas Edison showed up on his doorstep within a few years with a couple of new items, including the electric light, the phonograph, and the movie projector. Let us put new batteries in our crystal ball and venture a guess on where operating systems are going in the near future.

12.6.1 Large Address Space Operating Systems

As machines move from 32-bit address spaces to 64-bit address spaces, major shifts in operating system design become possible. A 32-bit address space is not really that big. If you tried to divide up 2^{32} bytes by giving everybody on earth his or her own byte, there would not be enough bytes to go around. In contrast, 2^{64} is about 2×10^{19}. Now everybody gets his or her personal 3 GB chunk.

IBM Ex-1009
Page 578

What could we do with an address space of 2×10^{19} bytes? For starters, we could eliminate the file system concept. Instead, all files could be conceptually held in (virtual) memory all the time. After all, there is enough room in there for over 1 billion full-length movies, each compressed to 4 GB.

Another possible use is a persistent object store. Objects could be created in the address space and kept there until all references to them were gone, at which time they would be automatically deleted. Such objects would be persistent in the address space, even over shutdowns and reboots of the computer. With a 64-bit address space, objects could be created at a rate of 100 MB/sec for 5000 years before we run out of address space. Of course, to actually store this amount of data, a lot of disk storage would be needed for the paging traffic, but for the first time in history, the limiting factor would be disk storage, not address space.

With large numbers of objects in the address space, it becomes interesting to allow multiple processes to run in the same address space at the same time, to share the objects in a general way. Such a design would clearly lead to very different operating systems than we now have. Some thoughts on this concept are contained in (Chase et al., 1994).

Another operating system issue that will have to be rethought with 64-bit addresses is virtual memory. With 2^{64} bytes of virtual address space and 8-KB pages we have 2^{51} pages. Conventional pages tables do not scale well to this size, so something else is needed. Inverted page tables are a possibility, but other ideas have been proposed as well (Talluri et al., 1995). In any event there is plenty of room for new research on 64-bit operating systems.

12.6.2 Networking

Current operating systems were designed for standalone computers. Networking was an afterthought and is generally accessed through special programs, such as Web browsers, FTP, or telnet. In the future, networking will probably be the basis for all operating systems. A standalone computer without a network connection will be as rare as a telephone without a connection to the network. And it is likely that multimegabit/sec connections will be the norm.

Operating systems will have to change to adapt to this paradigm shift. The difference between local data and remote data may blur to the point that virtually no one knows or cares where data are stored. Computers anywhere may be able to treat data anywhere as local data. To a limited extent, this is already true with NFS, but it is likely to become far more pervasive and better integrated.

Access to the Web, which now requires special programs (browsers) may also become completely integrated into the operating system in a seamless way. The standard way to store information may become Web pages, and these pages may contain a wide variety of nontext items, including audio, video, programs, and more, all managed as the operating system's fundamental data.

IBM Ex-1009
Page 579

12.6.3 Parallel and Distributed Systems

Another area that is up and coming is parallel and distributed systems. Current operating systems for multiprocessors and multicomputers are just standard uniprocessor operating systems with minor adjustments to the scheduler to handle parallelism a bit better. In the future, we may see operating systems where parallelism is much more central than it now is. This effect will be enormously stimulated if desktop machines soon have two, four, or more CPUs in a multiprocessor configuration. This may lead to many application programs being designed for multiprocessors, with the concurrent demand for better operating system support for them.

Multicomputers are likely to dominant large-scale scientific and engineering supercomputers in the coming years, but the operating systems for them are still fairly primitive. Process placement, load balancing, and communication need a lot of work.

Current distributed systems are often built as middleware because existing operating systems do not provide the right facilities for distributed applications. Future ones may be designed with distributed systems in mind, so all the necessary features are already present in the operating system from the start.

12.6.4 Multimedia

Multimedia systems are clearly a rising star in the computer world. It would surprise no one if computers, stereos, televisions, and telephones all merged into a single device capable of supporting high-quality still images, audio and video, and connected to high-speed networks so these files could easily be downloaded, exchanged, and accessed remotely. The operating systems for these devices, or even for standalone audio and video devices, will have to be substantially different from current ones. In particular, real-time guarantees will be needed, and these will drive the system design. Also, consumers will be very intolerant of hourly crashes of their digital television sets, so better software quality and fault tolerance will be required. Also, multimedia files tend to be very long, so file systems will have to change to be able to handle them efficiently.

12.6.5 Battery-Powered Computers

Powerful desktop PCs, probably with 64-bit address spaces, high-bandwidth networking, multiple processors, and high-quality audio and video will no doubt be commonplace soon. Their operating systems will have to be appreciably different from current ones to handle all these demands. However, an even faster growing segment of the market is battery-powered computers, including laptops, palmtops, Webpads, and various telephone hybrids. Some of these will have wireless connections to the outside world; others will run in disconnected mode

IBM Ex-1009
Page 580

when not docked at home. They will need different operating systems that are smaller, faster, more flexible, and more reliable than current ones. Various kinds of microkernel and extensible systems may form the basis here.

These operating systems will have to handle fully connected (i.e., wired), weakly connected (i.e., wireless), and disconnected operation, including data hoarding before going offline and consistency resolution when going back online better than current systems. They will also have to handle the problems of mobility better than current systems (e.g., find a laser printer, log onto it, and send it a file by radio). Power management, including extensive dialogs between the operating system and applications about how much battery power is left and how it can be best used, will be essential. Dynamic adaptation of applications to handle the limitations of tiny screens may become important. Finally, new input and output modes, including handwriting and speech, may require new techniques in the operating system to improve the quality. It is unlikely the operating system for a battery-powered, handheld wireless, voice-operated computer will have much in common with that of a desktop 64-bit four-CPU multiprocessor with a gigabit fiber-optic network connection. And of course, there will be innumerable hybrid machines with their own requirements.

12.6.6 Embedded Systems

One final area in which new operating systems will proliferate is embedded systems. The operating systems inside washing machines, microwave ovens, dolls, transistor (Internet?) radios, MP3 players, camcorders, elevators, and pacemakers will differ from all of the above and most likely from each other. Each one will probably be carefully tailored for its specific application, since it is unlikely anyone will ever stick a PCI card into a pacemaker to turn it into an elevator controller. Since all embedded systems run only a limited number of programs, known at design time, it may be possible to make optimizations not possible in general-purpose systems.

A promising idea for embedded systems are the extensible operating systems (e.g., Paramecium and Exokernel). These can be made as lightweight or heavyweight as the application in question demands, but in a consistent way across applications. Since embedded systems will be produced by the hundreds of millions, this will be a major market for new operating systems.

12.7 SUMMARY

Designing an operating system starts with determining what it should do. The interface should be simple, complete, and efficient. It should have a clear user interface paradigm, execution paradigm, and data paradigm.

The system should be well structured, using one of several known techniques, such as layering or client-server. The internal components should be orthogonal to one another and clearly separate policy from mechanism. Considerable thought should be given to issues such as static versus dynamic data structure, naming, binding time, and order of implementing modules.

Performance is important, but optimizations should be chosen carefully so as not to ruin the system's structure. Space-time trade-offs, caching, hints, exploiting locality, and optimizing the common case are often worth doing.

Writing a system with a couple of people is different than producing a big system with 300 people. In the latter case, team structure and project management play a crucial role in the success or failure of the project.

Finally, operating systems will have to change in the coming years to follow new trends and meet new challenges. These may include 64-bit address spaces, massive connectivity, desktop multiprocessors, multimedia, handheld wireless computers, and a huge variety of embedded systems. The coming years will be exciting times for operating system designers.

PROBLEMS

1. Moore's law describes a phenomenon of exponential growth similar to the population growth of an animal species introduced into a new environment with abundant food and no natural enemies. In nature, an exponential growth curve is likely eventually to become a sigmoid curve with an asymptotic limit when food supplies become limiting or predators learn to take advantage of new prey. Discuss some factors that may eventually limit the rate of improvement of computer hardware.

2. In Fig. 12-1, two paradigms are shown, algorithmic and event driven. For each of the following kinds of programs, which paradigm is likely to be easiest to use:

   (a) A compiler.
   (b) A photo editing program.
   (c) A payroll program.

3. On some of the early Apple Macintoshes, the GUI code was in ROM. Why?

4. Hierarchical file names always start at the top of the tree. Consider, for example, the file name */usr/ast/books/mos2/chap-12* rather than *chap-12/mos2/books/ast/usr*. In contrast, DNS names start at the bottom of the tree and work up. Is there some fundamental reason for this difference?

5. Corbató's dictum is that the system should provide minimal mechanism. Here is a list of POSIX calls that were also present in UNIX Version 7. Which ones are redundant, that is, could be removed with no loss of functionality because simple combinations of other ones could do the same job with about the same performance? Access, alarm, chdir, chmod, chown, chroot, close, creat, dup, exec, exit, fcntl, fork, fstat, ioctl, kill, link, lseek, mkdir, mknod, open, pause, pipe, read, stat, time, times, umask, unlink, utime, wait, and write.

IBM Ex-1009
Page 582

6. Suppose that layers 3 and 4 in Fig. 12-2 were exchanged. What implications would that have for the design of the system?

7. In a microkernel-based client-server system, the microkernel just does message passing and nothing else. Is it possible for user processes to nevertheless create and use semaphores? If so, how? If not, why not?

8. Careful optimization can improve system call performance. Consider the case in which one system call is made every 10 msec. The average time of a call is 2 msec. If the system calls can be speeded up by a factor of two, how long does a process that took 10 sec to run now take?

9. Give a short discussion of mechanism versus policy in the context of retail stores.

10. Operating systems often do naming at two different levels: external and internal. What are the differences between these names with respect to

    (a) Length
    (b) Uniqueness
    (c) Hierarchies

11. One way to handle tables whose size is not known in advance is to make them fixed, but when one fills up, to replace it with a bigger one, copy the old entries over to the new one, then release the old one. What are the advantages and disadvantages of making the new one 2× the size of the original one, as compared to making it only 1.5× as big?

12. In Fig. 12-4, a flag, *found*, is used to tell whether the PID was located. Would it have been possible to forget about *found* and just test *p* at the end of the loop to see whether it got to the end or not?

13. In Fig. 12-5, the differences between the Pentium and the UltraSPARC are hidden by conditional compilation. Could this same approach be used to hide the difference between Pentiums with an IDE disk as the only disk and Pentiums with a SCSI disk as the only disk? Would it be a good idea?

14. Indirection is a way of making an algorithm more flexible. Does it have any disadvantages, and if so, which ones?

15. Can reentrant procedures have private static global variables? Discuss your answer.

16. The macro of Fig. 12-6(b) is clearly much more efficient than the procedure of Fig. 12-6(a). One disadvantage, however, is that it is hard to read. Are there any other disadvantages? If so, what are they?

17. Suppose that we need a way of computing whether the number of bits in a 32-bit word is odd or even. Devise an algorithm for performing this computation as fast as possible. You may use up to 256 KB of RAM for tables if need be. Write a macro to carry out your algorithm. *Extra Credit*: Write a procedure to do the computation by looping over the 32 bits. Measure how many times faster your macro is than the procedure.

18. In Fig. 12-7, we saw how GIF files use 8-bit values to index into a color palette. The same idea can be used with a 16-bit-wide color palette. Under what circumstances, if any, might a 24-bit color palette be a good idea?

IBM Ex-1009
Page 583

19. One disadvantage of GIF is that the image must include the color palette, which increases the file size. What is the minimum image size for which an 8-bit-wide color palette breaks even? Now repeat this question for a 16-bit-wide color palette.

20. In the text we showed how caching path names can result in a significant speedup when looking up path names. Another technique that is sometimes used is having a daemon program that opens all the files in the root directory and keeps them open permanently, in order to force their i-nodes to be in memory all the time. Does pinning the i-nodes like this improve the path lookup even more?

21. Even if a remote file has not been removed since a hint was recorded, it may have been changed since the last time it was referenced. What other information might it be useful to record?

22. Consider a system that hoards references to remote files as hints, for example as (name, remote-host, remote-name). It is possible that a remote file will quietly be removed and then replaced. The hint may then retrieve the wrong file. How can this problem be made less likely to occur?

23. In the text it is stated that locality can often be exploited to improve performance. But consider a case where a program reads input from one source and continuously outputs to two or more files. Can an attempt to take advantage of locality in the file system lead to a decrease in efficiency here? Is there a way around this?

24. Fred Brooks claims that a programmer can write 1000 lines of debugged code per year, yet the first version of MINIX (13,000 lines of code) was produced by one person in under three years. How do you explain this discrepancy?

25. Using Brooks' figure of 1000 lines of code per programmer per year, make an estimate of the amount of money it took to produce Windows 2000. Assume that a programmer costs $100,000 per year (including overhead, such as computers, office space, secretarial support, and management overhead). Do you believe this answer? If not, what might be wrong with it?

26. As memory gets cheaper and cheaper, one could imagine a computer with a big battery-backed up RAM instead of a hard disk. At current prices, how much would a low-end RAM-only PC cost? Assume that a 1-GB RAM-disk is sufficient for a low-end machine. Is this machine likely to be competitive?

27. Name some features of a conventional operating system that are not needed in an embedded system used inside an appliance.

28. Write a procedure in C to do a double precision addition on two given parameters. Write the procedure using conditional compilation in such a way that it works on 16-bit machines and also on 32-bit machines.

29. Write programs that enter randomly generated short strings into an array and then can search the array for a given string using (a) a simple linear search (brute force), and (b) a more sophisticated method of your choice. Recompile your programs for array sizes ranging from small to as large as you can handle on your system. Evaluate the performance of both approaches). Where is the breakeven point?

30. Write a program to simulate an in-memory file system.

IBM Ex-1009
Page 584