had no protection hardware, eventually the PDP-11 did, and this feature led to multiprogramming and eventually to UNIX.

When the first microcomputers were built, they used the Intel 8080 CPU chip, which had no hardware protection, so we were back to monoprogramming. It wasn't until the Intel 80286 that protection hardware was added and multiprogramming became possible. Until this day, many embedded systems have no protection hardware and run just a single program.

Now let us look at operating systems. The first mainframes initially had no protection hardware and no support for multiprogramming, so they ran simple operating systems that handled one manually loaded program at a time. Later they acquired the hardware and operating system support to handle multiple programs at once, and then full timesharing capabilities.

When minicomputers first appeared, they also had no protection hardware and ran one manually loaded program at a time, even though multiprogramming was well established in the mainframe world by then. Gradually, they acquired protection hardware and the ability to run two or more programs at once. The first microcomputers were also capable of running only one program at a time, but later acquired the ability to multiprogram. Handheld computers and smart cards went the same route.

In all cases, the software development was dictated by technology. The first microcomputers, for example, had something like 4 KB of memory and no protection hardware. High-level languages and multiprogramming were simply too much for such a tiny system to handle. As the microcomputers evolved into modern personal computers, they acquired the necessary hardware and then the necessary software to handle more advanced features. It is likely that this development will continue for years to come. Other fields may also have this wheel of reincarnation, but in the computer industry it seems to spin faster.

Disks

Early mainframes were largely magnetic-tape based. They would read in a program from tape, compile it, run it, and write the results back to another tape. There were no disks and no concept of a file system. That began to change when IBM introduced the first hard disk—the RAMAC (RAndoM ACcess) in 1956. It occupied about 4 square meters of floor space and could store 5 million 7-bit characters, enough for one medium-resolution digital photo. But with an annual rental fee of $35,000, assembling enough of them to store the equivalent of a roll of film got pricey quite fast. But eventually prices came down and primitive file systems were developed.

Typical of these new developments was the CDC 6600, introduced in 1964 and for years by far the fastest computer in the world. Users could create so-called "permanent files" by giving them names and hoping that no other user had also decided that, say, "data" was a suitable name for a file. This was a single-level

directory. Eventually, mainframes developed complex hierarchical file systems, perhaps culminating in the MULTICS file system.

As minicomputers came into use, they eventually also had hard disks. The standard disk on the PDP-11 when it was introduced in 1970 was the RK05 disk, with a capacity of 2.5 MB, about half of the IBM RAMAC, but it was only about 40 cm in diameter and 5 cm high. But it, too, had a single-level directory initially. When microcomputers came out, CP/M was initially the dominant operating system, and it, too, supported just one directory on the (floppy) disk.

Virtual Memory

Virtual memory (discussed in Chap. 3), gives the ability to run programs larger than the machine's physical memory by moving pieces back and forth between RAM and disk. It underwent a similar development, first appearing on mainframes, then moving to the minis and the micros. Virtual memory also enabled the ability to have a program dynamically link in a library at run time instead of having it compiled in. MULTICS was the first system to allow this. Eventually, the idea propagated down the line and is now widely used on most UNIX and Windows systems.

In all these developments, we see ideas that are invented in one context and later thrown out when the context changes (assembly language programming, monoprogramming, single-level directories, etc.) only to reappear in a different context often a decade later. For this reason in this book we will sometimes look at ideas and algorithms that may seem dated on today's gigabyte PCs, but which may soon come back on embedded computers and smart cards.

1.6 SYSTEM CALLS

We have seen that operating systems have two main functions: providing abstractions to user programs and managing the computer's resources. For the most part, the interaction between user programs and the operating system deals with the former; for example, creating, writing, reading, and deleting files. The resource management part is largely transparent to the users and done automatically. Thus the interface between user programs and the operating system is primarily about dealing with the abstractions. To really understand what operating systems do, we must examine this interface closely. The system calls available in the interface vary from operating system to operating system (although the underlying concepts tend to be similar).

We are thus forced to make a choice between (1) vague generalities ("operating systems have system calls for reading files") and (2) some specific system ("UNIX has a read system call with three parameters: one to specify the file, one to tell where the data are to be put, and one to tell how many bytes to read").

We have chosen the latter approach. It's more work that way, but it gives more insight into what operating systems really do. Although this discussion specifically refers to POSIX (International Standard 9945-1), hence also to UNIX, System V, BSD, Linux, MINIX 3, and so on, most other modern operating systems have system calls that perform the same functions, even if the details differ. Since the actual mechanics of issuing a system call are highly machine dependent and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from C programs and often from other languages as well.

It is useful to keep the following in mind. Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the instruction following the system call. In a sense, making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not.

To make the system call mechanism clearer, let us take a quick look at the read system call. As mentioned above, it has three parameters: the first one specifying the file, the second one pointing to the buffer, and the third one giving the number of bytes to read. Like nearly all system calls, it is invoked from C programs by calling a library procedure with the same name as the system call: *read*. A call from a C program might look like this:

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

The system call (and the library procedure) return the number of bytes actually read in *count*. This value is normally the same as *nbytes*, but may be smaller, if, for example, end-of-file is encountered while reading.

If the system call cannot be carried out, either due to an invalid parameter or a disk error, *count* is set to −1, and the error number is put in a global variable, *errno*. Programs should always check the results of a system call to see if an error occurred.

System calls are performed in a series of steps. To make this concept clearer, let us examine the read call discussed above. In preparation for calling the *read* library procedure, which actually makes the read system call, the calling program first pushes the parameters onto the stack, as shown in steps 1-3 in Fig. 1-17.

C and C++ compilers push the parameters onto the stack in reverse order for historical reasons (having to do with making the first parameter to *printf*, the format string, appear on top of the stack). The first and third parameters are called by value, but the second parameter is passed by reference, meaning that the address of the buffer (indicated by &) is passed, not the contents of the buffer. Then

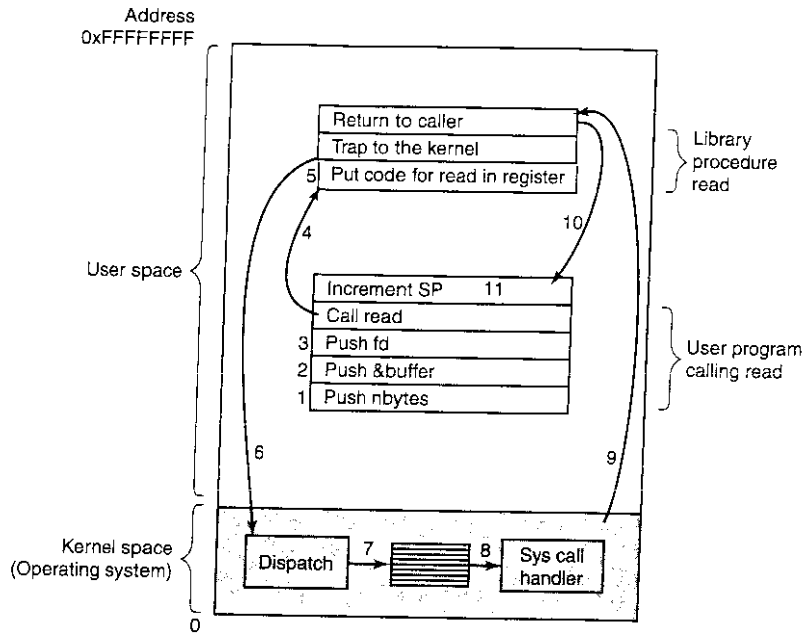

Figure 1-17. The 11 steps in making the system call read(fd, buffer, nbytes).

comes the actual call to the library procedure (step 4). This instruction is the normal procedure call instruction used to call all procedures.

The library procedure, possibly written in assembly language, typically puts the system call number in a place where the operating system expects it, such as a register (step 5). Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6). The TRAP instruction is actually fairly similar to the procedure call instruction in the sense that the instruction following it is taken from a distant location and the return address is saved on the stack for use later.

Nevertheless, the TRAP instruction also differs from the procedure call instruction in two fundamental ways. First, as a side effect, it switches into kernel mode. The procedure call instruction does not change the mode. Second, rather than giving a relative or absolute address where the procedure is located, the TRAP instruction cannot jump to an arbitrary address. Depending on the architecture, it either jumps to a single fixed location, there is an 8-bit field in the instruction giving the index into a table in memory containing jump addresses, or equivalent.

The kernel code that starts following the TRAP examines the system call number and then dispatches to the correct system call handler, usually via a table of pointers to system call handlers indexed on system call number (step 7). At that point the system call handler runs (step 8). Once the system call handler has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction (step 9). This procedure then returns to the user program in the usual way procedure calls return (step 10).

To finish the job, the user program has to clean up the stack, as it does after any procedure call (step 11). Assuming the stack grows downward, as it often does, the compiled code increments the stack pointer exactly enough to remove the parameters pushed before the call to *read*. The program is now free to do whatever it wants to do next.

In step 9 above, we said "may be returned to the user-space library procedure" for good reason. The system call may block the caller, preventing it from continuing. For example, if it is trying to read from the keyboard and nothing has been typed yet, the caller has to be blocked. In this case, the operating system will look around to see if some other process can be run next. Later, when the desired input is available, this process will get the attention of the system and steps 9–11 will occur.

In the following sections, we will examine some of the most heavily used POSIX system calls, or more specifically, the library procedures that make those system calls. POSIX has about 100 procedure calls. Some of the most important ones are listed in Fig. 1-18, grouped for convenience in four categories. In the text we will briefly examine each call to see what it does.

To a large extent, the services offered by these calls determine most of what the operating system has to do, since the resource management on personal computers is minimal (at least compared to big machines with multiple users). The services include things like creating and terminating processes, creating, deleting, reading, and writing files, managing directories, and performing input and output.

As an aside, it is worth pointing out that the mapping of POSIX procedure calls onto system calls is not one-to-one. The POSIX standard specifies a number of procedures that a conformant system must supply, but it does not specify whether they are system calls, library calls, or something else. If a procedure can be carried out without invoking a system call (i.e., without trapping to the kernel), it will usually be done in user space for reasons of performance. However, most of the POSIX procedures do invoke system calls, usually with one procedure mapping directly onto one system call. In a few cases, especially where several required procedures are only minor variations of one another, one system call handles more than one library call.

1.6.1 System Calls for Process Management

The first group of calls in Fig. 1-18 deals with process management. Fork is a good place to start the discussion. Fork is the only way to create a new process in POSIX. It creates an exact duplicate of the original process, including all the file

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure 1-18. Some of the major POSIX system calls. The return code *s* is −1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time. The parameters are explained in the text.

descriptors, registers—everything. After the fork, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the fork, but since the parent's data are copied to create the child, subsequent changes in one of them do not affect the other one. (The program text, which is unchangeable, is shared between parent and child.) The fork call returns a value, which is zero in the child and equal to the child's process identifier or **PID** in the parent. Using the returned PID, the two processes can see which one is the parent process and which one is the child process.

In most cases, after a fork, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a waitpid system call, which just waits until the child terminates (any child if more than one exists). Waitpid can wait for a specific child, or for any old child by setting the first parameter to −1. When waitpid completes, the address pointed to by the second parameter, *statloc*, will be set to the child's exit status (normal or abnormal termination and exit value). Various options are also provided, specified by the third parameter.

Now consider how fork is used by the shell. When a command is typed, the shell forks off a new process. This child process must execute the user command. It does this by using the execve system call, which causes its entire core image to be replaced by the file named in its first parameter. (Actually, the system call itself is exec, but several library procedures call it with different parameters and slightly different names. We will treat these as system calls here.) A highly simplified shell illustrating the use of fork, waitpid, and execve is shown in Fig. 1-19.

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
     type_prompt( );                            /* display prompt on the screen */
     read_command(command, parameters);         /* read input from terminal */

     if (fork( ) != 0) {                        /* fork off child process */
         /* Parent code. */
         waitpid(-1, &status, 0);               /* wait for child to exit */
     } else {
         /* Child code. */
         execve(command, parameters, 0);        /* execute command */
     }
}
```

Figure 1-19. A stripped-down shell. Throughout this book, *TRUE* is assumed to be defined as 1.

In the most general case, execve has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment

array. These will be described shortly. Various library routines, including *execl*, *execv*, *execle*, and *execve*, are provided to allow the parameters to be omitted or specified in various ways. Throughout this book we will use the name exec to represent the system call invoked by all of these.

Let us consider the case of a command such as

```
cp file1 file2
```

used to copy *file1* to *file2*. After the shell has forked, the child process locates and executes the file *cp* and passes to it the names of the source and target files.

The main program of *cp* (and main program of most other C programs) contains the declaration

```
main(argc, argv, envp)
```

where *argc* is a count of the number of items on the command line, including the program name. For the example above, *argc* is 3.

The second parameter, *argv*, is a pointer to an array. Element *i* of that array is a pointer to the *i*-th string on the command line. In our example, *argv*[0] would point to the string "cp", *argv*[1] would point to the string "file1" and *argv*[2] would point to the string "file2".

The third parameter of *main*, *envp*, is a pointer to the environment, an array of strings containing assignments of the form *name = value* used to pass information such as the terminal type and home directory name to programs. There are library procedures that programs can call to get the environment variables, which are often used to customize how a user wants to perform certain tasks (e.g., the default printer to use). In Fig. 1-19, no environment is passed to the child, so the third parameter of *execve* is a zero.

If exec seems complicated, do not despair; it is (semantically) the most complex of all the POSIX system calls. All the other ones are much simpler. As an example of a simple one, consider exit, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent via *statloc* in the waitpid system call.

Processes in UNIX have their memory divided up into three segments: the **text segment** (i.e., the program code), the **data segment** (i.e., the variables), and the **stack segment**. The data segment grows upward and the stack grows downward, as shown in Fig. 1-20. Between them is a gap of unused address space. The stack grows into the gap automatically, as needed, but expansion of the data segment is done explicitly by using a system call, brk, which specifies the new address where the data segment is to end. This call, however, is not defined by the POSIX standard, since programmers are encouraged to use the *malloc* library procedure for dynamically allocating storage, and the underlying implementation of *malloc* was not thought to be a suitable subject for standardization since few programmers use it directly and it is doubtful that anyone even notices that brk is not in POSIX.

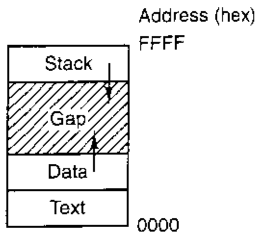

Figure 1-20. Processes have three segments: text, data, and stack.

1.6.2 System Calls for File Management

Many system calls relate to the file system. In this section we will look at calls that operate on individual files; in the next one we will examine those that involve directories or the file system as a whole.

To read or write a file, the file must first be opened using open. This call specifies the file name to be opened, either as an absolute path name or relative to the working directory, and a code of *O_RDONLY*, *O_WRONLY*, or *O_RDWR*, meaning open for reading, writing, or both. To create a new file, the *O_CREAT* parameter is used. The file descriptor returned can then be used for reading or writing. Afterward, the file can be closed by close, which makes the file descriptor available for reuse on a subsequent open.

The most heavily used calls are undoubtedly read and write. We saw read earlier. Write has the same parameters.

Although most programs read and write files sequentially, for some applications programs need to be able to access any part of a file at random. Associated with each file is a pointer that indicates the current position in the file. When reading (writing) sequentially, it normally points to the next byte to be read (written). The lseek call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file.

Lseek has three parameters: the first is the file descriptor for the file, the second is a file position, and the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file (in bytes) after changing the pointer.

For each file, UNIX keeps track of the file mode (regular file, special file, directory, and so on), size, time of last modification, and other information. Programs can ask to see this information via the stat system call. The first parameter specifies the file to be inspected; the second one is a pointer to a structure where the information is to be put. The fstat calls does the same thing for an open file.

1.6.3 System Calls for Directory Management

In this section we will look at some system calls that relate more to directories or the file system as a whole, rather than just to one specific file as in the previous section. The first two calls, mkdir and rmdir, create and remove empty directories, respectively. The next call is link. Its purpose is to allow the same file to appear under two or more names, often in different directories. A typical use is to allow several members of the same programming team to share a common file, with each of them having the file appear in his own directory, possibly under different names. Sharing a file is not the same as giving every team member a private copy; having a shared file means that changes that any member of the team makes are instantly visible to the other members—there is only one file. When copies are made of a file, subsequent changes made to one copy do not affect the others.

To see how link works, consider the situation of Fig. 1-21(a). Here are two users, *ast* and *jim*, each having his own directory with some files. If *ast* now executes a program containing the system call

```
link("/usr/jim/memo", "/usr/ast/note");
```

the file *memo* in *jim*'s directory is now entered into *ast*'s directory under the name *note*. Thereafter, */usr/jim/memo* and */usr/ast/note* refer to the same file. As an aside, whether user directories are kept in */usr*, */user*, */home*, or somewhere else is simply a decision made by the local system administrator.

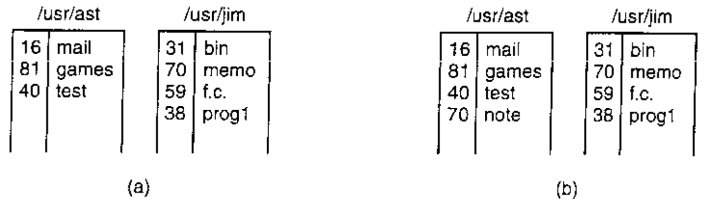

Figure 1-21. (a) Two directories before linking */usr/jim/memo* to ast's directory. (b) The same directories after linking.

Understanding how link works will probably make it clearer what it does. Every file in UNIX has a unique number, its i-number, that identifies it. This i-number is an index into a table of **i-nodes**, one per file, telling who owns the file, where its disk blocks are, and so on. A directory is simply a file containing a set of (i-number, ASCII name) pairs. In the first versions of UNIX, each directory entry was 16 bytes—2 bytes for the i-number and 14 bytes for the name. Now a more complicated structure is needed to support long file names, but conceptually a directory is still a set of (i-number, ASCII name) pairs. In Fig. 1-21, *mail* has i-number 16, and so on. What link does is simply create a new directory entry with a (possibly new) name, using the i-number of an existing file. In Fig. 1-21(b), two entries have the same i-number (70) and thus refer to the same file. If either one is later removed, using the unlink system call, the other one remains. If both are removed, UNIX 00sees that no entries to the file exist (a field in the i-node keeps track of the number of directory entries pointing to the file), so the file is removed from the disk.

As we have mentioned earlier, the mount system call allows two file systems to be merged into one. A common situation is to have the root file system containing the binary (executable) versions of the common commands and other heavily used files, on a hard disk. The user can then insert a CD-ROM disk with files to be read into the CD-ROM drive.

By executing the mount system call, the CD-ROM file system can be attached to the root file system, as shown in Fig. 1-22. A typical statement in C to perform the mount is

```
mount("/dev/fd0", "/mnt", 0);
```

where the first parameter is the name of a block special file for drive 0, the second parameter is the place in the tree where it is to be mounted, and the third parameter tells whether the file system is to be mounted read-write or read-only.

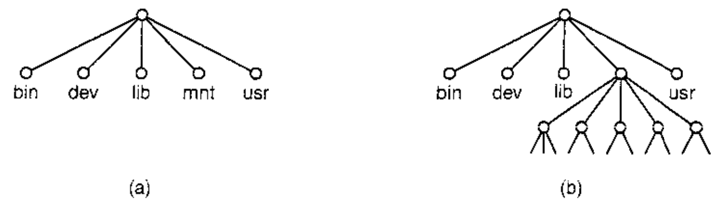

Figure 1-22. (a) File system before the mount. (b) File system after the mount.

After the mount call, a file on drive 0 can be accessed by just using its path from the root directory or the working directory, without regard to which drive it is on. In fact, second, third, and fourth drives can also be mounted anywhere in the tree. The mount call makes it possible to integrate removable media into a single integrated file hierarchy, without having to worry about which device a file is on. Although this example involves CD-ROMs, portions of hard disks (often called **partitions** or **minor devices**) can also be mounted this way, as well as external hard disks and USB sticks. When a file system is no longer needed, it can be unmounted with the umount system call.

1.6.4 Miscellaneous System Calls

A variety of other system calls exist as well. We will look at just four of them here. The chdir call changes the current working directory. After the call

```
chdir("/usr/ast/test");
```

an open on the file *xyz* will open */usr/ast/test/xyz*. The concept of a working directory eliminates the need for typing (long) absolute path names all the time.

In UNIX every file has a mode used for protection. The mode includes the read-write-execute bits for the owner, group, and others. The chmod system call makes it possible to change the mode of a file. For example, to make a file read-only by everyone except the owner, one could execute

```
chmod("file", 0644);
```

The kill system call is the way users and user processes send signals. If a process is prepared to catch a particular signal, then when it arrives, a signal handler is run. If the process is not prepared to handle a signal, then its arrival kills the process (hence the name of the call).

POSIX defines several procedures for dealing with time. For example, time just returns the current time in seconds, with 0 corresponding to Jan. 1, 1970 at midnight (just as the day was starting, not ending). On computers using 32-bit words, the maximum value time can return is $2^{32} - 1$ seconds (assuming an unsigned integer is used). This value corresponds to a little over 136 years. Thus in the year 2106, 32-bit UNIX systems will go berserk, not unlike the famous Y2K problem that would have wreaked havoc with the world's computers in 2000, were it not for the massive effort the IT industry put into fixing the problem. If you currently have a 32-bit UNIX system, you are advised to trade it in for a 64-bit one sometime before the year 2106.

1.6.5 The Windows Win32 API

So far we have focused primarily on UNIX. Now it is time to look briefly at Windows. Windows and UNIX differ in a fundamental way in their respective programming models. A UNIX program consists of code that does something or other, making system calls to have certain services performed. In contrast, a Windows program is normally event driven. The main program waits for some event to happen, then calls a procedure to handle it. Typical events are keys being struck, the mouse being moved, a mouse button being pushed, or a CD-ROM inserted. Handlers are then called to process the event, update the screen and update the internal program state. All in all, this leads to a somewhat different style of programming than with UNIX, but since the focus of this book is on operating system function and structure, these different programming models will not concern us much more.

Of course, Windows also has system calls. With UNIX, there is almost a one-to-one relationship between the system calls (e.g., read) and the library procedures (e.g., *read*) used to invoke the system calls. In other words, for each system call, there is roughly one library procedure that is called to invoke it, as indicated in Fig. 1-17. Furthermore, POSIX has only about 100 procedure calls.

With Windows, the situation is radically different. To start with, the library calls and the actual system calls are highly decoupled. Microsoft has defined a set of procedures called the **Win32 API (Application Program Interface)** that programmers are expected to use to get operating system services. This interface is (partially) supported on all versions of Windows since Windows 95. By decoupling the interface from the actual system calls, Microsoft retains the ability to change the actual system calls in time (even from release to release) without invalidating existing programs. What actually constitutes Win32 is also slightly ambiguous because Windows 2000, Windows XP, and Windows Vista have many new calls that were not previously available. In this section, Win32 means the interface supported by all versions of Windows.

The number of Win32 API calls is extremely large, numbering in the thousands. Furthermore, while many of them do invoke system calls, a substantial number are carried out entirely in user space. As a consequence, with Windows it is impossible to see what is a system call (i.e., performed by the kernel) and what is simply a user-space library call. In fact, what is a system call in one version of Windows may be done in user space in a different version, and vice versa. When we discuss the Windows system calls in this book, we will use the Win32 procedures (where appropriate) since Microsoft guarantees that these will be stable over time. But it is worth remembering that not all of them are true system calls (i.e., traps to the kernel).

The Win32 API has a huge number of calls for managing windows, geometric figures, text, fonts, scrollbars, dialog boxes, menus, and other features of the GUI. To the extent that the graphics subsystem runs in the kernel (true on some versions of Windows but not on all), these are system calls; otherwise they are just library calls. Should we discuss these calls in this book or not? Since they are not really related to the function of an operating system, we have decided not to, even though they may be carried out by the kernel. Readers interested in the Win32 API should consult one of the many books on the subject (e.g., Hart, 1997; Rector and Newcomer, 1997; and Simon, 1997).

Even introducing all the Win32 API calls here is out of the question, so we will restrict ourselves to those calls that roughly correspond to the functionality of the UNIX calls listed in Fig. 1-18. These are listed in Fig. 1-23.

Let us now briefly go through the list of Fig. 1-23. CreateProcess creates a new process. It does the combined work of fork and execve in UNIX. It has many parameters specifying the properties of the newly created process. Windows does not have a process hierarchy as UNIX does so there is no concept of a parent process and a child process. After a process is created, the creator and createe are equals. WaitForSingleObject is used to wait for an event. Many possible events can be waited for. If the parameter specifies a process, then the caller waits for the specified process to exit, which is done using ExitProcess.

The next six calls operate on files and are functionally similar to their UNIX counterparts although they differ in the parameters and details. Still, files can be

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Figure 1-23. The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1-18.

opened, closed, read, and written pretty much as in UNIX. The SetFilePointer and GetFileAttributesEx calls set the file position and get some of the file attributes.

Windows has directories and they are created with CreateDirectory and RemoveDirectory API calls, respectively. There is also a notion of a current directory, set by SetCurrentDirectory. The current time of day is acquired using GetLocalTime.

The Win32 interface does not have links to files, mounted file systems, security, or signals, so the calls corresponding to the UNIX ones do not exist. Of course, Win32 has a huge number of other calls that UNIX does not have, especially for managing the GUI. And Windows Vista has an elaborate security system and also supports file links.

One last note about Win32 is perhaps worth making. Win32 is not a terribly uniform or consistent interface. The main culprit here was the need to be backward compatible with the previous 16-bit interface used in Windows 3.x.

1.7 OPERATING SYSTEM STRUCTURE

Now that we have seen what operating systems look like on the outside (i.e., the programmer's interface), it is time to take a look inside. In the following sections, we will examine six different structures that have been tried, in order to get some idea of the spectrum of possibilities. These are by no means exhaustive, but they give an idea of some designs that have been tried in practice. The six designs are monolithic systems, layered systems, microkernels, client-server systems, virtual machines, and exokernels.

1.7.1 Monolithic Systems

By far the most common organization, in this approach the entire operating system runs as a single program in kernel mode. The operating system is written as a collection of procedures, linked together into a single large executable binary program. When this technique is used, each procedure in the system is free to call any other one, if the latter provides some useful computation that the former needs. Having thousands of procedures that can call each other without restriction often leads to an unwieldy and difficult to understand system.

To construct the actual object program of the operating system when this approach is used, one first compiles all the individual procedures (or the files containing the procedures) and then binds them all together into a single executable file using the system linker. In terms of information hiding, there is essentially none—every procedure is visible to every other procedure (as opposed to a structure containing modules or packages, in which much of the information is hidden away inside modules, and only the officially designated entry points can be called from outside the module).

Even in monolithic systems, however, it is possible to have some structure. The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction. This instruction switches the machine from user mode to kernel mode and transfers control to the operating system, shown as step 6 in Fig. 1-17. The operating system then fetches the parameters and determines which system call is to be carried out. After that, it indexes into a table that contains in slot *k* a pointer to the procedure that carries out system call *k* (step 7 in Fig. 1-17).

This organization suggests a basic structure for the operating system:

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

In this model, for each system call there is one service procedure that takes care of it and executes it. The utility procedures do things that are needed by several

service procedures, such as fetching data from user programs. This division of the procedures into three layers is shown in Fig. 1-24.

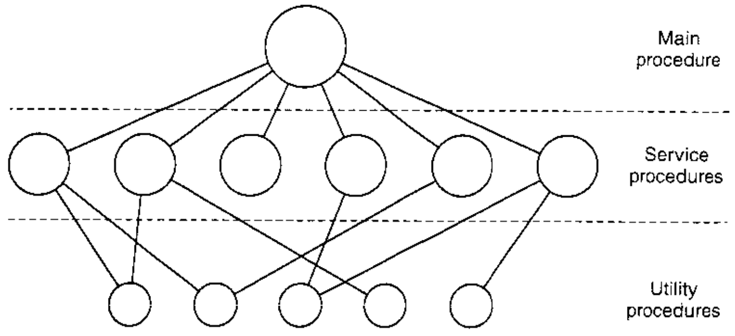

Figure 1-24. A simple structuring model for a monolithic system.

In addition to the core operating system that is loaded when the computer is booted, many operating systems support loadable extensions, such as I/O device drivers and file systems. These components are loaded on demand.

1.7.2 Layered Systems

A generalization of the approach of Fig. 1-24 is to organize the operating system as a hierarchy of layers, each one constructed upon the one below it. The first system constructed in this way was the THE system built at the Technische Hogeschool Eindhoven in the Netherlands by E. W. Dijkstra (1968) and his students. The THE system was a simple batch system for a Dutch computer, the Electrologica X8, which had 32K of 27-bit words (bits were expensive back then).

The system had six layers, as shown in Fig. 1-25. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

Layer 1 did the memory management. It allocated space for processes in main memory and on a 512K word drum used for holding parts of processes (pages) for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; the layer 1 software took care of making sure pages were brought into memory whenever they were needed.

Layer 2 handled communication between each process and the operator console (that is, the user). On top of this layer each process effectively had its own

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 1-25. Structure of the THE operating system.

operator console. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities. Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management. The system operator process was located in layer 5.

A further generalization of the layering concept was present in the MULTICS system. Instead of layers, MULTICS was described as having a series of concentric rings, with the inner ones being more privileged than the outer ones (which is effectively the same thing). When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to make the equivalent of a system call, that is, a TRAP instruction whose parameters were carefully checked for validity before allowing the call to proceed. Although the entire operating system was part of the address space of each user process in MULTICS, the hardware made it possible to designate individual procedures (memory segments, actually) as protected against reading, writing, or executing.

Whereas the THE layering scheme was really only a design aid, because all the parts of the system were ultimately linked together into a single executable program, in MULTICS, the ring mechanism was very much present at run time and enforced by the hardware. The advantage of the ring mechanism is that it can easily be extended to structure user subsystems. For example, a professor could write a program to test and grade student programs and run this program in ring n, with the student programs running in ring $n + 1$ so that they could not change their grades.

1.7.3 Microkernels

With the layered approach, the designers have a choice where to draw the kernel-user boundary. Traditionally, all the layers went in the kernel, but that is not necessary. In fact, a strong case can be made for putting as little as possible in

kernel mode because bugs in the kernel can bring down the system instantly. In contrast, user processes can be set up to have less power so that a bug there may not be fatal.

Various researchers have studied the number of bugs per 1000 lines of code (e.g., Basilli and Perricone, 1984; and Ostrand and Weyuker, 2002). Bug density depends on module size, module age, and more, but a ballpark figure for serious industrial systems is ten bugs per thousand lines of code. This means that a monolithic operating system of five million lines of code is likely to contain something like 50,000 kernel bugs. Not all of these are fatal, of course, since some bugs may be things like issuing an incorrect error message in a situation that rarely occurs. Nevertheless, operating systems are sufficiently buggy that computer manufacturers put reset buttons on them (often on the front panel), something the manufacturers of TV sets, stereos, and cars do not do, despite the large amount of software in these devices.

The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which—the microkernel—runs in kernel mode and the rest run as relatively powerless ordinary user processes. In particular, by running each device driver and file system as a separate user process, a bug in one of these can crash that component, but cannot crash the entire system. Thus a bug in the audio driver will cause the sound to be garbled or stop, but will not crash the computer. In contrast, in a monolithic system with all the drivers in the kernel, a buggy audio driver can easily reference an invalid memory address and bring the system to a grinding halt instantly.

Many microkernels have been implemented and deployed (Accetta et al., 1986; Haertig et al., 1997; Heiser et al., 2006; Herder et al., 2006; Hildebrand, 1992; Kirsch et al., 2005; Liedtke, 1993, 1995, 1996; Pike et al., 1992; and Zuberi et al., 1999). They are especially common in real-time, industrial, avionics, and military applications that are mission critical and have very high reliability requirements. A few of the better-known microkernels are Integrity, K42, L4, PikeOS, QNX, Symbian, and MINIX 3. We will now give a brief overview of MINIX 3, which has taken the idea of modularity to the limit, breaking most of the operating system up into a number of independent user-mode processes. MINIX 3 is a POSIX conformant, open-source system freely available at *www.minix3.org* (Herder et al., 2006a; Herder et al., 2006b).

The MINIX 3 microkernel is only about 3200 lines of C and 800 lines of assembler for very low-level functions such as catching interrupts and switching processes. The C code manages and schedules processes, handles interprocess communication (by passing messages between processes), and offers a set of about 35 kernel calls to allow the rest of the operating system to do its work. These calls perform functions like hooking handlers to interrupts, moving data between address spaces, and installing new memory maps for newly created processes. The process structure of MINIX 3 is shown in Fig. 1-26, with the kernel call handlers labeled *Sys*. The device driver for the clock is also in the kernel because the scheduler interacts closely with it. All the other device drivers run as separate user processes.

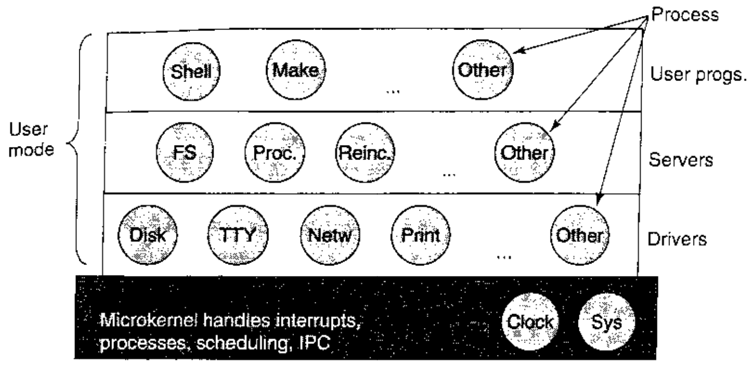

Figure 1-26. Structure of the MINIX 3 system.

Outside the kernel, the system is structured as three layers of processes all running in user mode. The lowest layer contains the device drivers. Since they run in user mode, they do not have physical access to the I/O port space and cannot issue I/O commands directly. Instead, to program an I/O device, the driver builds a structure telling which values to write to which I/O ports and makes a kernel call telling the kernel to do the write. This approach means that the kernel can check to see that the driver is writing (or reading) from I/O it is authorized to use. Consequently, (and unlike a monolithic design), a buggy audio driver cannot accidentally write on the disk.

Above the drivers is another user-mode layer containing the servers, which do most of the work of the operating system. One or more file servers manage the file system(s), the process manager creates, destroys, and manages processes, and so on. User programs obtain operating system services by sending short messages to the servers asking for the POSIX system calls. For example, a process needing to do a read sends a message to one of the file servers telling it what to read.

One interesting server is the **reincarnation server**, whose job is to check if the other servers and drivers are functioning correctly. In the event that a faulty one is detected, it is automatically replaced without any user intervention. In this way the system is self healing and can achieve high reliability.

The system has many restrictions limiting the power of each process. As mentioned, drivers can only touch authorized I/O ports, but access to kernel calls is also controlled on a per process basis, as is the ability to send messages to other processes. Processes can also grant limited permission for other processes to have the kernel access their address spaces. As an example, a file system can grant permission for the disk driver to let the kernel put a newly read in disk block at a specific address within the file system's address space. The sum total of all these restrictions is that each driver and server has exactly the power to do its work and nothing more, thus greatly limiting the damage a buggy component can do.

An idea somewhat related to having a minimal kernel is to put the **mechanism** for doing something in the kernel but not the **policy**. To make this point better, consider the scheduling of processes. A relatively simple scheduling algorithm is to assign a priority to every process and then have the kernel run the highest-priority process that is runnable. The mechanism—in the kernel—is to look for the highest-priority process and run it. The policy—assigning priorities to processes—can be done by user-mode processes. In this way policy and mechanism can be decoupled and the kernel can be made smaller.

1.7.4 Client-Server Model

A slight variation of the microkernel idea is to distinguish two classes of processes, the **servers**, each of which provides some service, and the **clients**, which use these services. This model is known as the **client-server** model. Often the lowest layer is a microkernel, but that is not required. The essence is the presence of client processes and server processes.

Communication between clients and servers is often by message passing. To obtain a service, a client process constructs a message saying what it wants and sends it to the appropriate service. The service then does the work and sends back the answer. If the client and server run on the same machine, certain optimizations are possible, but conceptually, we are talking about message passing here.

An obvious generalization of this idea is to have the clients and servers run on different computers, connected by a local or wide-area network, as depicted in Fig. 1-27. Since clients communicate with servers by sending messages, the clients need not know whether the messages are handled locally on their own machines, or whether they are sent across a network to servers on a remote machine. As far as the client is concerned, the same thing happens in both cases: requests are sent and replies come back. Thus the client-server model is an abstraction that can be used for a single machine or for a network of machines.

Increasingly many systems involve users at their home PCs as clients and large machines elsewhere running as servers. In fact, much of the Web operates this way. A PC sends a request for a Web page to the server and the Web page comes back. This is a typical use of the client-server model in a network.

1.7.5 Virtual Machines

The initial releases of OS/360 were strictly batch systems. Nevertheless, many 360 users wanted to be able to work interactively at a terminal, so various groups, both inside and outside IBM, decided to write timesharing systems for it. The

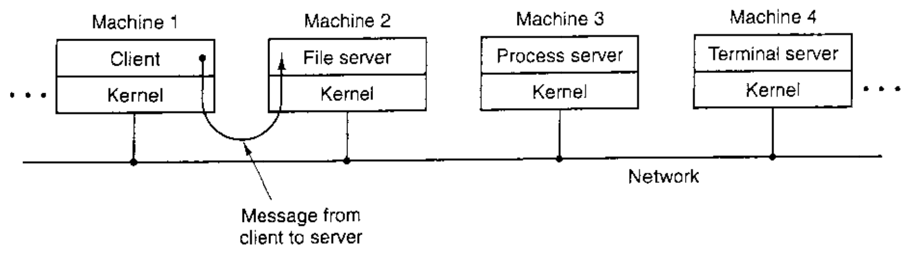

Figure 1-27. The client-server model over a network.

official IBM timesharing system, TSS/360, was delivered late, and when it finally arrived it was so big and slow that few sites converted to it. It was eventually abandoned after its development had consumed some $50 million (Graham, 1970). But a group at IBM's Scientific Center in Cambridge, Massachusetts, produced a radically different system that IBM eventually accepted as a product. A linear descendant of it, called **z/VM**, is now widely used on IBM's current mainframes, the zSeries, which are heavily used in large corporate data centers, for example, as e-commerce servers that handle hundreds or thousands of transactions per second and use databases whose sizes run to millions of gigabytes.

VM/370

This system, originally called CP/CMS and later renamed VM/370 (Seawright and MacKinnon, 1979), was based on an astute observation: a timesharing system provides (1) multiprogramming and (2) an extended machine with a more convenient interface than the bare hardware. The essence of VM/370 is to completely separate these two functions.

The heart of the system, known as the **virtual machine monitor**, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in Fig. 1-28. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are *exact* copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware. Different virtual machines can, and frequently do, run different operating systems. On the original VM/370 system, some ran OS/360 or one of the other large batch or transaction processing operating systems, while other ones ran a single-user, interactive system called **CMS (Conversational Monitor System)** for interactive timesharing users. The latter was popular with programmers.

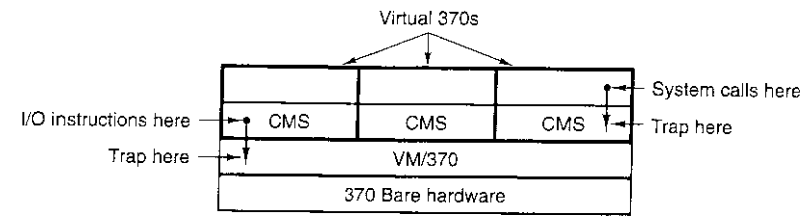

Figure 1-28. The structure of VM/370 with CMS.

When a CMS program executed a system call, the call was trapped to the operating system in its own virtual machine, not to VM/370, just as it would if it were running on a real machine instead of a virtual one. CMS then issued the normal hardware I/O instructions for reading its virtual disk or whatever was needed to carry out the call. These I/O instructions were trapped by VM/370, which then performed them as part of its simulation of the real hardware. By completely separating the functions of multiprogramming and providing an extended machine, each of the pieces could be much simpler, more flexible, and much easier to maintain.

In its modern incarnation, z/VM is usually used to run multiple complete operating systems rather than stripped-down single-user systems like CMS. For example, the zSeries is capable of running one or more Linux virtual machines along with traditional IBM operating systems.

Virtual Machines Rediscovered

While IBM has had a virtual machine product available for four decades, and a few other companies, including Sun Microsystems and Hewlett-Packard, have recently added virtual machine support to their high-end enterprise servers, the idea of virtualization has largely been ignored in the PC world until recently. But in the past few years, a combination of new needs, new software, and new technologies have combined to make it a hot topic.

First the needs. Many companies have traditionally run their mail servers, Web servers, FTP servers, and other servers on separate computers, sometimes with different operating systems. They see virtualization as a way to run them all on the same machine without having a crash of one server bring down the rest.

Virtualization is also popular in the Web hosting world. Without it, Web hosting customers are forced to choose between **shared hosting** (which just gives them a login account on a Web server, but no control over the server software) and dedicated hosting (which gives them their own machine, which is very flexible but not cost effective for small to medium Websites). When a Web hosting company offers virtual machines for rent, a single physical machine can run many virtual machines, each of which appears to be a complete machine. Customers who rent a virtual machine can run whatever operating system and software they want to, but at a fraction of the cost of a dedicated server (because the same physical machine supports many virtual machines at the same time).

Another use of virtualization is for end users who want to be able to run two or more operating systems at the same time, say Windows and Linux, because some of their favorite application packages run on one and some run on the other. This situation is illustrated in Fig. 1-29(a), where the term "virtual machine monitor" has been renamed type 1 **hypervisor** in recent years.

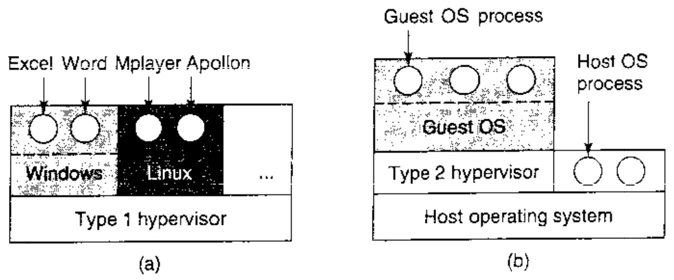

Figure 1-29. (a) A type 1 hypervisor. (b) A type 2 hypervisor.

Now the software. While no one disputes the attractiveness of virtual machines, the problem was implementation. In order to run virtual machine software on a computer, its CPU must be virtualizable (Popek and Goldberg, 1974). In a nutshell, here is the problem. When an operating system running on a virtual machine (in user mode) executes a privileged instruction), such as modifying the PSW or doing I/O, it is essential that the hardware trap to the virtual machine monitor so the instruction can be emulated in software. On some CPUs—notably the Pentium, its predecessors, and its clones—attempts to execute privileged instructions in user mode are just ignored. This property made it impossible to have virtual machines on this hardware, which explains the lack of interest in the PC world. Of course, there were interpreters for the Pentium that ran on the Pentium, but with a performance loss of typically 5–10x, they were not useful for serious work.

This situation changed as a result of several academic research projects in the 1990s, notably Disco at Stanford (Bugnion et al., 1997), which led to commercial products (e.g., VMware Workstation) and a revival of interest in virtual machines. VMware Workstation is a type 2 hypervisor, which is shown in Fig. 1-29(b). In contrast to type 1 hypervisors, which run on the bare metal, type 2 hypervisors run as application programs on top of Windows, Linux, or some other operating system, known as the **host operating system**. After a type 2 hypervisor is started, it

reads the installation CD-ROM for the chosen **guest operating system** and installs on a virtual disk, which is just a big file in the host operating system's file system.

When the guest operating system is booted, it does the same thing it does on the actual hardware, typically starting up some background processes and then a GUI. Some hypervisors translate the binary programs of the guest operating system block by block, replacing certain control instructions with hypervisor calls. The translated blocks are then executed and cached for subsequent use.

A different approach to handling control instructions is to modify the operating system to remove them. This approach is not true virtualization, but **paravirtualization**. We will discuss virtualization in more detail in Chap. 8.

The Java Virtual Machine

Another area where virtual machines are used, but in a somewhat different way, is for running Java programs. When Sun Microsystems invented the Java programming language, it also invented a virtual machine (i.e., a computer architecture) called the **JVM** (**Java Virtual Machine**). The Java compiler produces code for JVM, which then typically is executed by a software JVM interpreter. The advantage of this approach is that the JVM code can be shipped over the Internet to any computer that has a JVM interpreter and run there. If the compiler had produced SPARC or Pentium binary programs, for example, they could not have been shipped and run anywhere as easily. (Of course, Sun could have produced a compiler that produced SPARC binaries and then distributed a SPARC interpreter, but JVM is a much simpler architecture to interpret.) Another advantage of using JVM is that if the interpreter is implemented properly, which is not completely trivial, incoming JVM programs can be checked for safety and then executed in a protected environment so they cannot steal data or do any damage.

1.7.6 Exokernels

Rather than cloning the actual machine, as is done with virtual machines, another strategy is partitioning it, in other words, giving each user a subset of the resources. Thus one virtual machine might get disk blocks 0 to 1023, the next one might get blocks 1024 to 2047, and so on.

At the bottom layer, running in kernel mode, is a program called the **exokernel** (Engler et al., 1995). Its job is to allocate resources to virtual machines and then check attempts to use them to make sure no machine is trying to use somebody else's resources. Each user-level virtual machine can run its own operating system, as on VM/370 and the Pentium virtual 8086s, except that each one is restricted to using only the resources it has asked for and been allocated.

The advantage of the exokernel scheme is that it saves a layer of mapping. In the other designs, each virtual machine thinks it has its own disk, with blocks running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk addresses (and all other resources). With the exokernel, this remapping is not needed. The exokernel need only keep track of which virtual machine has been assigned which resource. This method still has the advantage of separating the multiprogramming (in the exokernel) from the user operating system code (in user space), but with less overhead, since all the exokernel has to do is keep the virtual machines out of each other's hair.

1.8 THE WORLD ACCORDING TO C

Operating systems are normally large C (or sometimes C++) programs consisting of many pieces written by many programmers. The environment used for developing operating systems is very different from what individuals (such as students) are used to when writing small Java programs. This section is an attempt to give a very brief introduction to the world of writing an operating system for small-time Java programmers.

1.8.1 The C Language

This is not a guide to C, but a short summary of some of the key differences between C and Java. Java is based on C, so there are many similarities between the two. Both are imperative languages with data types, variables, and control statements, for example. The primitive data types in C are integers (including short and long ones), characters, and floating-point numbers. Composite data types can be constructed using arrays, structures, and unions. The control statements in C are similar to those in Java, including if, switch, for, and while statements. Functions and parameters are roughly the same in both languages.

One feature that C has that Java does not is explicit pointers. A **pointer** is a variable that points to (i.e., contains the address of) a variable or data structure. Consider the statements

```
char c1, c2, *p;
c1 = 'x';
p = &c1;
c2 = *p;
```

which declare *c1* and *c2* to be character variables and *p* to be a variable that points to (i.e., contains the address of) a character. The first assignment stores the ASCII code for the character 'c' in the variable *c1*. The second one assigns the address of *c1* to the pointer variable *p*. The third one assigns the contents of the variable pointed to by *p* to the variable *c2*, so after these statements are executed, *c2* also contains the ASCII code for 'c'. In theory, pointers are typed, so you are not supposed to assign the address of a floating-point number to a character pointer, but

in practice compilers accept such assignments, albeit sometimes with a warning. Pointers are a very powerful construct, but also a great source of errors when used carelessly.

Some things that C does not have include built-in strings, threads, packages, classes, objects, type safety, and garbage collection. The last one is a show stopper for operating systems. All storage in C is either static or explicitly allocated and released by the programmer, usually with the library function *malloc* and *free*. It is the latter property—total programmer control over memory—along with explicit pointers that makes C attractive for writing operating systems. Operating systems are basically real-time systems to some extent, even general purpose ones. When an interrupt occurs, the operating system may have only a few microseconds to perform some action or lose critical information. Having the garbage collector kick in at an arbitrary moment is intolerable.

1.8.2 Header Files

An operating system project generally consists of some number of directories, each containing many *.c* files containing the code for some part of the system, along with some *.h* header files that contain declarations and definitions used by one or more code files. Header files can also include simple **macros**, such as

```
#define BUFFER_SIZE 4096
```

which allows the programmer to name constants, so that when *BUFFER_SIZE* is used in the code, it is replaced during compilation by the number 4096. Good C programming practice is to name every constant except 0, 1, and −1, and sometimes even them. Macros can have parameters, such as

```
#define max(a, b) (a > b ? a : b)
```

which allows the programmer to write

```
i = max(j, k+1)
```

and get

```
i = (j > k+1 ? j : k+1)
```

to store the larger of *j* and *k+1* in *i*. Headers can also contain conditional compilation, for example

```
#ifdef PENTIUM
intel_int_ack();
#endif
```

which compiles into a call to the function *intel_int_ack* if the macro *PENTIUM* is defined and nothing otherwise. Conditional compilation is heavily used to isolate architecture-dependent code so that certain code is inserted only when the system is compiled on the Pentium, other code is inserted only when the system is compiled on a SPARC, and so on. A *.c* file can bodily include zero or more header files using the *#include* directive. There are also many header files that are common to nearly every *.c* and are stored in a central directory.

1.8.3 Large Programming Projects

To build the operating system, each *.c* is compiled into an **object file** by the C compiler. Object files, which have the suffix *.o*, contain binary instructions for the target machine. They will later be directly executed by the CPU. There is nothing like Java byte code in the C world.

The first pass of the C compiler is called the **C preprocessor**. As it reads each *.c* file, every time it hits a *#include* directive, it goes and gets the header file named in it and processes it, expanding macros, handling conditional compilation (and certain other things) and passing the results to the next pass of the compiler as if they were physically included.

Since operating systems are very large (five million lines of code is not unusual), having to recompile the entire thing every time one file is changed would be unbearable. On the other hand, changing a key header file that is included in thousands of other files does require recompiling those files. Keeping track of which object files depend on which header files is completely unmanageable without help.

Fortunately, computers are very good at precisely this sort of thing. On UNIX systems, there is a program called *make* (with numerous variants such as *gmake*, *pmake*, etc.) that reads the *Makefile*, which tells it which files are dependent on which other files. What *make* does is see which object files are needed to build the operating system binary needed right now and for each one, check to see if any of the files it depends on (the code and headers) have been modified subsequent to the last time the object file was created. If so, that object file has to be recompiled. When *make* has determined which *.c* files have to recompiled, it invokes the C compiler to recompile them, thus reducing the number of compilations to the bare minimum. In large projects, creating the *Makefile* is error prone, so there are tools that do it automatically.

Once all the *.o* files are ready, they are passed to a program called the **linker** to combine all of them into a single executable binary file. Any library functions called are also included at this point, interfunction references are resolved, and machine address are relocated as need be. When the linker is finished, the result is an executable program, traditionally called *a.out* on UNIX systems. The various components of this process are illustrated in Fig. 1-30 for a program with three C files and two header files. Although we have been discussing operating system development here, all of this applies to developing any large program.

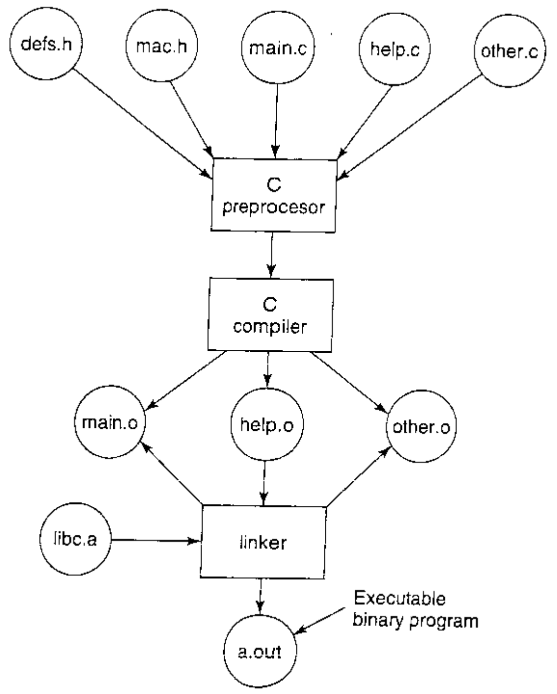

Figure 1-30. The process of compiling C and header files to make an executable.

1.8.4 The Model of Run Time

Once the operating system binary has been linked, the computer can be rebooted and the new operating system started. Once running, it may dynamically load pieces that were not statically included in the binary such as device drivers and file systems. At run time the operating system may consist of multiple segments, for the text (the program code), the data, and the stack. The text segment is normally immutable, not changing during execution. The data segment starts out at a certain size and initialized with certain values, but it can change and grow as need be. The stack is initially empty but grows and shrinks as functions are called and returned from. Often the text segment is placed near the bottom of memory, the data segment just above it, with the ability to grow upward, and the stack segment at a high virtual address, with the ability to grow downward, but different systems work differently.

In all cases, the operating system code is directly executed by the hardware, with no interpreter and no just-in-time compilation, as is normal with Java.

1.9 RESEARCH ON OPERATING SYSTEMS

Computer science is a rapidly advancing field and it is hard to predict where it is going. Researchers at universities and industrial research labs are constantly thinking up new ideas, some of which go nowhere but some of which become the cornerstone of future products and have massive impact on the industry and users. Telling which is which turns out to be easier to do in hindsight than in real time. Separating the wheat from the chaff is especially difficult because it often takes 20 to 30 years from idea to impact.

For example, when President Eisenhower set up the Dept. of Defense's Advanced Research Projects Agency (ARPA) in 1958, he was trying to keep the Army from killing the Navy and the Air Force over the Pentagon's research budget. He was not trying to invent the Internet. But one of the things ARPA did was fund some university research on the then-obscure concept of packet switching, which led to the first experimental packet-switched network, the ARPANET. It went live in 1969. Before long, other ARPA-funded research networks were connected to the ARPANET, and the Internet was born. The Internet was then happily used by academic researchers for sending e-mail to each other for 20 years. In the early 1990s, Tim Berners-Lee invented the World Wide Web at the CERN research lab in Geneva and Marc Andreesen wrote a graphical browser for it at the University of Illinois. All of a sudden the Internet was full of chatting teenagers. President Eisenhower is probably rolling over in his grave.

Research in operating systems has also led to dramatic changes in practical systems. As we discussed earlier, the first commercial computer systems were all batch systems, until M.I.T. invented interactive timesharing in the early 1960s. Computers were all text-based until Doug Engelbart invented the mouse and the graphical user interface at Stanford Research Institute in the late 1960s. Who knows what will come next?

In this section and in comparable sections throughout the book, we will take a brief look at some of the research in operating systems that has taken place during the past 5 to 10 years, just to give a flavor of what might be on the horizon. This introduction is certainly not comprehensive and is based largely on papers that have been published in the top research journals and conferences because these ideas have at least survived a rigorous peer review process in order to get published. Most of the papers cited in the research sections were published by either ACM, the IEEE Computer Society, or USENIX and are available over the Internet to (student) members of these organizations. For more information about these organizations and their digital libraries, see

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

Virtually all operating systems researchers realize that current operating systems are massive, inflexible, unreliable, insecure, and loaded with bugs, certain ones more than others (*names withheld here to protect the guilty*). Consequently, there is a lot of research on how to build better operating systems. Work has recently been published about new operating systems (Krieger et al., 2006), operating system structure (Fassino et al., 2002), operating system correctness (Elphinstone et al., 2007; Kumar and Li, 2002; and Yang et al., 2006), operating system reliability (Swift et al., 2006; and LeVasseur et al., 2004), virtual machines (Barham et al., 2003; Garfinkel et al., 2003; King et al., 2003; and Whitaker et al., 2002), viruses and worms (Costa et al., 2005; Portokalidis et al., 2006; Tucek et al., 2007; and Vrable et al., 2005), bugs and debugging (Chou et al., 2001; and King et al., 2005), hyperthreading and multithreading (Fedorova, 2005; and Bulpin and Pratt, 2005), and user behavior (Yu et al., 2006), among many other topics.

1.10 OUTLINE OF THE REST OF THIS BOOK

We have now completed our introduction and bird's-eye view of the operating system. It is time to get down to the details. As mentioned already, from the programmer's point of view, the primary purpose of an operating system is to provide some key abstractions, the most important of which are processes and threads, address spaces, and files. Accordingly the next three chapters are devoted to these critical topics.

Chapter 2 is about processes and threads. It discusses their properties and how they communicate with one another. It also gives a number of detailed examples of how interprocess communication works and how to avoid some of the pitfalls.

In Chap. 3 we will study address spaces and their adjunct, memory management, in detail. The important topic of virtual memory will be examined, along with closely related concepts such as paging and segmentation.

Then, in Chap. 4, we come to the all-important topic of file systems. To a considerable extent, what the user sees is largely the file system. We will look at both the file system interface and the file system implementation.

Input/Output is covered in Chap. 5. The concepts of device independence and device dependence will be looked at. Several important devices, including disks, keyboards, and displays, will be used as examples.

Chapter 6 is about deadlocks. We briefly showed what deadlocks are in this chapter, but there is much more to say. Ways to prevent or avoid them are discussed.

At this point we will have completed our study of the basic principles of single-CPU operating systems. However, there is more to say, especially about advanced topics. In Chap. 7, we examine multimedia systems, which have a number of properties and requirements that differ from conventional operating systems. Among other items, scheduling and the file system are affected by the nature of multimedia. Another advanced topic is multiple processor systems, including multiprocessors, parallel computers, and distributed systems. These subjects are covered in Chap. 8.

A hugely important subject is operating system security, which is covered in Chap 9. Among the topics discussed in this chapter are threats (e.g., viruses and worms), protection mechanisms, and security models.

Next we have some case studies of real operating systems. These are Linux (Chap. 10), Windows Vista (Chap. 11), and Symbian (Chap. 12). The book concludes with some wisdom and thoughts about operating system design in Chap. 13.

1.11 METRIC UNITS

To avoid any confusion, it is worth stating explicitly that in this book, as in computer science in general, metric units are used instead of traditional English units (the furlong-stone-fortnight system). The principal metric prefixes are listed in Fig. 1-31. The prefixes are typically abbreviated by their first letters, with the units greater than 1 capitalized. Thus a 1-TB database occupies 10^{12} bytes of storage and a 100 psec (or 100 ps) clock ticks every 10^{-10} seconds. Since milli and micro both begin with the letter "m," a choice had to be made. Normally, "m" is for milli and "μ" (the Greek letter mu) is for micro.

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

Figure 1-31. The principal metric prefixes.

It is also worth pointing out that for measuring memory sizes, in common industry practice, the units have slightly different meanings. There Kilo means 2^{10} (1024) rather than 10^3 (1000) because memories are always a power of two. Thus a 1-KB memory contains 1024 bytes, not 1000 bytes. Similarly, a 1-MB memory contains 2^{20} (1,048,576) bytes and a 1-GB memory contains 2^{30} (1,073,741,824) bytes. However, a 1-Kbps communication line transmits 1000 bits per second and a 10-Mbps LAN runs at 10,000,000 bits/sec because these speeds are not powers

of two. Unfortunately, many people tend to mix up these two systems, especially for disk sizes. To avoid ambiguity, in this book, we will use the symbols KB, MB, and GB for 2^{10}, 2^{20}, and 2^{30} bytes respectively, and the symbols Kbps, Mbps, and Gbps for 10^3, 10^6 and 10^9 bits/sec, respectively.

1.12 SUMMARY

Operating systems can be viewed from two viewpoints: resource managers and extended machines. In the resource manager view, the operating system's job is to manage the different parts of the system efficiently. In the extended machine view, the job of the system is to provide the users with abstractions that are more convenient to use than the actual machine. These include processes, address spaces, and files.

Operating systems have a long history, starting from the days when they replaced the operator, to modern multiprogramming systems. Highlights include early batch systems, multiprogramming systems, and personal computer systems.

Since operating systems interact closely with the hardware, some knowledge of computer hardware is useful to understanding them. Computers are built up of processors, memories, and I/O devices. These parts are connected by buses.

The basic concepts on which all operating systems are built are processes, memory management, I/O management, the file system, and security. Each of these will be treated in a subsequent chapter.

The heart of any operating system is the set of system calls that it can handle. These tell what the operating system really does. For UNIX, we have looked at four groups of system calls. The first group of system calls relates to process creation and termination. The second group is for reading and writing files. The third group is for directory management. The fourth group contains miscellaneous calls.

Operating systems can be structured in several ways. The most common ones are as a monolithic system, a hierarchy of layers, microkernel, client-server, virtual machine, or exokernel.

PROBLEMS

1. What is multiprogramming?
2. What is spooling? Do you think that advanced personal computers will have spooling as a standard feature in the future?
3. On early computers, every byte of data read or written was handled by the CPU (i.e., there was no DMA). What implications does this have for multiprogramming?

4. The family of computers idea was introduced in the 1960s with the IBM System/360 mainframes. Is this idea now dead as a doornail or does it live on?

5. One reason GUIs were initially slow to be adopted was the cost of the hardware needed to support them. How much video RAM is needed to support a 25 line × 80 row character monochrome text screen? How much for a 1024 × 768 pixel 24-bit color bitmap? What was the cost of this RAM at 1980 prices ($5/KB)? How much is it now?

6. There are several design goals in building an operating system, for example, resource utilization, timeliness, robustness, and so on. Give an example of two design goals that may contradict one another.

7. Which of the following instructions should be allowed only in kernel mode?

   (a) Disable all interrupts.
   (b) Read the time-of-day clock.
   (c) Set the time-of-day clock.
   (d) Change the memory map.

8. Consider a system that has two CPUs and each CPU has two threads (hyperthreading). Suppose three programs, *P0*, *P1*, and *P2*, are started with run times of 5, 10 and 20 mses, respectively. How long will it take to complete the execution of these programs? Assume that all three programs are 100% CPU bound, do not block during execution, and do not change CPUs once assigned.

9. A computer has a pipeline with four stages. Each stage takes the same time to do its work, namely, 1 nsec. How many instructions per second can this machine execute?

10. Consider a computer system that has cache memory, main memory (RAM) and disk, and the operating system uses virtual memory. It takes 2 nsec to access a word from the cache, 10 nsec to access a word from the RAM, and 10 ms to access a word from the disk. If the cache hit rate is 95% and main memory hit rate (after a cache miss) is 99%, what is the average time to access a word?

11. An alert reviewer notices a consistent spelling error in the manuscript of an operating systems textbook that is about to go to press. The book has approximately 700 pages, each with 50 lines of 80 characters each. How long will it take to electronically scan the text for the case of the master copy being in each of the levels of memory of Fig. 1-9? For internal storage methods, consider that the access time given is per character, for disk devices assume the time is per block of 1024 characters, and for tape assume the time given is to the start of the data with subsequent access at the same speed as disk access.

12. When a user program makes a system call to read or write a disk file, it provides an indication of which file it wants, a pointer to the data buffer, and the count. Control is then transferred to the operating system, which calls the appropriate driver. Suppose that the driver starts the disk and terminates until an interrupt occurs. In the case of reading from the disk, obviously the caller will have to be blocked (because there are no data for it). What about the case of writing to the disk? Need the caller be blocking awaiting completion of the disk transfer?

13. What is a trap instruction? Explain its use in operating systems.

14. What is the key difference between a trap and an interrupt?

15. Why is the process table needed in a timesharing system? Is it also needed in personal computer systems in which only one process exists, that process taking over the entire machine until it is finished?

16. Is there any reason why you might want to mount a file system on a nonempty directory? If so, what is it?

17. What is the purpose of a system call in an operating system?

18. For each of the following system calls, give a condition that causes it to fail: fork, exec, and unlink.

19. Can the

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

    call return any value in *count* other than *nbytes*? If so, why?

20. A file whose file descriptor is *fd* contains the following sequence of bytes: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. The following system calls are made:

```
lseek(fd, 3, SEEK_SET);
read(fd, &buffer, 4);
```

    where the lseek call makes a seek to byte 3 of the file. What does *buffer* contain after the read has completed?

21. Suppose that a 10-MB file is stored on a disk on the same track (track #: 50) in consecutive sectors. The disk arm is currently situated over track number 100. How long will it take to retrieve this file from the disk? Assume that moving the arm from one cylinder to the next takes about 1 ms and it takes about 5 ms for the sector where the beginning of the file is stored to rotate under the head. Also, assume that reading occurs at a rate of 100 MB/s.

22. What is the essential difference between a block special file and a character special file?

23. In the example given in Fig. 1-17, the library procedure is called *read* and the system call itself is called read. Is it essential that both of these have the same name? If not, which one is more important?

24. The client-server model is popular in distributed systems. Can it also be used in a single-computer system?

25. To a programmer, a system call looks like any other call to a library procedure. Is it important that a programmer know which library procedures result in system calls? Under what circumstances and why?

26. Figure 1-23 shows that a number of UNIX system calls have no Win32 API equivalents. For each of the calls listed as having no Win32 equivalent, what are the consequences for a programmer of converting a UNIX program to run under Windows?

27. A portable operating system is one that can be ported from one system architecture to another without any modification. Explain why it is infeasible to build an operating

system that is completely portable. Describe two high-level layers that you will have in designing an operating system that is highly portable.

28. Explain how separation of policy and mechanism aids in building microkernel-based operating systems.

29. Here are some questions for practicing unit conversions:

(a) How long is a microyear in seconds?
(b) Micrometers are often called microns. How long is a gigamicron?
(c) How many bytes are there in a 1-TB memory?
(d) The mass of the earth is 6000 yottagrams. What is that in kilograms?

30. Write a shell that is similar to Fig. 1-19 but contains enough code that it actually works so you can test it. You might also add some features such as redirection of input and output, pipes, and background jobs.

31. If you have a personal UNIX-like system (Linux, MINIX, Free BSD, etc.) available that you can safely crash and reboot, write a shell script that attempts to create an unlimited number of child processes and observe what happens. Before running the experiment, type sync to the shell to flush the file system buffers to disk to avoid ruining the file system. **Note**: Do not try this on a shared system without first getting permission from the system administrator. The consequences will be instantly obvious so you are likely to be caught and sanctions may follow.

32. Examine and try to interpret the contents of a UNIX-like or Windows directory with a tool like the UNIX *od* program or the MS-DOS *DEBUG* program. *Hint*: How you do this will depend upon what the OS allows. One trick that may work is to create a directory on a floppy disk with one operating system and then read the raw disk data using a different operating system that allows such access.

2

PROCESSES AND THREADS

We are now about to embark on a detailed study of how operating systems are designed and constructed. The most central concept in any operating system is the *process*: an abstraction of a running program. Everything else hinges on this concept, and it is important that the operating system designer (and student) have a thorough understanding of what a process is as early as possible.

Processes are one of the oldest and most important abstractions that operating systems provide. They support the ability to have (pseudo) concurrent operation even when there is only one CPU available. They turn a single CPU into multiple virtual CPUs. Without the process abstraction, modern computing could not exist. In this chapter we will go into considerable detail about processes and their first cousins, threads.

2.1 PROCESSES

All modern computers often do several things at the same time. People used to working with personal computers may not be fully aware of this fact, so a few examples may make the point clearer. First consider a Web server. Requests come in from all over asking for Web pages. When a request comes in, the server checks to see if the page needed is in the cache. If it is, it is sent back; if it is not, a disk request is started to fetch it. However, from the CPU's perspective, disk requests take eternity. While waiting for the disk request to complete, many more