This solution uses Java threads to simulate parallel processes. With this solution we have a class *producer* and a class *consumer*, which are instantiated in the variables *p* and *c*, respectively. Each of these classes is derived from a base class *Thread*, which has a method *run*. The *run* method contains the code for the thread. When the *start* method of an object derived from *Thread* is invoked, a new thread is started.

Each thread is like a process, except that all threads within a single Java program run in the same address space. This feature is convenient for having them share a common buffer. If the computer has two or more CPUs, each thread can be scheduled on a different CPU, allowing parallelism. If there is only one CPU, the threads are timeshared on the same CPU. We will continue to refer to the producer and consumer as processes (since we are really interested in parallel processes), even though Java supports only parallel threads and not true parallel processes.

The utility function *next* allows *in* and *out* to be incremented easily, without having to write code to check for the wraparound condition every time. If the parameter to *next* is 98 or lower, the next-higher integer is returned. If, however, the parameter is 99, we have hit the end of the buffer, so 0 is returned.

We need a way for either process to put itself to sleep when it cannot continue. The Java designers understood the need for this ability and included the methods *suspend* (sleep) and *resume* (wakeup) in the *Thread* class right from the first version of Java. They are used in Fig. 6-26.

Now we come to the actual code for the producer and consumer. First, the producer generates a new prime in P1. Notice the use of *m.MAX_PRIME* here. The prefix *m.* is needed to indicate that we mean the *MAX_PRIME* defined in class *m*. For the same reason, this prefix is needed for *in*, *out*, *buffer*, and *next*, as well.

Then the producer checks (in P2) to see if *in* is one behind *out*. If it is (e.g., *in* = 62 and *out* = 63), the buffer is full and the producer goes to sleep by calling *suspend* in P2. If the buffer is not full, the new prime is inserted into the buffer (P3) and *in* is incremented (P4). If the new value of *in* is 1 ahead of *out* (P5) (e.g., *in* = 17 and *out* = 16), *in* and *out* must have been equal before *in* was incremented. The producer concludes that the buffer was empty and that the consumer was, and still is, sleeping. Therefore, the producer calls *resume* to wake the consumer up (P5). Finally, the producer begins looking for the next prime.

The consumer's program is structurally similar. First, a test is made (C1) to see if the buffer is empty. If it is, there is no work for the consumer to do, so it goes to sleep. If the buffer is not empty, it removes the next number to be printed (C2) and increments *out* (*C*3). If *out* is two positions ahead of *in* at this point (C4), it must have been one position ahead of *in* before it was just incremented. Because *in* = *out* − 1 is the "buffer full" condition, the producer must have been sleeping, and thus the consumer wakes it up with *resume*. Finally, the number is printed (C5) and the cycle repeats.

Unfortunately, this design contains a fatal flaw, as illustrated in Fig. 6-27. Remember that the two processes run asynchronously and at different, possibly

varying, speeds. Consider the case where only one number is left in the buffer, in entry 21, and *in* = 22 and *out* = 21, as shown in Fig. 6-27(a). The producer is at statement P1 looking for a prime and the consumer is busy at C5 printing out the number in position 20. The consumer finishes printing the number, makes the test at C1, and takes the last number out of the buffer at C2. It then increments *out*. At this instant, both *in* and *out* have the value 22. The consumer prints the number and then goes to C1, where it fetches *in* and *out* from memory in order to compare them, as shown in Fig. 6-27(b).

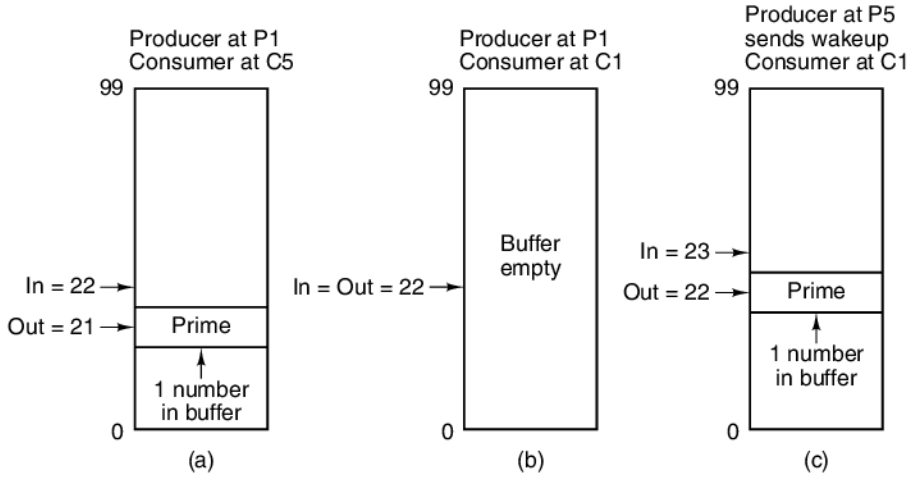

Figure 6-27. Failure of the producer-consumer communication mechanism.

At this very moment, after the consumer has fetched *in* and *out* but before it has compared them, the producer finds the next prime. It puts the prime into the buffer at P3 and increments *in* at P4. Now *in* = 23 and *out* = 22. At P5 the producer discovers that *in* = *next*(*out*). In other words, *in* is one higher than *out*, signifying that there is now one item in the buffer. The producer therefore (incorrectly) concludes that the consumer must be sleeping, so it sends a wakeup signal (i.e., calls *resume*), as shown in Fig. 6-27(c). Of course, the consumer is still awake, so the wakeup signal is lost. The producer begins looking for the next prime.

At this point in time the consumer continues. It has already fetched *in* and *out* from memory before the producer put the last number in the buffer. Because they both have the value 22, the consumer goes to sleep. Now the producer finds another prime. It checks the pointers and finds *in* = 24 and *out* = 22, therefore it assumes that there are two numbers in the buffer (true) and that the consumer is awake (false). The producer continues looping. Eventually, it fills the buffer and goes to sleep. Now both processes are sleeping and will remain so forever.

The big difficulty here is that between the moment when the consumer fetched *in* and *out* and the time it went to sleep, the producer snuck in, discovered that

in = *out* + 1, assumed that the consumer was sleeping (which it was not yet), and sent a wakeup signal that was lost because the consumer was still awake. This difficulty is known as a **race condition**, because the method's success depends on who wins the race to test *in* and *out* after *out* is incremented.

The problem of race conditions is well known. In fact it is so serious that several years after Java was introduced, Sun changed the *Thread* class and deprecated the *suspend* and *resume* calls because they led to race conditions so often. The solution offered was a language-based solution, but since we are studying operating systems here, we will discuss a different solution, one supported by many operating systems, including UNIX and Windows 7.

6.4.3 Process Synchronization Using Semaphores

The race condition can be solved in at least two ways. One solution consists of equipping each process with a "wakeup waiting bit." Whenever a wakeup is sent to a process that is still running, its wakeup waiting bit is set. Whenever the process goes to sleep when the wakeup waiting bit is set, it is immediately restarted and the wakeup waiting bit is cleared. The wakeup waiting bit stores the superfluous wakeup signal for future use.

Although this method solves the race condition when there are only two processes, it fails in the general case of n communicating processes because as many as $n - 1$ wakeups may have to be saved. Of course, each process could be equipped with $n - 1$ wakeup waiting bits to allow it to count to $n - 1$ in the unary system, but this solution is rather clumsy.

Dijkstra (1968b) proposed a more general solution to the problem of synchronizing parallel processes. Somewhere in the memory are some nonnegative integer variables called **semaphores**. Two system calls that operate on semaphores, up and down, are provided by the operating system. Up adds 1 to a semaphore and down subtracts 1 from a semaphore.

If a down operation is performed on a semaphore that is currently greater than 0, the semaphore is decremented by 1 and the process doing the down continues. If, however, the semaphore is 0, the down cannot complete; the process doing the down is put to sleep and remains asleep until the other process performs an up on that semaphore. Usually sleeping processes are strung together in a queue to keep track of them.

The up instruction checks to see if the semaphore is 0. If it is and the other process is sleeping on it, the semaphore is increased by 1. The sleeping process can then complete the down operation that suspended it, resetting the semaphore to 0 and allowing both processes to continue. An up instruction on a nonzero semaphore simply increases it by 1. In essence, a semaphore provides a counter to store wakeups for future use, so that they will not be lost. An essential property of semaphore instructions is that once a process has initiated an instruction on a semaphore, no other process may access the semaphore until the first one has either

completed its instruction or been suspended trying to perform a down on a 0. Figure 6-28 summarizes the essential properties of the up and down system calls.

Instruction	Semaphore = 0	Semaphore > 0
Up	Semaphore = semaphore + 1; if the other process was halted attempting to complete a down instruction on this semaphore, it may now complete the down and continue running	Semaphore = semaphore + 1
Down	Process halts until the other process ups this semaphore	Semaphore = semaphore – 1

Figure 6-28. The effect of a semaphore operation.

As mentioned above, Java has a language-based solution for dealing with race conditions, and we are discussing operating systems now. Thus we need a way to express semaphore usage in Java, even though it is not in the language or the standard classes. We will do this by assuming that two native methods have been written, *up* and *down*, which make the up and down system calls, respectively. By calling these with ordinary integers as parameters, we have a way to express the use of semaphores in Java programs.

Figure 6-29 shows how the race condition can be eliminated through the use of semaphores. Two semaphores are added to the *m* class, *available*, which is initially 100 (the buffer size), and *filled*, which is initially 0. The producer starts executing at P1 in Fig. 6-29 and the consumer starts executing at C1 as before. The *down* call on *filled* halts the consumer processor immediately. When the producer has found the first prime, it calls *down* with *available* as parameter, setting *available* to 99. At P5 it calls *up* with *filled* as parameter, making *filled* 1. This action releases the consumer, which is now able to complete its suspended *down* call. At this point, *filled* is 0 and both processes are running.

Let us now reexamine the race condition. At a certain point in time, *in* = 22, *out* = 21, the producer is at P1, and the consumer is at C5. The consumer finishes what it was doing and gets to C1 where it calls *down* on *filled*, which had the value 1 before the call and 0 after it. The consumer then takes the last number out of the buffer and ups *available*, making it 100. The consumer prints the number and goes to C1. Just before the consumer can call *down*, the producer finds the next prime and in quick succession executes statements P2, P3, and P4.

At this point, *filled* is 0. The producer is about to up it and the consumer is about to call *down*. If the consumer executes its instruction first, it will be suspended until the producer releases it (by calling *up*). On the other hand, if the producer goes first, the semaphore will be set to 1 and the consumer will not be suspended at all. In both cases, no wakeup is lost. This, of course, was our goal in introducing semaphores in the first place.

The essential property of the semaphore operations is that they are indivisible. Once a semaphore operation has been initiated, no other running process can use

the semaphore until the first process has either completed the operation or been suspended trying. Furthermore, with semaphores, no wakeups are lost. In contrast, the if statements of Figure 6-26 are not indivisible. Between the evaluation of the condition and the execution of the selected statement, another process can send a wakeup signal.

In effect the problem of process synchronization has been eliminated by declaring the up and down system calls made by *up* and *down* to be indivisible. In order for these operations to be indivisible, the operating system must prohibit two or more processes from using the same semaphore at the same time. At the very least, once an up or down system call has been made, no other user code will be run until the call has been completed. On single-processor systems, semaphores are sometimes implemented by disabling interrupts during semaphore operations. On multiple-processor systems, this trick does not work.

Synchronization using semaphores is a technique that works for arbitrarily many processes. Several processes may be sleeping, attempting to complete a down system call on the same semaphore. When some other process finally performs an up on that semaphore, one of the waiting processes is allowed to complete its down and continue running. The semaphore value remains 0 and the other processes continue waiting.

An analogy may make the nature of semaphores clearer. Imagine a picnic with 20 volleyball teams divided into 10 games (processes), each playing on its own court, and a large basket (the semaphore) for the volleyballs. Unfortunately, only seven volleyballs are available. At any instant, there are between zero and seven volleyballs in the basket (the semaphore has a value between 0 and 7). Putting a ball in the basket is an up because it increases the value of the semaphore; taking a ball out of the basket is a down because it decreases the value.

At the start of the picnic, each court sends a player to the basket to get a volleyball. Seven of them successfully manage to get a volleyball (complete the down); three are forced to wait for a volleyball (i.e., fail to complete the down). Their games are suspended temporarily. Eventually, one of the other games finishes and puts a ball into the basket (executes an up). This operation allows one of the three players waiting around the basket to get a ball (complete an unfinished down), allowing one game to continue. The other two games remain suspended until two more balls are put into the basket. When two more balls come back (two more ups are executed), the last two games can proceed.

6.5 EXAMPLE OPERATING SYSTEMS

In this section we will continue discussing our example systems, the Core i7 and the OMAP4430 ARM CPU. For each one we will look at an operating system used on that processor. For the Core i7 we will use Windows; for the OMAP4430 ARM CPU we will use UNIX. Since UNIX is simpler and in many ways more

```
public class m {
   final public static int BUF_SIZE = 100;                    // buffer runs from 0 to 99
   final public static long MAX_PRIME = 100000000000L;        // stop here
   public static int in = 0, out = 0;                         // pointers to the data
   public static long buffer[ ] = new long[BUF_SIZE];         // primes stored here
   public static producer p;                                  // name of the producer
   public static consumer c;                                  // name of the consumer
   public static int filled = 0, available = 100;             // semaphores

   public static void main(String args[ ]) {                  // main class
      p = new producer( );                                    // create the producer
      c = new consumer( );                                    // create the consumer
      p.start( );                                             // start the producer
      c.start( );                                             // start the consumer
   }

   // This is a utility function for circularly incrementing in and out
   public static int next(int k) {if (k < BUF_SIZE – 1) return(k+1); else return(0);}
}

class producer extends Thread {                               // producer class
   native void up(int s); native void down(int s);            // methods on semaphores
   public void run( ) {                                       // producer code
      long prime = 2;                                         // scratch variable

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);                           // statement P1
         down(m.available);                                   // statement P2
         m.buffer[m.in] = prime;                              // statement P3
         m.in = m.next(m.in);                                 // statement P4
         up(m.filled);                                        // statement P5
      }
   }

   private long next_prime(long prime){ ... }                 // function that computes next prime
}

class consumer extends Thread {                               // consumer class
   native void up(int s); native void down(int s);            // methods on semaphores
   public void run( ) {                                       // consumer code
      long emirp = 2;                                         // scratch variable

      while (emirp < m.MAX_PRIME) {
         down(m.filled);                                      // statement C1
         emirp = m.buffer[m.out];                             // statement C2
         m.out = m.next(m.out);                               // statement C3
         up(m.available);                                     // statement C4
         System.out.println(emirp);                           // statement C5
      }
   }
}
```

Figure 6-29. Parallel processing using semaphores.

elegant, we will begin with it. Also, UNIX was designed and implemented first and had a major influence on Windows 7, so this order makes more sense than the reverse.

6.5.1 Introduction

In this section we will give a brief introduction to our two example operating systems, UNIX and Windows 7, focusing on the history, structure, and system calls.

UNIX

UNIX was developed at Bell Labs in the early 1970s. The first version was written by Ken Thompson in assembler for the PDP-7 minicomputer. This was soon followed by a version for the PDP-11, written in a new language called C that was devised and implemented by Dennis Ritchie. In 1974, Ritchie and his colleague Ken Thompson published a landmark paper about UNIX (Ritchie and Thompson, 1974). For the work described in this paper they were later given the prestigious ACM Turing Award (Ritchie, 1984, Thompson, 1984). The publication of this paper stimulated many universities to ask Bell Labs for a copy of UNIX. Since Bell Labs' parent company, AT&T, was a regulated monopoly at the time and was not permitted to be in the computer business, it had no objection to licensing UNIX to universities for a modest fee.

In one of those coincidences that often shape history, the PDP-11 was the computer of choice at nearly all university computer science departments, and the operating systems that came with the PDP-11 were widely regarded as being dreadful by professors and students alike. UNIX quickly filled the void, not in the least because it was supplied with the complete source code, so people could, and did, tinker with it endlessly.

One of the many universities that acquired UNIX early on was the University of California at Berkeley. Because the complete source code was available, Berkeley was able to modify the system substantially. Foremost among the changes was a port to the VAX minicomputer and the addition of paged virtual memory, the extension of file names from 14 characters to 255 characters, and the inclusion of the TCP/IP networking protocol, which is now used on the Internet (largely due to the fact that it was in Berkeley UNIX).

While Berkeley was making all these changes, AT&T itself continued to develop UNIX, leading to System III in 1982 and then System V in 1984. By the late 1980s, two different, and quite incompatible, versions of UNIX were in widespread use: Berkeley UNIX and System V. This split in the UNIX world, together with the fact that there were no standards for binary program formats, greatly inhibited the commercial success of UNIX because it was impossible for software vendors to write and package UNIX programs with the expectation that they would run on any UNIX system (as was routinely done then with MS-DOS). After much

bickering, a standard called **POSIX** (**Portable Operating System-IX**) was created by the IEEE Standards Board. POSIX is also known by its IEEE Standards number, P1003. It later became an International Standard.

The standard is divided into many parts, each one covering a different area of UNIX. The first part, P1003.1, defines the system calls; the second part, P1003.2, defines the basic utility programs, and so on. The P1003.1 standard defines about 60 system calls that all conformant systems must support. These are the basic calls for reading and writing files, creating new processes, and so on. Nearly all UNIX systems now support the P1003.1 system calls. However many UNIX systems also support extra system calls, especially those defined by System V and/or those in Berkeley UNIX. Typically these add up to 200 system calls.

In 1987, one author of this book (Tanenbaum) released the source code for a tiny version of UNIX, called MINIX, for use at universities (Tanenbaum, 1987). One of the students who studied MINIX at his university in Helsinki and ran it on his home PC was Linus Torvalds. After becoming thoroughly familiar with MINIX, Torvalds decided to write his own clone of MINIX, which was called Linux and has become quite popular.

Many operating systems running today on ARM platforms are based on Linux. Both MINIX and Linux are POSIX conformant, and nearly everything said about UNIX in this chapter also applies to them unless stated otherwise.

A rough breakdown of the Linux system calls by category is given in Fig. 6-30. The file- and directory-management system calls are the largest and the most important categories. Linux is mostly POSIX P1003.1 compliant, although the developers did deviate from the specification in some areas. In general, however, it is not difficult to get POSIX-compliant programs to build and run on Linux.

Category	Some examples
File management	Open, read, write, close, and lock files
Directory management	Create and delete directories; move files around
Process management	Spawn, terminate, trace, and signal processes
Memory management	Share memory among processes; protect pages
Getting/setting parameters	Get user, group, process ID; set priority
Dates and times	Set file access times; use interval timer; profile execution
Networking	Establish/accept connection; send/receive message
Miscellaneous	Enable accounting; manipulate disk quotas; reboot the system

Figure 6-30. A rough breakdown of the UNIX system calls.

One area that is largely due to Berkeley UNIX rather than System V is networking. Berkeley invented the concept of a **socket**, which is the endpoint of a network connection. The four-pin wall plugs to which telephones can be connected served as the model for this concept. It is possible for a UNIX process to create a socket, attach to it, and establish a connection to a socket on a distant

machine. Over this connection it can then exchange data in both directions, typically using the TCP/IP protocol. Since networking technology has been in UNIX for decades and is very stable and mature, a substantial fraction of the servers on the Internet run UNIX.

Since there are many implementations of UNIX, it is difficult to say much about the structure of the operating system since each one is somewhat different from all the others. However, in general, Fig. 6-31 applies to most of them. At the bottom, there is a layer of device drivers that shield the file system from the bare hardware. Originally, each device driver was written as an independent entity, separate from all the others. This arrangement led to a lot of duplicated effort, since many drivers must deal with flow control, error handling, priorities, separating data from control, and so on. This observation led Dennis Ritchie to develop a framework called **streams** for writing drivers in a modular way. With a stream, it is possible to establish a two-way connection from a user process to a hardware device and to insert one or more modules along the path. The user process pushes data into the stream, which then is processed or transformed by each module until it gets to the hardware. The inverse processing occurs for incoming data.

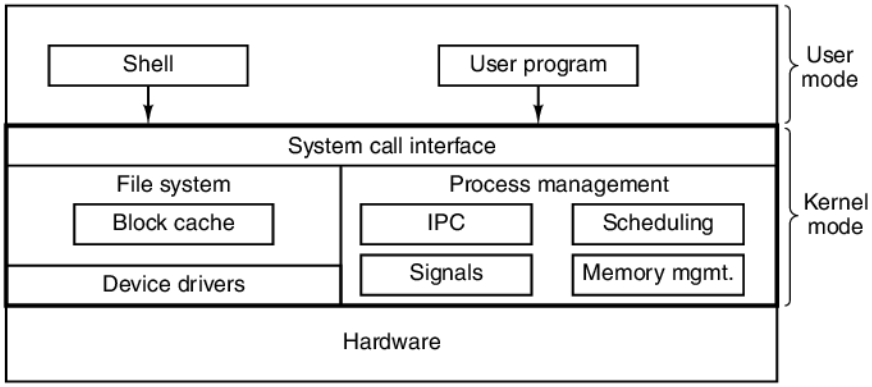

Figure 6-31. The structure of a typical UNIX system.

On top of the device drivers comes the file system. It manages file names, directories, disk block allocation, protection, and much more. Part of the file system is a **block cache**, for holding the blocks most recently read in from disk, in case they are needed again soon. A variety of file systems have been used over the years, including the Berkeley fast file system (McKusick et al., 1984), and log-structured file systems (Rosenblum and Ousterhout, 1991, and Seltzer et al., 1993).

The other part of the UNIX kernel is the process-management portion. Among its various other functions, it handles IPC (InterProcess Communication), which allows processes to communicate with one another and synchronize to avoid race conditions. A variety of mechanisms are provided. The process-management code

also handles process scheduling, which is based on priorities. Signals, which are a form of (asynchronous) software interrupt, are also managed here. Finally, memory management is done here as well. Most UNIX systems support demand-paged virtual memory, sometimes with a few extra features, such as the ability of multiple processes to share common regions of address space.

From its inception, UNIX has tried to be a small system, in order to enhance reliability and performance. The first versions of UNIX were entirely text based, using terminals that could display 24 or 25 lines of 80 ASCII characters. The user interface was handled by a user-level program called the **shell**, which offered a command-line interface. Since the shell was not part of the kernel, adding new shells to UNIX was easy, and over time a number of increasingly sophisticated ones were invented.

Later on, when graphics terminals came into existence, a windowing system for UNIX, called **X Windows**, was developed at M.I.T. Still later, a full-fledged **GUI** (**Graphical User Interface**), called **Motif**, was put on top of X Windows. These GUIs eventually developed into full-blown desktop environments with beautifully rendered window management, productivity tools, and utilities. Examples of these desktop environments include GNOME and KDE. In keeping with the UNIX philosophy of having a small kernel, nearly all the code of X Windows and its accompanying GUIs run in user mode, outside the kernel.

Windows 7

When the original IBM PC was launched in 1981, it came equipped with a 16-bit real-mode, single-user, command-line-oriented operating system called MS-DOS 1.0 This operating system consisted of 8 KB of memory-resident code. Two years later, a much more powerful 24-KB system, MS-DOS 2.0, appeared. It contained a command-line processor (shell), with a number of features borrowed from UNIX. When IBM released the 286-based PC/AT in 1984, it came equipped with 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.

Inspired by the success of the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface that it called **Windows**. The first three versions of Windows, culminating in Windows 3.x, were not true operating systems but graphical user interfaces on top of MS-DOS, which was still in control of the machine. 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 1995 still did not eliminate MS-DOS, although it introduced a new version, 7.0. Together, Windows 95 and 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 code (as well as some 32-bit code) and still used the MS-DOS file system, with nearly all its limitations.

The only major changes to the file system were the addition of long file names in place of the 8 + 3 character file names allowed in MS-DOS and the ability to have more than 65,536 blocks on a disk.

Even with the release of Windows 98 in 1998, MS-DOS was still there (now called version 7.1) and running 16-bit code. Although a bit 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, the Internet, and to some extent, even television 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. Windows 98 was followed by the short-lived Windows Millennium Edition (ME), which was a slightly improved Windows 98.

While all these developments were going on, Microsoft was also busy with a completely new 32-bit operating system being written from the ground up. This new system was called **Windows New Technology**, or **Windows NT**. It was initially hyped as the replacement for all other operating systems for Intel-based PCs (as well as the MIPS PowerPC chips), but it was somewhat slow to catch on and was later redirected to the upper end of the market, where it found a niche on large servers. The second version of NT was called Windows 2000 and became the mainstream version, also for the desktop market. The successor to Windows 2000 was Windows XP, but the changes here were relatively minor (better backward compatibility and a few more features). In 2007, the followup Windows Vista was released. Vista implemented many graphical enhancements over Windows XP, and it added many new user applications, such as a media center. Vista's adoption was slow because of its poor performance and high resource demands. A mere two years later, Windows 7 was released, which by all accounts is a tuned-up version of Windows Vista. Windows 7 runs much better on older hardware, and it requires significantly less hardware resources.

Windows 7 is sold in six different versions. Three are for home users in various countries, two are aimed at business users, and one combines all the features of all versions. These versions are nearly identical and differ primarily in focus, advanced features, and optimizations made. We will focus on the core features and not make any further distinction between these versions.

Before getting into the interface Windows 7 presents to programmers, let us take a very quick look at its internal structure, which is illustrated in Fig. 6-32. It consists of a number of modules that are structured in layers and work together to implement the operating system. Each module has some particular function and a well-defined interface to the other modules. Nearly all the modules are written in C, although part of the graphics device interface is written in C++ and tiny bits of the lowest layers are written in assembly language.

At the bottom is a thin layer called the **hardware abstraction layer**. Its job is to present the rest of the operating system with abstract hardware devices, devoid

of the warts and idiosyncracies with which real hardware is so richly endowed. Among the devices modeled are off-chip caches, timers, I/O buses, interrupt controllers, and DMA controllers. By exposing these to the rest of the operating system in idealized form, it becomes easier to port Windows 7 to other hardware platforms, since most of the modifications required are concentrated in one place.

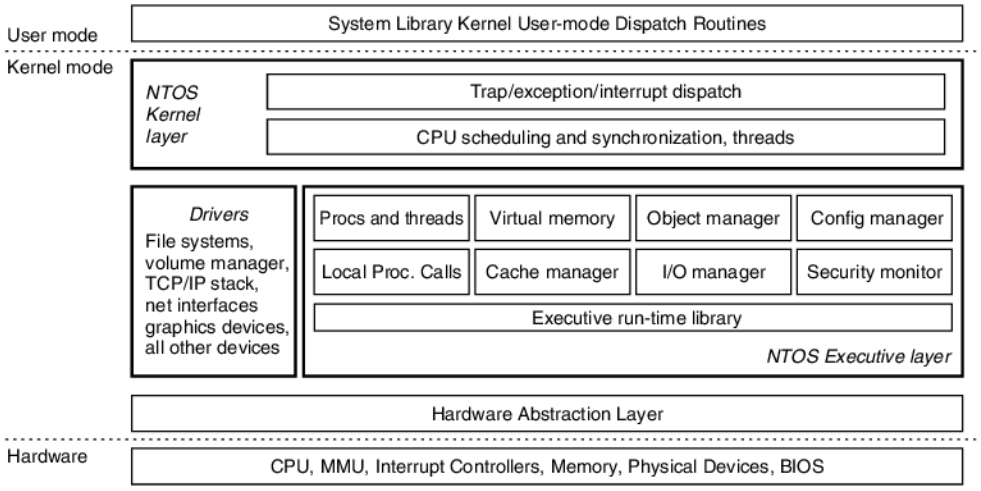

Figure 6-32. The structure of Windows 7.

Above the HAL, the code is divided into two major parts, the **NTOS executive** and the **Windows drivers**, which includes the file systems, networking, and graphics code. On top of that is the kernel layer. All of this code runs in protected kernel mode.

The executive manages the fundamental abstractions used in Windows 7, including threads, processes, virtual memory, kernel objects, and configurations. Also here are the managers for local procedure calls, the file cache, I/O, and security.

The kernel layer handles trap and exception handling, as well as scheduling and synchronization.

Outside the kernel are the user programs and the system library used to interface to the operating system. In contrast to UNIX systems, Microsoft does not encourage user programs to make direct system calls. Instead they are expected to call procedures in the library. To provide standardization across different versions of Windows (e.g., XP, Vista, and Windows 7), Microsoft defined a set of calls called the **Win32 API** (**Application Programming Interface**). These are library procedures that either make system calls to get the work done, or, in some case, do the work right in the user-space library procedure. Although many Windows 7 library calls have been added since Win32 was defined, these are the core calls and it is them we will focus on. Later, when Windows was ported to 64-bit machines,

Microsoft changed the name of Win32 to cover both the 32-bit and 64-bit versions, but for our purposes, looking at the 32-bit version is sufficient.

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 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, etc. Every call creating a kernel object returns a result called a **handle** to the called. This handle can be subsequently 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 can have a security descriptor associated with it, telling in detail who may and may not perform what kinds of operations on the object.

Windows 7 is sometimes said to be object oriented because the only way to manipulate kernel objects is by invoking methods (API functions) on their handles, which are returned when the objects are created. On the other hand, it lacks some of the most basic properties of object-oriented systems such as inheritance and polymorphism.

6.5.2 Examples of Virtual Memory

In this section we will look at virtual memory in both UNIX and Windows 7. For the most part, they are fairly similar from the programmer's point of view.

UNIX Virtual Memory

The UNIX memory model is simple. Each process has three segments: code, data, and stack, as illustrated in Fig. 6-33. In a machine with a single, linear address space, the code is generally placed near the bottom of memory, followed by the data. The stack is placed at the top of memory. The code size is fixed, but the data and stack may each grow, in opposite directions. This model is easy to implement on almost any machine and is the model used by Linux variants that run on OMAP4430 ARM CPUs.

Furthermore, if the machine has paging, the entire address space can be paged, without user programs even being aware of it. The only thing they notice is that it

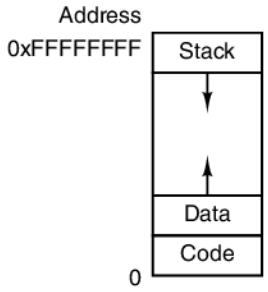

Figure 6-33. The address space of a single UNIX process.

is permitted to have programs larger than the machine's physical memory. UNIX systems that do not have paging generally swap entire processes between memory and disk to allow an arbitrarily large number of processes to be timeshared.

For Berkeley UNIX, the above description (demand-paged virtual memory) is basically the entire story. However, System V (and also Linux) include several features that allow users to manage their virtual memory in more sophisticated ways. Most important of these is the ability of a process to map a (portion of a) file onto part of its address space. For example, if a 12-KB file is mapped at virtual address 144K, a read to the word at address 144 KB reads the first word of the file. In this way file I/O can be done without making system calls. Since some files may exceed the size of the virtual address space, it is also possible to map in only a portion of a file instead of the whole file. The mapping is done by first opening the file and getting back a file descriptor, *fd*, which is used to identify the file to be mapped. Then the process makes a call

```
paddr = mmap(virtual_address, length, protection, flags, fd, file_offset)
```

which maps *length* bytes starting at *file_offset* in the file onto the virtual address space starting at *virtual_address*. Alternatively, the *flags* parameter can be set to ask the system to choose a virtual address, which it then returns as *paddr*. The mapped region must be an integral number of pages and aligned at a page boundary. The *protection* parameter can specify any combination of read, write, or execute permission. The mapping can be removed later with unmap.

Multiple processes can map onto the same file at the same time. Two options are provided for sharing. In the first one, all the pages are shared, so writes done by one process are visible to all the others. This option provides a high-bandwidth communication path between processes. The other option shares the pages as long as no process modifies them. However, as soon as any process attempts to write on a page, it gets a protection fault, which causes the operating system to give it a private copy of the page to write on. This scheme, known as **copy on write**, is used when each of multiple processes needs the illusion it is the only one mapped onto a file. In this model the sharing is an optimization, not part of the semantics.

Windows 7 Virtual Memory

In Windows 7, every user process has its own virtual address space. In the 32-bit version of Windows 7, virtual addresses are 32 bits long, so each process has 4 GB of virtual address space. The lower 2 GB are available for the process' code and data; the upper 2 GB allow (limited) access to kernel memory, except in Server versions of Windows, in which the split can be 3 GB for the user and 1 GB for the kernel. The virtual address space is demand paged, with a fixed page size (4 KB on the Core i7). The address space for the 64-bit version of Windows 7 is similar, however, the code and data space is the lower 8 terabytes of the virtual address space.

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. Reserved pages are used when a run of consecutive pages may be needed in the future, such as for the stack. In addition to the free, reserved, and committed attributes, pages also have other attributes, such as being readable, writable, and executable. The top 64 KB and bottom 64 KB of memory are always free, to catch pointer errors (uninitialized pointers are often 0 or −1).

Each committed page has a shadow page on the disk where it is kept when it is not in main memory. Free and reserved pages do not have shadow pages, so references to them cause page faults (the system cannot bring in a page from disk if there is no page on disk). The shadow pages on the disk are arranged into one or more paging files. 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 contains the shadow pages; for data pages, special paging files are used.

Windows 7, like System V, allows files to be mapped directly onto regions of the virtual address spaces (i.e., runs of 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 can be in the disk file instead of in the paging file. As a result, when a file is mapped in, the version in memory may not be identical to the disk version (due to recent writes to the virtual address space). However, when the file is unmapped or is flushed, the disk version is updated from memory.

Windows 7 explicitly allows two or more processes to map in the same file at the same time, possibly even at different virtual addresses. 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.

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. 6-34. All of them operate on a region consisting of either a single page or a sequence of two or more pages that are consecutive in the virtual address space.

API function	Meaning
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 6-34. The principal Windows 7 API calls for managing virtual memory

The first four API functions are self-explanatory. The next two give a process the ability to hardwire some number of 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. Only programs run on behalf of the system administrator may pin pages in memory. And a limit is enforced by the operating system to prevent even these processes from getting too greedy. Although not shown in Fig. 6-34, Windows 7 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 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. A mapped file is (part of) a disk file that can be read from or written to just by accessing the virtual address space, with no explicit I/O. 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.

These API functions are the basic ones upon which the rest of the memory management system is constructed. For example, there are API functions for

allocating and freeing data structures on one or more heaps. Heaps are used for storing data structures that are dynamically created and destroyed. The heaps are not garbage collected by the operating system, so it is up to language run-time systems or user software to free blocks of virtual memory that are no longer in use. (Garbage collection is the automatic removal of unused data structures.) Heap usage in Windows 7 is similar to the use of the *malloc* function in UNIX systems, except that there can be multiple independently managed heaps.

6.5.3 Examples of OS-Level I/O

The heart of any operating system is providing services to user programs, mostly I/O services such as reading and writing files. Both UNIX and Windows 7 offer a wide variety of I/O services to user programs. For most UNIX system calls, Windows 7 has an equivalent call, but the reverse is not true, as Windows 7 has far more calls and each is far more complicated than its UNIX counterpart.

UNIX I/O

Much of the popularity of the UNIX system can be traced directly to its simplicity, which, in turn, is a direct result of the organization of the file system. An ordinary file is a linear sequence of 8-bit bytes starting at 0 and going up to a maximum of typically $2^{64} - 1$ bytes. The operating system itself imposes no record structure on files, although many user programs regard ASCII text files as sequences of lines, each line terminated by a line feed.

Associated with every open file is a pointer to the next byte to be read or written. The read and write system calls read and write data starting at the file position indicated by the pointer. Both calls advance the pointer after the operation by an amount equal to the number of bytes transferred. However, random access to files is possible by explicitly setting the file pointer to a specific value.

In addition to ordinary files, the UNIX system also supports special files, which are used to access I/O devices. Each I/O device typically has one or more special files assigned to it. By reading and writing from the associated special file, a program can read or write from the I/O device. Disks, printers, terminals, and many other devices are handled this way.

The major UNIX file system calls are listed in Fig. 6-35. The creat call (without the *e*) can be used to create a new file. It is not strictly necessary any more, because open can also create a new file now. Unlink removes a file, assuming that the file is in only one directory.

Open is used to open existing files (and create new ones). The mode flag tells how to open it (for reading, for writing, etc.). The call returns a small integer called a **file descriptor** that identifies the file in subsequent calls. When the file is no longer needed, close is called to free up the file descriptor.

The actual file I/O is done with read and write, each having a file descriptor indicating which file to use, a buffer for the data to go to or come from, and a byte

System call	Meaning
creat(name, mode)	Create a file; *mode* specifies the protection mode
unlink(name)	Delete a file (assuming that there is only 1 link to it)
open(name, mode)	Open or create a file and return a file descriptor
close(fd)	Close a file
read(fd, buffer, count)	Read *count* bytes into *buffer*
write(fd, buffer, count)	Write *count* bytes from *buffer*
lseek(fd, offset, w)	Move the file pointer as required by *offset* and *w*
stat(name, buffer)	Return information about a file
chmod(name, mode)	Change the protection mode of a file
fcntl(fd, cmd, ...)	Do various control operations such as locking (part of) a file

Figure 6-35. The principal UNIX file system calls.

count telling how much data to transmit. Lseek is used to position the file pointer, making random access to files possible.

Stat returns information about a file, including its size, time of last access, owner, and more. Chmod changes the protection mode of a file, for example, allowing or forbidding users other than the owner from reading it. Finally, fcntl does various miscellaneous operations on a file, such as locking or unlocking it.

Figure 6-36 illustrates how the major file I/O calls work. This code is minimal and does not include the necessary error checking. Before entering the loop, the program opens an existing file, *data*, and creates a new file, *newf*. Each call returns a file descriptor, *infd*, and *outfd*, respectively. The second parameters to the two calls are protection bits specifying that the files are to be read and written, respectively. Both calls return a file descriptor. If either open or creat fails, a negative file descriptor is returned, telling that the call failed.

```
/* Open the file descriptors. */
infd = open("data", 0);
outfd = creat("newf", ProtectionBits);

/* Copy loop. */
do {
        count = read(infd, buffer, bytes);
        if (count > 0) write(outfd, buffer, count);
} while (count > 0);

/* Close the files. */
close(infd);
close(outfd);
```

Figure 6-36. A program fragment for copying a file using the UNIX system calls. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

The call to read has three parameters: a file descriptor, a buffer, and a byte count. The call tries to read the desired number of bytes from the indicated file into the buffer. The number of bytes actually read is returned in *count*, which will be smaller than *bytes* if the file was too short. The write call deposits the newly read bytes on the output file. The loop continues until the input file has been completely read, at which time the loop terminates and both files are closed.

File descriptors in UNIX are small integers (usually below 20). File descriptors 0, 1, and 2 are special and correspond to **standard input**, **standard output**, and **standard error**, respectively. Normally, these refer to the keyboard, the display, and the display, respectively, but they can be redirected to files by the user. Many UNIX programs get their input from standard input and write the processed output on standard output. Such programs are often called **filters**.

Closely related to the file system is the directory system. Each user may have multiple directories, with each directory containing both files and subdirectories. UNIX systems normally are configured with a main directory, called the **root directory**, containing subdirectories *bin* (for frequently executed programs), *dev* (for the special I/O device files), *lib* (for libraries), and *usr* (for user directories), as shown in Fig. 6-37. In this example, the *usr* directory contains subdirectories for *ast* and *jim*. The *ast* directory contains two files, *data* and *foo.c*, and a subdirectory, *bin*, containing four games.

When multiple disks or disk partitions are present, they can be mounted in the naming tree so that all files on all disks appear in the same directory hierarchy, all reachable from the root directory.

Files can be named by giving their **path** from the root directory. A path contains a list of all the directories traversed from the root to the file, with directory names separated by slashes. For example, the absolute path name of *game2* is */usr/ast/bin/game2*. A path starting at the root is called an **absolute path**.

At every instant, each running program has a **working directory**. Path names may also be relative to the working directory, in which case they do not begin with a slash, to distinguish them from absolute path names. Such paths are called **relative paths**. When */usr/ast* is the working directory, *game3* can be accessed using the path *bin/game3*. A user may create a **link** to someone else's file using the link system call. In the above example, */usr/ast/bin/game3* and */usr/jim/jotto* both access the same file. To prevent cycles in the directory system, links are not permitted to directories. The calls open and creat take either absolute or relative path names as arguments.

The major directory-management system calls in UNIX are listed in Fig. 6-38. Mkdir creates a new directory and rmdir deletes an existing (empty) directory. The next three calls are used to read directory entries. The first one opens the directory, the next one reads entries from it, and the last one closes the directory. Chdir changes the working directory.

Link makes a new directory entry with the new entry pointing to an existing file. For example, the entry */usr/jim/jotto* might have been created by the call

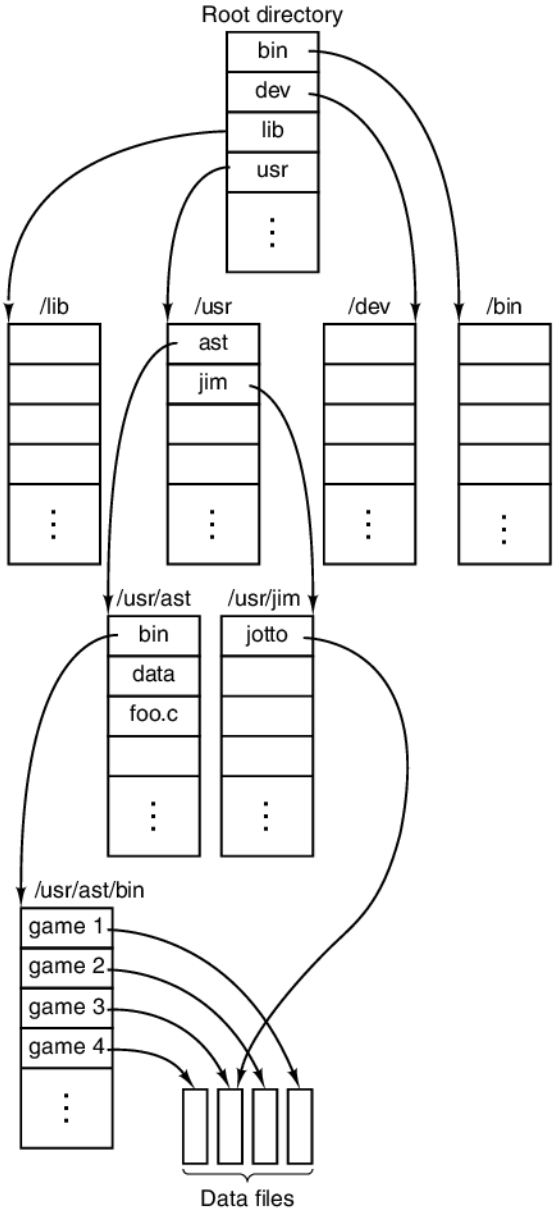

Figure 6-37. Part of a typical UNIX directory system.

```
link("/usr/ast/bin/game3", "/usr/jim/jotto")
```

or an equivalent call using relative path names, depending on the working directory of the program making the call. Unlink removes a directory entry. If the file has

System call	Meaning
mkdir(name, mode)	Create a new directory
rmdir(name)	Delete an empty directory
opendir(name)	Open a directory for reading
readdir(dirpointer)	Read the next entry in a directory
closedir(dirpointer)	Close a directory
chdir(dirname)	Change working directory to *dirname*
link(name1, name2)	Create a directory entry *name2* pointing to *name1*
unlink(name)	Remove *name* from its directory

Figure 6-38. The principal UNIX directory-management calls.

only one link, the file is deleted. If it has two or more links, it is kept. It does not matter whether a removed link is the original or a copy made later. Once a link is made, it is a first-class citizen, indistinguishable from the original. The call

```
unlink("/usr/ast/bin/game3")
```

makes *game3* accessible only via the path */usr/jim/jotto* henceforth. Link and unlink can be used in this way to "move" files from one directory to another.

Associated with every file (including directories, because they are also files) is a bit map telling who may access the file. The map contains three RWX fields, the first controlling the Read, Write, eXecute permissions for the owner, the second for others in the owner's group, and the third for everybody else. Thus RWX R-X --X means that the owner can read the file, write the file, and execute the file (obviously, it is an executable program, or execute would be off), whereas others in his group can read or execute it and strangers can only execute it. With these permissions, strangers can use the program but not steal (copy) it because they do not have read permission. The assignment of users to groups is done by the system administrator, usually called the **superuser**. The superuser also has the power to override the protection mechanism and read, write, or execute any file.

Let us now briefly examine how files and directories are implemented in UNIX. For a more complete treatment, see Vahalia (1996). Associated with each file (and each directory, because a directory is also a file) is a 64-byte block of information called an **i-node**. The i-node tells who owns the file, what the permissions are, where to find the data, and similar things. The i-nodes for the files on each disk are located either in numerical sequence at the beginning of the disk or, if the disk is split up into groups of cylinders, at the start of a cylinder group. Thus, given an i-node number, the UNIX system can locate the i-node by simply calculating its disk address.

A directory entry consists of two parts: a file name and an i-node number. When a program executes

```
open("foo.c", 0)
```

the system searches the working directory for the file name, "foo.c", in order to locate its i-node number. Having found the i-node number, it can then read in the i-node, which tells it all about the file.

When a longer path name is specified, the basic steps outlined above are repeated several times until the full path has been parsed. For example, to locate the i-node number for */usr/ast/data*, the system first searches the root directory for an entry *usr*. Having found the i-node for *usr*, it can read that file (a directory is a file in UNIX). In this file it looks for an entry *ast*, thus locating the i-node number for the file */usr/ast*. By reading */usr/ast*, the system can then find the entry for *data*, and thus the i-node number for */usr/ast/data*. Given the i-node number for the file, it can then find out everything about the file from the i-node.

The format, contents, and layout of an i-node vary somewhat from system to system (especially when networking is in use), but the following items are typically found in each i-node.

1. The file type, the 9 RWX protection bits, and a few other bits.
2. The number of links to the file (number of directory entries for it).
3. The owner's identity.
4. The owner's group.
5. The file length in bytes.
6. Thirteen disk addresses.
7. The time the file was last read.
8. The time the file was last written.
9. The time the i-node was last changed.

The file type distinguishes ordinary files, directories, and two kinds of special files, for block-structured and unstructured I/O devices, respectively. The number of links and the owner identification have already been discussed. The file length is an integer giving the highest byte that has a value. It is perfectly legal to create a file, do an lseek to position 1,000,000, and write 1 byte, which yields a file of length 1,000,001. The file would *not*, however, require storage for all the "missing" bytes.

The first 10 disk addresses point to data blocks. With a block size of 1024 bytes, files up to 10,240 bytes can be handled this way. Address 11 points to a disk block, called an **indirect block**, which contains more disk addresses. With a 1024-byte block and 32-bit disk addresses, the indirect block would contain 256 disk addresses. Files up to $10{,}240 + 256 \times 1024 = 272{,}384$ bytes are handled this way. For still larger files, address 12 points to a block containing the addresses of 256 indirect blocks, which takes care of files up to $272{,}384 + 256 \times 256 \times 1024 = 67{,}381{,}248$ bytes. If this **double indirect block** scheme is still too small, disk ad-

dress 13 is used to point to a **triple indirect block** containing the addresses of 256 double indirect blocks. Using the direct, single, double, and triple indirect addresses, up to 16,843,018 blocks can be addressed, giving a theoretical maximum file size of 17,247,250,432 bytes. If disk addresses are 64 bits instead of 32, and disk blocks are 4 KB, then files can be really, really, really big. Free disk blocks are kept on a linked list. When a new block is needed, the next block is plucked from the list. As a result, the blocks of each file are scattered around the disk.

In order to make disk I/O more efficient, when a file is opened, its i-node is copied to a table in main memory and is kept there for handy reference as long as the file remains open. In addition, a pool of recently referenced disk blocks is maintained in memory. Because most files are read sequentially, it often happens that a file reference requires the same disk block as the previous reference. To strengthen this effect, the system also tries to read the *next* block in a file, before it is referenced, in order to speed up processing. All this optimization is hidden from the user; when a user issues a read call, the program is suspended until the requested data are available in the buffer.

With this background information, we can now take a look to see how file I/O works. Open causes the system to search the directories for the specified path. If the search is successful, the i-node is read into an internal table. Reads and writes require the system to compute the block number from the current file position. The disk addresses of the first 10 blocks are always in main memory (in the i-node); higher-numbered blocks require one or more indirect blocks to be read first. Lseek just changes the current position pointer without doing any I/O.

Link and unlink are also simple to understand now. Link looks up its first argument to find the i-node number. Then it creates a directory entry for the second argument, putting the i-node number of the first file in that entry. Finally, it increases the link count in the i-node by one. Unlink removes a directory entry and decrements the link count in the i-node. If it is zero, the file is removed and all the blocks are put back on the free list.

Windows 7 I/O

Windows 7 supports several file systems, the most important of which are **NTFS** (**NT File System**) and the **FAT** (**File Allocation Table**) file system. The former is a new file system developed specifically for NT; the latter is the old MS-DOS file system, which was also used on Windows 95/98 (albeit with support for longer file names). Since the FAT file system is basically obsolete except for USB sticks and memory cards for cameras, we will study NTFS below.

File names in NTFS can be up to 255 characters long. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Japan, India, and Israel) to write file names in their native language. (In fact, Windows 7 uses Unicode throughout internally; versions starting with Windows 2000 have a single binary that can be used in any country and still use the local language because all

the menus, error messages, etc., are kept in country-dependent configuration files.) NTFS fully supports case-sensitive names (so *foo* is different from *FOO*). 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 using Win32.

As with UNIX, a file is just a linear sequence of bytes, although up to a maximum of $2^{64} - 1$ bytes. File pointers also exist, as in UNIX, but are 64 rather than 32 bits wide, to handle files of the maximum length. The Win32 API function calls for file and directory manipulation are roughly similar to their UNIX counterparts, except that 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. However, unlike in UNIX, handles are not small integers because they are used to identify all kernel objects, of which there are potentially millions. The principal Win32 API functions for file management are listed in Fig. 6-39.

API function	UNIX	Meaning
CreateFile	open	Create a file or open an existing file; return a handle
DeleteFile	unlink	Delete an existing file entry from a directory
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 6-39. The principal Win32 API functions for file I/O. The second column gives the nearest UNIX equivalent.

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 is also used to open existing files as there is no open API function. We have not listed the parameters for the Windows 7 API functions because they are so voluminous. As an example, CreateFile has seven parameters, 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 are to be cloned for the new file.

The next six API functions in Fig. 6-39 are fairly similar to the corresponding UNIX system calls. Note, however, that Windows 7 I/O is, in principle, asynchronous, although it is possible for a process to wait for completion. The last two functions 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 Figure 6-36. Such a procedure (without any error checking) is shown in Fig. 6-40. It has been designed to mimic the structure of Fig. 6-36. In practice, one would not have to program a copy file function since CopyFile is an API function (which executes something close to this program as a library procedure).

```
/* 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 > 0 && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Close the files. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Figure 6-40. A program fragment for copying a file using the Windows 7 API functions. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

Windows 7 supports 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. One significant difference, however, is that UNIX allows the file systems on different disks and machines to be mounted together in a single naming tree starting at a unique root, thus hiding the disk structure from all software. Early versions of Windows (pre Windows 2000) did not have this property, so absolute file names had to begin with a drive letter indicating which logical disk was meant, as in *C:\windows\system\foo.dll*. Starting with Windows 2000 UNIX-style mounting of file systems was added.

The major directory management API functions are given in Fig. 6-41, again along with their nearest UNIX equivalents. The functions should be self-explanatory.

Windows 7 has a much more elaborate security mechanism than most UNIX systems. Although there are hundreds of API functions relating to security, the

API function	UNIX	Meaning
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		Move a file from one directory to another
SetCurrentDirectory	chdir	Change the current working directory

Figure 6-41. The principal Win32 API functions for directory management. The second column gives the nearest UNIX equivalent, when one exists.

following brief description gives the general idea. When a user logs in, his or her initial process is given an **access token** by the operating system. The access token contains the user's **SID** (**Security ID**), a list of the security groups to which the user belongs, any special privileges available, the integrity level of the process, and a few other items. The point of the access token is to concentrate all the security information in one easy-to-find place. All processes created by this process inherit the same access token.

One of the parameters that can be supplied when any object is created is its **security descriptor**. The security descriptor contains a list of entries called an **ACL** (**Access Control List**). Each entry permits or prohibits some set of the operations on the object by some SID or group. For example, a file could have a security descriptor specifying that Elinor has no access to the file at all, Ken can read the file, Linda can read or write the file, and all members of the XYZ group can read the file's length but nothing else. Defaults can also be set up to deny access to anyone not explicitly listed.

When a process tries to perform some operation on an object using a handle, the security manager gets the process' access token and first checks the integrity level in the object's security descriptor against the integrity level in the token. A process cannot obtain a handle with write permission for any object with a higher integrity level. Integrity levels are primarily used to restrict what code loaded by Web browsers can do to modify the system. After the integrity-level check, the security manager goes down the list of entries in the ACL in order. As soon as it finds an entry that matches the called's SID or one of the called's groups, the access found there is taken as definitive. 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. The security descriptor also contains information used for auditing accesses to the object.

Let us now take a quick look at how files and directories are implemented in Windows 7. Each disk is statically divided up into self-contained volumes, which are the same as disk partitions in UNIX. Each volume contains bit maps, files,

directories, and other data structures for managing its information. Each volume is organized as a linear sequence of **clusters**, with the cluster size being fixed for each volume and ranging from 512 bytes to 64 KB, depending on the volume size. Clusters 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 has an entry for each file and directory in the volume. These entries are analogous to the i-nodes in UNIX. The MFT is itself a file, and as such can be placed anywhere within the volume. This property is useful in case there are defective disk blocks at the beginning of the volume where the MFT would normally be stored. UNIX systems normally store certain key information at the start of each volume and in the (extremely unlikely) case that one of these blocks is irreparably damaged, the entire volume has to be repositioned.

The MFT is shown in Fig. 6-42. It begins with a header containing information about the volume, such as (pointers to) the root directory, the boot file, the bad-block file, the free-list administration, etc. After that comes an entry per file or directory, 1 KB except when the cluster size is 2 KB or more. Each entry contains all the metadata (administrative information) about the file or directory. Several formats are allowed, one of which is shown in Fig. 6-42.

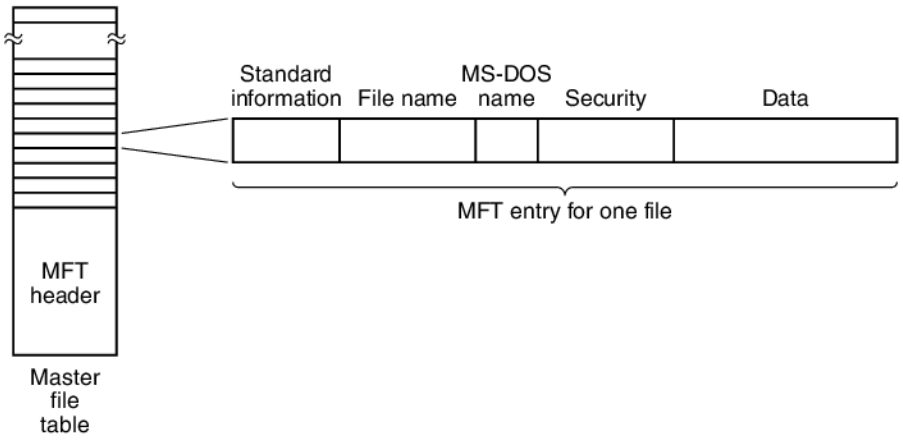

Figure 6-42. The Windows 7 master file table.

The standard information field contains information such as the time stamps needed by POSIX, the hard link count, the read-only and archive bits, etc. It is a fixed-length field and always present. The file name is of variable length, up to 255 Unicode characters. In order to make such files accessible to old 16-bit programs, files can also have a MS-DOS name, which consists of eight alphanumeric characters optionally followed by a dot and an extension of no more than three

alphanumeric characters. If the actual file name conforms to the MS-DOS 8+3 naming rule, a secondary MS-DOS name is not used.

Next comes the security information. In versions up to and including Windows NT 4.0, the security field contained the actual security descriptor. Starting with Windows 2000, all the security information was centralized in a single file, with the security field simply pointing to the relevant part of this file.

For small files, the file data itself is actually contained in the MFT entry, saving a disk access to fetch it. This idea is called an **immediate file** (Mullender and Tanenbaum, 1984). For somewhat larger files, this field contains pointers to the clusters containing the data, or more commonly, runs of consecutive clusters so a single cluster number and a length can represent an arbitrary amount of file data. If a single MFT entry is insufficiently large to hold whatever information it is supposed to hold, one or more additional entries can be chained to it.

The maximum file size is 2^{64} bytes. To get an idea of how big a 2^{64}-byte (i.e., 2^{67}-bit) file is, imagine that it were written out in binary, with each 0 or 1 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.

The NTFS file system has many other interesting properties including support for multiple data streams for each file, encryption, data compression, and fault tolerance using atomic transactions. Additional information about it can be found in Russinovich and Solomon (2005).

6.5.4 Examples of Process Management

Both UNIX and Windows 7 allow a job to be split up into multiple processes that can run in (pseudo)parallel and communicate with each other, in the style of the producer-consumer example discussed earlier. In this section we will discuss how processes are managed in both systems. Both systems also support parallelism within a single process using threads, so that will also be discussed.

UNIX Process Management

At any time, a UNIX process can create a subprocess that is an exact replica of itself by executing the fork system call. The original process is called the **parent** and the new one the **child**. Right after the fork, the two processes are identical and even share the same file descriptors. Thereafter, each one goes its own way and does whatever it wants to, independent of the other one.

In many cases, the child process juggles the file descriptors in certain ways and then executes the exec system call, which replaces its program and data with the program and data found in an executable file specified as parameter to the exec call. For example, when a user types a command *xyz* at a terminal, the command interpreter (shell) executes fork to create a child process. This child process then executes exec to run the *xyz* program.

The two processes run in parallel (with or without exec), unless the parent wishes to wait for the child to terminate before continuing. If the parent wishes to wait, it executes either the wait or waitpid system call, which causes it to be suspended until the child finishes by executing exit. After the child finishes, the parent continues.

Processes can execute fork as often as they want, giving rise to a tree of processes. In Fig. 6-43, for example, process *A* has executed fork twice, creating two children, *B* and *C*. Then *B* also executed fork twice, and *C* executed it once, giving the final tree of six processes.

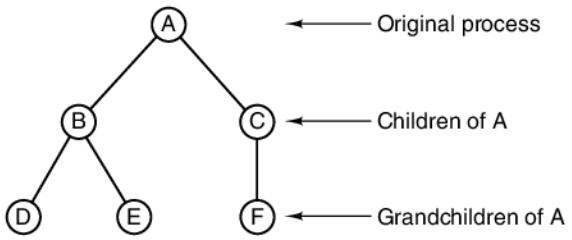

Figure 6-43. A process tree in UNIX.

Processes in UNIX can communicate with each other via a structure called a **pipe**. A pipe is a kind of buffer into which one process can write a stream of data and another can take it out. Bytes are always retrieved from a pipe in the order they were written. Random access is not possible. Pipes do not preserve message boundaries, so if one process does four 128-byte writes and the other does a 512-byte read, the reader will get all the data at once, with no indication that they were written in multiple operations.

In System V and Linux, another way for processes to communicate is by using **message queues**. A process can create a new message queue or open an existing one using msgget. Using a message queue, a process can send messages using msgsnd and receive them using msgrecv. Messages sent this way differ in several ways from data stuffed into a pipe. First, message boundaries are preserved, whereas a pipe is just a byte stream. Second, messages have priorities, so urgent ones can skip ahead of less important ones. Third, messages are typed, and a msgrecv can specify a particular type, if desired.

Another communication mechanism is the ability of two or more processes to share a region of their respective address spaces. UNIX handles this shared memory by mapping the same pages into the virtual address space of all the sharing processes. As a result, a write by one process into the shared region is immediately visible to the other processes. This mechanism provides a very high bandwidth communication path between processes. The system calls involved in shared memory go by names like shmat and shmop.

System V and Linux also provide semaphores. These work essentially as described in the producer-consumer example given in the text.

Yet another facility provided by all POSIX-conformant UNIX systems is the ability to have multiple threads of control within a single process. These threads of control, usually just called **threads**, are like lightweight processes that share a common address space and everything associated with that address space, such as file descriptors, environment variables, and outstanding timers. However, each thread has its own program counter, own registers, and own stack. When a thread blocks (i.e., has to stop temporarily until I/O completes or some other event happens), other threads in the same process are still able to run. Two threads in the same process operating as a producer and consumer are similar, but not identical, to two single-thread processes that are sharing a memory segment containing a buffer. The differences have to do with the fact that in the latter case, each process has its own file descriptors, etc., whereas in the former case all of these items are shared. We saw the use of Java threads in our producer-consumer example earlier. Often the Java runtime system uses an operating system thread for each of its threads, but it does not have to do this.

As an example of where threads might be useful, consider a World Wide Web server. Such a server might keep a cache of commonly used Web pages in main memory. If a request is for a page in the cache, the Web page is returned immediately. Otherwise, it is fetched from disk. Unfortunately, waiting for the disk takes a long time (typically 20 msec), during which the process is blocked and cannot serve new incoming requests, even those for Web pages in the cache.

The solution is to have multiple threads within the server process, all of which share the common Web page cache. When one thread blocks, other threads can handle new requests. To prevent blocking without threads, one could have multiple server processes, but this would probably entail replicating the cache, thus wasting valuable memory.

The UNIX standard for threads is called **pthreads**, and is defined by POSIX (P1003.1C). It contains calls for managing and synchronizing threads. It is not defined whether threads are managed by the kernel or entirely in user space. The most commonly used thread calls are listed in Fig. 6-44.

Let us briefly examine the thread calls shown in Fig. 6-44. The first call, pthread_create, creates a new thread. After successful completion, one more thread is running in the called's address space than before the call. A thread that has done its job and wants to terminate calls pthread_exit. A thread can wait for another thread to exit by calling pthread_join. If the thread waited for has already exited, the pthread_join finishes immediately. Otherwise it blocks.

Threads can synchronize using **mutexes**. A mutex guards some resource, such as a buffer shared by two threads. To make sure that only one thread at a time accesses the shared resource, threads are expected to lock the mutex before touching the resource and unlock it when they are done. As long as all threads obey this protocol, race conditions can be avoided. Mutexes are like binary semaphores (semaphores that can take on only the values of 0 and 1). The name "mutex" comes from the fact that mutexes are used to ensure mutual exclusion.

Thread call	Meaning
pthread_create	Create a new thread in the called's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Figure 6-44. The principal POSIX thread calls.

Mutexes can be created and destroyed by the calls pthread_mutex_init and pthread_mutex_destroy, respectively. A mutex can be in one of two states: locked or unlocked. When a thread needs to set a lock on an unlocked mutex (using pthread_mutex_lock), the lock is set and the thread continues. However, when a thread tries to lock a mutex that is already locked, it blocks. When the locking thread is finished with the shared resource, it is expected to unlock the corresponding mutex by calling pthread_mutex_unlock.

Mutexes are intended for short-term locking, such as protecting a shared variable. They are not intended for long-term synchronization, such as waiting for a tape drive to become free. For long-term synchronization, **condition variables** are provided. These are created and destroyed by calls to pthread_cond_init and pthread_cond_destroy, respectively.

A condition variable is used by having one thread wait on it, and another signal it. For example, having discovered that the tape drive it needs is busy, a thread would do pthread_cond_wait on a condition variable that all the threads have agreed to associate with the tape drive. When the thread using the tape drive is finally done with it (possibly hours later), it uses pthread_cond_signal to release exactly one thread waiting on that condition variable (if any). If no thread is waiting, the signal is lost. Condition variables do not count like semaphores. A few other operations are also defined on threads, mutexes, and condition variables.

Windows 7 Process Management

Windows 7 supports multiple processes, which can communicate and synchronize. Each process contains at least one thread. Together, processes and threads (which can be scheduled by the process itself) provide a general set of tools for

managing parallelism, both on uniprocessors (single-CPU machines) and on multiprocessors (multi-CPU machines).

New processes are created using the 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 called.

Windows 7 does not enforce any kind of parent-child or other hierarchy. All processes are created 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 handle suitable for another process and then give it the handle, so the implicit process hierarchy may not last long.

Each process in Windows 7 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: the security descriptor, the stack size, the starting address, a user-defined parameter, the initial state of the thread (ready or blocked), and 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).

When the kernel does scheduling, it looks only at the runnable threads and pays no any attention at all to which process each one is in. This means that the kernel is always aware of which threads are ready and which ones are blocked. Because threads are kernel objects, they have security descriptors and handles. Since a handle for a thread can be passed to another process, it is possible to have

one process control (or even create) threads in a different process. This feature is useful for debuggers, for example.

Processes can communicate in a wide variety of ways, including pipes, named pipes, 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 would 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.

Sockets are like pipes, except that they normally connect processes on different machines. However, they can also be used to connect processes on the same machine. In general, there is usually little advantage to using a socket connection over a pipe or named pipe for intramachine communication.

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. Instead, the object(s) pointed to have to be bundled up and sent to the destination 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 our producer-consumer example can be easily implemented.

Just as Windows 7 provides numerous interprocess communication mechanisms, it also provides numerous synchronization mechanisms, including semaphores, mutexes, critical sections, 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. Semaphores can also be given names when they are created so that other processes can open them by name. Calls for up and down are present, although they have the speculiar 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 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 entirely in user space, they are much faster than mutexes. Windows 7 also provides condition variables, lightweight reader/writer locks, lock-free operations, and other synchronization mechanisms that work only between the threads of a single process.

The last synchronization mechanism uses kernel objects called **events**. A thread can wait for an event to occur with WaitForSingleObject. A thread can release a single thread waiting on an event with SetEvent or it can release all threads waiting on an event with PulseEvent. Events come in several flavors and have a variety of options, too. Windows uses events to synchronize on the completion of asynchronous I/O and for other purposes.

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.

6.6 SUMMARY

The operating system can be regarded as an interpreter for certain architectural features not found at the ISA level. Chief among these are virtual memory, virtual I/O instructions, and facilities for parallel processing.

Virtual memory is an architectural feature whose purpose is to allow programs to use more address space than the machine has physical memory, or to provide a consistent and flexible mechanism for memory protection and sharing. It can be implemented as pure paging, pure segmentation, or a combination of the two. In pure paging, the address space is broken up into equal-sized virtual pages. Some of these are mapped onto physical page frames. Others are not mapped. A reference to a mapped page is translated by the MMU into the correct physical address. A reference to an unmapped page causes a page fault. Both the Core i7 and the OMAP4430 ARM CPU have MMUs that support virtual memory and paging.

The most important I/O abstraction present at this level is the file. A file consists of a sequence of bytes or logical records that can be read and written without knowledge of how disks, tapes, and other I/O devices work. Files can be accessed sequentially, randomly by record number, or randomly by key. Directories can be used to group files together. Files can be stored in consecutive sectors or scattered around the disk. In the latter case, normal on hard disks, data structures are needed

to locate all the blocks of a file. Free disk storage can be kept track of using a list or a bit map.

Parallel processing is often supported and is implemented by simulating multiple processors by timesharing a single CPU. Uncontrolled interaction between processes can lead to race conditions. To solve this problem, synchronization primitives are introduced, of which semaphores are a simple example. Using semaphores, producer-consumer problems can be solved simply and elegantly.

Two examples of sophisticated operating systems are UNIX and Windows 7. Both support paging and memory-mapped files. They also both support hierarchical file systems, with files consisting of byte sequences. Finally, both support processes and threads and provide ways to synchronize them.

PROBLEMS

1. Why does an operating system interpret only some of the level 3 instructions, whereas a microprogram interprets all the ISA-level instructions?
2. A machine has a 32-bit byte-addressable virtual address space. The page size is 4 KB. How many pages of virtual address space exist?
3. Is it necessary to have the page size be a power of 2? Could a page of size, say, 4000 bytes be implemented in theory? If so, would it be practical?
4. A virtual memory has a page size of 1024 words, eight virtual pages, and four physical page frames. The page table is as follows:

Virtual page	Page frame
0	3
1	1
2	not in main memory
3	not in main memory
4	2
5	not in main memory
6	0
7	not in main memory

   a. Make a list of all virtual addresses that will cause page faults.
   b. What are the physical addresses for 0, 3728, 1023, 1024, 1025, 7800, and 4096?

5. A computer has 16 pages of virtual address space but only four page frames. Initially, the memory is empty. A program references the virtual pages in the order

   0, 7, 2, 7, 5, 8, 9, 2, 4

a. Which references cause a page fault with LRU?
b. Which references cause a page fault with FIFO?

6. In Sec. 6.1.4 an algorithm was presented for implementing a FIFO page replacement strategy. Devise a more efficient one. *Hint:* It is possible to update the counter in the newly loaded page, leaving all the others alone.

7. In the paged systems discussed in the text, the page fault handler was part of the ISA level and thus was not present in any OSM-level program's address space. In reality, the page fault handler also occupies pages, and might, under some circumstances (e.g., FIFO page replacement policy), itself be removed. What would happen if the page fault handler were not present when a page fault occurred? How could this be fixed?

8. Not all computers have a hardware bit that is automatically set when a page is written to. Nevertheless, it is useful to keep track of which pages have been modified, to avoid having to assume worst case and write all pages back to the disk after use. Assuming that each page has hardware bits to separately enable access for reading, writing, and execution, how can the operating system keep track of which pages are clean and which are dirty?

9. A segmented memory has paged segments. Each virtual address has a 2-bit segment number, a 2-bit page number, and an 11-bit offset within the page. The main memory contains 32 KB, divided into 2-KB pages. Each segment is either read-only, read/execute, read/write, or read/write/execute. The page tables and protection are as follows:

Segment 0		Segment 1		Segment 2	Segment 3	
Read only		Read/execute		Read/write/execute	Read/write	
Virtual page	Page frame	Virtual page	Page frame		Virtual page	Page frame
0	9	0	On disk	Page table	0	14
1	3	1	0	not in	1	1
2	On disk	2	15	main	2	6
3	12	3	8	memory	3	On disk

For each of the following accesses to virtual memory, tell what physical address is computed. If a fault occurs, tell which kind.

Access	*Segment*	*Page*	*Offset within page*
1. fetch data	0	1	1
2. fetch data	1	1	10
3. fetch data	3	3	2047
4. store data	0	1	4
5. store data	3	1	2
6. store data	3	0	14
7. branch to it	1	3	100
8. fetch data	0	2	50
9. fetch data	2	0	5
10. branch to it	3	0	60

10. Some computers allow I/O directly to user space. For example, a program could start up a disk transfer to a buffer inside a user process. Does this cause any problems if compaction is used to implement the virtual memory? Discuss.

11. Operating systems that allow memory-mapped files always require files to be mapped at page boundaries. For example, with 4-KB pages, a file can be mapped in starting at virtual address 4096, but not starting at virtual address 5000. Why?

12. When a segment register is loaded on the Core i7, the corresponding descriptor is fetched and loaded into an invisible part of the segment register. Why do you think the Intel designers decided to do this?

13. A program on the Core i7 references local segment 10 with offset 8000. The BASE field of LDT segment 10 contains 10000. Which page directory entry does the Core i7 use? What is the page number? What is the offset?

14. Discuss some possible algorithms for removing segments in an unpaged, segmented memory.

15. Compare internal fragmentation to external fragmentation. What can be done to alleviate each?

16. Supermarkets are constantly faced with a problem similar to page replacement in virtual memory systems. They have a fixed amount of shelf space to display an ever-increasing number of products. If an important new product comes along, say, 100% efficient dog food, some existing product must be dropped from the inventory to make room for it. The obvious replacement algorithms are LRU and FIFO. Which of these would you prefer?

17. In some ways, caching and paging are very similar. In both cases there are two levels of memory (the cache and main memory in the former and main memory and disk in the latter). In this chapter we looked at some of the arguments in favor of large disk pages and small disk pages. Do the same arguments hold for cache line sizes?

18. Why do many file systems require that a file be explicitly opened with an open system call before being read?

19. Compare the bit-map and hole-list methods for keeping track of free space on a disk with 800 cylinders, each one having 5 tracks of 32 sectors. How many holes would it take before the hole list would be larger than the bit map? Assume that the allocation unit is the sector and that a hole requires a 32-bit table entry.

20. A third hole allocation scheme, in addition to best fit and first fit, is worst fit, where a process is allocated space from the largest remaning hole. What advantage can be gained by using the worst fit algorithm?

21. Describe a purpose for the file open system call that was not mentioned in the text.

22. To be able to make some predictions of disk performance, it is useful to have a model of storage allocation. Suppose that the disk is viewed as a linear address space of $N \gg 1$ sectors, consisting of a run of data blocks, then a hole, then another run of data blocks, and so on. If empirical measurements show that the probability distributions for data and hole lengths are the same, with the chance of either being i sectors as 2^{-i}, what is the expected number of holes on the disk?

23. On a certain computer, a program can create as many files as it needs, and all files may grow dynamically during execution without giving the operating system any advance information about their ultimate size. Do you think that files are stored in consecutive sectors? Explain.

24. Studies of different file systems have shown that more than half the files are a few KB or smaller, with the vast majority of files less than something like 8 KB. On the other hand, the largest 10 percent of all files usually occupies about 95 percent of the entire disk space in use. From this data, what conclusion can you draw about disk block size?

25. Consider the following method by which an operating system might implement semaphore instructions. Whenever the CPU is about to do an up or down on a semaphore (an integer variable in memory), it first sets the CPU priority or mask bits in such a way as to disable all interrupts. Then it fetches the semaphore, modifies it, and branches accordingly. Finally, it enables interrupts again. Does this method work if

a. There is a single CPU that switches between processes every 100 msec?
b. Two CPUs share a common memory in which the semaphore is located?

26. The description of semaphores in Sec. 6.3.3 states: "Usually sleeping processes are strung together in a queue to keep track of them." What advantage is gained by using a queue for waiting processes as opposed to waking a random sleeping processes when an up is performed?

27. The Nevercrash Operating System Company has been receiving complaints from some of its customers about its latest release, which includes semaphore operations. They feel it is immoral for processes to block (they call it "sleeping on the job"). Since it is company policy to give the customers what they want, it has been proposed to add a third operation, peek, to supplement up and down. peek simply examines the semaphore without changing it or blocking the process. In this way, programs that feel it is immoral to block can first inspect the semaphore to see if it is safe to do a down. Will this idea work if three or more processes use the semaphore? If two processes use the semaphore?

28. Make a table showing which of the processes P1, P2, and P3 are running and which are blocked as a function of time from 0 to 1000 msec. All three processes perform up and down instructions on the same semaphore. When two processes are blocked and an up is done, the process with the lower number is restarted, that is, P1 gets preference over P2 and P3, and so on. Initially, all three are running and the semaphore is 1.

At $t = 100$ P1 does a down
At $t = 200$ P1 does a down
At $t = 300$ P2 does an up
At $t = 400$ P3 does a down
At $t = 500$ P1 does a down
At $t = 600$ P2 does an up
At $t = 700$ P2 does a down
At $t = 800$ P1 does an up
At $t = 900$ P1 does an up

29. In an airline reservation system, it is necessary to ensure that while one process is busy using a file, no other process can also use it. Otherwise, two different processes,

working for two different ticket agents, might each inadvertently sell the last seat on some flight. Devise a synchronization method using semaphores that makes sure that only one process at a time accesses each file (assuming the processes obey the rules).

30. To make it possible to implement semaphores on a computer with multiple CPUs that share a common memory, computer architects often provide a Test and Set Lock instruction. `TSL X` tests the location X. If the contents are zero, they are set to 1 in a single, indivisible memory cycle, and the next instruction is skipped. If it is nonzero, the `TSL` acts like a no-op. Using `TSL` it is possible to write procedures *lock* and *unlock* with the following properties. *lock*(x) checks to see if x is locked. If not, it locks x and returns control. If x is already locked, it just waits until it becomes unlocked, then it locks x and returns control. *unlock* releases an existing lock. If all processes lock the semaphore table before using it, only one process at a time can fiddle with the variables and pointers, thus preventing races. Write *lock* and *unlock* in assembly language. (Make any reasonable assumptions you need.)

31. Show the values of *in* and *out* for a circular buffer of length 65 words after each of the following operations. Both start at 0.

a. 22 words are put in
b. 9 words are removed
c. 40 words are put in
d. 17 words are removed
e. 12 words are put in
f. 45 words are removed
g. 8 words are put in
h. 11 words are removed

32. Suppose that a version of UNIX uses 2-KB disk blocks and stores 512 disk addresses per indirect block (single, double, and triple). What would the maximum file size be? (Assume that file pointers are 64 bits wide).

33. Suppose that the UNIX system call

```
unlink("/usr/ast/bin/game3")
```

were executed in the context of Figure 6-37. Describe carefully what changes are made in the directory system.

34. Imagine that you had to implement the UNIX system on an embedded system where main memory was in short supply. After a considerable amount of shoehorning, it still did not quite fit, so you picked a system call at random to sacrifice for the general good. You picked `pipe`, which creates the pipes used to send byte streams from one process to another. Is it still possible to implement I/O redirection somehow? What about pipelines? Discuss the problems and possible solutions.

35. The Committee for Fairness to File Descriptors is organizing a protest against the UNIX system because whenever the latter returns a file descriptor, it always returns the lowest number not currently in use. Consequently, higher-numbered file descriptors are hardly ever used. Their plan is to return the lowest number not yet used by the program rather than the lowest number currently not in use. They claim that it is trivial to implement, will not affect existing programs, and is fairer. What do you think?

36. In Windows 7 it is possible to set up an access control list in such a way that Roberta has no access at all to a file, but everyone else has full access to it. How do you think this is implemented?

37. Describe two different ways to program producer-consumer problems using shared buffers and semaphores in Windows 7. Think about how to implement the shared buffer in each case.

38. It is common to test out page replacement algorithms by simulation. For this exercise, you are to write a simulator for a page-based virtual memory for a machine with 64 1-KB pages. The simulator should maintain a single table of 64 entries, one per page, containing the physical page number corresponding to that virtual page. The simulator should read in a file containing virtual addresses in decimal, one address per line. If the corresponding page is memory, just record a page hit. If it is not in memory, call a page replacement procedure to pick a page to evict (i.e., an entry in the table to overwrite) and record a page miss. No page transport actually occurs. Generate a file consisting of random addresses and test the performance for both LRU and FIFO. Now generate an address file in which x percent of the addresses are four bytes higher than the previous one (to simulate locality). Run tests for various values of x and report on your results.

39. The program of Fig. 6-26 has a fatal race condition because two threads access shared variables in an uncontrolled way, without using semaphores or any other mutual exclusion technique. Run this program and see how long it takes to hang. If you cannot make it hang, modify it to increase the size of the window of vulnerability by putting some computing between adjusting *m.in* and *m.out* and testing them. How much computing do you have to put in before it fails, say, once an hour?

40. Write a program for UNIX or Windows 7 that takes as input the name of a directory. The program should print a list of the files in the directory, one line per file, and after the file name, print the size of the file. Print the file names in the order they occur in the directory. Unused slots in the directory should be listed as (unused).

This page intentionally left blank

7

THE ASSEMBLY LANGUAGE LEVEL

In Chapters 4, 5, and 6 we discussed three different levels present on most contemporary computers. This chapter is concerned primarily with another level that is present on all computers: the assembly language level. The assembly language level differs in a significant respect from the microarchitecture, ISA, and operating system machine levels—it is implemented by translation rather than by interpretation.

Programs that convert a user's program written in some language to another language are called **translators**. The language in which the original program is written is called the **source language** and the language to which it is converted is called the **target language**. Both the source language and the target language define levels. If a processor that can directly execute programs written in the source language is available, there is no need to translate the source program into the target language.

Translation is used when a processor (either hardware or an interpreter) is available for the target language but not for the source language. If the translation has been performed correctly, running the translated program will give precisely the same results as the execution of the source program would have given had a processor for it been available. Consequently, it is possible to implement a new level for which there is no processor by first translating programs written for that level to a target level and then executing the resulting target-level programs.

It is important to note the difference between translation, on the one hand, and interpretation, on the other hand. In translation, the original program in the source

language is not directly executed. Instead, it is converted to an equivalent program called an **object program** or **executable binary program** whose execution is carried out only after the translation has been completed. In translation, there are two distinct steps:

1. Generation of an equivalent program in the target language.
2. Execution of the newly generated program.

These two steps do not occur simultaneously. The second step does not begin until the first has been completed. In interpretation, there is only one step: executing the original source program. No equivalent program need be generated first, although sometimes the source program is converted to an intermediate form (e.g., Java byte code) for easier interpretation.

While the object program is being executed, only three levels are in evidence: the microarchitecture level, the ISA level, and the operating system machine level. Consequently, three programs—the user's object program, the operating system, and the microprogram (if any)—can be found in the computer's memory at run time. All traces of the original source program have vanished. Thus the number of levels present at execution time may differ from the number of levels present before translation. It should be noted, however, that although we define a level by the instructions and linguistic constructs available to its programmers (and not by the implementation technique), other authors sometimes make a greater distinction between levels implemented by execution-time interpreters and levels implemented by translation.

7.1 INTRODUCTION TO ASSEMBLY LANGUAGE

Translators can be roughly divided into two groups, depending on the relationship between the source language and the target language. When the source language is essentially a symbolic representation for a numerical machine language, the translator is called an **assembler** and the source language is called an **assembly language**. When the source language is a high-level language such as Java or C and the target language is either a numerical machine language or a symbolic representation for one, the translator is called a **compiler**.

7.1.1 What Is an Assembly Language?

A pure assembly language is a language in which each statement produces exactly one machine instruction. In other words, there is a one-to-one correspondence between machine instructions and statements in the assembly program. If each line in the assembly language program contains exactly one statement and

each machine word contains exactly one machine instruction, then an *n*-line assembly program will produce an *n*-instruction machine language program.

The reason that people use assembly language, as opposed to programming in machine language (in binary or hexadecimal), is that it is much easier to program in assembly language. The use of symbolic names and symbolic addresses instead of binary or hexadecimal ones makes an enormous difference. Most people can remember that the abbreviations for add, subtract, multiply, and divide are ADD, SUB, MUL, and DIV, but few can remember the corresponding numerical values the machine uses. The assembly language programmer need only remember the symbolic names because the assembler translates them to the machine instructions.

The same remarks apply to addresses. The assembly language programmer can give symbolic names to memory locations and have the assembler worry about supplying the correct numerical values. The machine language programmer must always work with the numerical values of the addresses. As a consequence, no one programs in machine language today, although people did so decades ago, before assemblers had been invented.

Assembly languages have another property, besides the one-to-one mapping of assembly language statements onto machine instructions, that distinguishes them from high-level languages. The assembly programmer has access to all the features and instructions available on the target machine. The high-level language programmer does not. For example, if the target machine has an overflow bit, an assembly language program can test it, but a Java program cannot test it. An assembly language program can execute every instruction in the instruction set of the target machine, but the high-level language program cannot. In short, everything that can be done in machine language can be done in assembly language, but many instructions, registers, and similar features are not available for the high-level language programmer to use. Languages for system programming, like C, are a cross between these types, with the syntax of a high-level language but with some of the access to the machine of an assembly language.

One final difference that is worth making explicit is that an assembly language program can run only on one family of machines, whereas a program written in a high-level language can potentially run on many machines. For many applications, this ability to move software from one machine to another is of great practical importance.

7.1.2 Why Use Assembly Language?

Assembly language programming is difficult. Make no mistake about that. It is not for wimps and weaklings. Furthermore, writing a program in assembly language takes much longer than writing the same program in a high-level language. It also takes much longer to debug and is much harder to maintain.

Under these conditions, why would anyone ever program in assembly language? There are two reasons: performance and access to the machine. First of

all, an expert assembly language programmer working very hard can sometimes produce code that is much smaller and much faster than a high-level language programmer can. For some applications, speed and size are critical. Many embedded applications, such as the code on a smart card or RFID card, device drivers, string-manipulation libraries, BIOS routines, and the inner loops of performance-critical real-time applications fall in this category.

Second, some procedures need complete access to the hardware, something usually impossible in high-level languages. For example, the low-level interrupt and trap handlers in an operating system and the device controllers in many embedded real-time systems fall into this category.

Besides these reasons for programming in assembly language, there are also two additional reasons for studying it. First, a compiler must either produce output used by an assembler or perform the assembly process itself. Thus understanding assembly language is essential to understanding how compilers work. Someone has to write the compiler (and its assembler) after all.

Second, studying assembly language exposes the real machine to view. For students of computer architecture, writing some assembly code is the only way to get a feel for what a machine is really like at the architectural level.

7.1.3 Format of an Assembly Language Statement

Although the structure of an assembly language statement closely mirrors the structure of the machine instruction that it represents, assembly languages for different machines sufficiently resemble one another to allow a discussion of assembly language in general. Figure 7-1 shows fragments of assembly language programs for the x86 which performs the computation $N = I + J$. The statements below the blank line are commands to the assembler to reserve memory for the variables I, J, and N and are not symbolic representations of machine instructions.

Label	Opcode	Operands	Comments
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I + J
	MOV	N,EAX	; N = I + J
I	DD	3	; reserve 4 bytes initialized to 3
J	DD	4	; reserve 4 bytes initialized to 4
N	DD	0	; reserve 4 bytes initialized to 0

Figure 7-1. Computation of $N = I + J$ on the x86.

Several assemblers exist for the Intel family (i.e., x86), each with a different syntax. In this chapter we will use the Microsoft MASM assembly language for our example. There are many assemblers for the ARM, but the syntax is comparable to the x86 assembler, so one example should siffice.

Assembly language statements have up to four parts: first, a label field, second, an operation (opcode) field, third, an operand field, and fourth, a comments field. None of these is mandatory. Labels, which are used to provide symbolic names for memory addresses, are needed on executable statements so that the statements can be branched to. Additionally, they are needed for data words to permit the data stored there to be accessible by symbolic name. If a statement is labeled, the label (usually) begins in column 1.

The example of Fig. 7-1 has four labels: *FORMULA*, *I*, *J*, and *N*. The MASM requires colons on code labels but not on data labels. There is nothing fundamental about this. Other assemblers may have other requirements. Nothing in the underlying architecture suggests one choice or the other. One advantage of the colon notation is that with it a label can appear by itself on a line, with the opcode in column 1 of the next line. This style is convenient for compilers to generate. Without the colon, there would be no way to tell a label on a line all by itself from an opcode on a line all by itself. The colon eliminates this potential ambiguity.

It is an unfortunate characteristic of some assemblers that labels are restricted to six or eight characters. In contrast, most high-level languages allow the use of arbitrarily long names. Long, well-chosen names make programs much more readable and understandable by other people.

Each machine has some registers, so they need names. The x86 registers have names like EAX, EBX, ECX, and so on.

The opcode field contains either a symbolic abbreviation for the opcode—if the statement is a symbolic representation for a machine instruction—or a command to the assembler itself. The choice of an appropriate name is just a matter of taste, and different assembly language designers often make different choices. The designers of the MASM assembler decided to use MOV for both loading a register from memory and storing a register into memory but they could have chosen MOVE or LOAD and STORE.

Assembly progams often need to reserve space for variables. The MASM assembly language designers chose DD (Define Double), since a word on the 8088 was 16 bits.

The operand field of an assembly language statement is used to specify the addresses and registers used as operands by the machine instruction. The operand field of an integer addition instruction tells what is to be added to what. The operand field of a branch instruction tells where to branch to. Operands can be registers, constants, memory locations, and so on.

The comments field provides a place for programmers to put helpful explanations of how the program works for the benefit of other programmers who may subsequently use or modify the program (or for the benefit of the original programmer a year later). An assembly language program without such documentation is nearly incomprehensible to all programmers, frequently including the author as well. The comments field is solely for human consumption; it has no effect on the assembly process or on the generated program.

7.1.4 Pseudoinstructions

In addition to specifying which machine instructions to execute, an assembly language program can also contain commands to the assembler itself, for example, asking it to allocate some storage or to eject to a new page on the listing. Commands to the assembler itself are called **pseudoinstructions** or sometimes **assembler directives**. We have already seen a typical pseudoinstruction in Fig. 7-1(a): DD. Some other pseudoinstructions are listed in Fig. 7-2. These are taken from the Microsoft MASM assembler for the x86.

Pseudoinstruction	Meaning
SEGMENT	Start a new segment (text, data, etc.) with certain attributes
ENDS	End the current segment
ALIGN	Control the alignment of the next instruction or data
EQU	Define a new symbol equal to a given expression
DB	Allocate storage for one or more (initialized) bytes
DW	Allocate storage for one or more (initialized) 16-bit (word) data items
DD	Allocate storage for one or more (initialized) 32-bit (double) data items
DQ	Allocate storage for one or more (initialized) 64-bit (quad) data items
PROC	Start a procedure
ENDP	End a procedure
MACRO	Start a macro definition
ENDM	End a macro definition
PUBLIC	Export a name defined in this module
EXTERN	Import a name from another module
INCLUDE	Fetch and include another file
IF	Start conditional assembly based on a given expression
ELSE	Start conditional assembly if the IF condition above was false
ENDIF	End conditional assembly
COMMENT	Define a new start-of-comment character
PAGE	Generate a page break in the listing
END	Terminate the assembly program

Figure 7-2. Some of the pseudoinstructions available in the MASM assembler (MASM).

The SEGMENT pseudoinstruction starts a new segment, and ENDS terminates one. It is allowed to start a text segment, with code, then start a data segment, then go back to the text segment, and so on.

ALIGN forces the next line, usually data, to an address that is a multiple of its argument. For example, if the current segment has 61 bytes of data already, then after ALIGN 4 the next address allocated will be 64.

EQU is used to give a symbolic name to an expression. For example, after the pseudoinstruction

```
BASE  EQU  1000
```

the symbol BASE can be used everywhere instead of 1000. The expression that follows the EQU can involve multiple defined symbols combined with arithmetic and other operators, as in

```
LIMIT  EQU  4 * BASE + 2000
```

Most assemblers, including MASM, require that a symbol be defined before it is used in an expression like this.

The next four pseudoinstructions, DB, DW, DD, and DQ, allocate storage for one or more variables of size 1, 2, 4, or 8 bytes, respectively. For example,

```
TABLE DB 11, 23, 49
```

allocates space for 3 bytes and initializes them to 11, 23, and 49, respectively. It also defines the symbol *TABLE* and sets it equal to the address where 11 is stored.

The PROC and ENDP pseudoinstructions define the beginning and end of assembly language procedures, respectively. Procedures in assembly language have the same function as procedures in other programming languages. Similarly, MACRO and ENDM delimit the scope of a macro definition. We will study macros later in this chapter.

The next two pseudoinstructions, PUBLIC and EXTERN, control the visibility of symbols. It is common to write programs as a collection of files. Frequently, a procedure in one file needs to call a procedure or access a data word defined in another file. To make this cross-file referencing possible, a symbol that is to be made available to other files is exported using PUBLIC. Similarly, to prevent the assembler from complaining about the use of a symbol that is not defined in the current file, the symbol can be declared as EXTERN, which tells the assembler that it will be defined in some other file. Symbols that are not declared in either of these pseudoinstructions have a scope of the local file. This default means that using, say, *FOO* in multiple files does not generate a conflict because each definition is local to its own file.

The INCLUDE pseudoinstruction causes the assembler to fetch another file and include it bodily into the current one. Such included files often contain definitions, macros, and other items needed in multiple files.

Many assemblers, support conditional assembly. For example,

```
WORDSIZE EQU 32
IF WORDSIZE GT 32
WSIZE:    DD   64
ELSE
WSIZE:    DD   32
ENDIF
```

allocates a single 32-bit word and calls its address *WSIZE*. The word is initialized to either 64 or 32, depending on the value of *WORDSIZE*, in this case, 32. Typically this construction would be used to write a program that could be assembled for either 32-bit mode or 64-bit mode. IF and ENDIF, then by changing a single definition, *WORDSIZE*, the program can automatically be set to assemble for either size. Using this approach, it is possible to maintain one source program for multiple (different) target machines, which makes software development and maintenance easier. In many cases, all the machine-dependent definitions, like *WORDSIZE*, are collected into a single file, with different versions for different machines. By including the right definitions file, the program can be easily recompiled for different machines.

The COMMENT pseudoinstruction allows the user to change the comment delimiter to something other than semicolon. PAGE is used to control the listing the assembler can produce, if requested. END marks the end of the program.

Many other pseudoinstructions exist in MASM. Other x86 assemblers have a different collection of pseudoinstructions available because they are dictated not by the underlying architecture, but by the taste of the assembler writer.

7.2 MACROS

Assembly language programmers frequently need to repeat sequences of instructions several times within a program. The most obvious way to do so is simply to write the required instructions wherever they are needed. If a sequence is long, however, or must be used a large number of times, writing it repeatedly becomes tedious.

An alternative approach is to make the sequence into a procedure and call it wherever it is needed. This strategy has the disadvantage of requiring a procedure call instruction and a return instruction to be executed every time a sequence is needed. If the sequences are short—for example, two instructions—but are used frequently, the procedure call overhead may significantly slow the program down. Macros provide an easy and efficient solution to the problem of repeatedly needing the same or nearly the same sequences of instructions.

7.2.1 Macro Definition, Call, and Expansion

A **macro definition** is a way to give a name to a piece of text. After a macro has been defined, the programmer can write the macro name instead of the piece of program. A macro is, in effect, an abbreviation for a piece of text. Figure 7-3(a) shows an assembly language program for the x86 that exchanges the contents of the variables *p* and *q* twice. These sequences could be defined as macros, as shown in Fig. 7-3(b). After its definition, every occurrence of *SWAP* causes it to be replaced by the four lines:

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

The programmer has defined *SWAP* as an abbreviation for the four statements shown above.

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX
```

(a)

```
SWAP  MACRO
      MOV EAX,P
      MOV EBX,Q
      MOV Q,EAX
      MOV P,EBX
      ENDM

      SWAP

      SWAP
```

(b)

Figure 7-3. Assembly language code for interchanging P and Q twice. (a) Without a macro. (b) With a macro.

Although different assemblers have slightly different notations for defining macros, all require the same basic parts in a macro definition:

1. A macro header giving the name of the macro being defined.
2. The text that is the body of the macro.
3. A pseudoinstruction marking the end of the definition (e.g., ENDM).

When the assembler encounters a macro definition, it saves it in a macro definition table for subsequent use. From that point on, whenever the name of the macro (*SWAP* in the example of Fig. 7-3) appears as an opcode, the assembler replaces it by the macro body. The use of a macro name as an opcode is known as a **macro call** and its replacement by the macro body is called **macro expansion**.

Macro expansion occurs during the assembly process and not during execution of the program. This point is important. The program of Fig. 7-3(a) and that of Fig. 7-3(b) will produce precisely the same machine language code. Looking only at the machine language program, it is impossible to tell whether or not any macros were involved in its generation. The reason is that once macro expansion has been completed the macro definitions are discarded by the assembler. No trace of them is left in the generated program.

Macro calls should not be confused with procedure calls. The basic difference is that a macro call is an instruction to the assembler to replace the macro name