Part Two

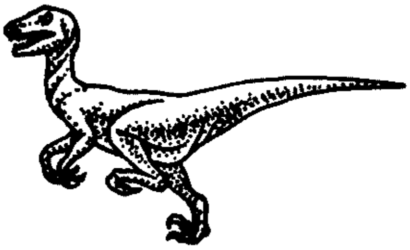
A *process* can be thought of as a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices—to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

A process is the unit of work in most systems. Systems consist of a collection of processes: Operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Although traditionally a process contained only a single *thread* of control as it ran, most modern operating systems now support processes that have multiple threads.

The operating system is responsible for the following activities in connection with process and thread management: the creation and deletion of both user and system processes, and the scheduling-of processes.

CHAPTER



Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern time-sharing system.

The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: operating-system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive.

3.1 [illegible]

A question that arises in discussing operating systems involves what to call all the CPU activities. A batch system executes *jobs*, whereas a time-shared system has *user programs*, or *tasks*. Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. Even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them *processes*.

The terms *job* and *process* are used almost interchangeably in this text. Although we personally prefer the term *process*, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was job processing. It would be misleading to avoid the use of commonly accepted terms that include the word *job* (such as *job scheduling*) simply because *process* has superseded *job*.

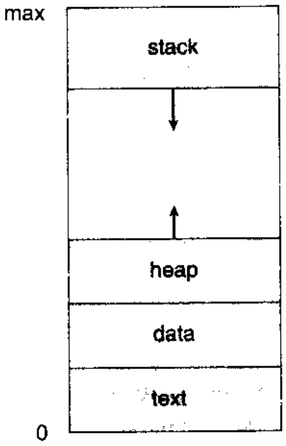


Figure 3.1 Process in memory.

3.1.1 The Process

Informally, as mentioned earlier, a process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure 3.1.

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an **executable file**), whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in `prog.exe` or `a.out`.)

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs. We discuss such matters in Section 3.4.

3.1.2 Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

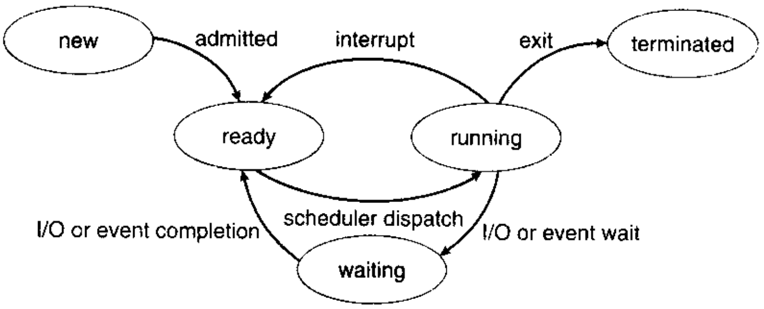


Figure 3.2 Diagram of process state.

New. The process is being created.

Running. Instructions are being executed.

Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

Ready. The process is waiting to be assigned to a processor.

Terminated. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*, however. The state diagram corresponding to these states is presented in Figure 3.2.

3.1.3 Process Control Block

Each process is represented in the operating system by a **process control block** (PCB)—also called a *task control block*. A PCB is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including these:

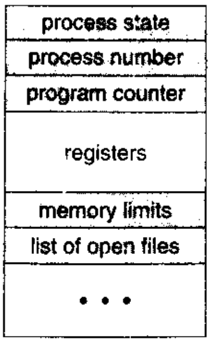


Figure 3.3 Process control block (PCB).

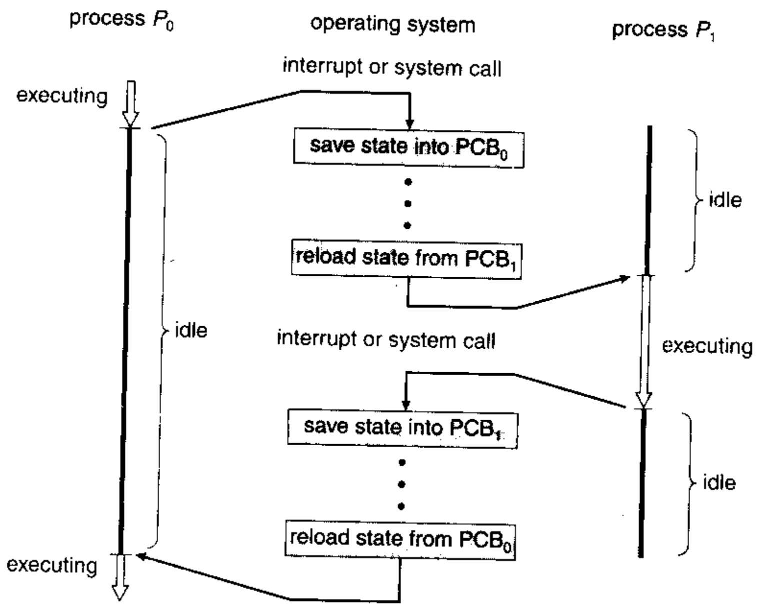


Figure 3.4 Diagram showing CPU switch from process to process.

Process state. The state may be new, ready, running, waiting, halted, and so on.

Program counter. The counter indicates the address of the next instruction to be executed for this process.

CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 3.4).

CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)

Memory-management information. This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 8).

Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

3.1.4 Threads

The process model discussed so far has implied that a process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. Chapter 4 explores multithreaded processes in detail.

3.2

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the **process scheduler** selects

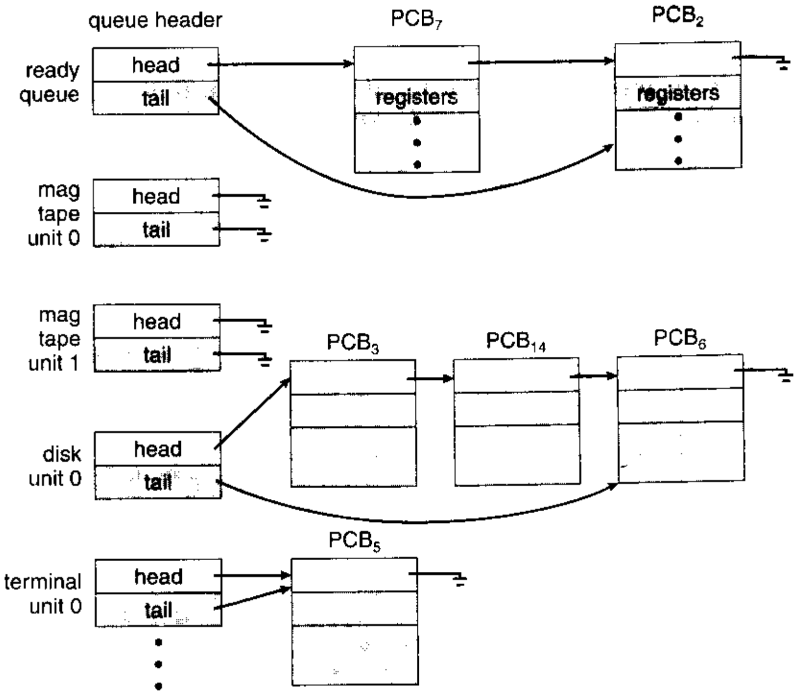


Figure 3.5 The ready queue and various I/O device queues.

an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

3.2.1 Scheduling Queues

As processes enter the system, they are put into a **job queue**, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue (Figure 3.5).

A common representation for a discussion of process scheduling is a **queueing diagram**, such as that in Figure 3.6. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.

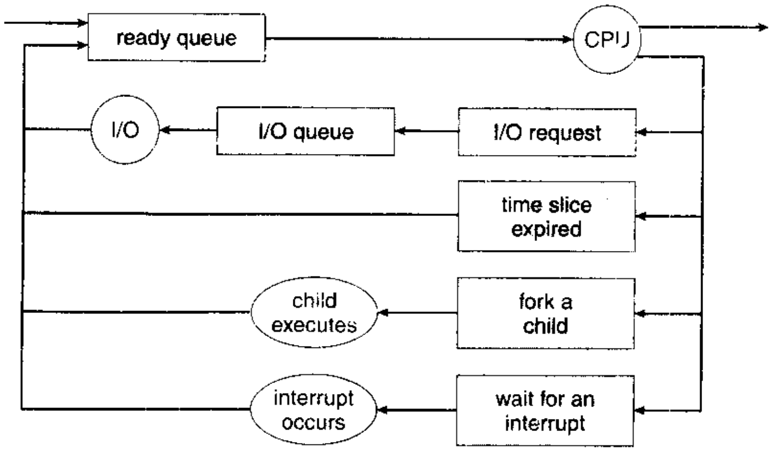


Figure 3.6 Queueing-diagram representation of process scheduling.

The process could create a new subprocess and wait for the subprocess's termination.

The process could be removed forcibly from the CPU as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

3.2.2 Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution. The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100 + 10) = 9 percent of the CPU is being used (wasted) simply for scheduling the work.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound. An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations. A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that the long-term scheduler select a good **process mix** of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the

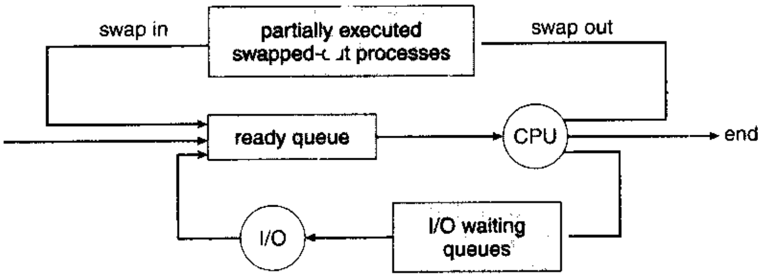


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

best performance will thus have a combination of CPU-bound and I/O-bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler. The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users. If the performance declines to unacceptable levels on a multiuser system, some users will simply quit.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in Figure 3.7. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. Swapping is discussed in Chapter 8.

3.2.3 Context Switch

As mentioned in 1.2.1, interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current **context** of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process; it includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a **state save** of the current state of the CPU, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds are a few milliseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the more work must be done during a context switch. As we will see in Chapter 8, advanced memory-management techniques may require extra data to be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.

3.3

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

3.3.1 Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. Figure 3.8 illustrates a typical process tree for the Solaris operating system, showing the name of each process and its pid. In Solaris, the process at the top of the tree is the `sched` process, with pid of 0. The `sched` process creates several children processes—including `pageout` and `fsflush`. These processes are responsible for managing memory and file systems. The `sched` process also creates the `init` process, which serves as the root parent process for all user processes. In Figure 3.8, we see two children of `init`—`inetd` and `dtlogin`. `inetd` is responsible for networking services such as `telnet` and `ftp`; `dtlogin` is the process representing a user login screen. When a user logs in, `dtlogin` creates an X-windows session (`Xsession`), which in turns creates the `sdt_shel` process. Below `sdt_shel`, a user's command-line shell—the C-shell or `csh`—is created. It is this command-line interface where the user then invokes various child processes, such as the `ls` and `cat` commands. We also see a `csh` process with pid of 7778 representing a user who has logged onto the system using `telnet`. This user has started the Netscape browser (pid of 7785) and the emacs editor (pid of 8105).

On UNIX, a listing of processes can be obtained using the ps command. For example, entering the command `ps -el` will list complete information for all processes currently active in the system. It is easy to construct a process tree similar to what is shown in Figure 3.8 by recursively tracing parent processes all the way to the `init` process.

In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process. For example, consider a process whose function is to display the contents of a file—say, *img.jpg*—on the screen of a terminal. When it is created, it will get, as an input from its parent process, the name of the file *img.jpg*, and it will use that file name, open the file, and write the contents out. It may also get the name of the output device. Some

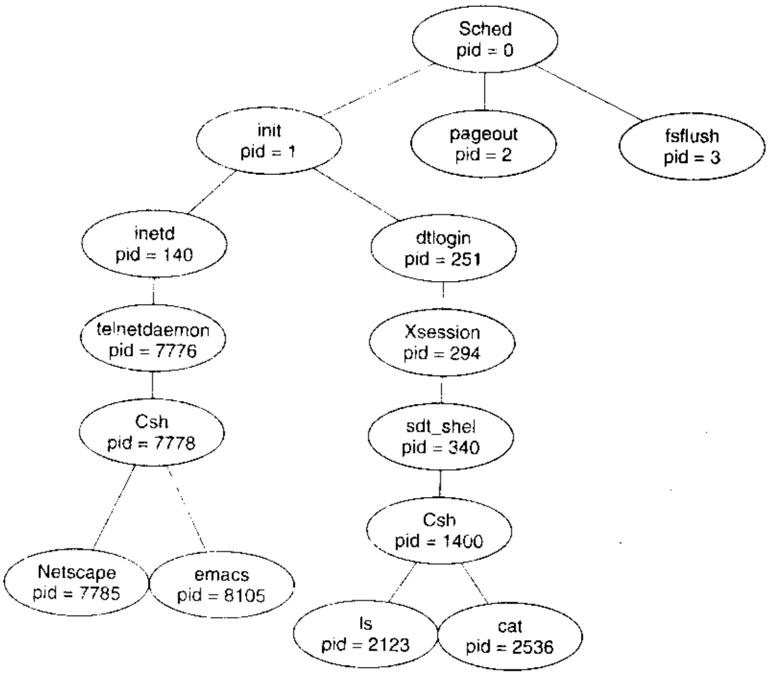


Figure 3.8 A tree of processes on a typical Solaris system.

operating systems pass resources to child processes. On such a system, the new process may get two open files, *img.jpg* and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities exist in terms of execution:

The parent continues to execute concurrently with its children.

The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

The child process is a duplicate of the parent process (it has the same program and data as the parent).

The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) {/* error occurred */
     fprintf(stderr, "Fork Failed");
     exit(-1);
   }
   else if (pid == 0) {/* child process */
     execlp("/bin/ls","ls",NULL);
   }
   else {/* parent process */
     /* parent will wait for the child to complete */
     wait(NULL);
     printf("Child Complete");
     exit(0);
   }
}
```

Figure 3.9 C program forking a separate process.

its child process. Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: The return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the `exec()` system call is used after a `fork()` system call by one of the two processes to replace the process's memory space with a new program. The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child.

The C program shown in Figure 3.9 illustrates the UNIX system calls previously described. We now have two different processes running a copy of the same program. The value of pid for the child process is zero; that for the parent is an integer value greater than zero. The child process overlays its address space with the UNIX command `/bin/ls` (used to get a directory listing) using the `execlp()` system call (`execlp()` is a version of the `exec()` system call). The parent waits for the child process to complete with the `wait()` system call. When the child process completes (by either implicitly or explicitly invoking `exit()`) the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call. This is also illustrated in Figure 3.10.

As an alternative example, we next consider process creation in Windows. Processes are created in the Win32 API using the `CreateProcess()` function, which is similar to `fork()` in that a parent creates a new child process. However, whereas `fork()` has the child process inheriting the address space of its parent, `CreateProcess()` requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas `fork()` is passed no parameters, `CreateProcess()` expects no fewer than ten parameters.

The C program shown in Figure 3.11 illustrates the `CreateProcess()` function, which creates a child process that loads the application `mspaint.exe`. We opt for many of the default values of the ten parameters passed to `CreateProcess()`. Readers interested in pursuing the details on process creation and management in the Win32 API are encouraged to consult the bibliographical notes at the end of this chapter.

Two parameters passed to `CreateProcess()` are instances of the STARTUPINFO and PROCESS_INFORMATION structures. STARTUPINFO specifies many properties of the new process, such as window size and appearance and han-

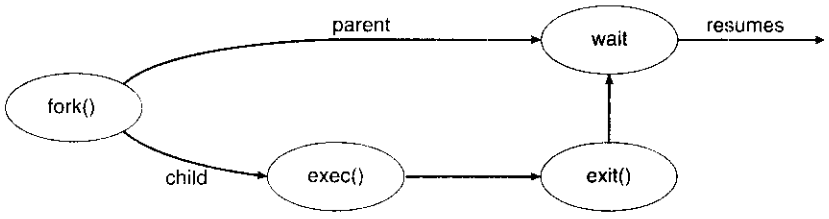


Figure 3.10 Process creation.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
     "C:\\WINDOWS\\system32\\mspaint.exe", // command line
     NULL, // don't inherit process handle
     NULL, // don't inherit thread handle
     FALSE, // disable handle inheritance
     0, // no creation flags
     NULL, // use parent's environment block
     NULL, // use parent's existing directory
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figure 3.11 Creating a separate process using the Win32 API.

dles to standard input and output files. The PROCESS_INFORMATION structure contains a handle and the identifiers to the newly created process and its thread. We invoke the `ZeroMemory()` function to allocate memory for each of these structures before proceeding with `CreateProcess()`.

The first two parameters passed to `CreateProcess()` are the application name and command line parameters. If the application name is NULL (which in this case it is), the command line parameter specifies the application to load. In this instance we are loading the Microsoft Windows *mspaint.exe* application. Beyond these two initial parameters, we use the default parameters for inheriting process and thread handles as well as specifying no creation flags. We also use the parent's existing environment block and starting directory.

Last, we provide two pointers to the STARTUPINFO and PROCESS_INFORMATION structures created at the beginning of the program. In Figure 3.9, the parent process waits for the child to complete by invoking the `wait()` system call. The equivalent of this in Win32 is `WaitForSingleObject()`, which is passed a handle of the child process—`pi.hProcess`— that it is waiting for to complete. Once the child process exits, control returns from the `WaitForSingleObject()` function in the parent process.

3.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, `TerminateProcess()` in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems, including VMS, do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in UNIX, we can terminate a process by using the `exit()` system call; its parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated. If the parent terminates, however, all its children have assigned as their new parent the `init` process. Thus, the children still have a parent to collect their status and execution statistics.

3.4 Interprocess Communication

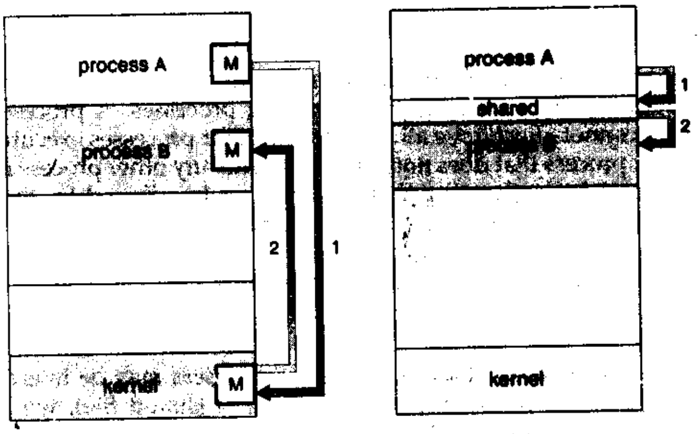
Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is **independent** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads, as we discussed in Chapter 2.
- **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **interprocess communication** (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: (1) **shared memory** and (2) **message passing**. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.12.

Both of the models just discussed are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In contrast, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required. In the remainder of this section, we explore each of these IPC models in more detail.



(a) (b)

Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

3.4.1 Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. The producer–consumer problem also provides a useful metaphor for the client–server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Let's look more closely at how the bounded buffer can be used to enable processes to share memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
   . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The buffer is empty when `in == out`; the buffer is full when ((`in` + 1) % BUFFER_SIZE) == `out`.

The code for the producer and consumer processes is shown in Figures 3.13 and 3.14, respectively. The producer process has a local variable `nextProduced` in which the new item to be produced is stored. The consumer process has a local variable `nextConsumed` in which the item to be consumed is stored.

This scheme allows at most BUFFER_SIZE − 1 items in the buffer at the same time. We leave it as an exercise for you to provide a solution where BUFFER_SIZE items can be in the buffer at the same time. In Section 3.5.1, we illustrate the POSIX API for shared memory.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently. In Chapter 6, we discuss how synchronization among cooperating processes can be implemented effectively in a shared-memory environment.

```
item nextProduced;

while (true) {
     /* produce an item in nextProduced */
     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.13 The producer process.

```
item nextConsumed;

while (true) {
     while (in == out)
        ; // do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

Figure 3.14 The consumer process.

3.4.2 Message-Passing Systems

In Section 3.4.1, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, a chat program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations: `send(message)` and `receive(message)`. Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.

If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in Chapter 14) but rather with its logical implementation. Here are several methods for logically implementing a link and the `send()`/`receive()` operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

We look at issues related to each of these features next.

3.4.2.1 Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the `send()` and `receive()` primitives are defined as:

- `send(P, message)`—Send a `message` to process `P`.
- `receive(Q, message)`—Receive a `message` from process `Q`.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits *symmetry* in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs *asymmetry* in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the `send()` and `receive()` primitives are defined as follows:

- `send(P, message)`—Send a `message` to process `P`.
- `receive(id, message)`—Receive a `message` from any process; the variable *id* is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such **hard-coding** techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

With **indirect communication**, the messages are sent to and received from **mailboxes**, or **ports**. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox, however. The `send()` and `receive()` primitives are defined as follows:

- `send(A, message)`—Send a `message` to mailbox `A`.
- `receive(A, message)`—Receive a `message` from mailbox `A`.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Now suppose that processes P_1, P_2, and P_3 all share mailbox A. Process P_1 sends a message to A, while both P_2 and P_3 execute a `receive()` from A. Which process will receive the message sent by P_1? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a `receive()` operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P_2 or P_3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, *round robin* where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

3.4.2.2 Synchronization

Communication between processes takes place through calls to `send()` and `receive()` primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**.

- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

Different combinations of `send()` and `receive()` are possible. When both `send()` and `receive()` are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer–consumer problem becomes trivial when we use blocking `send()` and `receive()` statements. The producer merely invokes the blocking `send()` call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes `receive()`, it blocks until a message is available.

Note that the concepts of synchronous and asynchronous occur frequently in operating-system I/O algorithms, as you will see throughout this text.

3.4.2.3 Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**. The queue has finite length *n*; thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The links capacity is finite , however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**. The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

3.5 Examples of IPC Systems

In this section, we explore three different IPC systems. We first cover the POSIX API for shared memory and then discuss message passing in the Mach operating system. We conclude with Windows XP, which interestingly uses shared memory as a mechanism for providing certain types of message passing.

3.5.1 An Example: POSIX Shared Memory

Several IPC mechanisms are available for POSIX systems, including shared memory and message passing. Here, we explore the POSIX API for shared memory.

A process must first create a shared memory segment using the `shmget()` system call (`shmget()` is derived from `SHared Memory GET`). The following example illustrates the use of `shmget()`:

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

This first parameter specifies the key (or identifier) of the shared-memory segment. If this is set to IPC_PRIVATE, a new shared-memory segment is created. The second parameter specifies the size (in bytes) of the shared memory segment. Finally, the third parameter identifies the mode, which indicates how the shared-memory segment is to be used—that is, for reading, writing, or both. By setting the mode to S_IRUSR | S_IWUSR, we are indicating that the owner may read or write to the shared memory segment. A successful call to `shmget()` returns an integer identifier for the shared-memory segment. Other processes that want to use this region of shared memory must specify this identifier.

Processes that wish to access a shared-memory segment must attach it to their address space using the `shmat()` (`SHared Memory ATtach`) system call. The call to `shmat()` expects three parameters as well. The first is the integer identifier of the shared-memory segment being attached, and the second is a pointer location in memory indicating where the shared memory will be attached. If we pass a value of NULL, the operating system selects the location on the user's behalf. The third parameter identifies a flag that allows the shared-memory region to be attached in read-only or read-write mode; by passing a parameter of 0, we allow both reads and writes to the shared region.

The third parameter identifies a mode flag. If set, the mode flag allows the shared-memory region to be attached in read-only mode; if set to 0, the flag allows both reads and writes to the shared region. We attach a region of shared memory using `shmat()` as follows:

```
shared_memory = (char *) shmat(id, NULL, 0);
```

If successful, `shmat()` returns a pointer to the beginning location in memory where the shared-memory region has been attached.

Once the region of shared memory is attached to a process's address space, the process can access the shared memory as a routine memory access using the pointer returned from `shmat()`. In this example, `shmat()` returns a pointer to a character string. Thus, we could write to the shared-memory region as follows:

```
sprintf(shared_memory, "Writing to shared memory");
```

Other processes sharing this segment would see the updates to the shared-memory segment.

Typically, a process using an existing shared-memory segment first attaches the shared-memory region to its address space and then accesses (and possibly updates) the region of shared memory. When a process no longer requires access to the shared-memory segment, it detaches the segment from its address space. To detach a region of shared memory, the process can pass the pointer of the shared-memory region to the `shmdt()` system call, as follows:

```
shmdt(shared_memory);
```

Finally, a shared-memory segment can be removed from the system with the `shmctl()` system call, which is passed the identifier of the shared segment along with the flag IPC_RMID.

The program shown in Figure 3.15 illustrates the POSIX shared-memory API discussed above. This program creates a 4,096-byte shared-memory segment. Once the region of shared memory is attached, the process writes the message `Hi There!` to shared memory. After outputting the contents of the updated memory, it detaches and removes the shared-memory region. We provide further exercises using the POSIX shared memory API in the programming exercises at the end of this chapter.

3.5.2 An Example: Mach

As an example of a message-based operating system, we next consider the Mach operating system, developed at Carnegie Mellon University. We introduced Mach in Chapter 2 as part of the Mac OS X operating system. The Mach kernel supports the creation and destruction of multiple tasks, which are similar to processes but have multiple threads of control. Most communication in Mach—including most of the system calls and all intertask information—is carried out by *messages*. Messages are sent to and received from mailboxes, called *ports* in Mach.

Even system calls are made by messages. When a task is created, two special mailboxes—the Kernel mailbox and the Notify mailbox—are also created. The Kernel mailbox is used by the kernel to communicate with the task. The kernel sends notification of event occurrences to the Notify port. Only three system calls are needed for message transfer. The `msg_send()` call sends a message to a mailbox. A message is received via `msg_receive()`. Remote procedure calls (RPCs) are executed via `msg_rpc()`, which sends a message and waits for exactly one return message from the sender. In this way, the RPC models a typical subroutine procedure call but can work between systems—hence the term *remote*.

The `port_allocate()` system call creates a new mailbox and allocates space for its queue of messages. The maximum size of the message queue defaults to eight messages. The task that creates the mailbox is that mailbox's

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the identifier for the shared memory segment */
int segment_id;
/* a pointer to the shared memory segment */
char* shared_memory;
/* the size (in bytes) of the shared memory segment */
const int size = 4096;

   /* allocate a shared memory segment */
   segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

   /* attach the shared memory segment */
   shared_memory = (char *) shmat(segment_id, NULL, 0);

   /* write a message to the shared memory segment */
   sprintf(shared_memory, "Hi there!");

   /* now print out the string from shared memory */
   printf("*%s\n", shared_memory);

   /* now detach the shared memory segment */
   shmdt(shared_memory);

   /* now remove the shared memory segment */
   shmctl(segment_id, IPC_RMID, NULL);

   return 0;
}
```

Figure 3.15 C program illustrating POSIX shared-memory API.

owner. The owner is also allowed to receive from the mailbox. Only one task at a time can either own or receive from a mailbox, but these rights can be sent to other tasks if desired.

The mailbox has an initially empty queue of messages. As messages are sent to the mailbox, the messages are copied into the mailbox. All messages have the same priority. Mach guarantees that multiple messages from the same sender are queued in first-in, first-out (FIFO) order but does not guarantee an absolute ordering. For instance, messages from two senders may be queued in any order.

The messages themselves consist of a fixed-length header followed by a variable-length data portion. The header indicates the length of the message and includes two mailbox names. One mailbox name is the mailbox to which the message is being sent. Commonly, the sending thread expects a reply; so the mailbox name of the sender is passed on to the receiving task, which can use it as a "return address."

The variable part of a message is a list of typed data items. Each entry in the list has a type, size, and value. The type of the objects specified in the message is important, since objects defined by the operating system—such as ownership or receive access rights, task states, and memory segments—may be sent in messages.

The send and receive operations themselves are flexible. For instance, when a message is sent to a mailbox, the mailbox may be full. If the mailbox is not full, the message is copied to the mailbox, and the sending thread continues. If the mailbox is full, the sending thread has four options:

1. Wait indefinitely until there is room in the mailbox.
2. Wait at most *n* milliseconds.
3. Do not wait at all but rather return immediately.
4. Temporarily cache a message. One message can be given to the operating system to keep, even though the mailbox to which it is being sent is full. When the message can be put in the mailbox, a message is sent back to the sender; only one such message to a full mailbox can be pending at any time for a given sending thread.

The final option is meant for server tasks, such as a line-printer driver. After finishing a request, such tasks may need to send a one-time reply to the task that had requested service; but they must also continue with other service requests, even if the reply mailbox for a client is full.

The receive operation must specify the mailbox or mailbox set from which a message is to be received. A **mailbox set** is a collection of mailboxes, as declared by the task, which can be grouped together and treated as one mailbox for the purposes of the task. Threads in a task can receive only from a mailbox or mailbox set for which the task has receive access. A `port_status()` system call returns the number of messages in a given mailbox. The receive operation attempts to receive from (1) any mailbox in a mailbox set or (2) a specific (named) mailbox. If no message is waiting to be received, the receiving thread can either wait at most *n* milliseconds or not wait at all.

The Mach system was especially designed for distributed systems, which we discuss in Chapters 14 through 16, but Mach is also suitable for single-processor systems, as evidenced by its inclusion in the Mac OS X system. The major problem with message systems has generally been poor performance caused by double copying of messages; the message is copied first from the sender to the mailbox and then from the mailbox to the receiver. The Mach message system attempts to avoid double-copy operations by using virtual-memory-management techniques (Chapter 9). Essentially, Mach maps the address space containing the sender's message into the receiver's address space. The message itself is never actually copied. This message-management technique provides a large performance boost but works for only intrasystem messages. The Mach operating system is discussed in an extra chapter posted on our website.

3.5.3 An Example: Windows XP

The Windows XP operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows XP provides support for multiple operating environments, or *subsystems*, with which application programs communicate via a message-passing mechanism. The application programs can be considered clients of the Windows XP subsystem server.

The message-passing facility in Windows XP is called the **local procedure-call** (LPC) facility. The LPC in Windows XP communicates between two processes on the same machine. It is similar to the standard RPC mechanism that is widely used, but it is optimized for and specific to Windows XP. Like Mach, Windows XP uses a port object to establish and maintain a connection between two processes. Every client that calls a subsystem needs a communication channel, which is provided by a port object and is never inherited. Windows XP uses two types of ports: connection ports and communication ports. They are really the same but are given different names according to how they are used. Connection ports are named *objects* and are visible to all processes; they give applications a way to set up communication channels (Chapter 22). The communication works as follows:

- The client opens a handle to the subsystem's connection port object.
- The client sends a connection request.
- The server creates two private communication ports and returns the handle to one of them to the client.
- The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Windows XP uses two types of message-passing techniques over a port that the client specifies when it establishes the channel. The simplest, which is used for small messages, uses the port's message queue as intermediate storage and copies the message from one process to the other. Under this method, messages of up to 256 bytes can be sent.

If a client needs to send a larger message, it passes the message through a **section object**, which sets up a region of shared memory. The client has to decide when it sets up the channel whether or not it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Similarly, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about the section object. This method is more complicated than the first method, but it avoids data copying. In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request. The callback mechanism allows them to perform asynchronous message handling. The structure of local procedure calls in Windows XP is shown in Figure 3.16.

It is important to note that the LPC facility in Windows XP is not part of the Win32 API and hence is not visible to the application programmer. Rather, applications using the Win32 API invoke standard remote procedure calls. When the RPC is being invoked on a process on the same system, the RPC is

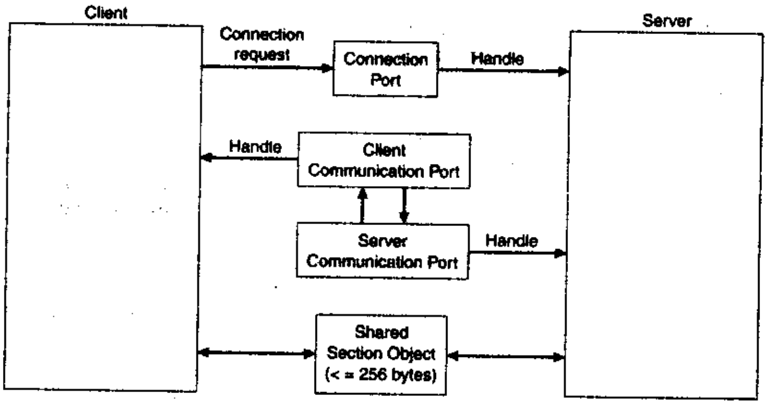


Figure 3.16 Local procedure calls in Windows XP.

indirectly handled through a local procedure call. LPCs are also used in a few other functions that are part of the Win32 API.

3.6 Communication in Client-Server Systems

In Section 3.4, we described how processes can communicate using shared memory and message passing. These techniques can be used for communication in client–server systems (Section 1.12.2) as well. In this section, we explore three other strategies for communication in client–server systems: sockets, remote procedure calls (RPCs), and Java's remote method invocation (RMI).

3.6.1 Sockets

A **socket** is defined as an endpoint for communication. A pair of processes communicating over a network employ a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services (such as telnet, ftp, and http) listen to well-known ports (a telnet server listens to port 23, an ftp server listens to port 21, and a web, or http, server listens to port 80). All ports below 1024 are considered *well known*; we can use them to implement standard services.

When a client process initiates a request for a connection, it is assigned a port by the host computer. This port is some arbitrary number greater than 1024. For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 3.17. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

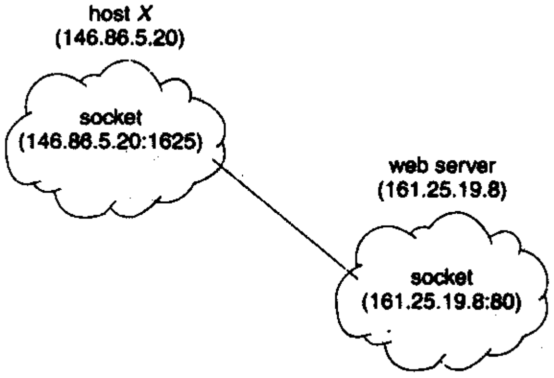


Figure 3.17 Communication using sockets.

All connections must be unique. Therefore, if another process also on host X wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Although most program examples in this text use C, we will illustrate sockets using Java, as it provides a much easier interface to sockets and has a rich library for networking utilities. Those interested in socket programming in C or C++ should consult the bibliographical notes at the end of the chapter.

Java provides three different types of sockets. **Connection-oriented (TCP) sockets** are implemented with the `Socket` class. **Connectionless (UDP) sockets** use the `DatagramSocket` class. Finally, the `MulticastSocket` class is a subclass of the `DatagramSocket` class. A multicast socket allows data to be sent to multiple recipients.

Our example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from the server. The server listens to port 6013, although the port could have any arbitrary number greater than 1024. When a connection is received, the server returns the date and time to the client.

The date server is shown in Figure 3.18. The server creates a `ServerSocket` that specifies it will listen to port 6013. The server then begins listening to the port with the `accept()` method. The server blocks on the `accept()` method waiting for a client to request a connection. When a connection request is received, `accept()` returns a socket that the server can use to communicate with the client.

The details of how the server communicates with the socket are as follows. The server first establishes a `PrintWriter` object that it will use to communicate with the client. A `PrintWriter` object allows the server to write to the socket using the routine `print()` and `println()` methods for output. The server process sends the date to the client, calling the method `println()`. Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.

A client communicates with the server by creating a socket and connecting to the port on which the server is listening. We implement such a client in the

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         // now listen for connections
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            // write the Date to the socket
            pout.println(new java.util.Date().toString());

            // close the socket and resume
            // listening for connections
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Figure 3.18 Date server.

Java program shown in Figure 3.19. The client creates a **Socket** and requests a connection with the server at IP address 127.0.0.1 on port 6013. Once the connection is made, the client can read from the socket using normal stream I/O statements. After it has received the date from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the **loopback**. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the date server. In addition to an IP address, an actual host name, such as *www.westminstercollege.edu*, can be used as well.

Communication using sockets—although common and efficient—is considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data. In the next two subsections, we look at two higher-level methods of communication: remote procedure calls (RPCs) and remote method invocation (RMI).

```
import java.net.*;
import java.io.*;

public class DateClient
{
  public static void main(String[] args) {
    try {
      //make connection to server socket
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      // read the date from the socket
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      // close the socket connection
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Figure 3.19 Date client.

3.6.2 Remote Procedure Calls

One of the most common forms of remote service is the RPC paradigm, which we discussed briefly in Section 3.5.2. The RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. It is similar in many respects to the IPC mechanism described in Section 3.4, and it is usually built on top of such a system. Here, however, because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service. In contrast to the IPC facility, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data. Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

A *port* is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses a message to the proper port. For instance, if a system wished to allow other systems to be able to list its current users, it would have a daemon supporting such an RPC attached to a port—say, port 3027. Any remote system could obtain the needed information (that is, the list of current users) by sending an RPC message to port 3027 on the server; the data would be received in a reply message.

The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the details that allow communication to take place by providing a **stub** on the client side. Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and *marshals* the parameters. Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.

One issue that must be dealt with concerns differences in data representation on the client and server machines. Consider the representation of 32-bit integers. Some systems (known as *big-endian*) use the high memory address to store the most significant byte, while other systems (known as *little-endian*) store the least significant byte at the high memory address. To resolve differences like this, many RPC systems define a machine-independent representation of data. One such representation is known as **external data representation (XDR)**. On the client side, parameter marshalling involves converting the machine-dependent data into XDR before they are sent to the server. On the server side, the XDR data are unmarshalled and converted to the machine-dependent representation for the server.

Another important issue involves the semantics of a call. Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors. One way to address this problem is for the operating system to ensure that messages are acted on *exactly once*, rather than *at most once*. Most local procedure calls have the "exactly once" functionality, but it is more difficult to implement.

First, consider "at most once". This semantic can be assured by attaching a timestamp to each message. The server must keep a history of all the timestamps of messages it has already processed or a history large enough to ensure that repeated messages are detected. Incoming messages that have a timestamp already in the history are ignored. The client can then send a message one or more times and be assured that it only executes once. (Generation of these timestamps is discussed in Section 16.1.)

For "exactly once," we need to remove the risk that the server never receives the request. To accomplish this, the server must implement the "at most once" protocol described above but must also acknowledge to the client that the RPC call was received and executed. These ACK messages are common throughout networking. The client must resend each RPC call periodically until it receives the ACK for that call.

Another important issue concerns the communication between a server and a client. With standard procedure calls, some form of binding takes place during link, load, or execution time (Chapter 8) so that a procedure call's name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other because they do not share memory.

Two approaches are common. First, the binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a **matchmaker**) daemon on a fixed RPC port. A client then sends a message containing the name of the RPC to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls can be sent

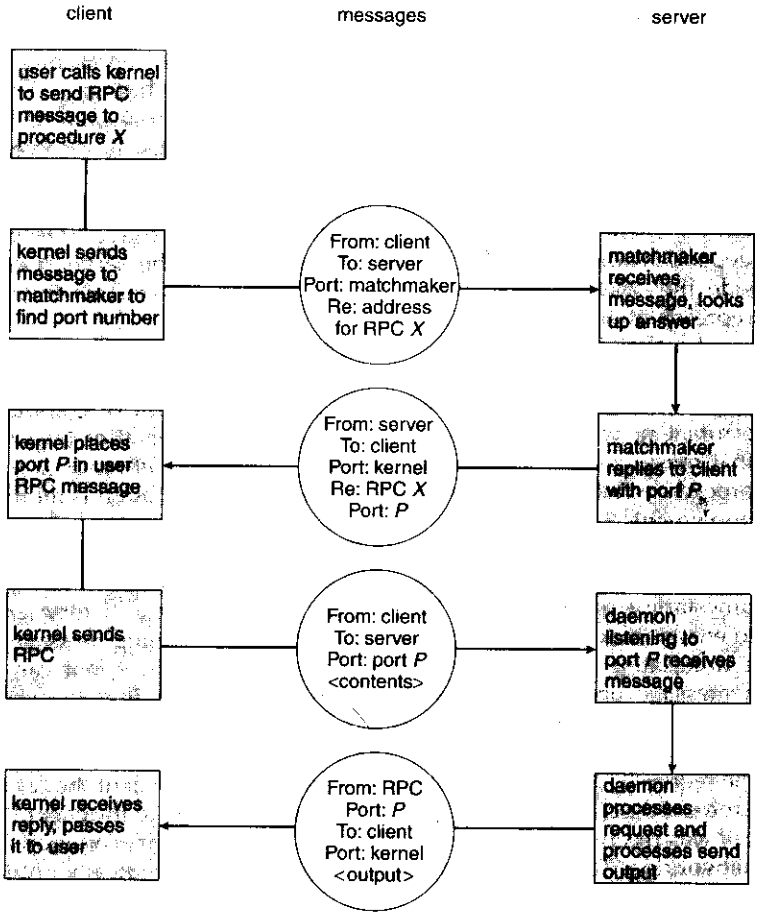


Figure 3.20 Execution of a remote procedure call (RPC).

to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request but is more flexible than the first approach. Figure 3.20 shows a sample interaction.

The RPC scheme is useful in implementing a distributed file system (Chapter 15). Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the distributed file system port on a server on which a file operation is to take place. The message contains the disk operation to be performed. The disk operation might be `read`, `write`, `rename`, `delete`, or `status`, corresponding to the usual file-related system calls. The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client. For instance, a message might contain a request to transfer a whole file to a client or be limited to a simple block request. In the latter case, several such requests may be needed if a whole file is to be transferred.

3.6.3 Remote Method Invocation

Remote method invocation (RMI) is a Java feature similar to RPCs. RMI allows a thread to invoke a method on a remote object. Objects are considered remote if they reside in a different Java virtual machine (JVM). Therefore, the remote object may be in a different JVM on the same computer or on a remote host connected by a network. This situation is illustrated in Figure 3.21.

RMI and RPCs differ in two fundamental ways. First, RPCs support procedural programming, whereby only remote *procedures* or *functions* can be called. In contrast, RMI is object-based: It supports invocation of *methods* on remote objects. Second, the parameters to remote procedures are ordinary data structures in RPC; with RMI, it is possible to pass objects as parameters to remote methods. By allowing a Java program to invoke methods on remote objects, RMI makes it possible for users to develop Java applications that are distributed across a network.

To make remote methods transparent to both the client and the server, RMI implements the remote object using stubs and skeletons. A **stub** is a proxy for the remote object; it resides with the client. When a client invokes a remote method, the stub for the remote object is called. This client-side stub is responsible for creating a **parcel** consisting of the name of the method to be invoked on the server and the marshalled parameters for the method. The stub then sends this parcel to the server, where the skeleton for the remote object receives it. The **skeleton** is responsible for unmarshalling the parameters and

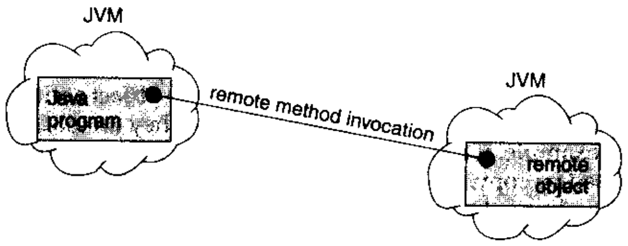


Figure 3.21 Remote method invocation.

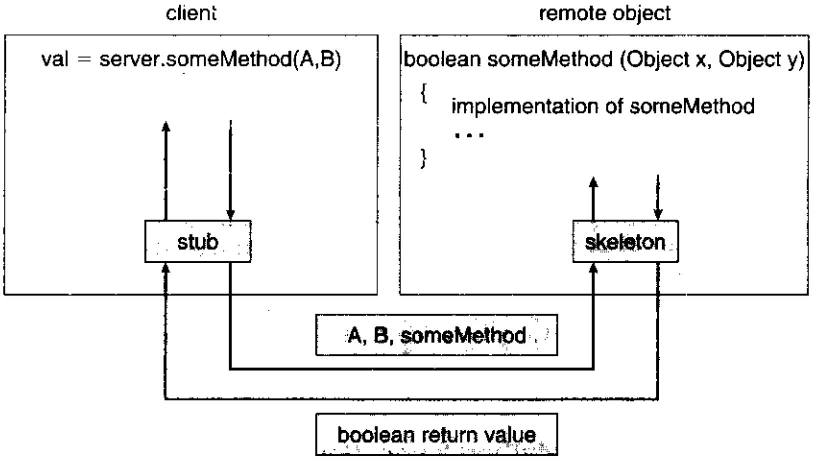


Figure 3.22 Marshalling parameters.

invoking the desired method on the server. The skeleton then marshals the return value (or exception, if any) into a parcel and returns this parcel to the client. The stub unmarshals the return value and passes it to the client.

Lets look more closely at how this process works. Assume that a client wishes to invoke a method on a remote object `server` with a signature `someMethod(Object, Object)` that returns a `boolean` value. The client executes the statement

```
boolean val = server.someMethod(A, B);
```

The call to `someMethod()` with the parameters `A` and `B` invokes the stub for the remote object. The stub marshals into a parcel the parameters `A` and `B` and the name of the method that is to be invoked on the server, then sends this parcel to the server. The skeleton on the server unmarshals the parameters and invokes the method `someMethod()`. The actual implementation of `someMethod()` resides on the server. Once the method is completed, the skeleton marshals the `boolean` value returned from `someMethod()` and sends this value back to the client. The stub unmarshals this return value and passes it to the client. The process is shown in Figure 3.22.

Fortunately, the level of abstraction that RMI provides makes the stubs and skeletons transparent, allowing Java developers to write programs that invoke distributed methods just as they would invoke local methods. It is crucial, however, to understand a few rules about the behavior of parameter passing.

- If the marshalled parameters are **local** (or **nonremote**) objects, they are passed by copy using a technique known as **object serialization**. However, if the parameters are also remote objects, they are passed by reference. In our example, if `A` is a local object and `B` a remote object, `A` is serialized and passed by copy, and `B` is passed by reference. This in turn allows the server to invoke methods on `B` remotely.
- If local objects are to be passed as parameters to remote objects, they must implement the interface `java.io.Serializable`. Many objects in the core

Java API implement `Serializable`, allowing them to be used with RMI. Object serialization allows the state of an object to be written to a byte stream.

3.7

A process is a program in execution. As a process executes, it changes state. The state of a process is defined by that process's current activity. Each process may be in one of the following states: new, ready, running, waiting, or terminated. Each process is represented in the operating system by its own process-control block (PCB).

A process, when it is not executing, is placed in some waiting queue. There are two major classes of queues in an operating system: I/O request queues and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Each process is represented by a PCB, and the PCBs can be linked together to form a ready queue. Long-term (job) scheduling is the selection of processes that will be allowed to contend for the CPU. Normally, long-term scheduling is heavily influenced by resource-allocation considerations, especially memory management. Short-term (CPU) scheduling is the selection of one process from the ready queue.

Operating systems must provide a mechanism for parent processes to create new child processes. The parent may wait for its children to terminate before proceeding, or the parent and children may execute concurrently. There are several reasons for allowing concurrent execution: information sharing, computation speedup, modularity, and convenience.

The processes executing in the operating system may be either independent processes or cooperating processes. Cooperating processes require an interprocess communication mechanism to communicate with each other. Principally, communication is achieved through two schemes: shared memory and message passing. The shared-memory method requires communicating processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory system, the responsibility for providing communication rests with the application programmers; the operating system needs to provide only the shared memory. The message-passing method allows the processes to exchange messages. The responsibility for providing communication may rest with the operating system itself. These two schemes are not mutually exclusive and can be used simultaneously within a single operating system.

Communication in client–server systems may use (1) sockets, (2) remote procedure calls (RPCs), or (3) Java's remote method invocation (RMI). A socket is defined as an endpoint for communication. A connection between a pair of applications consists of a pair of sockets, one at each end of the communication channel. RPCs are another form of distributed communication. An RPC occurs when a process (or thread) calls a procedure on a remote application. RMI is the Java version of RPCs. RMI allows a thread to invoke a method on a remote object just as it would invoke a method on a local object. The primary distinction between RPCs and RMI is that in RPCs data are passed to a remote procedure in the form of an ordinary data structure, whereas RMI allows objects to be passed in remote method calls.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

    pid = fork();

    if (pid == 0) {/* child process */
        value += 15;
    }
    else if (pid > 0) {/* parent process */
        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE A */
        exit(0);
    }
}
```

Figure 3.23 C program.

Exercises

3.1 Describe the differences among short-term, medium-term, and long-term scheduling.

3.2 Describe the actions taken by a kernel to context-switch between processes.

3.3 Using the program shown in Figure 3.23, explain what will be output at `Line A`.

3.4 What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.

  a. Synchronous and asynchronous communication

  b. Automatic and explicit buffering

  c. Send by copy and send by reference

  d. Fixed-sized and variable-sized messages

3.5 The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_n = fib_{n-1} + fib_{n-2}$$

Write a C program using the `fork()` system call that that generates the Fibonacci sequence in the child process. The number of the sequence will be provided in the command line. For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child process. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the `wait()` call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

3.6 Repeat the preceding exercise, this time using the `CreateProcess()` in the Win32 API. In this instance, you will need to specify a separate program to be invoked from `CreateProcess()`. It is this separate program that will run as a child process outputting the Fibonacci sequence. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

3.7 Modify the date server shown in Figure 3.18 so that it delivers random fortunes rather than the current date. Allow the fortunes to contain multiple lines. The date client shown in Figure 3.19 can be used to read the multi-line fortunes returned by the fortune server.

3.8 In Exercise 3.5, the child process must output the Fibonacci sequence, since the parent and child have their own copies of the data. Another approach to designing this program is to establish a shared-memory segment between the parent and child processes. This technique allows the child to write the contents of the Fibonacci sequence to the shared-memory segment and has the parent output the sequence when the child completes. Because the memory is shared, any changes the child makes to the shared memory will be reflected in the parent process as well.

This program will be structured using POSIX shared memory as described in Section 3.5.1. The program first requires creating the data structure for the shared-memory segment. This is most easily accomplished using a `struct`. This data structure will contain two items: (1) a fixed-sized array of size `MAX_SEQUENCE` that will hold the Fibonacci values; and (2) the size of the sequence the child process is to generate —`sequence_size` where `sequence_size` $\leq$ `MAX_SEQUENCE`. These items can be represented in a `struct` as follows:

```
#define MAX_SEQUENCE 10

typedef struct {
   long fib_sequence[MAX_SEQUENCE];
   int sequence_size;
}shared_data;
```

The parent process will progress through the following steps:

a. Accept the parameter passed on the command line and perform error checking to ensure that the parameter is $\leq$ MAX_SEQUENCE.

b. Create a shared-memory segment of size `shared_data`.

c. Attach the shared-memory segment to its address space.

d. Set the value of `sequence_size` to the parameter on the command line.

e. Fork the child process and invoke the `wait()` system call to wait for the child to finish.

f. Output the value of the Fibonacci sequence in the shared-memory segment.

g. Detach and remove the shared-memory segment.

Because the child process is a copy of the parent, the shared-memory region will be attached to the child's address space as well. The child process will then write the Fibonacci sequence to shared memory and finally will detach the segment.

One issue of concern with cooperating processes involves synchronization issues. In this exercise, the parent and child processes must be synchronized so that the parent does not output the Fibonacci sequence until the child finishes generating the sequence. These two processes will be synchronized using the `wait()` system call; the parent process will invoke `wait()`, which will cause it to be suspended until the child process exits.

3.9 Most UNIX and Linux systems provide the `ipcs` command. This command lists the status of various POSIX interprocess communication mechanisms, including shared-memory segments. Much of the information for the command comes from the data structure `struct shmid_ds`, which is available in the `/usr/include/sys/shm.h` file. Some of the fields of this structure include:

- `int shm_segsz`—size of the shared-memory segment
- `short shm_nattch`—number of attaches to the shared-memory segment
- `struct ipc_perm shm_perm`—permission structure of the shared-memory segment

The `struct ipc_perm` data structure (which is available in the file `/usr/include/sys/ipc.h`) contains the fields:

- `unsigned short uid`—identifier of the user of the shared-memory segment
- `unsigned short mode`—permission modes
- `key_t key` (on Linux systems, `__key`)—user-specified key identifier

The permission modes are set according to how the shared-memory segment is established with the `shmget()` system call. Permissions are identified according to the following:

mode	meaning
0400	Read permission of owner.
0200	Write permission of owner.
0040	Read permission of group.
0020	Write permission of group.
0004	Read permission of world.
0002	Write permission of world.

Permissions can be accessed by using the bitwise *AND* operator &. For example, if the statement `mode & 0400` evaluates to true, the permission mode allows read permission by the owner of the shared-memory segment.

Shared-memory segments can be identified according to a user-specified key or according to the integer value returned from the `shmget()` system call, which represents the integer identifier of the shared-memory segment created. The `shm_ds` structure for a given integer segment identifier can be obtained with the following `shmctl()` system call:

```
/* identifier of the shared memory segment*/
int segment_id;
shm_ds shmbuffer;

shmctl(segment.id, IPC_STAT, &shmbuffer);
```

If successful, `shmctl()` returns 0; otherwise, it returns -1.

Write a C program that is passed an identifier for a shared-memory segment. This program will invoke the `shmctl()` function to obtain its `shm_ds` structure. It will then output the following values of the given shared-memory segment:

- Segment ID
- Key
- Mode
- Owner UID
- Size
- Number of attaches

This project consists of modifying a C program which serves as a shell interface that accepts user commands and then executes each command in a separate process. A shell interface provides the user a prompt after which the next command is entered. The example below illustrates the prompt sh> and the user's next command: `cat prog.c`. This command displays the file `prog.c` on the terminal using the UNIX `cat` command.

```
sh> cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (i.e. `cat prog.c`), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to what is illustrated in Figure 3.10. However, UNIX shells typically also allow the child process to run in the background—or concurrently—as well by specifying the ampersand (&) at the end of the command. By rewriting the above command as

```
sh> cat prog.c &
```

the parent and child processes now run concurrently.

The separate child process is created using the `fork()` system call and the user's command is executed by using one of the system calls in the `exec()` family (as described in Section 3.3.1).

Simple Shell

A C program that provides the basic operations of a command line shell is supplied in Figure 3.24. This program is composed of two functions: `main()` and `setup()`. The `setup()` function reads in the user's next command (which can be up to 80 characters), and then parses it into separate tokens that are used to fill the argument vector for the command to be executed. (If the command is to be run in the background, it will end with '&', and `setup()` will update the parameter `background` so the `main()` function can act accordingly. This program is terminated when the user enters `<Control><D>` and `setup()` then invokes `exit()`.

The `main()` function presents the prompt `COMMAND->` and then invokes `setup()`, which waits for the user to enter a command. The contents of the command entered by the user is loaded into the `args` array. For example, if the user enters `ls -l` at the `COMMAND->` prompt, `args[0]` becomes equal to the string `ls` and `args[1]` is set to the string to `-l`. (By "string", we mean a null-terminated, C-style string variable.)

This project is organized into two parts: (1) creating the child process and executing the command in the child, and (2) modifying the shell to allow a history feature.

Creating a Child Process

The first part of this project is to modify the `main()` function in Figure 3.24 so that upon returning from `setup()`, a child process is forked and executes the command specified by the user.

As noted above, the `setup()` function loads the contents of the `args` array with the command specified by the user. This `args` array will be passed to the `execvp()` function, which has the following interface:

```
execvp(char *command, char *params[]);
```

```
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80

/** setup() reads in the next command line, separating it into
distinct tokens using whitespace as delimiters.
setup() modifies the args parameter so that it holds pointers
to the null-terminated strings that are the tokens in the most
recent user command line as well as a NULL pointer, indicating
the end of the argument list, which comes after the string
pointers that have been assigned to args. */

void setup(char inputBuffer[], char *args[],int *background)
{
   /** full source code available online */
}

int main(void)
{
char inputBuffer[MAX_LINE]; /* buffer to hold command entered */
int background; /* equals 1 if a command is followed by '&' */
char *args[MAX_LINE/2 + 1]; /* command line arguments */

   while (1) {
      background = 0;
      printf(" COMMAND->");
      /* setup() calls exit() when Control-D is entered */
      setup(inputBuffer, args, &background);

      /** the steps are:
      (1) fork a child process using fork()
      (2) the child process will invoke execvp()
      (3) if background == 1, the parent will wait,
      otherwise it will invoke the setup() function again. */
   }
}
```

Figure 3.24 Outline of simple shell

where `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as `execvp(args[0],args)`; be sure to check the value of `background` to determine if the parent process is to wait for the child to exit or not.

Creating a History Feature

The next task is to modify the program in Figure 3.24 so that it provides a *history* feature that allows the user access up to the 10 most recently entered commands. These commands will be numbered starting at 1 and will continue to grow larger even past 10, e.g. if the user has entered 35 commands, the 10 most recent commands should be numbered 26 to 35. This history feature will be implementing using a few different techniques.

First, the user will be able to list these commands when he/she presses `<Control> <C>`, which is the `SIGINT` signal. UNIX systems use **signals** to notify a process that a particular event has occurred. Signals may be either synchronous or asynchronous, depending upon the source and the reason for the event being signaled. Once a signal has been generated by the occurrence of a certain event (e.g., division by zero, illegal memory access, user entering `<Control> <C>`, etc.), the signal is delivered to a process where it must be **handled**. A process receiving a signal may handle it by one of the following techniques:

Ignoring the signal

using the default signal handler, or

providing a separate signal-handling function.

Signals may be handled by first setting certain fields in the C structure `struct sigaction` and then passing this structure to the `sigaction()` function. Signals are defined in the include file `/usr/include/sys/signal.h`. For example, the signal `SIGINT` represents the signal for terminating a program with the control sequence `<Control> <C>`. The default signal handler for `SIGINT` is to terminate the program.

Alternatively, a program may choose to set up its own signal-handling function by setting the `sa handler` field in `struct sigaction` to the name of the function which will handle the signal and then invoking the `sigaction()` function, passing it (1) the signal we are setting up a handler for, and (2) a pointer to `struct sigaction`.

In Figure 3.25 we show a C program that uses the function `handle_SIGINT()` for handling the `SIGINT` signal. This function prints out the message "`Caught Control C`" and then invokes the `exit()` function to terminate the program. (We must use the `write()` function for performing output rather than the more common `printf()` as the former is known as being **signal-safe**, indicating it can be called from inside a signal-handling function; such guarantees cannot be made of `printf()`.) This program will run in the `while(1)` loop until the user enters the sequence `<Control> <C>`. When this occurs, the signal-handling function `handle_SIGINT()` is invoked.

The signal-handling function should be declared above `main()` and because control can be transferred to this function at any point, no parameters may be passed to it this function. Therefore, any data that it must access in your program must be declared globally, i.e. at the top of the source file before your function declarations. Before returning from the signal-handling function, it should reissue the command prompt.

If the user enters `<Control><C>`, the signal handler will output a list of the most recent 10 commands. With this list, the user can run any of the previous 10 commands by entering `r x` where 'x' is the first letter of that command. If more than one command starts with 'x', execute the most recent one. Also, the user should be able to run the most recent command again by just entering '`r`'. You can assume that only one space will separate the '`r`' and the first letter and

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

#define BUFFER_SIZE 50
char buffer[BUFFER_SIZE];

/* the signal handling function */
void handle_SIGINT()
{
   write(STDOUT_FILENO,buffer,strlen(buffer));

   exit(0);
}

int main(int argc, char *argv[])
{
   /* set up the signal handler */
   struct sigaction handler;
   handler.sa_handler = handle_SIGINT;
   sigaction(SIGINT, &handler, NULL);

   /* generate the output message */
   strcpy(buffer,"Caught Control C\n");

   /* loop until we receive <Control><C> */
   while (1)
      ;

   return 0;
}
```

Figure 3.25 Signal-handling program.

that the letter will be followed by '\n'. Again, 'r' alone will be immediately followed by the \n character if it is wished to execute the most recent command.

Any command that is executed in this fashion should be echoed on the user's screen and the command is also placed in the history buffer as the next command. (`r x` does not go into the history list; the actual command that it specifies, though, does.)

It the user attempts to use this history facility to run a command and the command is detected to be *erroneous*, an error message should be given to the user and the command not entered into the history list, and the `execvp()` function should not be called. (It would be nice to know about improperly formed commands that are handed off to `execvp()` that appear to look valid and are not, and not include them in the history as well, but that is beyond the capabilities of this simple shell program.) You should also modify `setup()` so it returns an `int` signifying if has successfully created a valid args list or not, and the `main()` should be updated accordingly.

Bibliographical Notes

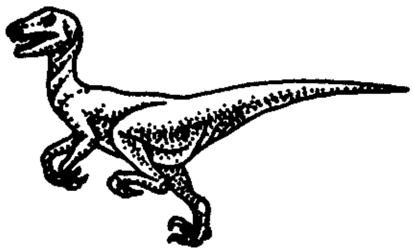
Interprocess communication in the RC 4000 system was discussed by Brinch-Hansen [1970]. Schlichting and Schneider [1982] discussed asynchronous message-passing primitives. The IPC facility implemented at the user level was described by Bershad et al. [1990].

Details of interprocess communication in UNIX systems were presented by Gray [1997]. Barrera [1991] and Vahalia [1996] described interprocess communication in the Mach system. Solomon and Russinovich [2000] and Stevens [1999] outlined interprocess communication in Windows 2000 and UNIX respectively.

The implementation of RPCs was discussed by Birrell and Nelson [1984]. A design of a reliable RPC mechanism was described by Shrivastava and Panzieri [1982], and Tay and Ananda [1990] presented a survey of RPCs. Stankovic [1982] and Staunstrup [1982] discussed procedure calls versus message-passing communication. Grosso [2002] discussed RMI in significant detail. Calvert and Donahoo [2001] provided coverage of socket programming in Java.

CHAPTER

Multithreaded Programming



The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control. Most modern operating systems now provide features enabling a process to contain multiple threads of control. This chapter introduces many concepts associated with multithreaded computer systems, including a discussion of the APIs for the Pthreads, Win32, and Java thread libraries. We look at many issues related to multithreaded programming and how it affects the design of operating systems. Finally, we explore how the Windows XP and Linux operating systems support threads at the kernel level.

4.1 Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or **heavyweight**) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 4.1 illustrates the difference between a traditional **single-threaded** process and a **multithreaded** process.

4.1.1 Motivation

Many software packages that run on modern desktop PCs are multithreaded. An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network, for example. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands) of clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one

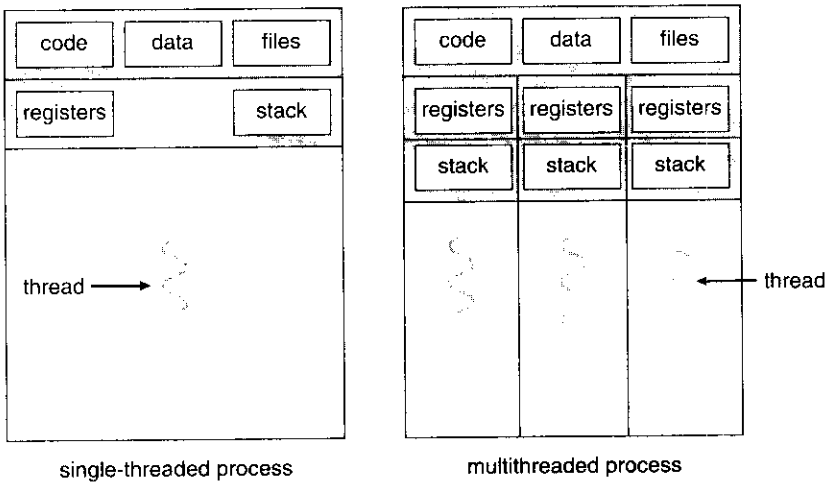


Figure 4.1 Single-threaded and multithreaded processes.

client at a time. The amount of time that a client might have to wait for its request to be serviced could be enormous.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, as was shown in the previous chapter. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. This approach would multithread the web-server process. The server would create a separate thread that would listen for client requests; when a request was made, rather than creating another process, the server would create another thread to service the request.

Threads also play a vital role in remote procedure call (RPC) systems. Recall from Chapter 3 that RPCs allow interprocess communication by providing a communication mechanism similar to ordinary function or procedure calls. Typically, RPC servers are multithreaded. When a server receives a message, it services the message using a separate thread. This allows the server to service several concurrent requests. Java's RMI systems work similarly.

Finally, many operating system kernels are now multithreaded; several threads operate in the kernel, and each thread performs a specific task, such as managing devices or interrupt handling. For example, Solaris creates a set of threads in the kernel specifically for interrupt handling; Linux uses a kernel thread for managing the amount of free memory in the system.

4.1.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

Responsiveness. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.

Resource sharing. By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Economy. Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

Utilization of multiprocessor architectures. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

4.2

Our discussion so far has treated threads in a generic sense. However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows XP, Linux, Mac OS X, Solaris, and Tru64 UNIX (formerly Digital UNIX)—support kernel threads.

Ultimately, there must exist a relationship between user threads and kernel threads. In this section, we look at three common ways of establishing this relationship.

4.2.1 Many-to-One Model

The many-to-one model (Figure 4.2) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. **Green threads**—a thread library available for Solaris—uses this model, as does **GNU Portable Threads**.

4.2.2 One-to-One Model

The one-to-one model (Figure 4.3) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another

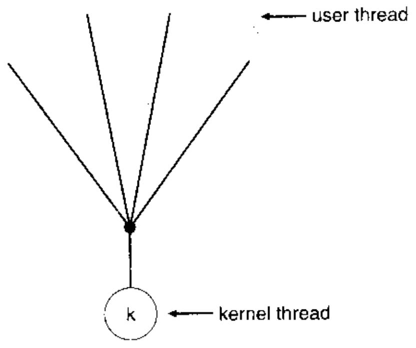


Figure 4.2 Many-to-one model.

thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems—including Windows 95, 98, NT, 2000, and XP—implement the one-to-one model.

4.2.3 Many-to-Many Model

The many-to-many model (Figure 4.4) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor). Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time. The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create). The many-to-many model suffers from neither of these shortcomings: Developers can create as many user

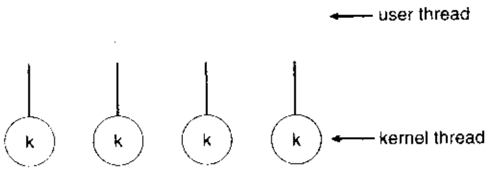


Figure 4.3 One-to-one model.

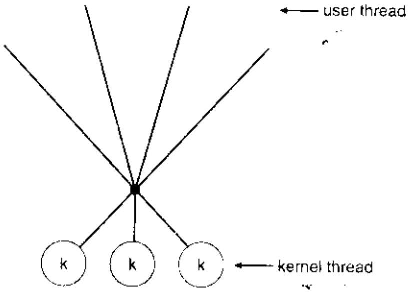


Figure 4.4 Many-to-many model.

threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One popular variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation, sometimes referred to as the *two-level model* (Figure 4.5), is supported by operating systems such as IRIX, HP-UX, and Tru64 UNIX. The Solaris operating system supported the two-level model in versions older than Solaris 9. However, beginning with Solaris 9, this system uses the one-to-one model.

4.3

A **thread library** provides the programmer an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support.

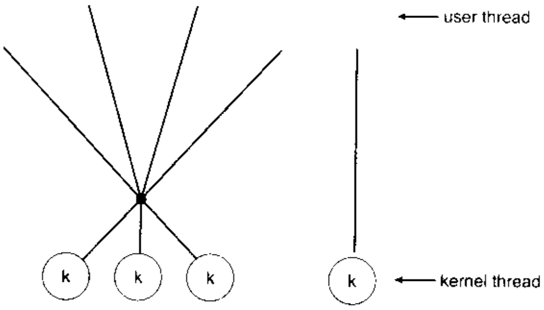


Figure 4.5 Two-level model.

All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today: (1) POSIX Pthreads, (2) Win32, and (3) Java. Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library. The Win32 thread library is a kernel-level library available on Windows systems. The Java thread API allows thread creation and management directly in Java programs. However, because in most instances the JVM is running on top of a host operating system, the Java thread API is typically implemented using a thread library available on the host system. This means that on Windows systems, Java threads are typically implemented using the Win32 API; UNIX and Linux systems often use Pthreads.

In the remainder of this section, we describe basic thread creation using these three thread libraries. As an illustrative example, we design a multithreaded program that performs the summation of a non-negative integer in a separate thread using the well-known summation function:

$$sum = \sum_{i=0}^{N} i$$

For example, if N were 5, this function would represent the summation from 0 to 5, which is 15. Each of the three programs will be run with the upper bounds of the summation entered on the command line; thus, if the user enters 8, the summation of the integer values from 0 to 8 will be output.

4.3.1 Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a *specification* for thread behavior, not an *implementation*. Operating system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification, including Solaris, Linux, Mac OS X, and Tru64 UNIX. *Shareware* implementations are available in the public domain for the various Windows operating systems as well.

The C program shown in Figure 4.6 demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread. In a Pthreads program, separate threads begin execution in a specified function. In Figure 4.6, this is the `runner()` function. When this program begins, a single thread of control begins in `main()`. After some initialization, `main()` creates a second thread that begins control in the `runner()` function. Both threads share the global data `sum`.

Let's look more closely at this program. All Pthreads programs must include the `pthread.h` header file. The statement `pthread_t tid` declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The `pthread_attr_t attr` declaration represents the attributes for the thread. We set the attributes in

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

the function call `pthread_attr_init(&attr)`. Because we did not explicitly set any attributes, we use the default attributes provided. (In Chapter 5, we will discuss some of the scheduling attributes provided by the Pthreads API.) A separate thread is created with the `pthread_create()` function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the `runner()` function. Last, we pass the integer parameter that was provided on the command line, `argv[1]`.

At this point, the program has two threads: the initial (or parent) thread in `main()` and the summation (or child) thread performing the summation operation in the `runner()` function. After creating the summation thread, the parent thread will wait for it to complete by calling the `pthread_join()` function. The summation thread will complete when it calls the function `pthread_exit()`. Once the summation thread has returned, the parent thread will output the value of the shared data `sum`.

4.3.2 Win32 Threads

The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways. We illustrate the Win32 thread API in the C program shown in Figure 4.7. Notice that we must include the `windows.h` header file when using the Win32 API.

Just as in the Pthreads version shown in Figure 4.6, data shared by the separate threads—in this case, `Sum`—are declared globally (the `DWORD` data type is an unsigned 32-bit integer. We also define the `Summation()` function that is to be performed in a separate thread. This function is passed a pointer to a `void`, which Win32 defines as LPVOID. The thread performing this function sets the global data `Sum` to the value of the summation from 0 to the parameter passed to `Summation()`.

Threads are created in the Win32 API using the `CreateThread()` function and—just as in Pthreads—a set of attributes for the thread is passed to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state. In this program, we use the default values for these attributes (which do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler). Once the summation thread is created, the parent must wait for it to complete before outputting the value of `Sum`, as the value is set by the summation thread. Recall that the Pthread program (Figure 4.6) had the parent thread wait for the summation thread using the `pthread_join()` statement. We perform the equivalent of this in the Win32 API using the `WaitForSingleObject()` function, which causes the creating thread to block until the summation thread has exited. (We will cover synchronization objects in more detail in Chapter 6.)

4.3.3 Java Threads

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a `main()` method runs as a single thread in the JVM.

There are two techniques for creating threads in a Java program. One approach is to create a new class that is derived from the `Thread` class and to override its `run()` method. An alternative—and more commonly used—technique is to define a class that implements the `Runnable` interface. The `Runnable` interface is defined as follows:

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
  DWORD Upper = *(DWORD*)Param;
  for (DWORD i = 0; i <= Upper; i++)
    Sum += i;
  return 0;
}

int main(int argc, char *argv[])
{
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param;
  /* perform some basic error checking */
  if (argc != 2) {
    fprintf(stderr,"An integer parameter is required\n")
    return -1;
  }
  Param = atoi(argv[1]);
  if (Param < 0) {
    fprintf(stderr,"An integer >= 0 is required\n");
    return -1;
  }

  // create the thread
  ThreadHandle = CreateThread(
    NULL, // default security attributes
    0, // default stack size
    Summation, // thread function
    &Param, // parameter to thread function
    0, // default creation flags
    &ThreadId); // returns the thread identifier

  if (ThreadHandle != NULL) {
    // now wait for the thread to finish
    WaitForSingleObject(ThreadHandle,INFINITE);

    // close the thread handle
    CloseHandle(ThreadHandle);

    printf("sum = %d\n",Sum);
  }
}
```

Figure 4.7 Multithreaded C program using the Win32 API.

```
public interface Runnable
{
    public abstract void run();
}
```

When a class implements `Runnable`, it must define a `run()` method. The code implementing the `run()` method is what runs as a separate thread.

Figure 4.8 shows the Java version of a multithreaded program that determines the summation of a non-negative integer. The `Summation` class implements the `Runnable` interface. Thread creation is performed by creating an object instance of the `Thread` class and passing the constructor a `Runnable` object.

Creating a `Thread` object does not specifically create the new thread; rather, it is the `start()` method that actually creates the new thread. Calling the `start()` method for the new object does two things:

It allocates memory and initializes a new thread in the JVM.

It calls the `run()` method, making the thread eligible to be run by the JVM. (Note that we never call the `run()` method directly. Rather, we call the `start()` method, and it calls the `run()` method on our behalf.)

When the summation program runs, two threads are created by the JVM. The first is the parent thread, which starts execution in the `main()` method. The second thread is created when the `start()` method on the `Thread` object is invoked. This child thread begins execution in the `run()` method of the `Summation` class. After outputting the value of the summation, this thread terminates when it exits from its `run()` method.

Sharing of data between threads occurs easily in Win32 and Pthreads, as shared data are simply declared globally. As a pure object-oriented language, Java has no such notion of global data; if two or more threads are to share data in a Java program, the sharing occurs by passing reference to the shared object to the appropriate threads. In the Java program shown in Figure 4.8, the main thread and the summation thread share the the object instance of the `Sum` class. This shared object is referenced through the appropriate `getSum()` and `setSum()` methods. (You might wonder why we don't use an `Integer` object rather than designing a new `sum` class. The reason is that the `Integer` class is **immutable**—that is, once its value is set, it cannot change.)

Recall that the parent threads in the Pthreads and Win32 libraries use `pthread_join()` and `WaitForSingleObject()` (respectively) to wait for the summation threads to finish before proceeding. The `join()` method in Java provides similar functionality. (Notice that `join()` can throw an `InterruptedException`, which we choose to ignore.)

4.4

In this section, we discuss some of the issues to consider with multithreaded programs.

```
class Sum
{
  private int sum;

  public int getSum() {
    return sum;
  }

  public void setSum(int sum) {
    this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
  }

  public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
      sum += i;
    sumValue.setSum(sum);
  }
}

public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
     if (Integer.parseInt(args[0]) < 0)
      System.err.println(args[0] + " must be >= 0.");
     else {
      // create the object to be shared
      Sum sumObject = new Sum();
      int upper = Integer.parseInt(args[0]);
      Thread thrd = new Thread(new Summation(upper, sumObject));
      thrd.start();
      try {
         thrd.join();
         System.out.println
                 ("The sum of "+upper+" is "+sumObject.getSum());
     } catch (InterruptedException ie) { }
     }
    }
    else
     System.err.println("Usage: Summation <integer value>"); }
}
```

Figure 4.8 Java program for the summation of a non-negative integer.

4.4.1 The fork() and exec() System Calls

In Chapter 3, we described how the `fork()` system call is used to create a separate, duplicate process. The semantics of the `fork()` and `exec()` system calls change in a multithreaded program.

If one thread in a program calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of `fork()`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork()` system call.

The `exec()` system call typically works in the same way as described in Chapter 3. That is, if a thread invokes the `exec()` system call, the program specified in the parameter to `exec()` will replace the entire process—including all threads.

Which of the two versions of `fork()` to use depends on the application. If `exec()` is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to `exec()` will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call `exec()` after forking, the separate process should duplicate all threads.

4.4.2 Cancellation

Thread cancellation is the task of terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page is loaded using several threads—each image is loaded in a separate thread. When a user presses the *stop* button on the browser, all threads loading the page are canceled.

A thread that is to be canceled is often referred to as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

- **Asynchronous cancellation**. One thread immediately terminates the target thread.
- **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not. This allows a thread to check whether it should be canceled at a point when it can be canceled safely. Pthreads refers to such points as **cancellation points**.

4.4.3 Signal Handling

A **signal** is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern:

- A signal is generated by the occurrence of a particular event.
- A generated signal is delivered to a process.
- Once delivered, the signal must be handled.

Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as `<control><C>`) and having a timer expire. Typically, an asynchronous signal is sent to another process.

Every signal may be *handled* by one of two possible handlers:

- A default signal handler
- A user-defined signal handler

Every signal has a **default signal handler** that is run by the kernel when handling that signal. This default action can be overridden by a **user-defined signal handler** that is called to handle the signal. Signals may be handled in different ways. Some signals (such as changing the size of a window) may simply be ignored; others (such as an illegal memory access) may be handled by terminating the program.

Handling signals in single-threaded programs is straightforward; signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

In general, the following options exist:

- Deliver the signal to the thread to which the signal applies.
- Deliver the signal to every thread in the process.
- Deliver the signal to certain threads in the process.
- Assign a specific thread to receive all signals for the process.

The method for delivering a signal depends on the type of signal generated. For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process. However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (`<control><C>`, for example)—should be sent to all threads.

Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block. Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it. However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it. The standard UNIX function for delivering a signal is `kill(aid_t aid, int signal)`; here, we specify the process (`aid`) to which a particular signal is to be delivered. However, POSIX Pthreads also provides the `pthread_kill(pthread_t tid, int signal)` function, which allows a signal to be delivered to a specified thread (`tid`.)

Although Windows does not explicitly provide support for signals, they can be emulated using **asynchronous procedure calls (APCs)**. The APC facility allows a user thread to specify a function that is to be called when the user thread receives notification of a particular event. As indicated by its name, an APC is roughly equivalent to an asynchronous signal in UNIX. However, whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward, as an APC is delivered to a particular thread rather than a process.

4.4.4 Thread Pools

In Section 4.1, we mentioned multithreading in a web server. In this situation, whenever the server receives a request, it creates a separate thread to service the request. Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems. The first concerns the amount of time required to create the thread prior to servicing the request, together with the fact that this thread will be discarded once it has completed its work. The second issue is more troublesome: If we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this issue is to use a **thread pool**.

The general idea behind a thread pool is to create a number of threads at process startup and place them into a *pool,* where they sit and wait for work. When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the request to service. Once the thread completes its service, it returns to the pool and awaits more work. If the pool contains no available thread, the server waits until one becomes free.

Thread pools offer these benefits:

- Servicing a request with an existing thread is usually faster than waiting to create a thread.
- A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests. More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns. Such architectures provide the further benefit of having a smaller pool—thereby consuming less memory—when the load on the system is low.

The Win32 API provides several functions related to thread pools. Using the thread pool API is similar to creating a thread with the `Thread Create()` function, as described in Section 4.3.2. Here, a function that is to run as a separate thread is defined. Such a function may appear as follows:

```
DWORD WINAPI PoolFunction(AVOID Param) {
   /**
   * this function runs as a separate thread.
   **/
}
```

A pointer to `PoolFunction()` is passed to one of the functions in the thread pool API, and a thread from the pool executes this function. One such member in the thread pool API is the `QueueUserWorkItem()` function, which is passed three parameters:

`LPTHREAD_START_ROUTINE Function`—a pointer to the function that is to run as a separate thread

`PVOID Param`—the parameter passed to `Function`

`ULONG Flags`—flags indicating how the thread pool is to create and manage execution of the thread

An example of an invocation is:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

This causes a thread from the thread pool to invoke `PoolFunction()` on behalf of the programmer. In this instance, we pass no parameters to `PoolFunction()`. Because we specify `0` as a flag, we provide the thread pool with no special instructions for thread creation.

Other members in the Win32 thread pool API include utilities that invoke functions at periodic intervals or when an asynchronous I/O request completes. The `java.util.concurrent` package in Java 1.5 provides a thread pool utility as well.

4.4.5 Thread-Specific Data

Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread-specific data**. For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction may be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific

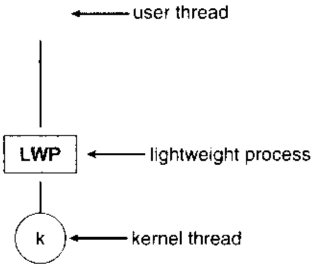


Figure 4.9 Lightweight process (LWP.)

data. Most thread libraries—including Win32 and Pthreads—provide some form of support for thread-specific data. Java provides support as well.

4.4.6 Scheduler Activations

A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models discussed in Section 4.2.3. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

Many systems implementing either the many-to-many or two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a lightweight process, or LWP—is shown in Figure 4.9. To the user-thread library, the LWP appears to be a *virtual processor* on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.

An application may require any number of LWPs to run efficiently. Consider a CPU-bound application running on a single processor. In this scenario, only one thread can run at once, so one LWP is sufficient. An application that is I/O-intensive may require multiple LWPs to execute, however. Typically, an LWP is required for each concurrent blocking system call. Suppose, for example, that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/O completion in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor. One event that triggers an upcall occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor. After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

4.5 Operating-System Examples

In this section, we explore how threads are implemented in Windows XP and Linux systems.

4.5.1 Windows XP Threads

Windows XP implements the Win32 API. The Win32 API is the primary API for the family of Microsoft operating systems (Windows 95, 98, NT, 2000, and XP). Indeed, much of what is mentioned in this section applies to this entire family of operating systems.

A Windows XP application runs as a separate process, and each process may contain one or more threads. The Win32 API for creating threads is covered in Section 4.3.2. Windows XP uses the one-to-one mapping described in Section 4.2.2, where each user-level thread maps to an associated kernel thread. However, Windows XP also provides support for a **fiber** library, which provides the functionality of the many-to-many model (Section 4.2.3). By using the thread library, any thread belonging to a process can access the address space of the process.

The general components of a thread include:

- A thread ID uniquely identifying the thread
- A register set representing the status of the processor
- A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
- A private storage area used by various run-time libraries and dynamic link libraries (DLLs)

The register set, stacks, and private storage area are known as the **context** of the thread. The primary data structures of a thread include:

- ETHREAD—executive thread block
- KTHREAD—kernel thread block
- TEB—thread environment block

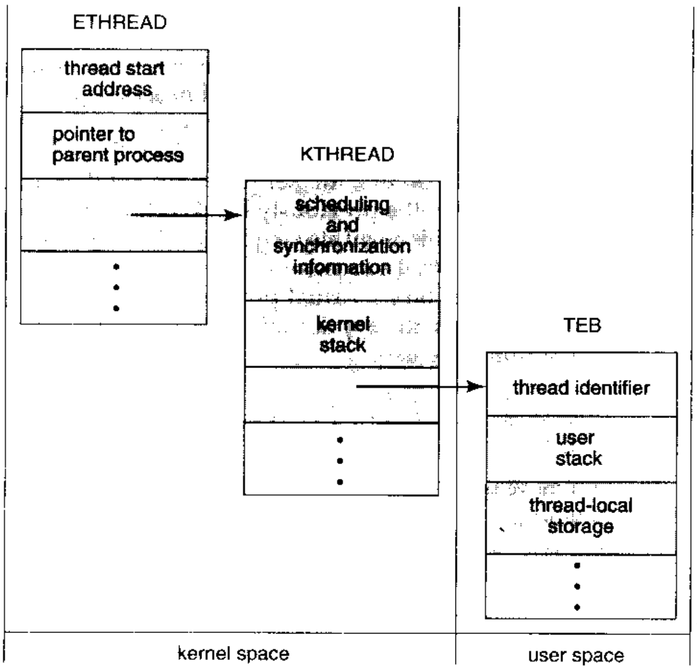


Figure 4.10 Data structures of a Windows XP thread.

The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.

The KTHREAD includes scheduling and synchronization information for the thread. In addition, the KTHREAD includes the kernel stack (used when the thread is running in kernel mode) and a pointer to the TEB.

The ETHREAD and the KTHREAD exist entirely in kernel space; this means that only the kernel can access them. The TEB is a user-space data structure that is accessed when the thread is running in user mode. Among other fields, the TEB contains the thread identifier, a user-mode stack, and an array for thread-specific data (which Windows XP terms **thread-local storage**). The structure of a Windows XP thread is illustrated in Figure 4.10.

4.5.2 Linux Threads

Linux provides the `fork()` system call with the traditional functionality of duplicating a process, as described in Chapter 3. Linux also provides the ability to create threads using the `clone()` system call. However, Linux does not distinguish between processes and threads. In fact, Linux generally uses the term *task*—rather than *process* or *thread*—when referring to a flow of control within a program. When `clone()` is invoked, it is passed a set of flags, which determine how much sharing is to take place between the parent and child tasks. Some of these flags are listed below:

flag	meaning
`CLONE_FS`	File-system information is shared.
`CLONE_VM`	The same memory space is shared.
`CLONE_SIGHAND`	Signal handlers are shared.
`CLONE_FILES`	The set of open files is shared.

For example, if `clone()` is passed the flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES`, the parent and child tasks will share the same file-system information (such as the current working directory), the same memory space, the same signal handlers, and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread as described in this chapter, since the parent task shares most of its resources with its child task. However, if none of these flags are set when `clone()` is invoked, no sharing takes place, resulting in functionality similar to that provided by the `fork()` system call.

The varying level of sharing is possible because of the way a task is represented in the Linux kernel. A unique kernel data structure (specifically, `struct task struct`) exists for each task in the system. This data structure, instead of storing data for the task, contains pointers to other data structures where these data are stored—for example, data structures that represent the list of open files, signal-handling information, and virtual memory. When `fork()` is invoked, a new task is created, along with a *copy* of all the associated data structures of the parent process. A new task is also created when the `clone()` system call is made. However, rather than copying all data structures, the new task *points* to the data structures of the parent task, depending on the set of flags passed to `clone()`.

4.6

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and the ability to take advantage of multiprocessor architectures.

User-level threads are threads that are visible to the programmer and are unknown to the kernel. The operating-system kernel supports and manages kernel-level threads. In general, user-level threads are faster to create and manage than are kernel threads, as no intervention from the kernel is required. Three different types of models relate user and kernel threads: The many-to-one model maps many user threads to a single kernel thread. The one-to-one model maps each user thread to a corresponding kernel thread. The many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads.

Most modern operating systems provide kernel support for threads; among these are Windows 98, NT, 2000, and XP, as well as Solaris and Linux.

Thread libraries provide the application programmer with an API for creating and managing threads. Three primary thread libraries are in common use: POSIX Pthreads, Win32 threads for Windows systems, and Java threads.

Multithreaded programs introduce many challenges for the programmer, including the semantics of the `fork()` and `exec()` system calls. Other issues include thread cancellation, signal handling, and thread-specific data.

4.1 Provide two programming examples in which multithreading does *not* provide better performance than a single-threaded solution.

4.2 Describe the actions taken by a thread library to context switch between user-level threads.

4.3 Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values

b. Heap memory

c. Global variables

d. Stack memory

4.4 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system?

4.5 The program shown in Figure 4.11 uses the Pthreads API. What would be output from the program at `LINE C` and `LINE P`?

4.6 Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be more than the number of processors in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processors.

b. The number of kernel threads allocated to the program is equal to the number of processors.

c. The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user-level threads.

```
#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
int pid;
pthread_t tid;
pthread_attr_t attr;

  pid = fork();

  if (pid == 0) {/* child process */
    pthread_attr_init(&attr);
    pthread_create(&tid,&attr,runner,NULL);
    pthread_join(tid,NULL);
    printf("CHILD: value = %d",value); /* LINE C */
  }
  else if (pid > 0) {/* parent process */
    wait(NULL);
    printf("PARENT: value = %d",value); /* LINE P */
  }
}

void *runner(void *param) {
  value = 5;
  pthread_exit(0);
}
```

Figure 4.11 C program for question 4.5.

4.7 Modify the socket-based date server (Figure 3.18) in Chapter 3 so that the server services each client request in a separate thread.

4.8 The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_n = fib_{n-1} + fib_{n-2}$$

Write a multithreaded program that generates the Fibonacci series using either the Java, Pthreads, or Win32 thread library. This program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that is shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish, using the techniques described in Section 4.3.

Given two matrices A and B, where A is a matrix with M rows and K columns and matrix B contains K rows and N columns, the **matrix product** of A and B is matrix C, where C contains M rows and N columns. The entry in matrix C for row i column j ($C_{i,j}$) is the sum of the products of the elements for row i in matrix A and column j in matrix B. That is,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} \times B_{n,j}$$

For example, if A were a 3-by-2 matrix and B were a 2-by-3 matrix, element $C_{3,1}$ would be the sum of $A_{3,1} \times B_{1,1}$ and $A_{3,2} \times B_{2,1}$.

For this project, calculate each element $C_{i,j}$ in a separate *worker* thread. This will involve creating $M \times N$ worker threads. The main—or parent—thread will initialize the matrices A and B and allocate sufficient memory for matrix C, which will hold the product of matrices A and B. These matrices will be declared as global data so that each worker thread has access to A, B, and C.

Matrices A and B can be initialized statically, as shown below:

```
#define M 3
#define K 2
#define N 3

int A [M][K] = { {1,4}, {2,5}, {3,6} };
int B [K][N] = { {8,7,6}, {5,4,3} };
int C [M] [N];
```

Alternatively, they can be populated by reading in values from a file.

Passing Parameters to Each Thread

The parent thread will create $M \times N$ worker threads, passing each worker the values of row i and column j that it is to use in calculating the matrix product. This requires passing two parameters to each thread. The easiest approach with Pthreads and Win32 is to create a data structure using a `struct`. The members of this structure are i and j, and the structure appears as follows:

```
/* structure for passing data to threads */
struct v
{
   int i; /* row */
   int j; /* column */
};
```

Both the Pthreads and Win32 programs will create the worker threads using a strategy similar to that shown below:

```
/* We have to create M * N worker threads */
for (i = 0; i < M, i++)
   for (j = 0; j < N; j++ ) {
      struct v *data = (struct v *) malloc(sizeof(struct v));
      data->i = i;
      data->j = j;
      /* Now create the thread passing it data as a parameter */
   }
}
```

The data pointer will be passed to either the `pthread_create()` (Pthreads) function or the `CreateThread()` (Win32) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

Sharing of data between Java threads is different from sharing between threads in Pthreads or Win32. One approach is for the main thread to create and initialize the matrices *A*, *B*, and *C*. This main thread will then create the worker threads, passing the three matrices—along with row *i* and column *j*—to the constructor for each worker. Thus, the outline of a worker thread appears as follows:

```
public class WorkerThread implements Runnable
{
    private int row;
    private int col;
    private int[][] A;
    private int[][] B;
    private int[][] C;

  public WorkerThread(int row, int col, int[][] A,
    int[][] B, int[][] C) {
     this.row = row;
     this.col = col;
     this.A = A;
     this.B = B;
     this.C = C;
  }
  public void run() {
     /* calculate the matrix product in C[row] [col] */
  }
}
```

Waiting for Threads to Complete

Once all worker threads have completed, the main thread will output the product contained in matrix *C*. This requires the main thread to wait for all worker threads to finish before it can output the value of the matrix product. Several different strategies can be used to enable a thread to wait

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.12 Phtread code for joining ten threads.

for other threads to finish. Section 4.3 describes how to wait for a child thread to complete using the Win32, Pthreads, and Java thread libraries. Win32 provides the `WaitForSingleObject()` function, whereas Pthreads and Java use `pthread_join()` and `join()`, respectively. However, in these programming examples, the parent thread waits for a single child thread to finish; completing this exercise will require waiting for multiple threads.

In Section 4.3.2, we describe the `WaitForSingleObject()` function, which is used to wait for a single thread to finish. However, the Win32 API also provides the `WaitForMultipleObjects()` function, which is used when waiting for multiple threads to complete. `WaitForMultipleObjects()` is passed four parameters:

- The number of objects to wait for
- A pointer to the array of objects
- A flag indicating if all objects have been signaled
- A timeout duration (or `INFINITE`)

For example, if `THandles` is an array of thread `HANDLE` objects of size `N`, the parent thread can wait for all its child threads to complete with the statement:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

A simple strategy for waiting on several threads using the Pthreads `pthread_join()` or Java's `join()` is to enclose the join operation within a simple `for` loop. For example, you could join on ten threads using the Pthread code depicted in Figure 4.12. The equivalent code using Java threads is shown in Figure 4.13.

```
final static int NUM_THREADS = 10;

/* an array of threads to be joined upon */
Thread[] workers = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
   try {
      workers[i].join();
     }catch (InterruptedException ie) {}
}
```

Figure 4.13 Java code for joining ten threads.

Thread performance issues were discussed by Anderson et al. [1989], who continued their work in Anderson et al. [1991] by evaluating the performance of user-level threads with kernel support. Bershad et al. [1990] describe combining threads with RPC. Engelschall [2000] discusses a technique for supporting user-level threads. An analysis of an optimal thread-pool size can be found in Ling et al. [2000]. Scheduler activations were first presented in Anderson et al. [1991], and Williams [2002] discusses scheduler activations in the NetBSD system. Other mechanisms by which the user-level thread library and the kernel cooperate with each other are discussed in Marsh et al. [1991], Govindan and Anderson [1991], Draves et al. [1991], and Black [1990]. Zabatta and Young [1998] compare Windows NT and Solaris threads on a symmetric multiprocessor. Pinilla and Gill [2003] compare Java thread performance on Linux, Windows, and Solaris.

Vahalia [1996] covers threading in several versions of UNIX. Mauro and McDougall [2001] describe recent developments in threading the Solaris kernel. Solomon and Russinovich [2000] discuss threading in Windows 2000. Bovet and Cesati [2002] explain how Linux handles threading.

Information on Pthreads programming is given in Lewis and Berg [1998] and Butenhof [1997]. Information on threads programming in Solaris can be found in Sun Microsystems [1995]. Oaks and Wong [1999], Lewis and Berg [2000], and Holub [2000] discuss multithreading in Java. Beveridge and Wiener [1997] and Cohen and Woodring [1997] describe multithreading using Win32.