This page intentionally left blank

Processes

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 among processes, the operating system can make the computer more productive. In this chapter, you will read about what processes are and how they work.

CHAPTER OBJECTIVES

- To introduce the notion of a process — a program in execution, which forms the basis of all computation.
- To describe the various features of processes, including scheduling, creation and termination, and communication.
- To describe communication in client–server systems.

3.1 Process Concept

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. And 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*.

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`.)

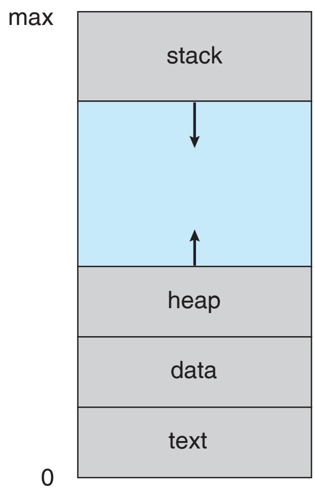

Figure 3.1 Process in memory.

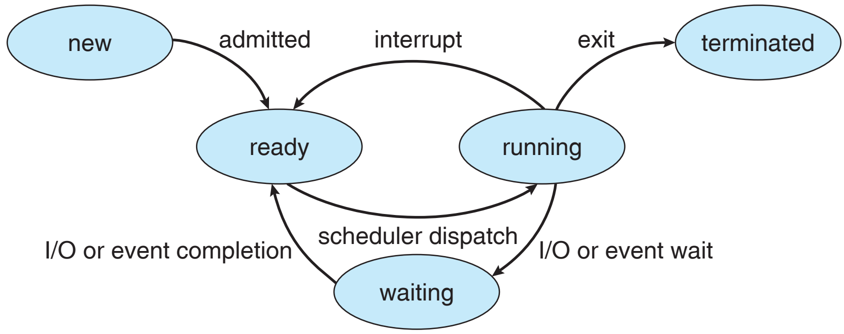

Figure 3.2 Diagram of process state.

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:

- **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 delineate process states more finely. 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:

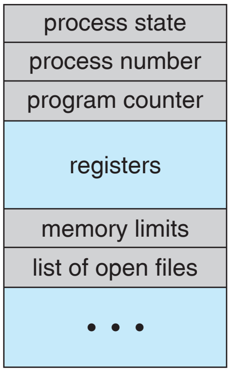

Figure 3.3 Process control block (PCB).

- **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 7).
- **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

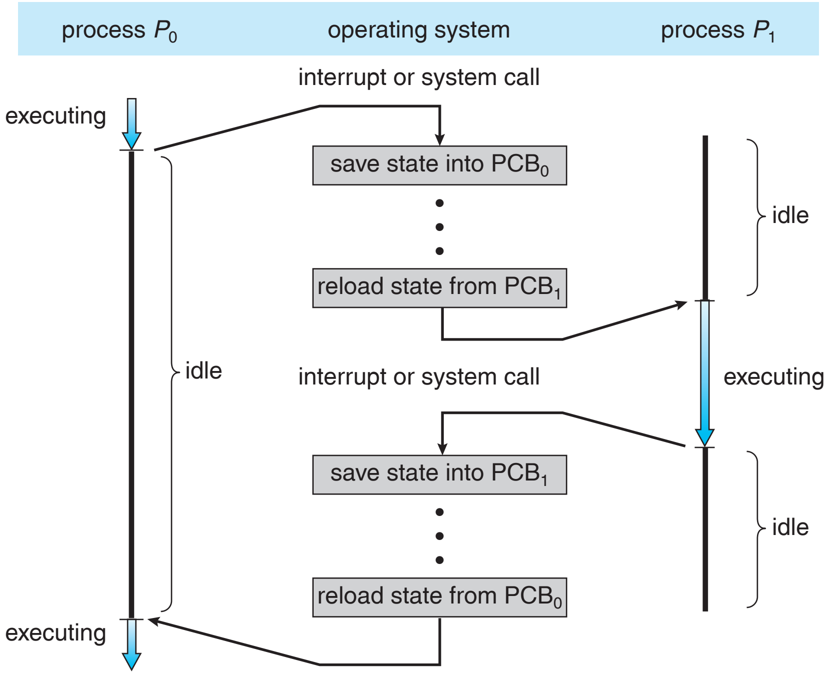

Figure 3.4 Diagram showing CPU switch from process to process.

task at a 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. On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads. Chapter 4 explores multithreaded processes in detail.

3.2 Process Scheduling

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 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**.

PROCESS REPRESENTATION IN LINUX

The process control block in the Linux operating system is represented by the C structure `task_struct`. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and any of its children. (A process's *parent* is the process that created it; its *children* are any processes that it creates.) Some of these fields include:

```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`, and the kernel maintains a pointer —`current`—to the process currently executing on the system. This is shown in Figure 3.5.

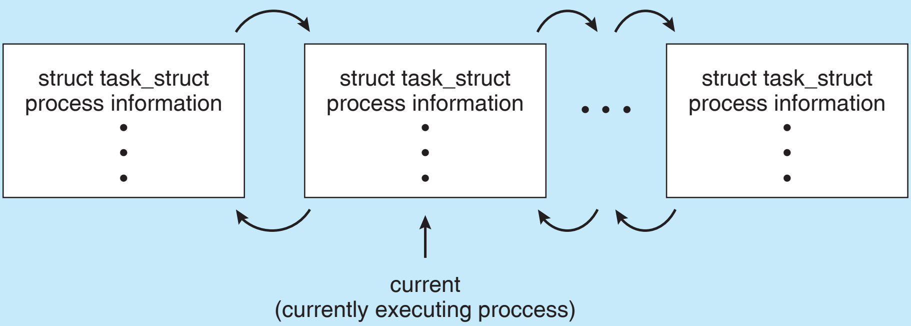

Figure 3.5 Active processes in Linux.

As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

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.

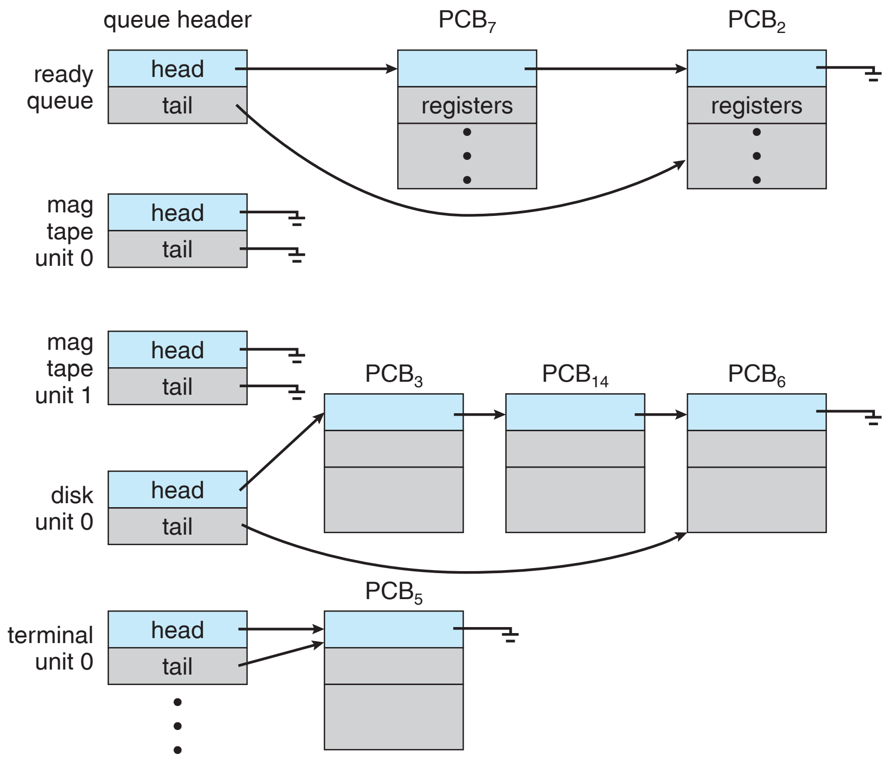

Figure 3.6 The ready queue and various I/O device queues.

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 3.6).

A common representation of process scheduling is a **queueing diagram**, such as that in Figure 3.7. 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.
- 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.

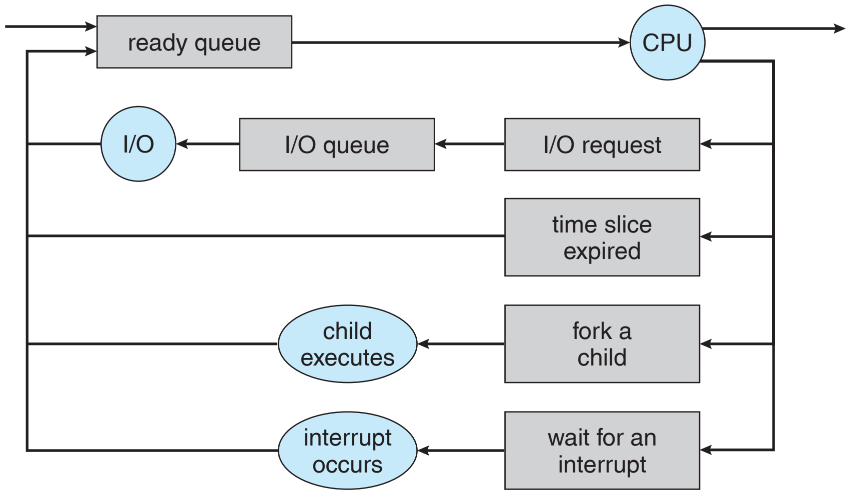

Figure 3.7 Queueing-diagram representation of process scheduling.

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 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 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 3.8. 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 7.

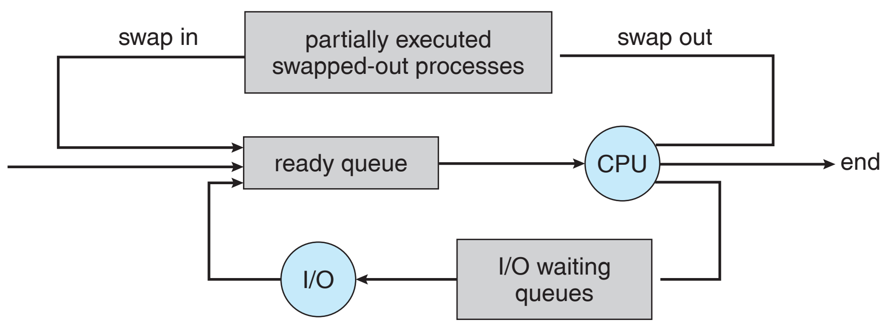

Figure 3.8 Addition of medium-term scheduling to the queueing diagram.

3.2.3 Context Switch

As mentioned in Section 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 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 7, 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 Operations on Processes

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 3.9 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 3.9, 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. In this command-line interface, the user can then invoke 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, we can obtain a listing of processes by using the `ps` command. For example, 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 that shown in Figure 3.9 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

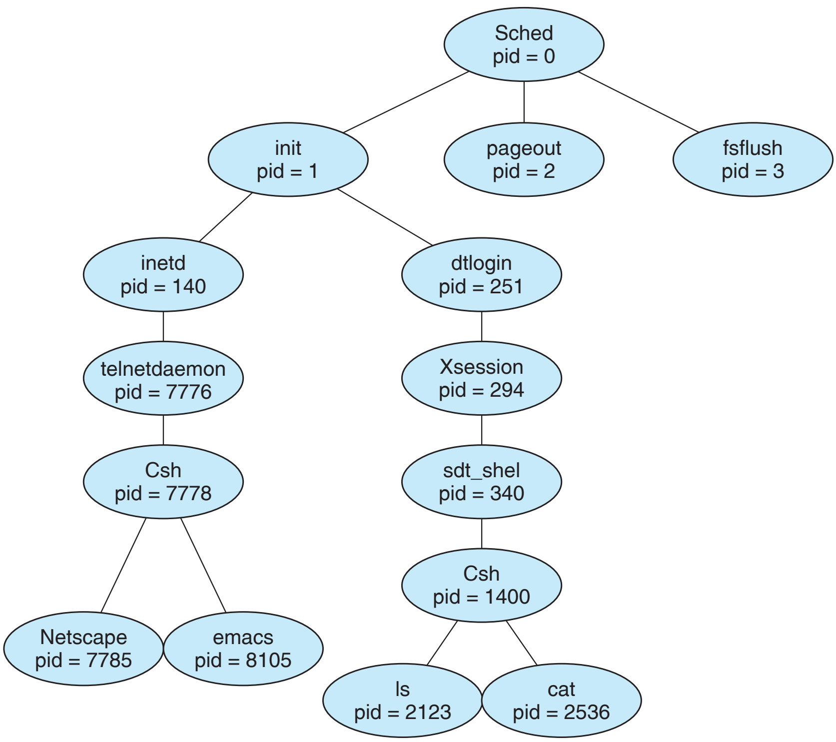

Figure 3.9 A tree of processes on a typical Solaris system.

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 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 for execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

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

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. 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 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 3.10 illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of `pid` (the process identifier) for the child process is zero, while that for the parent is an integer

```
#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");
       return 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");
    }

    return 0;
}
```

Figure 3.10 Creating a separate process using the UNIX `fork()` system call.

value greater than zero (in fact, it is the actual pid of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then 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 3.11.

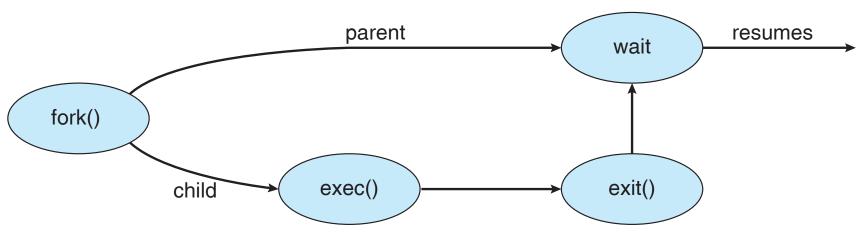

Figure 3.11 Process creation using `fork()` system call.

```
#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 3.12 Creating a separate process using the Win32 API.

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 3.12 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 of 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 handles 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 (as it is in this case), 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 3.10, 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`—and waits for this process 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 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

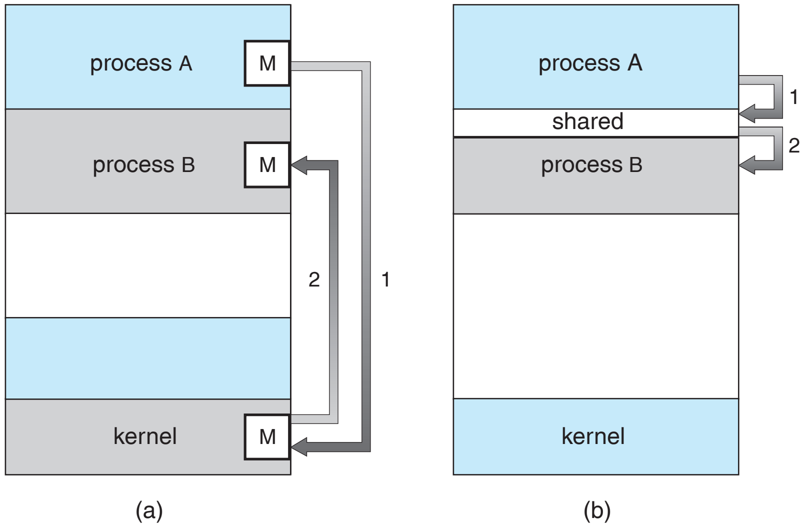

Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

between the cooperating processes. The two communications models are contrasted in Figure 3.13.

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 shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication. 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.

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 3.14 and 3.15, 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.

```
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 3.14 The producer process.

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.

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

```
item nextConsumed;

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

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

Figure 3.15 The consumer process.

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), 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 (say, *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 (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process 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 link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

- **Unbounded capacity**. The queue's 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, 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. 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 3.16 illustrates the POSIX shared-memory API just discussed. 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

```
#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 3.16 C program illustrating POSIX shared-memory API.

defaults to eight messages. The task that creates the mailbox is that mailbox's 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's message queue is initially empty. 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 that message 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, 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 8). 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

The Windows operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows 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 is called the **local procedure-call (LPC)** facility. The LPC in Windows 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. Like Mach, Windows 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 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. 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 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 4KB 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 is shown in Figure 3.17.

It is important to note that the LPC facility in Windows 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

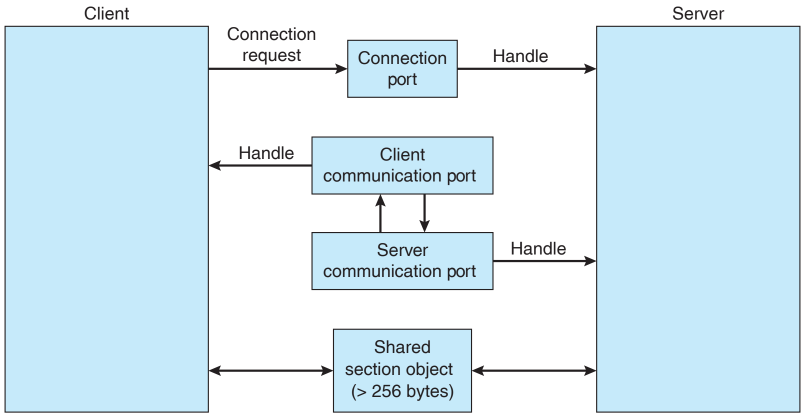

Figure 3.17 Local procedure calls in Windows.

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 two other strategies for communication in client–server systems: sockets and remote procedure calls.

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 its 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 3.18. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

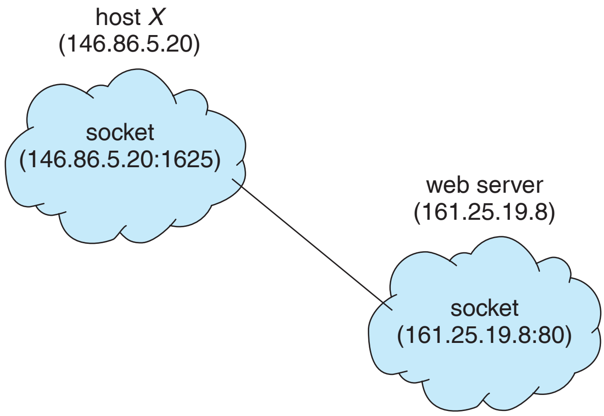

Figure 3.18 Communication using sockets.

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 3.19. 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 3.19 Date server.

Java program shown in Figure 3.20. 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 subsection, we look at remote procedure calls (RPCs), which provide a higher-level method of communication.

```
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 3.20 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*) store the most significant byte first, while other systems (known as *little-endian*) store the least significant byte first. Neither order is "better" per se; rather, the choice is arbitrary within a computer architecture. 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 implemented 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.

For "exactly once," we need to remove the risk that the server will never receive 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 7) 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 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 3.21 shows a sample interaction.

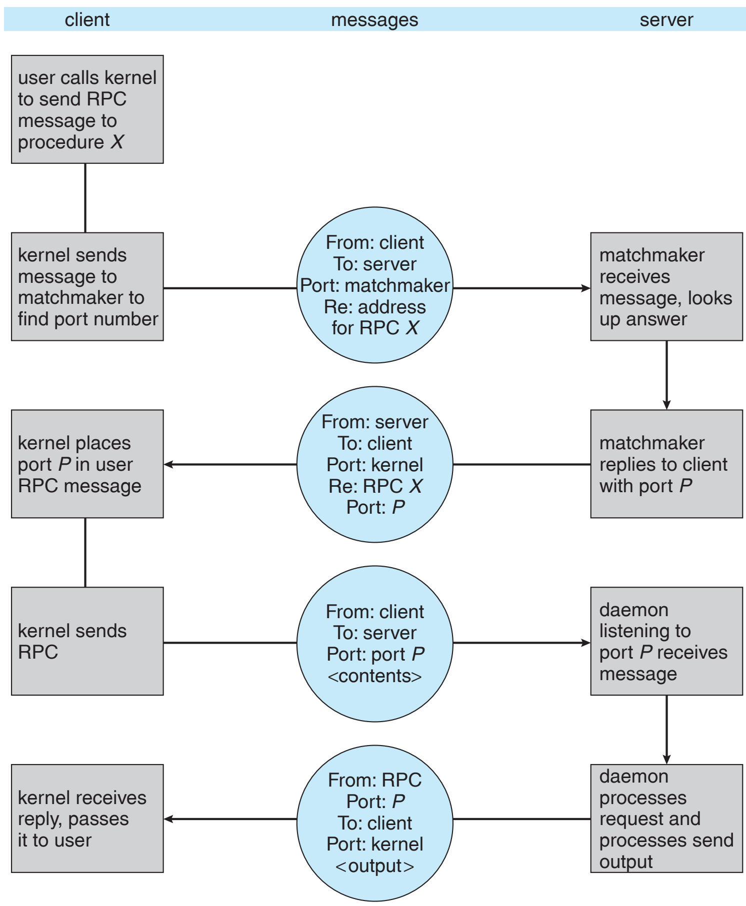

Figure 3.21 Execution of a remote procedure call (RPC).

The RPC scheme is useful in implementing a distributed file system. 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.

3.7 Summary

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 sockets or remote procedure calls (RPCs). 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.

Practice Exercises

3.1 Palm OS provides no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

3.2 The Sun UltraSPARC processor has multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

3.3 When a process creates a new process using the `fork()` operation, which of the following states is shared between the parent process and the child process?

a. Stack

b. Heap

c. Shared memory segments

3.4 With respect to the RPC mechanism, consider the "exactly once" semantic. Does the algorithm for implementing this semantic execute correctly even if the ACK message back to the client is lost due to a network problem? Describe the sequence of messages and discuss whether "exactly once" is still preserved.

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

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.22 How many processes are created?

3.5 Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the "exactly once" semantic for execution of RPCs?

Exercises

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

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

3.8 Construct a process tree similar to Figure 3.9. To obtain process information for the UNIX or Linux system, use the command `ps -ael`. Use the command `man ps` to get more information about the `ps` command. On Windows systems, you will have to use the task manager.

3.9 Including the initial parent process, how many processes are created by the program shown in Figure 3.22?

3.10 Using the program in Figure 3.23, identify the values of `pid` at lines `A`, `B`, `C`, and `D`. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

3.11 Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.

3.12 Using the program shown in Figure 3.24, explain what the output will be at `Line A`.

3.13 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

Programming Problems

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

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

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

int main()
{
pid_t pid, pid1;

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

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      pid1 = getpid();
      printf("child: pid = %d",pid); /* A */
      printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
      pid1 = getpid();
      printf("parent: pid = %d",pid); /* C */
      printf("parent: pid1 = %d",pid1); /* D */
      wait(NULL);
    }

    return 0;
}
```

Figure 3.23 What are the pid values?

Write a C program using the `fork()` system call 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.

3.15 Repeat the preceding exercise, this time using the `CreateProcess()` function 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.

3.16 Modify the date server shown in Figure 3.19 so that it delivers random jokes rather than the current date. Allow the jokes to contain multiple

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

Figure 3.24 What output will be at Line A?

lines. The date client shown in Figure 3.20 can be used to read the multi-line jokes returned by the joke server.

3.17 An echo server echoes back whatever it receives from a client. For example, if a client sends the server the string *Hello there!* the server will respond with the exact data it received from the client—that is, *Hello there!*

Write an echo server using the Java networking API described in Section 3.6.1. This server will wait for a client connection using the `accept()` method. When a client connection is received, the server will loop, performing the following steps:

- Read data from the socket into a buffer.
- Write the contents of the buffer back to the client.

The server will break out of the loop only when it has determined that the client has closed the connection.

The server shown in Figure 3.19 uses the `java.io.BufferedReader` class. `BufferedReader` extends the `java.io.Reader` class, which is used for reading character streams. However, the echo server cannot guarantee that it will read characters from clients; it may receive binary data as well. The class `java.io.InputStream` deals with data at the byte level rather than the character level. Thus, this echo server must use an object that extends `java.io.InputStream`. The `read()` method in the `java.io.InputStream` class returns −1 when the client has closed its end of the socket connection.

3.18 In Exercise 3.14, 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 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` ≤ `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 ≤ 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 as the parent's. 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.

3.19 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 in 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 gives read permission to the owner of the shared-memory segment.

A shared-memory segment 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 indicating an error condition (the global variable `errno` can be accessed to determine the error condition).

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 shared-memory segment:

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

Programming Projects

POSIX Message Passing

This project consists of using POSIX message queues for communicating temperatures between each of four external processes and a central process. The project can be completed on systems that support POSIX message passing, such as UNIX, Linux, and Mac OS X.

Part 1: Overview

Four external processes will communicate temperatures to a central process, which in turn will reply with its own temperature and will indicate whether the entire system has stabilized. Each process will receive its initial temperature upon creation and will recalculate a new temperature according to two formulas:

new external temp =
*(myTemp * 3 + 2 * centralTemp) / 5;*

new central temp =
*(2 * centralTemp + four temps received from external processes) / 6;*

Initially, each external process will send its temperature to the mailbox for the central process. If all four temperatures are exactly the same as those sent by the four processes during the last iteration, the system has stabilized. In this case, the central process will notify each external process that it is now finished (along with the central process itself), and each process will output the final stabilized temperature. If the system has not yet become stable, the central process will send its new temperature to the mailbox for each of the outer processes and await their replies. The processes will continue to run until the temperature has stabilized.

Part 2: The Message-Passing System

Processes can exchange messages by using four system calls: `msgget()`, `msgsnd()`, `msgrcv()`, and `msgctl()`. The `msgget()` function converts a mailbox name to a message queue id, `msqid`. (A mailbox name is an externally known message queue name that is shared among the cooperating processes.) `msqid`, the internal identifier returned by `msgget()`, must be passed to all subsequent system calls using this message queue to facilitate interprocess communication. A typical invocation of `msgget()` is seen below:

```
msqid = msgget(1234, 0600 | IPC_CREAT);
```

The first parameter is the name of the mailbox, and the second parameter instructs the operating system to create the message queue if it does not already exist, with read and write privileges only for processes with the same user id as this process. If a message queue already exists for this mailbox name, `msgget()` returns the `msqid` of the existing mailbox. To avoid attaching to an existing message queue, a process can first attempt to attach to the mailbox by omitting `IPC_CREAT` and then checking the return value from `msgget()`. If `msqid` is negative, an error has occurred during the system call, and the globally accessible variable `errno` can be consulted to determine whether the error occurred because the message queue already exists or for some other reason. If the process determines that the mailbox does not currently exist, it can then create it by including `IPC_CREAT`. (For the current project, this strategy should not be necessary if students are using standalone PCs or are assigned unique ranges of mailbox names by the instructor.)

Once a valid `msqid` has been established, a process can begin to use `msgsnd()` to send messages and `msgrcv()` to receive messages. The messages being sent and received are similar in format to those described in Section 3.5.2, as they include a fixed-length portion at the beginning followed by a variable-length portion. Obviously, the senders and receivers must agree on the format of the messages being exchanged.

Since the operating system specifies one field in the fixed-length portion of every message format, and at least one piece of information will be sent to the receiving process, it is logical to create a data aggregate for each type of message using a `struct`. The first field of any such `struct` must be a `long`, and it will contain the priority of the message. (This project does not use this functionality; we recommend that you simply make the first field in every message equal to the same integral value, such as 2.) Other fields in the messages contain the information to be shared between the communicating processes. Three additional fields are recommended: (1) the temperature being sent, (2) the process number of the external process sending the message (0 for the central process), and (3) a flag that is set to 0 but that the central process will set to 1 when it notices stability. A recommended `struct` appears as follows:

```
struct {
     long priority;
     int temp;
     int pid;
     int stable;
} msgp;
```

Assuming the `msqid` has been established, examples of `msgsnd()` and `msgrcv()` appear as such:

```
int stat, msqid;

stat = msgsnd(msqid, &msgp,
              sizeof(msgp)−sizeof(long), 0);

stat = msgrcv(msqid, &msgp,
              sizeof(msgp)−sizeof(long), 2, 0);
```

The first parameter in both system calls must be a valid `msqid`; otherwise a negative value is returned. (Both functions return the number of bytes sent or received upon successful completion of the operation.) The second parameter is the address of where to find or store the message to be sent or received, followed by the number of information bytes to be sent or received. The final parameter of 0 indicates that the operations will be synchronous and that the sender will block if the message queue is full. (`IPC_NOWAIT` would be used if asynchronous, or nonblocking, operations were desired. Each individual message queue can hold a maximum number of messages—or bytes—so it is possible for the queue to become filled, which is one reason a sender may block when attempting to transmit a message.) The 2 that appears before this final parameter in `msgrcv()` indicates the minimum priority level of the messages the process wishes to receive; the receiver will wait until a message of that priority (or higher) is sent to the `msqid` if this is a synchronous operation.

Once a process is finished using a message queue, it must be removed so that the mailbox can be reused by other processes. Unless it is removed, the message queue—and any messages that have not yet been received—will remain in the storage space that has been provided for this mailbox by the kernel. To remove the message queue and delete any unread messages therein, it is necessary to invoke `msgctl()`, as follows:

```
struct msgid_ds dummyParam;
status = msgctl(msqid, IPC_RMID, &dummyParam);
```

The third parameter is necessary because this function requires it, but it is used only if the programmer wishes to collect some statistics about the usage of the message queue. This is accomplished by substituting IPC_STAT as the second parameter.

All programs should include the following three header files, which are found in /usr/include/sys: ipc.h, types.h, and msg.h. One possibly confusing artifact of the message queue implementation bears mentioning at this point. After a mailbox is removed via msgctl(), any subsequent attempts to create another mailbox with that same name using msgget() will typically generate a different msqid.

Part 3: Creating the Processes

Each external process, as well as the central server, will create its own mailbox with the name $X + i$, where i is a numeric identifier of the external process 1..4 or zero for the central process. Thus, if X were 70, then the central process would receive messages in the mailbox named 70, and it would send its replies to mailboxes 71–74. Outer process 2 would receive in mailbox 72 and would send to mailbox 70, and so forth. Thus, each external process will attach to two mailboxes, and the central process will attach to five. If each process specifies IPC_CREAT when invoking msgget(), the first process that invokes msgget() actually creates the mailbox; subsequent calls to msgget() attach to the existing mailbox. The protocol for removal should be that the mailbox/message queue that each process is listening to should be the only one it removes —via msgctl().)

Each external process will be uniquely identified by a command-line parameter. The first parameter to each external process will be its initial temperature, and the second parameter will be its unique number: 1, 2, 3, or 4. The central server will be passed one parameter—its initial temperature. Assuming the executable name of the external process is external and the central server is central, invoking all five processes will be done as follows:

```
./external 100 1 &
./external 22 2 &
./external 50 3 &
./external 40 4 &
./central 60 &
```

Part 4: Implementation Hints

It might be best to start by sending one message successfully from the central process to a single outer process, and vice versa, before trying to write all the code to solve this problem. It is also wise to check all the return values from the four message queue system calls for possible failed requests and to output a message to the screen after each one successfully completes. The message should indicate what was accomplished and by whom—for instance, "mailbox 71 has been created by outer process 1," "message received by central process from external process 2," and so forth. These messages can be removed or commented out after the problem is solved. Processes should also verify that they were passed the correct number of command-line parameters (via the argc parameter in main()). Finally, extraneous messages residing in a queue can cause a collection of cooperating processes that function correctly to appear erroneous. For that reason, it is wise to remove all mailboxes relevant to this project to ensure that mailboxes are empty before the processes begin. The easiest way to do this is to use the `ipcs` command to list all message queues and the `ipcrm` command to remove existing message queues. The `ipcs` command lists the `msqid` of all message queues on the system. Use `ipcrm` to remove message queues according to their `msqid`. For example, if `msqid` 163845 appears with the output of `ipcs`, it can be deleted with the following command:

```
ipcrm -q 163845
```

Bibliographical Notes

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

Details of interprocess communication in UNIX systems are presented by Gray [1997]. Barrera [1991] and Vahalia [1996] describe interprocess communication in the Mach system. Russinovich and Solomon [2009], Solomon and Russinovich [2000], and Stevens [1999] outline interprocess communication in WinSEVEN, Windows 2000 and UNIX respectively. Hart [2005] covers Windows systems programming in detail.

The implementation of RPCs is discussed by Birrell and Nelson [1984]. Shrivastava and Panzieri [1982] describes the design of a reliable RPC mechanism, and Tay and Ananda [1990] presents a survey of RPCs. Stankovic [1982] and Staunstrup [1982] discuss procedure calls versus message-passing communication. Harold [2005] provides coverage of socket programming in Java.