2

PROCESSES

We are now about to embark on a detailed study of how operating systems, in
general, and MINIX, in particular, are designed and constructed. The most central
concept in any operating system is the \textit{process}: an abstraction of a running pro-
gram. Everything else hinges on this concept, and it is important that the operat-
ing system designer (and student) know what a process is as early as possible.

2.1 \textsc{Introduction to Processes}

All modern computers can do several things at the same time. While running
a user program, a computer can also be reading from a disk and outputting text to
a screen or printer. In a multiprogramming system, the CPU also switches from
program to program, running each for tens or hundreds of milliseconds. While,
strictly speaking, at any instant of time, the CPU is running only one program, in
the course of 1 second, it may work on several programs, thus giving the users the
illusion of parallelism. Sometimes people speak of \textit{pseudoparallelism} to mean
this rapid switching back and forth of the CPU between programs, to contrast it
with the true hardware parallelism of \textit{multiprocessor} systems (which have two or
more CPUs sharing the same physical memory). Keeping track of multiple, par-
allel activities is hard for people to do. Therefore, operating system designers over
the years have evolved a model (sequential processes) that makes parallelism
easier to deal with. That model and its uses are the subject of this chapter.
2.1.1 The Process Model

In this model, all the runnable software on the computer, often including the operating system, is organized into a number of sequential processes, or just processes for short. A process is just an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called multiprogramming, as we saw in the previous chapter.

In Fig. 2-1(a) we see a computer multiprogramming four programs in memory. In Fig. 2-1(b) we see four processes, each with its own flow of control (i.e., its own program counter), and each one running independently of the other ones. In Fig. 2-1(c) we see that viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

![Diagram of processes and program counters](image)

Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.

With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform, and probably not even reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing. Consider, for example, an I/O process that starts a streamer tape to restore backed up files, executes an idle loop 10,000 times to let it get up to speed, and then issues a command to read the first record. If the CPU decides to switch to another process during the idle loop, the tape process might not run again until after the first record was already past the read head. When a process has critical real-time requirements like this, that is, particular events must occur within a specified number of milliseconds,
special measures must be taken to ensure that they do occur. Normally, however, most processes are not affected by the underlying multiprogramming of the CPU or the relative speeds of different processes.

The difference between a process and a program is subtle, but crucial. An analogy may help make this point clearer. Consider a culinary-minded computer scientist who is baking a birthday cake for his daughter. He has a birthday cake recipe and a kitchen well-stocked with the necessary input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the recipe is the program (i.e., an algorithm expressed in some suitable notation), the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake.

Now imagine that the computer scientist's son comes running in crying, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher priority process (administering medical care), each having a different program (recipe vs. first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one.

**Process Hierarchies**

Operating systems that support the process concept must provide some way to create all the processes needed. In very simple systems, or in systems designed for running only a single application (e.g., controlling a device in real time), it may be possible to have all the processes that will ever be needed be present when the system comes up. In most systems, however, some way is needed to create and destroy processes as needed during operation. In MINIX, processes are created by the FORK system call, which creates an identical copy of the calling process. The child process can also execute FORK, so it is possible to get a whole tree of processes. In other operating systems, system calls exist to create a process, load its memory, and start it running. Whatever the exact nature of the system call, processes need a way to create other processes. Note that each process has one parent but zero, one, two, or more children.

As a simple example of how process trees are used, let us look at how MINIX initializes itself when it is started. A special process, called init, is present in the boot image. When it starts running, it reads a file telling how many terminals there are. Then it forks off one new process per terminal. These processes wait
for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes, and so forth. Thus, all the processes in the whole system belong to a single tree, with init at the root. (The code for init is not listed in the book; neither is the shell. The line had to be drawn somewhere.)

Process States

Although each process is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. One process may generate some output that another process uses as input. In the shell command

```
cat chapter1 chapter2 chapter3 | grep tree
```

the first process, running `cat`, concatenates and outputs three files. The second process, running `grep`, selects all lines containing the word “tree.” Depending on the relative speeds of the two processes (which depends on both the relative complexity of the programs and how much CPU time each one has had), it may happen that `grep` is ready to run, but there is no input waiting for it. It must then block until some input is available.

When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. It is also possible for a process that is conceptually ready and able to run to be stopped because the operating system has decided to allocate the CPU to another process for a while. These two conditions are completely different. In the first case, the suspension is inherent in the problem (you cannot process the user’s command line until it has been typed). In the second case, it is a technicality of the system (not enough CPUs to give each process its own private processor). In Fig. 2-2 we see a state diagram showing the three states a process may be in:

1. Running (actually using the CPU at that instant).
2. Ready (runnable; temporarily stopped to let another process run).
3. Blocked (unable to run until some external event happens).

Logically, the first two states are similar. In both cases the process is willing to run, only in the second one, there is temporarily no CPU available for it. The third state is different from the first two in that the process cannot run, even if the CPU has nothing else to do.

Four transitions are possible among these three states, as shown. Transition 1 occurs when a process discovers that it cannot continue. In some systems the process must execute a system call, BLOCK, to get into blocked state. In other systems, including MINIX, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically blocked.

Transitions 2, 3, and 4 occur when the CPU is available. When the CPU is available, the CPU is made available to processes that have not been blocked. Transitions 2, 3, and 4 are not instantaneous, the CPU is not always available.

Using the operating system's scheduling algorithm, the system may schedule processes that are running, ready, or blocked. When the CPU is available, the system selects one from among the ready processes, and runs it. The system may also schedule blocked processes that are not executable, and schedules them when they come back (e.g., when their disk read or their terminal is ready).
Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Transitions 2 and 3 are caused by the process scheduler, a part of the operating system, without the process even knowing about them. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again. The subject of scheduling, that is, deciding which process should run when and for how long, is an important one; we will look at it later in this chapter. Many algorithms have been devised to try to balance the competing demands of efficiency for the system as a whole and fairness to individual processes.

Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that instant, transition 3 will be triggered immediately, and the process will start running. Otherwise it may have to wait in ready state for a little while until the CPU is available.

Using the process model, it becomes much easier to think about what is going on inside the system. Some of the processes run programs that carry out commands typed in by a user. Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive. When a disk interrupt occurs, the system makes a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt. Thus, instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk block has been read or the character typed, the process waiting for it is unblocked and is eligible to run again.

This view gives rise to the model shown in Fig. 2-3. Here the lowest level of the operating system is the scheduler, with a variety of processes on top of it. All the interrupt handling and details of actually starting and stopping processes are hidden away in the scheduler, which is actually quite small. The rest of the operating system is nicely structured in process form. The model of Fig. 2-3 is used in MINIX, with the understanding that “scheduler” really means not just process scheduling, but also interrupt handling and all the interprocess communication. Nevertheless, to a first approximation, it does show the basic structure.
2.1.2 Implementation of Processes

To implement the process model, the operating system maintains a table (an array of structures), called the **process table**, with one entry per process. This entry contains information about the process' state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from **running** to **ready** state so that it can be restarted later as if it had never been stopped.

In MINIX the process management, memory management, and file management are each handled by separate modules within the system, so the process table is partitioned, with each module maintaining the fields that it needs. Figure 2-4 shows some of the more important fields. The fields in the first column are the only ones relevant to this chapter. The other two columns are provided just to give an idea of what information is needed elsewhere in the system.

Now that we have looked at the process table, it is possible to explain a little more about how the illusion of multiple sequential processes is maintained on a machine with one CPU and many I/O devices. What follows is technically a description of how the "scheduler" of Fig. 2-3 works in MINIX but most modern operating systems work essentially the same way. Associated with each I/O device class (e.g., floppy disks, hard disks, timers, terminals) is a location near the bottom of memory called the **interrupt vector**. It contains the address of the interrupt service procedure. Suppose that user process 3 is running when a disk interrupt occurs. The program counter, program status word, and possibly one or more registers are pushed onto the (current) stack by the interrupt hardware. The computer then jumps to the address specified in the disk interrupt vector. That is all the hardware does. From here on, it is up to the software.

The interrupt service procedure starts out by saving all the registers in the process table entry for the current process. The current process number and a pointer to its entry are kept in global variables so they can be found quickly. Then the information deposited by the interrupt is removed from the stack, and the stack pointer is set to a temporary stack used by the process handler. Actions such as saving the registers and setting the stack pointer cannot even be expressed in C, so
they are performed by a small assembly language routine. When this routine is finished, it calls a C procedure to do the rest of the work.

Interprocess communication in MINIX is via messages, so the next step is to build a message to be sent to the disk process, which will be blocked waiting for it. The message says that an interrupt occurred, to distinguish it from messages from user processes requesting disk blocks to be read and things like that. The state of the disk process is now changed from blocked to ready and the scheduler is called. In MINIX, different processes have different priorities, to give better service to I/O device handlers than to user processes. If the disk process is now the highest priority runnable process, it will be scheduled to run. If the process that was interrupted is just as important or more so, then it will be scheduled to run again, and the disk process will have to wait a little while.

Either way, the C procedure called by the assembly language interrupt code now returns, and the assembly language code loads up the registers and memory map for the now-current process and starts it running. Interrupt handling and scheduling are summarized in Fig. 2-5. It is worth noting that the details vary slightly from system to system.

### 2.1.3 Threads

In a traditional process, of the type we have just studied, there is a single thread of control and a single program counter in each process. However, in some modern operating systems, support is provided for multiple threads of control.
within a process. These threads of control are usually just called **threads**, or occasionally **lightweight processes**.

In Fig. 2-6(a) we see three traditional processes. Each process has its own address space and a single thread of control. In contrast, in Fig. 2-6(b) we see a single process with three threads of control. Although in both cases we have three threads, in Fig. 2-6(a) each of them operates in a different address space, whereas in Fig. 2-6(b) all three of them share the same address space.

![Figure 2-6](image)

**Figure 2-6.** (a) Three processes each with one thread. (b) One process with three threads.

As an example of where multiple threads might be used, consider a file server process. It receives requests to read and write files and sends back the requested data or accepts updated data. To improve performance, the server maintains a cache of recently used files in memory, reading from the cache and writing to the cache when possible.

This situation lends itself well to the model of Fig. 2-6(b). When a request comes in, it is handed to a thread for processing. If that thread blocks part way through waiting for a disk transfer, other threads are still able to run, so the server
can keep processing new requests even while disk I/O is taking place. The model of Fig. 2-6(a) is not suitable, because it is essential that all file server threads access the same cache, and the three threads of Fig. 2-6(a) do not share the same address space and thus cannot share the same memory cache.

Another example of where threads are useful is in browsers for the World Wide Web, such as Netscape and Mosaic. Many Web pages contain multiple small images. For each image on a Web page, the browser must set up a separate connection to the page’s home site and request the image. A great deal of time is wasted establishing and releasing all these connections. By having multiple threads within the browser, many images can be requested at the same time, greatly speeding up performance in most cases, since with small images, the setup time is the limiting factor, not the speed of the transmission line.

When multiple threads are present in the same address space, a few of the fields of Fig. 2-4 are not per process, but per thread, so a separate thread table is needed, with one entry per thread. Among the per-thread items are the program counter, registers, and state. The program counter is needed because threads, like processes, can be suspended and resumed. The registers are needed because when threads are suspended, their registers must be saved. Finally, threads, like processes, can be in running, ready, or blocked state.

In some systems, the operating system is not aware of the threads. In other words, they are managed entirely in user space. When a thread is about to block, for example, it chooses and starts its successor before stopping. Several user-level threads packages are in common use, including the POSIX P-threads and Mach C-threads packages.

In other systems, the operating system is aware of the existence of multiple threads per process. So when a thread blocks, the operating system chooses the next one to run, either from the same process or a different one. To do scheduling, the kernel must have a thread table that lists all the threads in the system, analogous to the process table.

Although these two alternatives may seem equivalent, they differ considerably in performance. Switching threads is much faster when thread management is done in user space than when a kernel call is needed. This fact argues strongly for doing thread management in user space. On the other hand, when threads are managed entirely in user space and one thread blocks (e.g., waiting for I/O or a page fault to be handled), the kernel blocks the entire process, since it is not even aware that other threads exist. This fact argues strongly for doing thread management in the kernel. As a consequence, both systems are in use, and various hybrid schemes have been proposed as well (Anderson et al., 1992).

No matter whether threads are managed by the kernel or in user space, they introduce a raft of problems that must be solved and which change the programming model appreciably. To start with, consider the effects of the FORK system call. If the parent process has multiple threads, should the child also have them? If not, the process may not function properly, since all of them may be essential.
However, if the child process gets as many threads as the parent, what happens if a thread was blocked on a READ call, say, from the keyboard? Are two threads now blocked on the keyboard? When a line is typed, do both threads get a copy of it? Only the parent? Only the child? The same problem exists with open network connections.

Another class of problems is related to the fact that threads share many data structures. What happens if one thread closes a file while another one is still reading from it? Suppose that one thread notices that there is too little memory and starts allocating more memory. Then, part way through, a thread switch occurs, and the new thread also notices that there is too little memory and also starts allocating more memory. Does the allocation happen once or twice? In nearly all systems that were not designed with threads in mind, the libraries (such as the memory allocation procedure) are not reentrant, and will crash if a second call is made while the first one is still active.

Another problem relates to error reporting. In UNIX, after a system call, the status of the call is put into a global variable, *errno*. What happens if a thread makes a system call, and before it is able to read *errno*, another thread makes a system call, wiping out the original value?

Next, consider signals. Some signals are logically thread specific, whereas others are not. For example, if a thread calls ALARM, it makes sense for the resulting signal to go to the thread that made the call. When the kernel is aware of threads, it can usually make sure the right thread gets the signal. When the kernel is not aware of threads, somehow the threads package must keep track of alarms. An additional complication for user-level threads exists when (as in UNIX) a process may only have one alarm at a time pending and several threads call ALARM independently.

Other signals, such as keyboard interrupt, are not thread specific. Who should catch them? One designated thread? All the threads? A newly created thread? All these solutions have problems. Furthermore, what happens if one thread changes the signal handlers without telling other threads?

One last problem introduced by threads is stack management. In many systems, when stack overflow occurs, the kernel just provides more stack, automatically. When a process has multiple threads, it must also have multiple stacks. If the kernel is not aware of all these stacks, it cannot grow them automatically upon stack fault. In fact, it may not even realize that a memory fault is related to stack growth.

These problems are certainly not insurmountable, but they do show that just introducing threads into an existing system without a fairly substantial system redesign is not going to work at all. The semantics of system calls have to be redefined and libraries have to be rewritten, at the very least. And all of these things must be done in such a way as to remain backward compatible with existing programs for the limiting case of a process with only one thread. For additional information about threads, see (Hauser et al., 1993; and Marsh et al., 1991).

2.2 Process Creation

Proper process creation is complex. For example, in UNIX, a child process is created by the parent process, and the parent process sends the child process to the fork and exec functions. Each function has the following behaviors

**Process Creation**

When a process creates a child process, it must ensure two things: (1) the child shares the same critical data structures and (2) the child has the same memory layout. The first is easily accomplished if procfs is used, as described above. Reducing system calls to a minimum might start the process.

2.2.1 Delayed Memory Layouts

In general, it is possible for a child processes to be in memory at the time it is created. This does not mean that the parent process has to see how memory is laid out, but contador must be entered to, as it is a **daemon**. Thus, the child is not ready for the parent as it performs.

Image data is always loaded into the child's address space, much like the parent. However, some of this data might not be loaded immediately. For example, if the child does not have enough memory, it might not be loaded at all. However, if the parent is not full, the parent might load the data. Consequently, the data is then available when the child is started.

In just about any system, such as the **next_free** data structure, the child is not always ready. This is a little confusing. However, **next_free** data structure is a key part of the process. A process is a collection of data structures.
2.2 INTERPROCESS COMMUNICATION

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts. In the following sections we will look at some of the issues related to this InterProcess Communication or IPC.

Very briefly, there are three issues here. The first was alluded to above: how one process can pass information to another. The second has to do with making sure two or more processes do not get into each other’s way when engaging in critical activities (suppose two processes each try to grab the last 100K of memory). The third concerns proper sequencing when dependencies are present: if process A produces data and process B prints it, B has to wait until A has produced some data before starting to print. We will examine all three of these issues starting in the next section.

2.2.1 Race Conditions

In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory or it may be a shared file; the location of the shared memory does not change the nature of the communication or the problems that arise. To see how interprocess communication works in practice, let us consider a simple but common example, a print spooler. When a process wants to print a file, it enters the file name in a special spooler directory. Another process, the printer daemon, periodically checks to see if there are any files to be printed, and if there are it prints them and then removes their names from the directory.

Imagine that our spooler directory has a large (potentially infinite) number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables, out, which points to the next file to be printed, and in, which points to the next free slot in the directory. These two variables might well be kept on a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes A and B decide they want to queue a file for printing. This situation is shown in Fig. 2-7.

In jurisdictions where Murphy’s law† is applicable, the following might happen. Process A reads in and stores the value, 7, in a local variable called next_free_slot. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B. Process B also reads in,

† If something can go wrong, it will.
and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things.

Eventually, process $A$ runs again, starting from the place it left off. It looks at $\text{next\_free\_slot}$, finds a 7 there, and writes its file name in slot 7, erasing the name that process $B$ just put there. Then it computes $\text{next\_free\_slot} + 1$, which is 8, and sets in to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process $B$ will never get any output.

Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Debugging programs containing race conditions is no fun at all. The results of most test runs are fine, but once in a rare while something weird and unexplained happens.

### 2.2.2 Critical Sections

How do we avoid race conditions? The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is mutual exclusion—some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. The difficulty above occurred because process $B$ started using one of the shared variables before process $A$ was finished with it. The choice of appropriate primitive operations for achieving mutual exclusion is a major design issue in any operating system, and a subject that we will examine in great detail in the following sections.

The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that involve accessing shared memory. There are always critical regions, where the processes use the shared conditions.

Although our spooler assumes that there will be no having parts. We need for

1. Need
2. Need
3. Need
4. Need

### 2.2.3 Mutual Exclusion

In this section, so far, in this section, no

Disabling interrupts

The system is written in such a way that once entering a critical section, the interrupts are turned off. This is because if an interrupt ever occurs while we are in a critical section, we may never turn it on again, and the system may crash. If a process affects only one file, it may be safe to disable interrupts.

On the other hand, if a process has a critical section, it has a critical section, and therefore, it may require disabling of interrupts. If a process interrupts the critical section, it may be safe to disable interrupts.

SEC. 2.2

things that involve accessing shared memory. There are always critical regions, where the processes use the shared conditions. Although our spooler assumes that there will be no having parts. We need for
things that do not lead to race conditions. However, sometimes a process may be accessing shared memory or files, or doing other critical things that can lead to races. That part of the program where the shared memory is accessed is called the critical region or critical section. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions.

Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

2.2.3 Mutual Exclusion with Busy Waiting

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

Disabling Interrupts

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it, and never turned them on again? That could be the end of the system. Furthermore, if the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

On the other hand, it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur. The conclusion is: disabling interrupts is often
a useful technique within the operating system itself but is not appropriate as a
general mutual exclusion mechanism for user processes.

**Lock Variables**

As a second attempt, let us look for a software solution. Consider having a
single, shared, (lock) variable, initially 0. When a process wants to enter its criti-
cal region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters
the critical region. If the lock is already 1, the process just waits until it becomes
0. Thus, a 0 means that no process is in its critical region, and a 1 means that
some process is in its critical region.

Unfortunately, this idea contains exactly the same fatal flaw that we saw in
the spooler directory. Suppose that one process reads the lock and sees that it is 0.
Before it can set the lock to 1, another process is scheduled, runs, and sets the lock
to 1. When the first process runs again, it will also set the lock to 1, and two
processes will be in their critical regions at the same time.

Now you might think that we could get around this problem by first reading
out the lock value, then checking it again just before storing into it, but that really
does not help. The race now occurs if the second process modifies the lock just
after the first process has finished its second check.

**Strict Alternation**

A third approach to the mutual exclusion problem is shown in Fig. 2-8. This
program fragment, like nearly all the others in this book, is written in C. C was
chosen here because real operating systems are commonly written in C (or occa-
sionally C++), but hardly ever in languages like Modula 2 or Pascal.

```c
while (TRUE) {
  while (turn != 0) /* wait */;
  critical_region();
  turn = 1;
  noncritical_region();
}
```

(a)

```c
while (TRUE) {
  while (turn != 1) /* wait */;
  critical_region();
  turn = 0;
  noncritical_region();
}
```

(b)

**Figure 2-8.** A proposed solution to the critical region problem.

In Fig. 2-8, the integer variable turn, initially 0, keeps track of whose turn it is
to enter the critical region and examine or update the shared memory. Initially,
process 0 inspects turn, finds it to be 0, and enters its critical region. Process 1
also finds it to be 0 and therefore sits in a tight loop continually testing turn to see
when it becomes 1. Continuously testing a variable until some value appears is

called a spin. The CPU is very busy and may waste time, and only the process
waiting on the lock will return to its work. When the lock is available, the
process that was forced to spin around quickly returns, allows the process
with turn to 0 to continue, and the second process begins its critical region.
Put differently, turn is a stricter form of mutual exclusion.

This solves the problem of a process going back to the beginning of
the directory, as was done in the earlier example.

In fact, giving each user a number

(their turn) allows more than one

process to be in the critical region

simultaneously.

**Petersen’s Algorithm**

By now you have no doubt that a

warning about the dangers of mutual exclusion.

In Fig. 2-8, the use of mutual exclusion has been illustrated. A number is shown in Fig. 2-9, which

defines a priority level, a process

or subsecond.

Before entering the critical region,

each process sets its number

character. This number is considered

finished when it is set and stays at

it is done.

Let us say that the process that

a critical region. Multiple process

returns is
called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used.

When process 0 leaves the critical region, it sets \textit{turn} to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so both processes are in their noncritical regions, with \textit{turn} set to 0. Now process 0 executes its whole loop quickly, coming back to its noncritical region with \textit{turn} set to 1. At this point, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because \textit{turn} is 1 and process 1 is busy with its noncritical region. Put differently, taking turns is not a good idea when one of the processes is much slower than the other.

This situation violates condition 3 set out above: process 0 is being blocked by a process not in its critical region. Going back to the spooler directory discussed above, if we now associate the critical region with reading and writing the spooler directory, process 0 would not be allowed to print another file because process 1 was doing something else.

In fact, this solution requires that the two processes strictly alternate in entering their critical regions, for example, in spooling files. Neither one would be permitted to spool two in a row. While this algorithm does avoid all races, it is not really a serious candidate as a solution because it violates condition 3.

**Peterson's Solution**

By combining the idea of taking turns with the idea of lock variables and warning variables, a Dutch mathematician, T. Dekker, was the first one to devise a software solution to the mutual exclusion problem that does not require strict alternation. For a discussion of Dekker's algorithm, see (Dijkstra, 1965).

In 1981, G.L. Peterson discovered a much simpler way to achieve mutual exclusion, thus rendering Dekker's solution obsolete. Peterson's algorithm is shown in Fig. 2-9. This algorithm consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used. However, to save space, we will not show the prototypes in this or subsequent examples.

Before using the shared variables (i.e., before entering its critical region), each process calls \textit{enter\_region} with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls \textit{leave\_region} to indicate that it is done and to allow the other process to enter, if it so desires.

Let us see how this solution works. Initially neither process is in its critical region. Now process 0 calls \textit{enter\_region}. It indicates its interest by setting its array element and sets \textit{turn} to 0. Since process 1 is not interested, \textit{enter\_region} returns immediately. If process 1 now calls \textit{enter\_region}, it will hang there until
#define FALSE   0
#define TRUE    1
#define N       2     /* number of processes */

int turn;        /* whose turn is it? */
int interested[N];     /* all values initially 0 (FALSE) */

void enter_region(int process);    /* process is 0 or 1 */
{ int other;                        /* number of the other process */

  other = 1 - process;          /* the opposite of process */
  interested[process] = TRUE;  /* show that you are interested */
  turn = process;               /* set flag */
  while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process)    /* process: who is leaving */
{ interested[process] = FALSE;    /* indicate departure from critical region */
}

Figure 2-9. Peterson's solution for achieving mutual exclusion.

interested[0] goes to FALSE, an event that only happens when process 0 calls
leave_region to exit the critical region.

Now consider the case that both processes call enter_region almost simultaneously. Both will store their process number in turn. Whichever store is done last is the one that counts; the first one is lost. Suppose that process 1 stores last, so turn is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region.

The TSL Instruction

Now let us look at a proposal that requires a little help from the hardware. Many computers, especially those designed with multiple processors in mind, have an instruction TEST AND SET LOCK (TSL) that works as follows. It reads the contents of the memory word into a register and then stores a nonzero value at that memory address. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.
To use the TSL instruction, we will use a shared variable, \textit{lock}, to coordinate access to shared memory. When \textit{lock} is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets \textit{lock} back to 0 using an ordinary MOVE instruction.

How can this instruction be used to prevent two processes from simultaneously entering their critical regions? The solution is given in Fig. 2-10. There a four-instruction subroutine in a fictitious (but typical) assembly language is shown. The first instruction copies the old value of \textit{lock} to the register and then sets \textit{lock} to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. Clearing the lock is simple. The program just stores a 0 in \textit{lock}. No special instructions are needed.

\begin{verbatim}
enter_region:
  tsl register,lock        I copy lock to register and set lock to 1
  cmp register,#0          I was lock zero?
  jne enter_region         I if it was non zero, lock was set, so loop
  ret                       I return to caller; critical region entered

leave_region:
  move lock,#0             I store a 0 in lock
  ret                       I return to caller
\end{verbatim}

\textbf{Figure 2-10.} Setting and clearing locks using TSL.

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls \textit{enter\_region}, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls \textit{leave\_region}, which stores a 0 in \textit{lock}. As with all solutions based on critical regions, the processes must call \textit{enter\_region} and \textit{leave\_region} at the correct times for the method to work. If a process cheats, the mutual exclusion will fail.

\subsection*{2.2.4 Sleep and Wakeup}

Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting. In essence, what these solutions do is this: when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.

Not only does this approach waste CPU time, but it can also have unexpected effects. Consider a computer with two processes, \textit{H}, with high priority and \textit{L},
with low priority. The scheduling rules are such that $H$ is run whenever it is in ready state. At a certain moment, with $L$ in its critical region, $H$ becomes ready to run (e.g., an I/O operation completes). $H$ now begins busy waiting, but since $L$ is never scheduled while $H$ is running, $L$ never gets the chance to leave its critical region, so $H$ loops forever. This situation is sometimes referred to as the priority inversion problem.

Now let us look at some interprocess communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the pair SLEEP and WAKEUP. SLEEP is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The WAKEUP call has one parameter, the process to be awakened. Alternatively, both SLEEP and WAKEUP each have one parameter, a memory address used to match up SLEEPs with WAKEUPS.

**The Producer-Consumer Problem**

As an example of how these primitives can be used, let us consider the producer-consumer problem (also known as the bounded buffer problem). Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. (It is also possible to generalize the problem to have $m$ producers and $n$ consumers, but we will only consider the case of one producer and one consumer because this assumption simplifies the solution).

Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory. To keep track of the number of items in the buffer, we will need a variable, `count`. If the maximum number of items the buffer can hold is $N$, the producer's code will first test to see if `count` is $N$. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment `count`.

The consumer's code is similar: first test `count` to see if it is 0. If it is, go to sleep; if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be sleeping, and if not, wakes it up. The code for both producer and consumer is shown in Fig. 2-11.

To express system calls such as SLEEP and WAKEUP in C, we will show them as calls to library routines. They are not part of the standard C library but presumably would be available on a system that actually had these system calls. The procedures `enter_item` and `remove_item`, which are not shown, handle the bookkeeping of putting items into the buffer and taking items out of the buffer.
```c
#define N 100
int count = 0;

void producer(void)
{
    while (TRUE) { /* repeat forever */
        produce_item(); /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        enter_item(); /* put item in buffer */
        count = count + 1; /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}

void consumer(void)
{
    while (TRUE) { /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, go to sleep */
        remove_item(); /* take item out of buffer */
        count = count - 1; /* decrement count of items in buffer */
        if (count == N-1) wakeup(producer); /* was buffer full? */
        consume_item(); /* print item */
    }
}
```

Figure 2.11. The producer-consumer problem with a fatal race condition.

Now let us get back to the race condition. It can occur because access to `count` is unconstrained. The following situation could possibly occur. The buffer is empty and the consumer has just read `count` to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer enters an item in the buffer, increments `count`, and notices that it is now 1. Reasoning that `count` was just 0, and thus the consumer must be sleeping, the producer calls `wakeup` to wake the consumer up.

Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of `count` it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a `wakeup` sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work. A quick fix is to modify the rules to add a `wakeup waiting bit` to the picture. When a `wakeup` is sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the `wakeup waiting bit` is on, it will be turned off, but the process will stay awake. The `wakeup waiting bit` is a piggy bank for `wakeup` signals.
While the wakeup waiting bit saves the day in this simple example, it is easy
to construct examples with three or more processes in which one wakeup waiting
bit is insufficient. We could make another patch and add a second wakeup waiting
bit, or maybe 8 or 32 of them, but in principle the problem is still there.

2.2.5 Semaphores

This was the situation in 1965, when E. W. Dijkstra (1965) suggested using an
integer variable to count the number of wakeups saved for future use. In his pro-
posal, a new variable type, called a semaphore, was introduced. A semaphore
could have the value 0, indicating that no wakeups were saved, or some positive
value if one or more wakeups were pending.

Dijkstra proposed having two operations, DOWN and UP (generalizations of
SLEEP and WAKEUP, respectively). The DOWN operation on a semaphore checks
to see if the value is greater than 0. If so, it decrements the value (i.e., uses up
one stored wakeup) and just continues. If the value is 0, the process is put to sleep
without completing the DOWN for the moment. Checking the value, changing it,
and possibly going to sleep is all done as a single, indivisible, atomic action. It is
guaranteed that once a semaphore operation has started, no other process can
access the semaphore until the operation has completed or blocked. This atomic-
ity is absolutely essential to solving synchronization problems and avoiding race
conditions.

The UP operation increments the value of the semaphore addressed. If one or
more processes were sleeping on that semaphore, unable to complete an earlier
DOWN operation, one of them is chosen by the system (e.g., at random) and is
allowed to complete its DOWN. Thus, after an UP on a semaphore with processes
sleeping on it, the semaphore will still be 0, but there will be one fewer process
sleeping on it. The operation of incrementing the semaphore and waking up one
process is also indivisible. No process ever blocks doing an UP, just as no process
ever blocks doing a WAKEUP in the earlier model.

As an aside, in Dijkstra's original paper, he used the names P and V instead of
DOWN and UP, respectively, but since these have no mnemonic significance to
people who do not speak Dutch (and only marginal significance to those who do),
we will use the terms DOWN and UP instead. These were first introduced in Algol
68.

Solving the Producer-Consumer Problem using Semaphores

Semaphores solve the lost-wakeup problem, as shown in Fig. 2-12. It is essen-
tial that they be implemented in an indivisible way. The normal way is to
implement UP and DOWN as system calls, with the operating system briefly dis-
abling all interrupts while it is testing the semaphore, updating it, and putting
the process to sleep, if necessary. As all of these actions take only a few instruc-

SEC. 2.2.5
no harm is done in disabling interrupts. If multiple CPUs are being used, each
semaphore should be protected by a lock variable, with the TSL instruction used to
make sure that only one CPU at a time examines the semaphore. Be sure you
understand that using TSL to prevent several CPUs from accessing the semaphore
at the same time is quite different from busy waiting by the producer or consumer
waiting for the other to empty or fill the buffer. The semaphore operation will
only take a few microseconds, whereas the producer or consumer might take arbi-
trarily long.

```c
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        produce_item(&item);
        down(&empty);
        down(&mutex);
        enter_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        remove_item(&item);
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Figure 2-12. The producer-consumer problem using semaphores.
This solution uses three semaphores: one called full for counting the number of slots that are full, one called empty for counting the number of slots that are empty, and one called mutex to make sure the producer and consumer do not access the buffer at the same time. Full is initially 0, empty is initially equal to the number of slots in the buffer, and mutex is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called binary semaphores. If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.

Now that we have a good interprocess communication primitive at our disposal, let us go back and look at the interrupt sequence of Fig. 2-5 again. In a system using semaphores, the natural way to hide interrupts is to have a semaphore, initially set to 0, associated with each I/O device. Just after starting an I/O device, the managing process does a down on the associated semaphore, thus blocking immediately. When the interrupt comes in, the interrupt handler then does an up on the associated semaphore, which makes the relevant process ready to run again. In this model, step 6 in Fig. 2-5 consists of doing an up on the device’s semaphore, so that in step 7 the scheduler will be able to run the device manager. Of course, if several processes are now ready, the scheduler may choose to run an even more important process next. We will look at how scheduling is done later in this chapter.

In the example of Fig. 2-12, we have actually used semaphores in two different ways. This difference is important enough to make explicit. The mutex semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. This mutual exclusion is required to prevent chaos.

The other use of semaphores is for synchronization. The full and empty semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and the consumer stops running when it is empty. This use is different from mutual exclusion.

Although semaphores have been around for more than a quarter of a century, people are still doing research about their use. As an example, see (Tai and Carver, 1996).

2.2.6 Monitors

With semaphores interprocess communication looks easy, right? Forget it. Look closely at the order of the downs before entering or removing items from the buffer in Fig. 2-12. Suppose that the two downs in the producer’s code were reversed in order, so mutex was decremented before empty instead of after it. If the buffer were completely full, the producer would block, with mutex set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a
DOWN on mutex, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a \textbf{deadlock}. We will study deadlocks in detail in Chap. 3.

This problem is pointed out to show how careful you must be when using semaphores. One subtle error and everything comes to a grinding halt. It is like programming in assembly language, only worse, because the errors are race conditions, deadlocks, and other forms of unpredictable and irreproducible behavior.

To make it easier to write correct programs, Hoare (1974) and Brinch Hansen (1975) proposed a higher level synchronization primitive called a \textbf{monitor}. Their proposals differed slightly, as described below. A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor. Figure 2.13 illustrates a monitor written in an imaginary language, pidgin Pascal.

\begin{verbatim}
monitor example
  integer i;
  condition c;

  procedure producer(x);
  ...
  end;

  procedure consumer(x);
  ...
  end;
end monitor:
\end{verbatim}

\textbf{Figure 2.13.} A monitor.

Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. Typically, when a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

It is up to the compiler to implement the mutual exclusion on monitor entries, but a common way is to use a binary semaphore. Because the compiler, not the
programmer, is arranging for the mutual exclusion, it is much less likely that something will go wrong. In any event, the person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion. It is sufficient to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time.

Although monitors provide an easy way to achieve mutual exclusion, as we have seen above, that is not enough. We also need a way for processes to block when they cannot proceed. In the producer-consumer problem, it is easy enough to put all the tests for buffer-full and buffer-empty in monitor procedures, but how should the producer block when it finds the buffer full?

The solution lies in the introduction of condition variables, along with two operations on them, WAIT and SIGNAL. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a WAIT on some condition variable, say, full. This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now.

This other process, for example, the consumer, can wake up its sleeping partner by doing a SIGNAL on the condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a SIGNAL. Hoare proposed letting the newly awakened process run, suspending the other one. Brinch Hansen proposed finesse the problem by requiring that a process doing a SIGNAL must exit the monitor immediately. In other words, a SIGNAL statement may appear only as the final statement in a monitor procedure. We will use Brinch Hansen’s proposal because it is conceptually simpler and is also easier to implement. If a SIGNAL is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.

Condition variables are not counters. They do not accumulate signals for later use the way semaphores do. Thus if a condition variable is signaled with no one waiting on it, the signal is lost. The WAIT must come before the SIGNAL. This rule makes the implementation much simpler. In practice it is not a problem because it is easy to keep track of the state of each process with variables, if need be. A process that might otherwise do a SIGNAL can see that this operation is not necessary by looking at the variables.

A skeleton of the producer-consumer problem with monitors is given in Fig. 2-14 in pidgin Pascal.

You may be thinking that the operations WAIT and SIGNAL look similar to SLEEP and WAKEUP, which we saw earlier had fatal race conditions. They are very similar, but with one crucial difference: SLEEP and WAKEUP failed because while one process was trying to go to sleep, the other one was trying to wake it up. With monitors, that cannot happen. The automatic mutual exclusion on monitor procedures guarantees that if, say, the producer inside a monitor procedure discovers that the buffer is full, it will be able to complete the WAIT operation.
monitor ProducerConsumer
condition full, empty;
integer count;

procedure enter;
begin
  if count = N then wait(full);
  enter_item;
  count := count + 1;
  if count = 1 then signal(empty)
end:

procedure remove;
begin
  if count = 0 then wait(empty);
  remove_item;
  count := count - 1;
  if count = N - 1 then signal(full)
end:

count := 0;
end monitor:

procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter
  end
end:

procedure consumer;
begin
  while true do
  begin
    ProducerConsumer.remove;
    consume_item
  end
end

Figure 2-14. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has $N$ slots.
without having to worry about the possibility that the scheduler may switch to the consumer just before the \texttt{wait} completes. The consumer will not even be let into the monitor at all until the \texttt{wait} is finished and the producer has been marked as no longer runnable.

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error-prone than with semaphores. Still, they too have some drawbacks. It is not for nothing that Fig. 2.14 is written in a strange kind of pidgin Pascal rather than in C, as are the other examples in this book. As we said earlier, monitors are a programming language concept. The compiler must recognize them and arrange for the mutual exclusion somehow. C, Pascal, and most other languages do not have monitors, so it is unreasonable to expect their compilers to enforce any mutual exclusion rules. In fact, how could the compiler even know which procedures were in monitors and which were not?

These same languages do not have semaphores either, but adding semaphores is easy: All you need to do is add two short assembly code routines to the library to issue the \texttt{up} and \texttt{down} system calls. The compilers do not even have to know that they exist. Of course, the operating systems have to know about the semaphores, but at least if you have a semaphore-based operating system, you can still write the user programs for it in C or C++ (or even BASIC if you are masochistic enough). With monitors, you need a language that has them built in. A few languages, such as Concurrent Euclid (Holt, 1983) have them, but they are rare.

Another problem with monitors, and also with semaphores, is that they were designed for solving the mutual exclusion problem on one or more CPUs that all have access to a common memory. By putting the semaphores in the shared memory and protecting them with TSL instructions, we can avoid races. When we go to a distributed system consisting of multiple CPUs, each with its own private memory, connected by a local area network, these primitives become inapplicable. The conclusion is that monitors are too low level and monitors are not usable except in a few programming languages. Also, none of the primitives provide for information exchange between machines. Something else is needed.

### 2.2.7 Message Passing

That something else is message passing. This method of interprocess communication uses two primitives \texttt{send} and \texttt{receive}, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

\begin{verbatim}
send(destination, &message);
and
receive(source, &message);
\end{verbatim}

The former call sends a message to a given destination and the latter one receives a message from a given source. Thus, a message is a piece of data that could return a message.

### Design Issues

Messages are not limited to sending and receiving a single message. There are algorithms that do not use semaphores at all. In these algorithms, processes are asynchronous: Messages can be sent and received concurrently. The receiver can individually check the buffer and send back a new message with an acknowledgment.

Now consider the message acknowledgment. The sender waits for an acknowledgment, and the receiver waits to check the buffer. The acknowledgment is not a message itself, but a small message that tells the receiver that the message was received.

Messages can be unbalanced and considered correct if a message is communicated.

At the core of message passing is the concept of blocking. When a message is being sent, the sending process blocks and waits for the message to arrive. When a message is being received, the receiving process blocks and waits for the message to be sent. This blocking behavior is crucial in ensuring that messages are sent and received in the correct order, but it also has implications for performance. For example, if a process is continuously sending messages while another process is receiving them, the sender may block for extended periods of time, leading to a bottleneck in the communication channel.

### The Producer-Consumer Problem

Now let us look at the producer-consumer problem again. It is still intended to demonstrate that all messages are sent and received in the correct order, but now we have a new twist: We are using a buffered automatic variable that is read by the consumer and written by the producer. The consumer starts out with an empty buffer and the producer starts out with a full buffer. The consumer will check the buffer periodically for new messages. If the buffer is empty, the consumer will block and wait for a new message to arrive. If the buffer is full, the consumer will not block and will continue to receive messages. Similarly, the producer will check the buffer for available space. If the buffer is full, the producer will block and wait for the consumer to remove a message from the buffer. If the buffer is empty, the producer will not block and will continue to send messages. This example demonstrates how message passing is used to solve common problems in concurrent systems.
a message from a given source (or from ANY, if the receiver does not care). If no message is available, the receiver could block until one arrives. Alternatively, it could return immediately with an error code.

**Design Issues for Message Passing Systems**

Message passing systems have many challenging problems and design issues that do not arise with semaphores or monitors, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.

Now consider what happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver can distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored.

Message systems also have to deal with the question of how processes are named, so that the process specified in a SEND or RECEIVE call is unambiguous. **Authentication** is also an issue in message systems: how can the client tell that he is communicating with the real file server, and not with an imposter?

At the other end of the spectrum, there are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying messages from one process to another is always slower than doing a semaphore operation or entering a monitor. Much work has gone into making message passing efficient. Cheriton (1984), for example, has suggested limiting message size to what will fit in the machine’s registers, and then doing message passing using the registers.

**The Producer-Consumer Problem with Message Passing**

Now let us see how the producer-consumer problem can be solved with message passing and no shared memory. A solution is given in Fig. 2-15. We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. In this solution, a total of $N$ messages is used, analogous to the $N$ slots in a shared memory buffer. The consumer starts out by sending $N$ empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance.
If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer; the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empty, waiting for the producer to fill them up; the consumer will be blocked, waiting for a full message.

#define N 100 /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m; /* message buffer */

    while (TRUE) {
        produce_item(&item); /* generate something to put in buffer */
        receive(consumer, &m); /* wait for an empty to arrive */
        build_message(&m, item); /* construct a message to send */
        send(consumer, &m); /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; ++i)
        send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        extract_item(&m, &item); /* extract item from message */
        send(producer, &m); /* send empty reply */
        consume_item(item); /* do something with the item */
    }
}

Figure 2-15. The producer–consumer problem with N messages.

Many variants are possible with message passing. For starters, let us look at how messages are addressed. One way to assign each process a unique address and have messages be addressed to processes. A different way is to invent a new data structure, called a mailbox. A mailbox is a place to buffer a certain number of messages, typically specified when the mailbox is created. When mailboxes are used, the address parameters in the SEND and RECEIVE calls are mailboxes, not processes. When a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox, making room for a new one.

SEC. 2.2 Processes

For the producer process to create mailboxes, it may have to create messages that are large enough to buffer empties. This is because the send empties may not have been sent by the consumer.

The operation of this approach is that the producer process is copied directly into the receiving process. Similarly, the receiveeds are forced on the producer.

The interaction between the two processes is via pipes, where each process can send a message to the other. The mailboxes preserve the direction of data flow by providing a pair of 100 bytes each. If both processes are reader and writer, they would both write and read messages at the same time. MINIX operating systems provide support for communication.

2.3 CLASSES

The operating systems are widely distributed among the better systems.

2.3.1 The dining philosophers

In 1965, David Lamport published an article on a synchronization problem. The problem is called the dining philosophers problem. The problem is that philosophers are seated around a round table. Each philosopher has a pair of plates in front of them, and a fork to eat with. Each philosopher has a plate on each side of them. Each philosopher can eat from one of the plates, but can only take one fork at a time. The problem is that if two philosophers try to take a fork from the same side of the table, they will both be blocked, waiting for the other philosopher to release the fork. This can lead to a deadlock, where no one can eat.
For the producer-consumer problem, both the producer and consumer would create mailboxes large enough to hold \( N \) messages. The producer would send messages containing data to the consumer's mailbox, and the consumer would send empty messages to the producer's mailbox. When mailboxes are used, the buffering mechanism is clear; the destination mailbox holds messages that have been sent to the destination process but have not yet been accepted.

The other extreme from having mailboxes is to eliminate all buffering. When this approach is followed, if the SEND is done before the RECEIVE, the sending process is blocked until the RECEIVE happens, at which time the message can be copied directly from the sender to the receiver, with no intermediate buffering. Similarly, if the RECEIVE is done first, the receiver is blocked until a SEND happens. This strategy is often known as a rendezvous. It is easier to implement than a buffered message scheme but is less flexible since the sender and receiver are forced to run in lockstep.

The interprocess communication between user processes in MINIX (and UNIX) is via pipes, which are effectively mailboxes. The only real difference between a message system with mailboxes and the pipe mechanism is that pipes do not preserve message boundaries. In other words, if one process writes 10 messages of 100 bytes to a pipe and another process reads 1000 bytes from that pipe, the reader will get all 10 messages at once. With a true message system, each READ should return only one message. Of course, if the processes agree always to read and write fixed-size messages from the pipe, or to end each message with a special character (e.g., linfeed), no problems arise. The processes that make up the MINIX operating system itself use a true message scheme with fixed size messages for communication among themselves.

### 2.3 CLASSICAL IPC PROBLEMS

The operating systems literature is full of interesting problems that have been widely discussed and analyzed. In the following sections we will examine three of the better-known problems.

#### 2.3.1 The Dining Philosophers Problem

In 1965, Dijkstra posed and solved a synchronization problem he called the dining philosophers problem. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosophers problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in Fig. 2-16.
The life of a philosopher consists of alternate periods of eating and thinking. (This is something of an abstraction, even for philosophers, but the other activities are irrelevant here.) When a philosopher gets hungry, she tries to acquire her left and right fork, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks and continues to think. The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck? (It has been pointed out that the two-fork requirement is somewhat artificial; perhaps we should switch from Italian to Chinese food, substituting rice for spaghetti and chopsticks for forks.)

Figure 2-17 shows the obvious solution. The procedure take_fork waits until the specified fork is available and then seizes it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation. (It is called starvation even when the problem does not occur in an Italian or a Chinese restaurant.)

Now you might think, “If the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that every philosopher would get a fork would be 100%.” It is an observation that we always want to prove. (Think about it.)

One idea is to keep the philosophers waiting in some other place and only allow them to acquire forks after they have been acquired. The procedure take_fork would be replaced with an acquire procedure, which would try to get the fork. This solution is not good either, because the philosophers may be unable to all be served.

The solution is to use a semaphores. A semaphore is a set of devices that keep track of availability. Each philosopher uses a semaphore to acquire forks. The mutex lock associated with the semaphore is used when one philosopher is eating. The semaphore does not allow competing threads to get the same resource. (See chapter 29.1.2.1.1.)
```c
#define N 5

void philosopher(int i) { /* i: philosopher number, from 0 to 4 */
    while (TRUE) { /* philosopher is thinking */
        think();
        take_fork(i); /* take left fork */
        take_fork((i+1) % N); /* take right fork; % is modulo operator */
        eat();
        yum-yum, spaghetti */
        put_fork(i); /* put left fork back on the table */
        put_fork((i+1) % N); /* put right fork back on the table */
    }
}
```

Figure 2-17. A nonsolution to the dining philosophers problem.

that everything would continue in lockstep for even an hour is very small.” This observation is true, but in some applications one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers. (Think about safety control in a nuclear power plant.)

One improvement to Fig. 2-17 that has no deadlock and no starvation is to protect the five statements following the call to think by a binary semaphore. Before starting to acquire forks, a philosopher would do a DOWN on mutex. After replacing the forks, she would do an UP on mutex. From a theoretical viewpoint, this solution is adequate. From a practical one, it has a performance bug: only one philosopher can be eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

The solution presented in Fig. 2-18 is correct and also allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move only into eating state if neither neighbor is eating. Philosopher i’s neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure philosopher as its main code, but the other procedures, take_forks, put_forks, and test are ordinary procedures and not separate processes.

### 2.3.2 The Readers and Writers Problem

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem (Courtois et al., 1971), which models access to a data base. Imagine, for example, an airline
#define N 5
#define LEFT (i-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2

typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    int j;
    int k;
    int status;
    int flag;
    int philosopher = i;
    int hungry = HUNGRY;
    int thinking = THINKING;
    int eating = EATING;
    int state = state[i];
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}

void take_forks(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks()
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i)
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

/* number of philosophers */
/* number of i's left neighbor */
/* number of i's right neighbor */
/* philosopher is thinking */
/* philosopher is trying to get forks */
/* philosopher is eating */
/* semaphores are a special kind of int */
/* array to keep track of everyone's state */
/* mutual exclusion for critical regions */
/* one semaphore per philosopher */
/* i: philosopher number, from 0 to N-1 */
/* repeat forever */
/* philosopher is thinking */
/* acquire two forks or block */
/* yum-yum, spaghetti */
/* put both forks back on table */
/* i: philosopher number, from 0 to N-1 */
/* enter critical region */
/* record fact that philosopher i is hungry */
/* try to acquire 2 forks */
/* exit critical region */
/* block if forks were not acquired */
/* i: philosopher number, from 0 to N-1 */
/* enter critical region */
/* philosopher has finished eating */
/* see if left neighbor can now eat */
/* see if right neighbor can now eat */
/* exit critical region */

/* i: philosopher number, from 0 to N-1 */

void writer(int i)
{
    while (TRUE) {
        down(&mutex);
        state[i] = EATING;
        test(LEFT);
        test(RIGHT);
        up(&mutex);
        print();
    }
}

Figure 2-18. A solution to the dining philosopher's problem.
reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the data base at the same time, but if one process is updating (writing) the data base, no other processes may have access to the data base, not even readers. The question is how do you program the readers and the writers? One solution is shown in Fig. 2-19.

typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

/* use your imagination */
/* controls access to 'rc' */
/* controls access to the data base */
/* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        think_up_data();
        /* noncritical region */
        think_up_data();
        /* get exclusive access */
        down(&db);
        write_data_base();
        up(&db);
        /* release exclusive access */
    }
}

Figure 2-19. A solution to the readers and writers problem.

In this solution, the first reader to get access to the data base does a \texttt{DOWN} on the semaphore \texttt{db}. Subsequent readers merely increment a counter, \texttt{rc}. As readers leave, they decrement the counter and the last one out does an \texttt{UP} on the semaphore, allowing a blocked writer, if there is one, to get in.

The solution presented here implicitly contains a subtle decision that is worth commenting on. Suppose that while a reader is using the data base, another
reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. A third and subsequent readers can also be admitted if they come along.

Now suppose that a writer comes along. The writer cannot be admitted to the data base, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.

To prevent this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance. Courtois et al. present a solution that gives priority to writers. For details, we refer you to the paper.

2.3.3 The Sleeping Barber Problem

Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and \( n \) chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in Fig. 2-20. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions.

Our solution uses three semaphores: customers, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), barbers, the number of barbers who are idle, waiting for customers (0 or 1), and mutex, which is used for mutual exclusion. We also need a variable, waiting, which also counts the waiting customers. It is essentially a copy of customers. The reason for having waiting is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

Our solution is shown in Fig. 2-21. When the barber shows up for work in the morning, he executes the procedure barber, causing him to block on the semaphore customers until somebody arrives. He then goes to sleep as shown in Fig. 2-20.

When a customer arrives, he executes customer, starting by acquiring mutex to enter a critical region. If another customer enters shortly thereafter, the second
one will not be able to do anything until the first one has released \textit{mutex}. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases \textit{mutex} and leaves without a haircut.

If there is an available chair, the customer increments the integer variable, \textit{waiting}. Then he does an UP on the semaphore \textit{customers}, thus waking up the barber. At this point, the customer and barber are both awake. When the customer releases \textit{mutex}, the barber grabs it, does some housekeeping, and begins the haircut.

When the haircut is over, the customer exits the procedure and leaves the shop. Unlike our earlier examples, there is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, another haircut is given. If not, the barber goes to sleep.

As an aside, it is worth pointing out that although the readers and writers and sleeping barber problems do not involve data transfer, they are still belong to the area of IPC because they involve synchronization between multiple processes.
#define CHAIRS 5
typedef int semaphore;

semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;

void barber(void)
{
  while (TRUE) {
    down(customers);
    down(mutex);
    waiting = waiting - 1;
    up(barbers);
    up(mutex);
    cut_hair();
  }
}

void customer(void)
{
  down(mutex);
  if (waiting < CHAIRS) {
    waiting = waiting + 1;
    up(customers);
    up(mutex);
    down(barbers);
    get_haircut();
  } else {
    up(mutex);
  }
}

Figure 2-21. A solution to the sleeping barber problem.

2.4 PROCESS SCHEDULING

In the examples of the previous sections, we have often had situations in which two or more processes (e.g., producer and consumer) were logically runnable. When more than one process is runnable, the operating system must decide which one to run first. The part of the operating system that makes this decision is called the scheduler; the algorithm it uses is called the scheduling algorithm.
Back in the old days of batch systems with input in the form of card images on a magnetic tape, the scheduling algorithm was simple: just run the next job on the tape. With timesharing systems, the scheduling algorithm is more complex, as there are often multiple users waiting for service, and there may be one or more batch streams as well (e.g., at an insurance company, for processing claims). Even on personal computers, there may be several user-initiated processes competing for the CPU, not to mention background jobs, such as network or electronic mail daemons sending or receiving e-mail.

Before looking at specific scheduling algorithms, we should think about what the scheduler is trying to achieve. After all, the scheduler is concerned with deciding on policy, not providing a mechanism. Various criteria come to mind as to what constitutes a good scheduling algorithm. Some of the possibilities include:

1. Fairness—make sure each process gets its fair share of the CPU.
2. Efficiency—keep the CPU busy 100 percent of the time.
3. Response time—minimize response time for interactive users.
4. Turnaround—minimize the time batch users must wait for output.
5. Throughput—maximize the number of jobs processed per hour.

A little thought will show that some of these goals are contradictory. To minimize response time for interactive users, the scheduler should not run any batch jobs at all (except maybe between 3 A.M. and 6 A.M., when all the interactive users are snug in their beds). The batch users probably will not like this algorithm, however; it violates criterion 4. It can be shown (Kleinrock, 1975) that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all. To give one user more you have to give another user less. Such is life.

A complication that schedulers have to deal with is that every process is unique and unpredictable. Some spend a lot of time waiting for file I/O, while others would use the CPU for hours at a time if given the chance. When the scheduler starts running some process, it never knows for sure how long it will be until that process blocks, either for I/O, or on a semaphore, or for some other reason. To make sure that no process runs too long, nearly all computers have an electronic timer or clock built in, which causes an interrupt periodically. A frequency of 50 or 60 times a second (called 50 or 60 Hertz and abbreviated Hz) is common, but on many computers the operating system can set the timer frequency to anything it wants. At each clock interrupt, the operating system gets to run and decide whether the currently running process should be allowed to continue, or whether it has had enough CPU time for the moment and should be suspended to give another process the CPU.

The strategy of allowing processes that are logically runnable to be temporarily suspended is called preemptive scheduling, and is in contrast to the run
to completion method of the early batch systems. Run to completion is also called nonpreemptive scheduling. As we have seen throughout this chapter, a process can be suspended at an arbitrary instant, without warning, so another process can be run. This leads to race conditions and necessitates semaphores, monitors, messages, or some other sophisticated method for preventing them. On the other hand, a policy of letting a process run as long as it wanted to would mean that some process computing $\pi$ to a billion places could deny service to all other processes indefinitely.

Thus although nonpreemptive scheduling algorithms are simple and easy to implement, they are usually not suitable for general-purpose systems with multiple competing users. On the other hand, for a dedicated system, such as a data base server, it may well be reasonable for the master process to start a child process working on a request and let it run until it completes or blocks. The difference from the general-purpose system is that all processes in the data base system are under the control of a single master, which knows what each child is going to do and about how long it will take.

### 2.4.1 Round Robin Scheduling

Now let us look at some specific scheduling algorithms. One of the oldest, simplest, fairest, and most widely used algorithms is **round robin**. Each process is assigned a time interval, called its quantum, which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course. Round robin is easy to implement. All the scheduler needs to do is maintain a list of runnable processes, as shown in Fig. 2-22(a). When the process uses up its quantum, it is put on the end of the list, as shown in Fig. 2-22(b).

![Figure 2-22](image)

**Figure 2-22.** Round robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after $B$ uses up its quantum.

The only interesting issue with round robin is the length of the quantum. Switching from one process to another requires a certain amount of time for doing the administration—saving and loading registers and memory maps, updating various tables and lists, etc. Suppose that this **process switch** or **context switch**, as it is sometimes called, takes 5 msec. Also suppose that the quantum is set at 20 msec. With one process this means that we have to spend 15 msec. per process, which would be wasted otherwise.

To improve performance consider the following: The idea is to keep a set of processes running as though they were each executing on a dedicated CPU, but to share the resources of a single CPU among them. This is known as **time-sharing** or **multi-programming**. If a process is currently running on the CPU and it has no work to do, it is said to be **idle**. The CPU is then said to be **idle**, too. Now the way it works is this: If the time-sharing program runs about 1/2 second after getting a character from the user, it computes the character's value and displays it. The user perceives a 5-msec. delay, which is 10 times what actually occurs on a typical CPU.

The converse of this problem is that causes too much overhead to be too long, and this means an overhead of around 100 msec.

### 2.4.2 Priorities

Another class of scheduling algorithms is **priorities**. These are usually used in real-time systems, where swamping the computer with data is important. Examples include interactive systems (that is, systems designed to be used interactively), scientific systems (that is, systems designed to perform scientific calculations), and real-time systems (systems designed to perform real-time calculations). In these systems, the order of the processes is determined by their priorities, which are usually based on the amount of data that each process is expected to produce. The higher the priority, the more important the process is considered to be. Priorities are usually assigned by the system administrator, and they are used to control the order in which processes are selected for execution. The most common way to assign priorities is to use a scheduling algorithm that takes into account the priority of each process. There are many different scheduling algorithms, each with its own advantages and disadvantages. Some of the most popular scheduling algorithms are the [Round Robin](https://en.wikipedia.org/wiki/Preemptive_round_robin), [Shortest Job First](https://en.wikipedia.org/wiki/Shortest_job_first), and [Shortest Remaining Time First](https://en.wikipedia.org/wiki/Shortest_remaining_time_first) algorithms.
msec. With these parameters, after doing 20 msec of useful work, the CPU will have to spend 5 msec on process switching. Twenty percent of the CPU time will be wasted on administrative overhead.

To improve the CPU efficiency, we could set the quantum to, say, 500 msec. Now the wasted time is less than 1 percent. But consider what happens on a timesharing system if ten interactive users hit the carriage return key at roughly the same time. Ten processes will be put on the list of runnable processes. If the CPU is idle, the first one will start immediately, the second one may not start until about 1/2 sec later, and so on. The unlucky last one may have to wait 5 sec before getting a chance, assuming all the others use their full quanta. Most users will perceive a 5-sec response to a short command as terrible. The same problem can occur on a personal computer that supports multiprogramming.

The conclusion can be formulated as follows: setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests. A quantum around 100 msec is often a reasonable compromise.

2.4.2 Priority Scheduling

Round robin scheduling makes the implicit assumption that all processes are equally important. Frequently, the people who own and operate multiuser computers have different ideas on that subject. At a university, the pecking order may be deans first, then professors, secretaries, janitors, and finally students. The need to take external factors into account leads to priority scheduling. The basic idea is straightforward: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

Even on a PC with a single owner, there may be multiple processes, some more important than others. For example, a daemon process sending electronic mail in the background should be assigned a lower priority than a process displaying a video film on the screen in real time.

To prevent high-priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick (i.e., at each clock interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs. Alternatively, each process may be assigned a maximum quantum that it is allowed to hold the CPU continuously. When this quantum is used up, the next highest priority process is given a chance to run.

Priorities can be assigned to processes statically or dynamically. On a military computer, processes started by generals might begin at priority 100, processes started by colonels at 90, majors at 80, captains at 70, lieutenants at 60, and so on. Alternatively, at a commercial computer center, high-priority jobs might cost 100 dollars an hour, medium priority 75 dollars an hour, and low priority 50 dollars an
hour. The UNIX system has a command, *nice*, which allows a user to voluntarily reduce the priority of his process, in order to be nice to the other users. Nobody ever uses it.

Priorities can also be assigned dynamically by the system to achieve certain system goals. For example, some processes are highly I/O bound and spend most of their time waiting for I/O to complete. Whenever such a process wants the CPU, it should be given the CPU immediately, to let it start its next I/O request, which can then proceed in parallel with another process actually computing. Making the I/O bound process wait a long time for the CPU will just mean having it around occupying memory for an unnecessarily long time. A simple algorithm for giving good service to I/O bound processes is to set the priority to $1/f$, where $f$ is the fraction of the last quantum that a process used. A process that used only 2 msec of its 100 msec quantum would get priority 50, while a process that ran 50 msec before blocking would get priority 2, and a process that used the whole quantum would get priority 1.

It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class. Figure 2-23 shows a system with four priority classes. The scheduling algorithm is as follows: as long as there are runnable processes in priority class 4, just run each one for one quantum, round-robin fashion, and never bother with lower priority classes. If priority class 4 is empty, then run the class 3 processes round robin. If classes 4 and 3 are both empty, then run class 2 round robin, and so on. If priorities are not adjusted occasionally, lower priority classes may all starve to death.

![Figure 2-23. A scheduling algorithm with four priority classes.](image)

### 2.4.3 Multiple Queues

One of the earliest priority schedulers was in CTSS (Corbato et al., 1962). CTSS had the problem that process switching was very slow because the 7094 could hold only one process in memory. Each switch meant swapping the current process to disk and reading in a new one from disk. The CTSS designers quickly realized that it was more efficient to give CPU-bound processes a large quantum once in a while (sleeping). On the other hand, response time was not a concern for the CTSS class. Processes that could not run for four quanta because their quantum was allocated to it, it would be put into the sleep state for 100 quanta. When it awoke, it would get 4, and the final 64 quanta would be scheduled after the initial 100, with the process keeping only the highest 64 quantum and less frequently.

The following long time was almost never finished. When the CPU was waiting to that time out, it was about the time bound process returns, irregular runs at random time, and all his friends would be getting it right.

Many of the early classes. For example, Chicago, Berkeley, had four classes. Whenever the process was running when the quantum ended, it went to the disk block and was running when the next quantum returned. However, if a process had been running for terminal I/O, the system uses a separate background one.

### 2.4.4 Short Queues

Most of this chapter we have looked at one class. The classes are known in many systems. The class of quite accurate, and the work is done