

MODERN OPERATING SYSTEMS

FIFTH EDITION

**ANDREW S. TANENBAUM
HERBERT BOS**

*Vrije Universiteit
Amsterdam, The Netherlands*



2

PROCESSES AND THREADS

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

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

2.1 PROCESSES

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

eternity. While waiting for a disk request to complete, many more requests may come in. If there are multiple disks present, some or all of the newer ones may be fired off to other disks long before the first request is satisfied. Clearly some way is needed to model and control this concurrency. Processes (and especially threads) can help here.

Now consider a user PC. When the system is booted, many processes are secretly started, often unknown to the user. For example, a process may be started up to wait for incoming email. Another process may run on behalf of the antivirus program to check periodically if any new virus definitions are available. In addition, explicit user processes may be running, printing files and backing up the user's photos on a USB stick, all while the user is surfing the Web. All this activity has to be managed, and a multiprogramming system supporting multiple processes comes in very handy here. Even simple computing devices, such as smartphones and tablets, can support multiple processes.

In any multiprogramming system, each CPU switches from process to process quickly, running each for tens or maybe hundreds of milliseconds. While strictly speaking, at any one instant each CPU is running only one process, in the course of 1 second it may work on several of them, giving the illusion of parallelism. Sometimes people speak of **pseudoparallelism** in this context, to contrast it with the true hardware parallelism of **multiprocessor** systems (which have two or more CPUs sharing the same physical memory). Keeping track of multiple, parallel activities is hard for people to do. Therefore, operating system designers over the years have evolved a conceptual model (sequential processes) that makes parallelism easier to deal with. That model, its uses, and some of its consequences form the subject of this chapter.

2.1.1 The Process Model

In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just **processes** for short. A process is just an instance of 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, each 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 each CPU switches from program to program. Switching rapidly back and forth like this is known as **multiprogramming**, as we saw in Chap. 1.

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 logical program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is

finished (for the time being), the physical program counter is saved in the process' stored logical program counter in memory. 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.

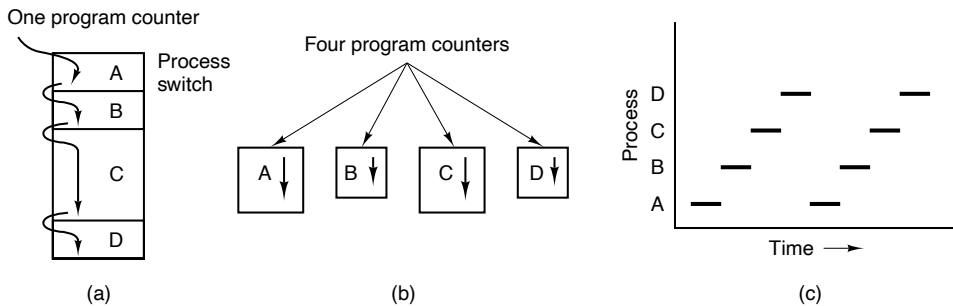


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

In this chapter, we will assume there is only one CPU. Often that assumption does not hold, since new chips are often multicore, with two, four, or more cores. We will look at multicore chips and multiprocessors in general in Chap. 8, but for the time being, it is simpler just to think of one CPU at a time. So when we say that a CPU can really run only one process at a time, if there are two cores (or CPUs) each of them can run only one process at a time.

With the CPU switching back and forth among multiple 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 their speed. Consider, for example, an audio process that plays music to accompany a high-quality video run by another core. Because the audio should start a little later than the video, it signals the video server to start playing, and then runs an idle loop 10,000 times before playing back the audio. All goes well, if the loop is a reliable timer, but if the CPU decides to switch to another process during the idle loop, the audio process may not run again until the corresponding video frames have already come and gone, and the video and audio will be annoyingly out of sync. 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 absolutely crucial. An analogy may help you here. Consider a culinary-minded computer scientist who is baking a birthday cake for his young daughter. He has a birthday cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar, extract of

vanilla, and so on. In this analogy, the recipe is the program, that is, 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 screaming his head off, 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 versus 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 accustomed to determine when to stop work on one process and service a different one. In contrast, a program is something that may be stored on disk, not doing anything.

It is worth noting that if a program is running twice, it counts as two processes. For example, it is often possible to start a word processor twice or print two files at the same time if two printers are available. The fact that two processes happen to be running the same program does not matter; they are distinct processes. The operating system may be able to share the code between them so only one copy is in memory, but that is a technical detail that does not change the conceptual situation of two processes running.

2.1.2 Process Creation

Operating systems need some way to create processes. In very simple systems, or in systems designed for running only a single application (e.g., the controller in a microwave oven), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-purpose systems, however, some way is needed to create and terminate processes as needed during operation. We will now look at some of the issues.

Four principal events cause processes to be created:

1. System initialization
2. Execution of a process-creation system call by a running process
3. A user request to create a new process
4. Initiation of a batch job

When an operating system is booted, typically numerous processes are created. Some of these processes are foreground processes, that is, processes that interact

with (human) users and perform work for them. Others run in the background and are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming email, sleeping most of the day but suddenly springing to life when email arrives. Another background process may be designed to accept incoming requests for Web pages hosted on that machine, waking up when a request arrives to service the request. Processes that stay in the background to handle some activity such as email, Web pages, news, printing, and so on are called **daemons**. Large systems commonly have dozens of them. In UNIX[†], the *ps* program can be used to list the running processes. In Windows, the task manager can be used.

In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes. For example, if a large amount of data are being fetched over a network for subsequent processing, it may be convenient to create one process to fetch the data and put them in a shared buffer while a second process removes the data items and processes them. On a multiprocessor, allowing each process to run on a different CPU may also make the job go faster.

In interactive systems, users can start a program by typing a command or (double) clicking on an icon. Taking either of these actions starts a new process and runs the selected program in it. In command-based UNIX systems running the X Window System, the new process takes over the window in which it was started. In Windows, when a process is started it does not have a window, but it can create one (or more) and most do. In both systems, users may have multiple windows open at once, each running some process. Using the mouse, the user can select a window and interact with the process, for example, providing input when needed.

The last situation in which processes are created applies only to batch systems found on large mainframes. Think of inventory management at the end of a day at a chain of stores—calculating what to order, analyzing per-store product popularity, etc. Here users can submit batch jobs to the system (possibly remotely). When the operating system decides it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

Technically, in all these cases, a new process is created by having an existing process execute a process creation system call. That process may be a running user process, a system process invoked from the keyboard or mouse, or a batch-manager process. What that process does is execute a system call to create the new process. This system call tells the operating system to create a new process and indicates, directly or indirectly, which program to run in it. To get the ball rolling, the very first process is hard-crafted when the system is booted.

[†] In this chapter, UNIX should be interpreted as including almost all POSIX-based systems, including Linux, FreeBSD, MacOS, Solaris, etc., and to some extent, Android and iOS as well.

In UNIX, there is only one system call to create a new process: `fork`. This call creates an exact clone of the calling process. After the `fork`, the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files. That is all there is. Usually, the child process then executes `execve` or a similar system call to change its memory image and run a new program. For example, when a user types a command, say, *sort*, to the shell, the shell forks off a child process and the child executes *sort*. The reason for this two-step process is to allow the child to manipulate its file descriptors after the `fork` but before the `execve` in order to accomplish redirection of standard input, standard output, and standard error.

In Windows, in contrast, a single Win32 function call, `CreateProcess`, handles both process creation and loading the correct program into the new process. This call has 10 parameters, which include the program to be executed, the command-line parameters to feed that program, various security attributes, bits that control whether open files are inherited, priority information, a specification of the window to be created for the process (if any), and a pointer to a structure in which information about the newly created process is returned to the caller. In addition to `CreateProcess`, Win32 has about 100 other functions for managing and synchronizing processes and related topics.

In both UNIX and Windows systems, after a process is created, the parent and child have their own distinct address spaces. If either process changes a word in its address space, the change is not visible to the other process. In traditional UNIX, the child's initial address space is a *copy* of the parent's, but there are definitely two distinct address spaces involved; no writable memory is shared. Some UNIX implementations share the program text between the two since that cannot be modified. Alternatively, the child may share all of the parent's memory, but in that case the memory is shared **copy-on-write**, which means that whenever either of the two wants to modify part of the memory, that chunk of memory is explicitly copied first to make sure the modification occurs in a private memory area. Again, no writable memory is shared. It is, however, possible for a newly created process to share some of its creator's other resources, such as open files. In Windows, the parent's and child's address spaces are different from the start.

2.1.3 Process Termination

After a process has been created, it starts running and does whatever its job is. However, nothing lasts forever, not even processes. Sooner or later the new process will terminate, usually due to one of the following conditions:

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. This call is `exit` in UNIX and `ExitProcess` in Windows. Screen-oriented programs also support voluntary termination. Word processors, Internet browsers, and similar programs always have an icon or menu item that the user can click to tell the process to remove any temporary files it has open and then terminate.

The second reason for termination is that the process discovers a fatal error. For example, if a user types the command

```
cc foo.c
```

to compile the program *foo.c* and no such file exists, the compiler simply announces this fact and exits. Screen-oriented interactive processes generally do not exit when given bad parameters. Instead they pop up a dialog box and ask the user to try again.

The third reason for termination is an error caused by the process, often due to a program bug. Examples include executing an illegal instruction, referencing non-existent memory, or dividing by zero. In some systems (e.g., UNIX), a process can tell the operating system that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated when one of the errors occurs.

The fourth reason a process might terminate is that the process executes a system call telling the operating system to kill some other process. In UNIX this call is `kill`. The corresponding Win32 function is `TerminateProcess`. In both cases, the killer must have the necessary authorization to do in the killee. In some systems, when a process terminates, either voluntarily or otherwise, all processes it created are immediately killed as well. Neither UNIX nor Windows works this way, however.

2.1.4 Process Hierarchies

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy. Note that unlike plants and animals that use sexual reproduction, a process has only one parent (but zero, one, two, or more children). So a process is more like a hydra than like, say, a cow.

In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard (e.g., by pressing CTRL-C), the signal is delivered to all members of the process group currently associated with the keyboard (usually all active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

As another example of where the process hierarchy plays a key role, let us look at how UNIX initializes itself when it is started, just after the computer is booted. 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 a 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.

In contrast, Windows has no concept of a process hierarchy. All processes are equal. The only hint of a process hierarchy is that when a process is created, the parent is given a special token (called a **handle**) that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

2.1.5 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 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 fundamentally different from the first two in that the process cannot run, even if the CPU is idle and has nothing else to do.

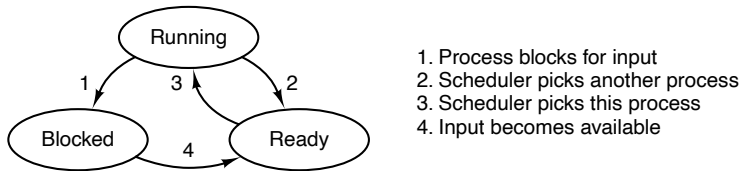


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Four transitions are possible among these three states, as shown. Transition 1 occurs when the operating system discovers that a process cannot continue right now. In some systems the process can execute a system call, such as `pause`, to get into blocked state. In other systems, including UNIX, 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 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. We will study some of them later in this chapter.

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 and the process will start running. Otherwise it may have to wait in *ready* state for a little while until the CPU is available and its turn comes.

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 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 what is here called the scheduler, which is actually not much code. The rest of the operating system is nicely structured in process form. Few real systems are as nicely structured as this, however.

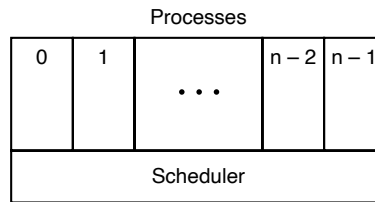


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

2.1.6 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. (Some authors call these entries **process control blocks**.) This entry contains important information about the process' state, including 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* or *blocked* state so that it can be restarted later as if it had never been stopped.

Figure 2-4 shows some of the key fields in a typical system. The fields in the first column relate to process management. The other two relate to memory management and file management, respectively. It should be noted that precisely which fields the process table has is highly system dependent, but this figure gives a general idea of the kinds of information needed.

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 one (or each) CPU and also explain interrupts in more detail than we were able to do in Chap. 1. Associated with each I/O class is a location (typically at a fixed location near the bottom of memory) called the **interrupt vector**. It contains the address of the **ISR (Interrupt Service Routine)**. Suppose that user process 3 is running when a disk interrupt happens. User process 3's program counter, program status word, and sometimes one or more registers are pushed onto the (current) stack by the interrupt hardware. The computer then jumps to the address in the interrupt vector. That is all the hardware does. From here on, it is up to the ISR in software.

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process-table entry.

All interrupts start by saving the registers, often in the process table entry for the current process. Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point 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 high-level languages such as C, so they are performed by a small assembly-language routine, usually the same one for all interrupts since the work of saving the registers is identical, no matter the cause of the interrupt.

When this routine is finished, it calls a C procedure to do the rest of the work for this specific interrupt type. (We assume the operating system is written in C, the usual choice for all operating systems used in production.) When it has done its job, possibly making some process now ready, the scheduler is called to see what to run next. After that, control is passed back to the assembly-language code to load up the registers and memory map for the now-current process and start it running. Interrupt handling and scheduling are summarized in Fig. 2-5. It is worth noting that the details vary somewhat from system to system.

A process may be interrupted thousands of times during its execution, but the key idea is that after each interrupt the interrupted process returns to precisely the same state it was in before the interrupt occurred.

2.1.7 Modeling Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20% of the time it is sitting in memory, then with five processes in memory at once the CPU should be busy all the time. This model is unrealistically optimistic, however, since it tacitly assumes that all five processes will never be waiting for I/O at the same time.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly-language procedure saves registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs. The details may vary between operating systems.

A better model is to look at CPU usage from a probabilistic viewpoint. Suppose that a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n . The CPU utilization is then given by the formula

$$\text{CPU utilization} = 1 - p^n$$

Figure 2-6 shows, for different values of p (or “I/O wait”), the CPU utilization as a function of n , which is called the **degree of multiprogramming**.

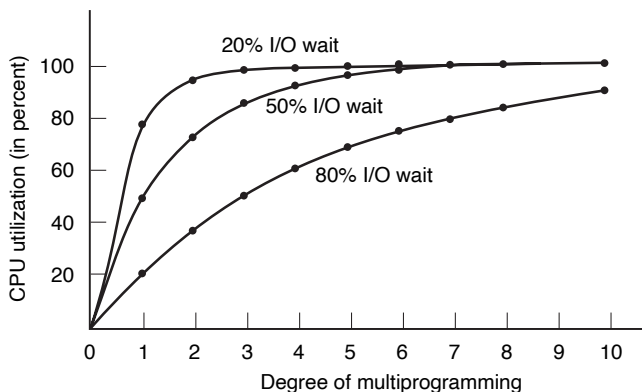


Figure 2-6. CPU utilization as a function of the number of processes in memory.

From the figure it is clear that if processes spend 80% of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10%. When you realize that an interactive process waiting for a user to type something at a terminal (or click on an icon) is in I/O wait state, it should be clear that I/O wait times of 80% and more are not unusual. But even on servers, processes doing a lot of disk I/O will often have this percentage or more.

For the sake of accuracy, it should be pointed out that the probabilistic model just described is only an approximation. It implicitly assumes that all n processes are independent, meaning that it is quite acceptable for a system with five processes in memory to have three running and two waiting. But with a single CPU, we cannot have three processes running at once, so a process becoming ready while the CPU is busy will have to wait. Thus, the processes are not independent. A more accurate model can be constructed using queueing theory, but the point we are making—multiprogramming lets processes use the CPU when it would otherwise become idle—is, of course, still valid, even if the true curves of Fig. 2-6 are slightly different from those shown in the figure.

Even though the model of Fig. 2-6 is fairly simple-minded, it can nevertheless be used to make specific, although approximate, predictions about CPU performance. Suppose, for example, that a computer has 8 GB of memory, with the operating system and its tables taking up 2 GB and each user program also taking up 2 GB. These sizes allow three user programs to be in memory at once. With an 80% average I/O wait, we have a CPU utilization (ignoring operating system overhead) of $1 - 0.8^3$ or about 49%. Adding another 8 GB of memory allows the system to go from three-way multiprogramming to seven-way multiprogramming, thus raising the CPU utilization to 79%. In other words, the additional 8 GB will raise the throughput by 30%.

Adding yet another 8 GB would increase CPU utilization only from 79% to 91%, thus raising the throughput by only another 12%. Using this model, the computer's owner might decide that the first addition was a good investment but that the second was not.

2.2 THREADS

In traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, in many situations, it is useful to have multiple threads of control in the same address space running in quasi-parallel, as though they were (almost) separate processes (except for the shared address space). In the following sections, we will discuss threads and their implications. Later we will look at an alternative solution.

2.2.1 Thread Usage

Why would anyone want to have a kind of process within a process? It turns out there are several reasons for having these miniprocesses, called **threads**. Let us now examine some of them. The main reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

We have seen this argument once before. It is precisely the argument for having processes. Instead of thinking about interrupts, timers, and context switches, we can think about parallel processes. Only now with threads we add a new element: the ability for the parallel entities to share an address space and all of its data among themselves. This ability is essential for certain applications, which is why having multiple processes (with their separate address spaces) will not work.

A second argument for having threads is that since they are lighter weight than processes, they are easier (i.e., faster) to create and destroy. In many systems, creating a thread goes 10–100 times faster than creating a process. When the number of threads needed changes dynamically and rapidly, this property is useful to have.

A third reason for having threads is also a performance argument. Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.

Finally, threads are useful on systems with multiple CPUs, where real parallelism is possible. We will come back to this issue in Chap. 8.

It is easiest to see why threads are useful by looking at some concrete examples. As a first example, consider a word processor. Word processors usually display the document being created on the screen formatted exactly as it will appear on the printed page. In particular, all the line breaks and page breaks are in their correct and final positions, so that the user can inspect them and change the document if need be (e.g., to eliminate widows and orphans—incomplete top and bottom lines on a page, which are considered esthetically displeasing).

Suppose that the user is writing a book. From the author's point of view, it is easiest to keep the entire book as a single file to make it easier to search for topics, perform global substitutions, and so on. Alternatively, each chapter might be a separate file. However, having every section and subsection as a separate file is a real nuisance when global changes have to be made to the entire book, since then hundreds of files have to be individually edited, one at a time. For example, if proposed standard xxxx is approved just before the book goes to press, all occurrences of "Draft Standard xxxx" have to be changed to "Standard xxxx" at the last minute. If the entire book is one file, typically a single command can do all the substitutions. In contrast, if the book is spread over 300 files, each one must be edited separately.

Now consider what happens when the user suddenly deletes one sentence from page 1 of an 800-page book. After checking the changed page for correctness, she now wants to make another change on page 600 and types in a command telling the word processor to go to that page (possibly by searching for a phrase occurring only there). The word processor is now forced to reformat the entire book up to page 600 on the spot because it does not know what the first line of page 600 will be until it has processed all the previous pages. There may be a substantial delay before page 600 can be displayed, leading to an unhappy user.

Threads can help here. Suppose that the word processor is written as a two-threaded program. One thread interacts with the user and the other handles reformatting in the background. As soon as the sentence is deleted from page 1, the interactive thread tells the reformatting thread to reformat the whole book. Meanwhile, the interactive thread continues to listen to the keyboard and mouse and responds to simple commands like scrolling page 1 while the other thread is computing madly in the background. With a little luck, the reformatting will be completed before the user asks to see page 600, so it can be displayed instantly.

While we are at it, why not add a third thread? Many word processors have a feature of automatically saving the entire file to disk every few minutes to protect the user against losing a day's work in the event of a program crash, system crash, or power failure. The third thread can handle the disk backups without interfering with the other two. The situation with three threads is shown in Fig. 2-7.

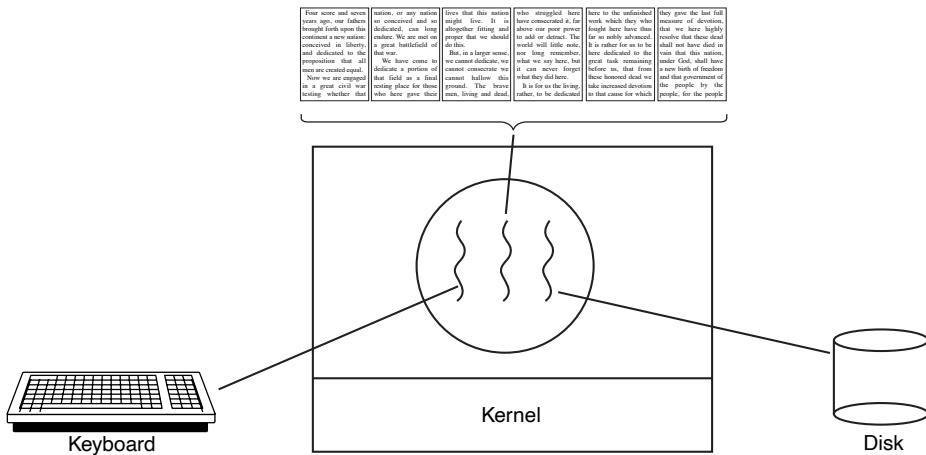


Figure 2-7. A word processor with three threads.

If the program were single-threaded, then whenever a disk backup started, commands from the keyboard and mouse would be ignored until the backup was finished. The user would surely perceive this as sluggish performance. Alternatively, keyboard and mouse events could interrupt the disk backup, allowing good performance but leading to a complex interrupt-driven programming model. With three threads, the programming model is much simpler. The first thread just interacts with the user. The second thread reformats the document when told to. The third thread writes the contents of RAM to disk periodically.

It should be clear that having three separate processes would not work here because all three threads need to operate on the document. By having three threads instead of three processes, they share a common memory and thus all have access to the document being edited. With three processes this would be impossible.

An analogous situation exists with many other interactive programs. For example, an electronic spreadsheet is a program that allows a user to maintain a matrix, some of whose elements are data provided by the user. Other elements are computed based on the input data using potentially complex formulas. A spreadsheet that computes the predicted annual profit of a substantial company might have hundreds of pages and thousands of complex formulas based on hundreds of input variables. When a user changes one input variable, many cells may have to be recomputed. By having a background thread do the recomputation, the interactive thread can allow the user to make additional changes while the computation is going on. Similarly, a third thread can handle periodic backups to disk on its own.

Now consider yet another example of where threads can be useful: a server for a Website. Requests for pages come in and the requested page is sent back to the client. At most Websites, some pages are more commonly accessed than other pages. For example, Samsung's home page is accessed far more than a page deep in the tree containing the detailed technical specifications of any smartphone model. Web servers use this fact to improve performance by maintaining a collection of heavily used pages in main memory to eliminate the need to go to disk to get them. Such a collection is called a **cache** and is used in many other contexts as well. We saw CPU caches in Chap. 1, for example.

One way to organize the Web server is shown in Fig. 2-8(a). Here one thread, the **dispatcher**, reads incoming requests for work from the network. After examining the request, it chooses an idle (i.e., blocked) **worker thread** and hands it the request, possibly by writing a pointer to the message into a special word associated with each thread. The dispatcher then wakes up the sleeping worker, moving it from blocked state to ready state.

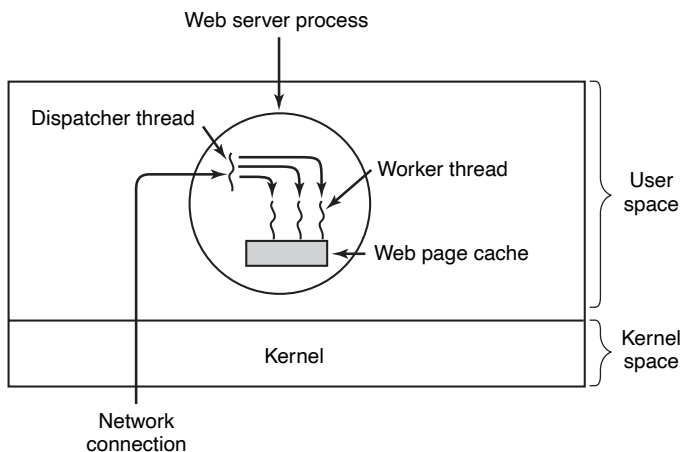


Figure 2-8. A multithreaded Web server.

When the worker wakes up, it checks to see if the request can be satisfied from the Web page cache, to which all threads have access. If not, it starts a **read**

operation to get the page from the disk and blocks until the disk operation completes. When the thread blocks on the disk operation, another thread is chosen to run, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

This model allows the server to be written as a collection of sequential threads. The dispatcher's program consists of an infinite loop for getting a work request and handing it off to a worker. Each worker's code consists of an infinite loop consisting of accepting a request from the dispatcher and checking the Web cache to see if the page is present. If so, it is returned to the client, and the worker blocks waiting for a new request. If not, it gets the page from the disk, returns it to the client, and blocks waiting for a new request.

A rough outline of the code is given in Fig. 2-9. Here, as in the rest of this book, *TRUE* is assumed to be the constant 1. Also, *buf* and *page* are structures appropriate for holding a work request and a Web page, respectively.

<pre>while (TRUE) { get_next_request(&buf); handoff_work(&buf); }</pre>	<pre>while (TRUE) { wait_for_work(&buf) look_for_page_in_cache(&buf, &page); if (page_not_in_cache(&page)) read_page_from_disk(&buf, &page); return_page(&page); }</pre>
(a)	(b)

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread.
(b) Worker thread.

Consider how the Web server could be written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the Web server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other incoming requests. If the Web server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the Web server is waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus, threads gain considerable performance, but each thread is programmed sequentially, in the usual way. We will look at an alternative, event-driven, approach later.

A third example where threads are useful is in applications that must process very large amounts of data. The normal approach is to read in a block of data, process it, and then write it out again. The problem here is that if only blocking system calls are available, the process blocks while data are coming in and data are going out. Having the CPU go idle when there is lots of computing to do is clearly wasteful and should be avoided if possible.

Threads offer a solution. The process could be structured with an input thread, a processing thread, and an output thread. The input thread reads data into an input buffer. The processing thread takes data out of the input buffer, processes them,

and puts the results in an output buffer. The output buffer writes these results back to disk. In this way, input, output, and processing can all be going on at the same time. Of course, this model works only if a system call blocks only the calling thread, not the entire process.

2.2.2 The Classical Thread Model

Now that we have seen why threads might be useful and how they can be used, let us investigate the idea a bit more closely. The process model is based on two independent concepts: resource grouping and execution. Sometimes it is useful to separate them; this is where threads come in. First we will look at the classical thread model; after that we will examine the Linux thread model, which blurs the line between processes and threads.

One way of looking at a process is that it is a convenient way to group together related resources. A process has an address space that contains program text and data, as well as other resources. These resources may include open files, child processes, pending alarms, signal handlers, accounting information, and more. By putting them together in the form of a process, they can be managed more easily.

The other concept a process has is a thread of execution, usually shortened to just **thread**. The thread has a program counter associated with it that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It also has a stack, which contains the thread's execution history, one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

What threads add to the process model is to allow multiple executions to take place in the same process environment (and address space), to a large degree independent of one another. Having multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer. In the former case, the threads share an address space and other resources. In the latter case, processes share physical memory, disks, printers, and other resources. Because threads have some of the properties of processes, they are sometimes called **lightweight processes**. The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process. As we saw in Chap. 1, some CPUs have direct hardware support for multithreading and allow a thread switch to happen on a nanosecond time scale.

In Fig. 2-10(a) we see three traditional processes. Each process has its own address space and a single thread of control. In contrast, in Fig. 2-10(b) we see a single process with three threads of control. Although in both cases we have three threads, in Fig. 2-10(a) each of them operates in a different address space, whereas in Fig. 2-10(b) all three of them share the same address space.

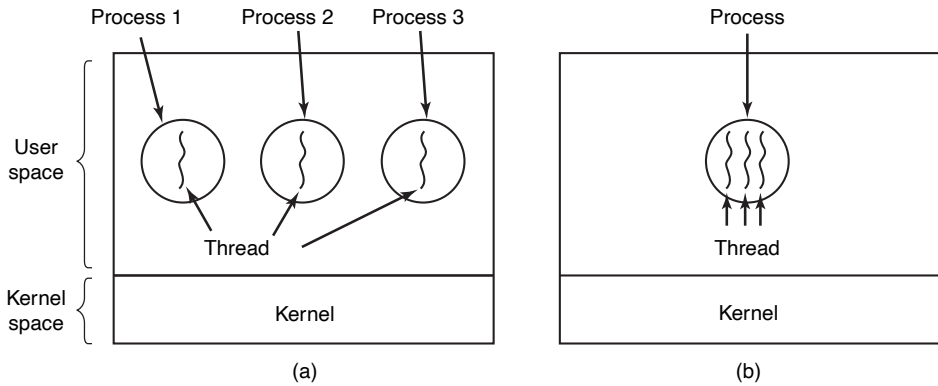


Figure 2-10. (a) Three processes each with one thread. (b) One process with three threads.

When a multithreaded process is run on a single-CPU system, the threads take turns running. In Fig. 2-1, we saw how multiprogramming of processes works. By switching back and forth among multiple processes, the system gives the illusion of separate sequential processes running in parallel. Multithreading works the same way. The CPU switches rapidly back and forth among the threads, providing the illusion that the threads are running in parallel, albeit on a slower CPU. With three compute-bound threads in a process, the threads would appear to be running in parallel, each one on a CPU with one-third the speed of the real CPU.

The different threads in a process are not as independent as different processes. All threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every memory address within the process' address space, one thread can read, write, or even wipe out another thread's stack. There is no protection between threads because (1) it is impossible, and (2) it should not be necessary. Unlike different processes, which may be from different users and which may be mutually hostile to one another, a process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight with each other. In addition to sharing an address space, all the threads can share the same set of open files, child processes, signals, alarms, and so forth, as shown in Fig. 2-11. Thus, the organization of Fig. 2-10(a) would be used when the three processes are essentially unrelated, whereas Fig. 2-10(b) would be appropriate when the three threads are actually part of the same job and are actively and closely cooperating with each other.

The items in the first column are process properties, not thread properties. For example, if one thread opens a file, that file is visible to the other threads in the process and they can read and write it. This is logical, since the process is the unit of resource management, not the thread. If each thread had its own address space, open files, pending alarms, and so on, it would be a separate process. What we are

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-11. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

trying to achieve with the thread concept is the ability for multiple threads of execution to share a set of resources so that they can work together closely to perform some task.

Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states: running, blocked, ready, or terminated. A running thread currently has the CPU and is active. In contrast, a blocked thread is waiting for some event to unblock it. For example, when a thread performs a system call to read from the keyboard, it is blocked until input is typed. A thread can block waiting for some external event to happen or for some other thread to unblock it. A ready thread is scheduled to run and will as soon as its turn comes up. The transitions between thread states are the same as those between process states and are illustrated in Fig. 2-2.

It is important to realize that each thread has its own stack, as illustrated in Fig. 2-12. Each thread's stack contains one frame for each procedure called but not yet returned from. This frame contains the procedure's local variables and the return address to use when the procedure call has finished. For example, if procedure *X* calls procedure *Y* and *Y* calls procedure *Z*, then while *Z* is executing, the frames for *X*, *Y*, and *Z* will all be on the stack. Each thread will generally call different procedures and thus have a different execution history. This is why each thread needs its own stack.

When multithreading is present, processes usually start with a single thread present. This thread has the ability to create new threads by calling a library procedure such as *thread_create*. A parameter to *thread_create* specifies the name of a procedure for the new thread to run. It is not necessary (or even possible) to specify anything about the new thread's address space, since it automatically runs in the address space of the creating thread. Sometimes threads are hierarchical, with a parent-child relationship, but often no such relationship exists, with all threads being equal. With or without a hierarchical relationship, the creating thread is usually returned a thread identifier that names the new thread.

When a thread has finished its work, it can exit by calling a library procedure, say, *thread_exit*. It then vanishes and is no longer schedulable. In some thread

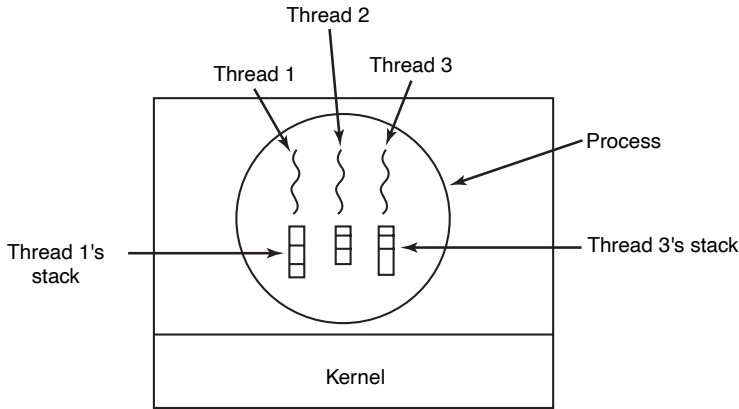


Figure 2-12. Each thread has its own stack.

systems, one thread can wait for a (specific) thread to exit by calling a procedure, for example, *thread_join*. This procedure blocks the calling thread until a (specific) thread has exited. In this regard, thread creation and termination is very much like process creation and termination, with approximately the same options as well.

Another common thread call is *thread_yield*, which allows a thread to voluntarily give up the CPU to let another thread run. Such a call is important because there is no clock interrupt to actually enforce multiprogramming as there is with processes. Thus, it is important for threads to be polite and voluntarily surrender the CPU from time to time to give other threads a chance to run. Other calls allow one thread to wait for another thread to finish some work, for a thread to announce that it has finished some work, and so on.

While threads are often useful, they also introduce a number of serious complications into the programming model. To start with, consider the effects of the UNIX *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 in the parent was blocked on a *read* call, say, from the keyboard? Are two threads now blocked on the keyboard, one in the parent and one in the child? 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. The designers of the operating system have to make choices clear and carefully define the semantics so users will understand the behavior of the threads.

We will look at *some* of these issues and observe that the solutions are often pragmatic. For instance, on a system such as Linux, a *fork* of a multithreaded process will create only a single thread in the child. However, using Posix threads a program can use the *pthread_atfork()* call to register fork handlers (procedures that are called when a fork occurs), so it can start additional threads and do whatever is

needed to get going properly again. Even so, it is important to note that many of these issues are design choices and different systems may opt for different solutions. For now, the most important thing to remember is that the relation between threads and fork can be quite complex.

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 one thread notices that there is too little memory and starts allocating more memory. Partway through, a thread switch occurs, and the new thread also notices that there is too little memory and also starts allocating more memory. Memory will probably be allocated twice. These problems can be solved with some effort, but careful thought and design are needed to make multithreaded programs work correctly.

2.2.3 POSIX Threads

To make it possible to write portable threaded programs, IEEE has defined a standard for threads in IEEE standard 1003.1c. The threads package it defines is called **Pthreads**. Most UNIX systems support it. The standard defines over 60 function calls, which is far too many to go over here. Instead, we will just describe a few of the major ones to give an idea of how it works. The calls we will describe below are listed in Fig. 2-13.

Thread call	Description
<code>pthread_create</code>	Create a new thread
<code>pthread_exit</code>	Terminate the calling thread
<code>pthread_join</code>	Wait for a specific thread to exit
<code>pthread_yield</code>	Release the CPU to let another thread run
<code>pthread_attr_init</code>	Create and initialize a thread's attribute structure
<code>pthread_attr_destroy</code>	Remove a thread's attribute structure

Figure 2-13. Some of the Pthreads function calls.

All Pthreads threads have certain properties. Each one has an identifier, a set of registers (including the program counter), and a set of attributes, which are stored in an attribute structure. The attributes include the stack size, scheduling parameters, and other items needed to use the thread.

A new thread is created using the *pthread_create* call. The thread identifier of the newly created thread is returned as the function value. This call is intentionally very much like the fork system call (except with parameters), with the thread identifier playing the role of the PID, mostly for identifying threads referenced in other calls. When a thread has finished the work it has been assigned, it can terminate by calling *pthread_exit*. This call stops the thread and releases its stack.

Often a thread has a need to wait for another thread to finish its work and exit before it can continue. The thread that is waiting calls *pthread_join* to wait for a

specific other thread to terminate. The thread identifier of the thread to wait for is given as a parameter.

Sometimes it happens that a thread is not logically blocked, but feels that it has run long enough and wants to give another thread a chance to run. It can accomplish this goal by calling *pthread_yield*. There is no such call for processes because the assumption there is that processes are fiercely competitive and each wants all the CPU time it can get (although a very public-spirited process could call *sleep* to yield the CPU briefly). However, since the threads of a process are working together and their code is invariably written by the same programmer, sometimes the programmer wants them to give each other another chance.

The next two thread calls deal with attributes. *Pthread_attr_init* creates the attribute structure associated with a thread and initializes it to the default values. These values (such as the priority) can be changed by manipulating fields in the attribute structure.

Finally, *pthread_attr_destroy* removes a thread's attribute structure, freeing up its memory. It does not affect threads using it; they continue to exist.

To get a better feel for how Pthreads works, consider the simple example of Fig. 2-14. Here the main program loops *NUMBER_OF_THREADS* times, creating a new thread on each iteration, after announcing its intention. If the thread creation fails, it prints an error message and then exits. After creating all the threads, the main program exits.

When a thread is created, it prints a one-line message announcing itself, then it exits. The order in which the various messages are interleaved is nondeterminate and may vary on consecutive runs of the program.

The Pthreads calls described above are not the only ones. We will examine some of the others after we have discussed process and thread synchronization.

2.2.4 Implementing Threads in User Space

There are two main places to implement threads: user space and the kernel. The choice is a bit controversial, and a hybrid implementation is also possible. We will now describe these methods, along with their advantages and disadvantages.

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads. All operating systems used to fall into this category, and even now some still do. With this approach, threads are implemented by a library.

All of these implementations have the same general structure, illustrated in Fig. 2-15(a). The threads run on top of a run-time system, which is a collection of procedures that manage threads. We have seen four of these already: *pthread_create*, *pthread_exit*, *pthread_join*, and *pthread_yield*, but usually there are more.

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

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread %d\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

Figure 2-14. An example program using threads.

When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process. This table is analogous to the kernel's process table, except that it keeps track only of the per-thread properties, such as each thread's program counter, stack pointer, registers, state, and so forth. The thread table is managed by the run-time system. When a thread is moved to the ready or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

When a thread does something that may cause it to become blocked locally, for example, waiting for another thread in its process to complete some work, it calls a run-time system procedure. This procedure checks to see if the thread must be put into blocked state. If so, it stores the thread's registers (i.e., its own) in the thread table, looks in the table for a ready thread to run, and reloads the machine registers with the new thread's saved values. As soon as the stack pointer and program

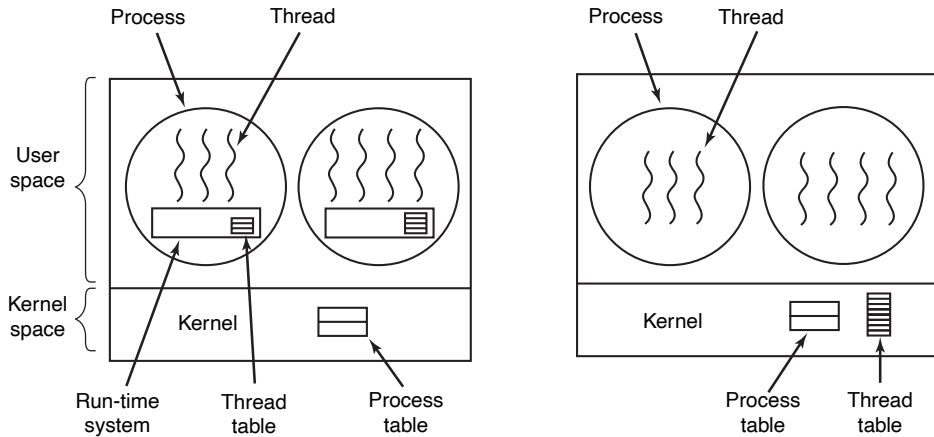


Figure 2-15. (a) A user-level threads package. (b) A threads package managed by the kernel.

counter have been switched, the new thread comes to life again automatically. If the machine happens to have an instruction to store all the registers and another one to load them all, the entire **thread switch** can be done in just a handful of instructions. Doing thread switching like this is at least an order of magnitude—maybe more—faster than trapping to the kernel and is a strong argument in favor of user-level threads packages.

In addition, when a thread is finished running for the moment, for example, when it calls *thread_yield*, the code of *thread_yield* saves the thread's information in the thread table and then calls the thread scheduler to pick another thread to run. The procedure that saves the thread's state and the scheduler are just local procedures, so invoking them is much more efficient than making a kernel call. There is no need for a trap, a context switch, flushing of caches, and so on. This makes thread scheduling very fast.

User-level threads also have other advantages. They allow each process to have its own customized scheduling algorithm. For some applications, for example, those with a garbage-collector thread, not having to worry about a thread being stopped at an inconvenient moment is a plus. They also scale better, since kernel threads invariably require some table space and stack space in the kernel, which can be a problem if there is a very large number of threads.

Despite their better performance, user-level threads packages have some major problems. First among these is the problem of how blocking system calls are implemented. Suppose that a thread reads from the keyboard before any keys have been hit. Letting the thread actually make the system call is unacceptable, since this will stop all the threads. One of the main goals of having threads in the first place was to allow each thread to use blocking calls, but to prevent one blocked

thread from affecting the others. With blocking system calls, it is hard to see how this goal can be achieved readily.

The system calls could all be changed to be nonblocking (e.g., a `read` on the keyboard would just return 0 bytes if no characters were already buffered), but requiring changes to the operating system is unattractive. Besides, one argument for user-level threads was precisely that they could run with *existing* operating systems. In addition, changing the semantics of `read` will require changes to many user programs.

Another alternative is available in the event that it is possible to tell in advance if a call will block. In most versions of UNIX, a system call, `select`, exists, which allows the caller to tell whether a prospective `read` will block. When this call is present, the library procedure `read` can be replaced with a new one that first does a `select` call and then does the `read` call only if it is safe (i.e., will not block). If the `read` call will block, the call is not made. Instead, another thread is run. The next time the run-time system gets control, it can check again to see if the `read` is now safe. This approach requires rewriting parts of the system call library, and is inefficient and inelegant, but there is little choice. The code placed around the system call to do the checking is called a **wrapper**. (As we shall see, many operating systems have even more efficient mechanisms for asynchronous I/O, such as `epoll` on Linux and `kqueue` on FreeBSD).

Somewhat analogous to the problem of blocking system calls is the problem of page faults. We will study these in Chap. 3. For the moment, suffice it to say that computers can be set up in such a way that not all of the program is in main memory at once. If the program calls or jumps to an instruction that is not in memory, a page fault occurs and the operating system will go and get the missing instruction (and its neighbors) from disk. This is called a page fault. The process is blocked while the necessary instruction is being located and read in. If a thread causes a page fault, the kernel, unaware of even the existence of threads, naturally blocks the entire process until the disk I/O is complete, even though other threads might be runnable.

Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU. Within a single process, there are no clock interrupts, making it impossible to schedule processes in round-robin fashion (taking turns). Unless a thread exits the run-time system of its own free will, the scheduler will never run.

One possible solution to the problem of threads running forever is to have the run-time system request a clock signal (interrupt) once a second to give it control, but this, too, is crude and messy to program. Periodic clock interrupts at a higher frequency are not always possible, and even if they are, the total overhead may be substantial. Furthermore, a thread might also need a clock interrupt, interfering with the run-time system's use of the clock.

Another, and really the most devastating, argument against user-level threads is that programmers typically want threads precisely in applications where threads

block often, as, for example, in a multithreaded Web server. These threads are constantly making system calls. Once a trap has occurred to the kernel to carry out the system call, it is hardly any more work for the kernel to switch threads if the old one has blocked, and having the kernel do this eliminates the need for constantly making `select` system calls to see if `read` system calls are safe. For applications that are CPU bound and rarely block, what is the point of having threads at all? No one would seriously propose computing the first n prime numbers or playing chess using threads because there is nothing to be gained by doing it that way.

2.2.5 Implementing Threads in the Kernel

Now let us consider having the kernel know about and manage the threads. As shown in Fig. 2-15(b), there is now no need for a run-time system or thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.

The kernel's thread table holds each thread's registers, state, and other information. The information is the same as with user-level threads, but now kept in the kernel instead of in user space (inside the run-time system). This information is a subset of the information that traditional kernels maintain about their single-threaded processes, that is, the process state. In addition, the kernel also maintains the traditional process table to keep track of processes.

All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure. When a thread blocks, the kernel may choose to run either another thread from the same process (if one is ready) or a thread from a different process. With user-level threads, the run-time system keeps running threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run).

Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be created, an old thread is reactivated, saving some overhead. Thread recycling is also possible for user-level threads, but since the thread-management overhead is much smaller, there is less incentive to do this.

Kernel threads do not require any new, nonblocking system calls. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk. Their main disadvantage is that the cost of a system call is substantial, so if thread operations (creation, termination, etc.) are common, much more overhead will be incurred.

While kernel threads solve some problems, they do not solve all problems. For example, we still have to think about what happens when a multithreaded process forks. Does the new process have as many threads as the old one did, or does it have just one? In many cases, the best choice depends on what the process is planning to do next. If it is going to call `exec` to start a new program, probably one thread is the correct choice, but if it continues to execute, reproducing all the threads is probably best.

Another issue with threads is signals. Remember that signals are sent to processes, not to threads, at least in the classical model. When a signal comes in, which thread should handle it? Possibly threads could register their interest in certain signals, so when a signal came in it would be given to the thread that said it wants it. On Linux, for instance, a signal may be handled by any thread and the lucky winner is selected by the operating system, but we can simply block the signal on all threads except one. If two or more threads register for the same signal, the operating system picks a thread (say, at random) and lets it handle the signal. Anyway, these are only some of the problems threads introduce, and there are more. Unless the programmer is very careful, it is easy to make mistakes.

2.2.6 Hybrid Implementations

Various ways have been investigated to try to combine the advantages of user-level threads with kernel-level threads. One way is use kernel-level threads and then multiplex user-level threads onto some or all of them, as shown in Fig. 2-16. When this approach is used, the programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one. This model gives the ultimate flexibility.

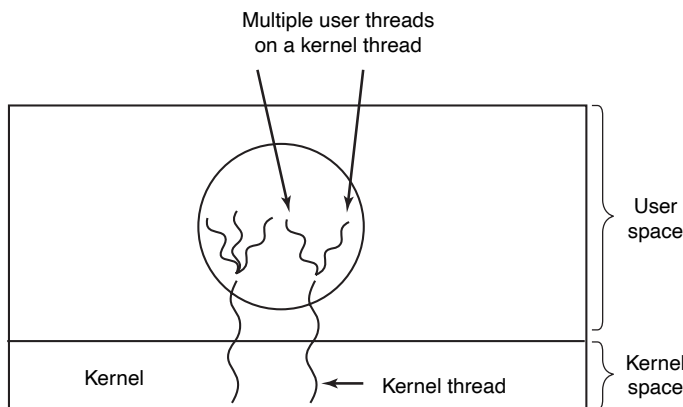


Figure 2-16. Multiplexing user-level threads onto kernel-level threads.

With this approach, the kernel is aware of *only* the kernel-level threads and thus schedules those. Some of those threads may have multiple user-level threads

multiplexed on top of them. These user-level threads are created, destroyed, and scheduled just like user-level threads in a process that runs on an operating system without multithreading capability. In this model, each kernel-level thread has some set of user-level threads that take turns using it.

2.2.7 Making Single-Threaded Code Multithreaded

Many existing programs were written for single-threaded processes. Converting these to multithreading is much trickier than it may at first appear. Below we will examine just a few of the pitfalls.

As a start, the code of a thread normally consists of multiple procedures, just like a process. These may have local variables, global variables, and parameters. Local variables and parameters do not cause any trouble, but variables that are global to a thread but not global to the entire program are a problem. These are variables that are global in the sense that many procedures within the thread use them (as they might use any global variable), but other threads should logically leave them alone.

As an example, consider the *errno* variable maintained by UNIX. When a process (or a thread) makes a system call that fails, the error code is put into *errno*. In Fig. 2-17, thread 1 executes the system call *access* to find out if it has permission to access a certain file. The operating system returns the answer in the global variable *errno*. After control has returned to thread 1, but before it has a chance to read *errno*, the scheduler decides that thread 1 has had enough CPU time for the moment and switches to thread 2. Thread 2 executes an *open* call that fails, which causes *errno* to be overwritten and thread 1's *access* code to be lost forever. When thread 1 starts up later, it will read the wrong value and behave incorrectly.

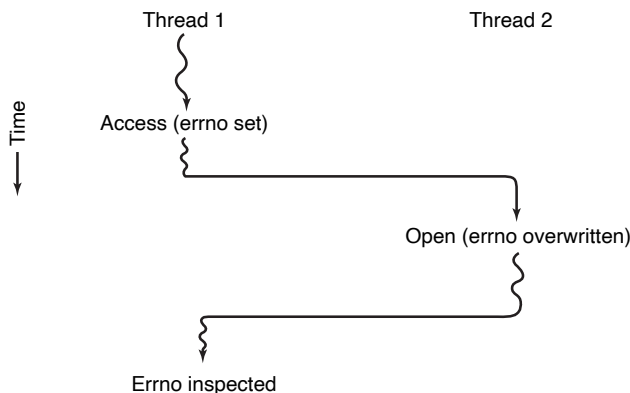


Figure 2-17. Conflicts between threads over the use of a global variable.

Various solutions to this problem are possible. One is to prohibit global variables altogether. However worthy this ideal may be, it conflicts with much existing

software. Another is to assign each thread its own private global variables, as shown in Fig. 2-18. In this way, each thread has its own private copy of *errno* and other global variables, so conflicts are avoided. In effect, this decision creates a new scoping level, variables visible to all the procedures of a thread (but not to other threads), in addition to the existing scoping levels of variables visible only to one procedure and variables visible everywhere in the program.

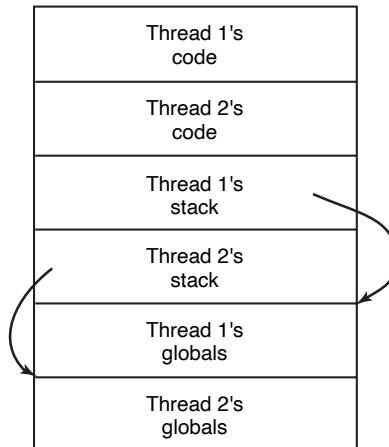


Figure 2-18. Threads can have private global variables.

Accessing the private global variables is a bit tricky, however, since most programming languages have a way of expressing local variables and global variables, but not intermediate forms. It is possible to allocate a chunk of memory for the globals and pass it to each procedure in the thread as an extra parameter. While hardly an elegant solution, it works.

Alternatively, new library procedures can be introduced to create, set, and read these threadwide global variables. The first call might look like this:

```
create_global("bufptr");
```

It allocates storage for a pointer called *bufptr* on the heap or in a special storage area reserved for the calling thread. No matter where the storage is allocated, only the calling thread has access to the global variable. If another thread creates a global variable with the same name, it gets a different storage location that does not conflict with the existing one.

Two calls are needed to access global variables: one for writing them and the other for reading them. For writing, something like

```
set_global("bufptr", &buf);
```

will do. It stores the value of a pointer in the storage location previously created by the call to *create_global*. To read a global variable, the call might look like


```
bufptr = read_global("bufptr");
```

It returns the address stored in the global variable, so its data can be accessed.

The next problem in turning a single-threaded program into a multithreaded one is that many library procedures are not reentrant. That is, they were not designed to have a second call made to any given procedure while a previous call has not yet finished. For example, sending a message over the network may well be programmed to assemble the message in a fixed buffer within the library, then to trap to the kernel to send it. What happens if one thread has assembled its message in the buffer, then a clock interrupt forces a switch to a second thread that immediately overwrites the buffer with its own message?

Similarly, memory-allocation procedures such as *malloc* in UNIX, maintain crucial tables about memory usage, for example, a linked list of available chunks of memory. While *malloc* is busy updating these lists, they may temporarily be in an inconsistent state, with pointers that point nowhere. If a thread switch happens to occur while the tables are inconsistent and a new call comes in from a different thread, an invalid pointer may be used, leading to a program crash. Fixing all these problems effectively means rewriting the entire library. Doing so is a nontrivial activity with a real possibility of introducing subtle errors.

A different solution is to provide each procedure with a wrapper that sets a bit to mark the library as being in use. Any attempt for another thread to use a library procedure while a previous call has not yet completed is blocked. Although this approach can be made to work, it greatly eliminates potential parallelism.

Next, consider signals. Some signals are logically thread specific; 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. However, when threads are implemented entirely in user space, the kernel does not even know about threads and can hardly direct the signal to the right one. An additional complication occurs if a process may only have one alarm pending at a time 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? Furthermore, what happens if one thread changes the signal handlers without telling other threads about it? And what happens if one thread wants to catch a particular signal (say, the user hitting CTRL-C), and another thread wants this signal to terminate the process? This situation can arise if one or more threads run standard library procedures and others are user-written. Clearly, these wishes are incompatible. In general, signals are difficult enough to manage even in a single-threaded environment. Going to a multithreaded environment does not make them any easier to handle.

One last problem introduced by threads is stack management. In many systems, when a process' stack overflows, the kernel just provides that process with 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 the growth of some thread's stack.

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 may have to be redefined and libraries 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 Cook (2008) and Rodrigues et al. (2010).

2.3 EVENT-DRIVEN SERVERS

In the previous section, we have seen two possible designs for a Web server: a fast multithreaded one and a slow single-threaded one. Suppose that threads are not available or not desirable but the system designers find the performance loss due to single threading, as described so far, unacceptable. If nonblocking versions of system calls, such as `read`, are available, a third approach is possible. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a nonblocking disk operation is started.

The server records the state of the current request in a table and then goes and gets the next event. The next event may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed. With nonblocking disk I/O, a reply probably will have to take the form of a signal or interrupt.

In this design, the “sequential process” model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table every time the server switches from working on one request to another. In effect, we are simulating the threads and their stacks the hard way. A design like this, in which each computation has a saved state, and there exists some set of events that can occur to change the state, is called a **finite-state machine**. This concept is widely used throughout computer science.

In fact, it is very popular in high-throughput servers where even threads are considered too expensive and instead an **event-driven programming paradigm** is used. By implementing the server as a finite state machine that responds to events (e.g., the availability of data on a socket) and interacting with the operating system using non-blocking (or **asynchronous**) system calls, the implementation can be very efficient. Every event leads to a burst of activity, but it never blocks.

Fig. 2-19 shows a pseudo-code example of an event-driven thank-you server (the server thanks every client that sends it a message) that uses the `select` call to monitor multiple network connections (line 17). The `select` determines which file descriptors are ready for receiving or sending data and, looping over them, receives all the messages it can and then tries to send thank-you messages on all corresponding connections that are ready to receive data. In case the server could not

```

0.  /* Preliminaries:
1.  svrSock  : the main server socket, bound to TCP port 12345
2.  toSend   : database to track what data we still have to send to the client
3.             - toSend.put (fd, msg) will register that we need to send msg on fd
4.             - toSend.get (fd) returns the string we need to send msg on fd
5.             - toSend.destroy (fd) removes all information about fd from toSend */
6.
7.  inFds     = { svrSock }           /* file descriptors to watch for incoming data */
8.  outFds     = { }                 /* file descriptors to watch to see if sending is possible */
9.  exceptFds = { }                 /* file descriptors to watch for exception conditions (not used) */
10.
11. char msgBuf [MAX_MSG_SIZE]      /* buffer in which to receive messages */
12. char *thankYouMsg = "Thank you!" /* reply to send back */
13.
14. while (TRUE)
15. {
16.     /* block until some file descriptors are ready to be used */
17.     rdyIns, rdyOuts, rdyExcepts = select (inFds, outFds, exceptFds, NO_TIMEOUT)
18.
19.     for (fd in rdyIns) /* iterate over all the connections that have something for us */
20.     {
21.         if (fd == svrSock)           /* a new connection from a client */
22.         {
23.             newSock = accept (svrSock) /* create new socket for client */
24.             inFds = inFds  $\cup$  { newSock } /* must monitor it also */
25.         }
26.         else
27.         { /* receive the message from the client */
28.             n = receive (fd, msgBuf, MAX_MSG_SIZE)
29.             printf ("Received: %s.\n", msgBuf)
30.
31.             toSend.put (fd, thankYouMsg) /* must still send thankYouMsg on fd */
32.             outFds = outFds  $\cup$  { fd } /* so must monitor this fd */
33.         }
34.     }
35.     for (fd in rdyOuts) /* iterate over all the connections that we can now thank */
36.     {
37.         msg = toSend.get (fd) /* see what we need to send on this connection */
38.         n = send (fd, msg, strlen(msg))
39.         if (n < strlen (thankYouMsg))
40.         {
41.             toSend.put (fd, msg+n) /* remaining characters to send next time */
42.         } else
43.         {
44.             toSend.destroy (fd)
45.             outFds = outFds  $\setminus$  { fd } /* we have thanked this one already */
46.         }
47.     }
47. }

```

Figure 2-19. An event-driven thank-you server (pseudo code).

send the full thank you message, it remembers which bytes it still needs to send, so it can try again later, when there is more space available. We simplified the program to keep it relatively short, by means of pseudo code and not worrying about errors or connections closing. Nevertheless, it illustrates that a single-threaded event-driven server can handle many clients concurrently.

Most popular operating systems offer special, highly optimized event notification interfaces for asynchronous I/O that are much more efficient than `select`. Well-known examples include the `epoll` system call on Linux, and the similar `kqueue` interface on FreeBSD. Windows and Solaris have slightly different solutions. They all allow the server to monitor many network connections at once without blocking on any of them. Because of this Web servers such as `nginx` can comfortably handle ten thousand concurrent connections. This is no trivial feat and even has its own name: the **C10k** problem.

Single-Threaded Versus Multi-Threaded Versus Event-driven Servers

Finally, let us compare the three different ways to build a server. It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking calls (e.g., for disk I/O) and still achieve parallelism. Blocking system calls make programming easier, and parallelism improves performance. The single-threaded server retains the simplicity of blocking system calls but gives up performance.

The third approach, event-driven programming, also achieves high performance through parallelism but uses nonblocking calls and interrupts to do so. It is considered harder to program. These models are summarized in Fig. 2-20.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine/event-driven	Parallelism, nonblocking system calls, interrupts

Figure 2-20. Three ways to construct a server.

These three approaches to handle requests from a client apply not only to user programs, but also to the kernel itself where concurrency is just as important for performance. In fact, this is a good moment to point out that this book introduces many operating system concepts with an emphasis on what they mean for user programs, but of course the operating system itself uses these concepts internally also (and some are even more relevant to the operating system than to user programs). Thus, the operating system kernel itself may consist of multithreaded or event-driven software. For instance, the Linux kernel on modern Intel CPUs is a multithreaded operating system kernel. In contrast, MINIX 3 consists of many servers implemented following the model of finite state machine and events.

2.4 SYNCHRONIZATION AND INTERPROCESS COMMUNICATION

Processes frequently need to synchronize and 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 **IPC (InterProcess Communication)**.

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 or threads do not get in each other's way, for example, two threads in an airline reservation system each trying to grab the last seat on a plane for different customers. The third concerns proper sequencing when dependencies are present: if thread *A* produces data and thread *B* prints them, *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.

It is also important to mention that two of these issues apply to threads as well as to processes with shared memory. The first one—passing information—is clearly easier for threads since they share a common address space by nature. However, the other two—keeping out of each other's hair and proper sequencing—are complicated for threads also. Below we will discuss the problems in the context of processes, but please keep in mind that the same problems and solutions apply to threads.

2.4.1 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 (possibly in a kernel data structure) 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 now 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 very large 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 in a two-word file available to all processes. At a certain instant, slots 0–3 are empty (the files have already been printed) and slots 4–6 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. 2-21.

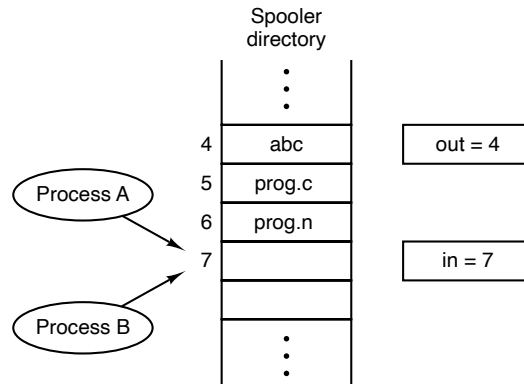


Figure 2-21. Two processes want to access shared memory at the same time.

In jurisdictions where Murphy's law[†] is applicable, the following could easily 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* and also gets a 7. It, too, stores it in *its* local variable *next_free_slot*. At this instant, both processes think that the next available slot is 7.

Process *B* now continues to run. 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 *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 *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 receive any output. User *B* will hang around the printer for years, wistfully hoping for output that never comes. 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 blue moon something weird and unexplained happens. Unfortunately, with increasing parallelism due to increasing numbers of cores, race conditions are becoming more common.

2.4.2 Critical Regions

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

[†] If something can go wrong, it will.

writing the shared data at the same time. Put in other words, what we need is **mutual exclusion**, that is, 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 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 do not lead to race conditions. However, sometimes a process has to access shared memory or files, or do 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 races.

Although this requirement avoids race conditions, it 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 any process.
4. No process should have to wait forever to enter its critical region.

In an abstract sense, the behavior that we want is shown in Fig. 2-22. Here process *A* enters its critical region at time T_1 . A little later, at time T_2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, *B* is temporarily suspended until time T_3 when *A* leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at T_4) and we are back to the original situation with no processes in their critical regions.

2.4.3 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

On a single-processor system, 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

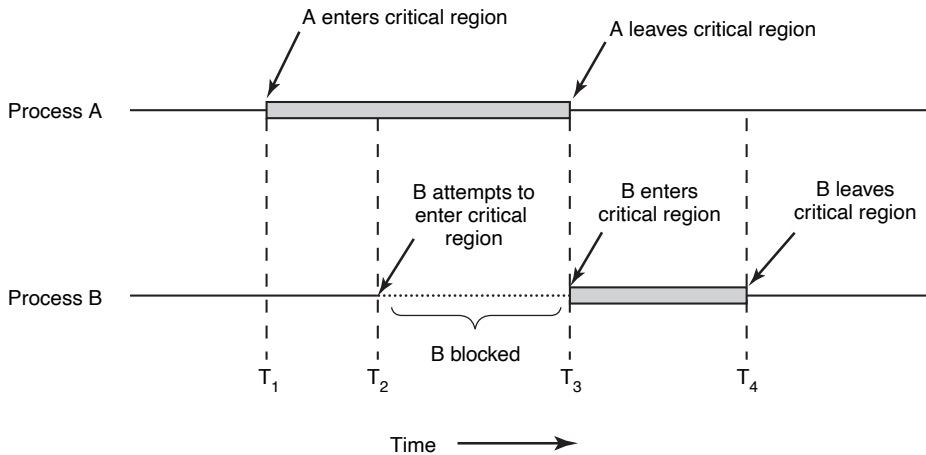


Figure 2-22. Mutual exclusion using critical regions.

is only switched from process to process as a result of clock interrupt 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 and make a mess of things.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. What if 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 especially critical lists. If an interrupt occurs while the list of ready processes, for example, is 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. The kernel should not disable interrupts for more than a few instructions lest it miss interrupts.

The possibility of achieving mutual exclusion by disabling interrupts—even within the kernel—is becoming less every day due to the increasing number of multicore chips even in low-end PCs. Two cores are already common, 4 are present in many machines, and 8, 16, or 32 are not far behind. In a multicore (i.e., multiprocessor system), disabling the interrupts of one CPU does not prevent other CPUs from interfering with operations the first CPU is performing. Consequently, more sophisticated schemes are needed.

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 critical 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. 2-23. This program fragment, like nearly all the others in this book, is written in C. C was chosen here because real operating systems are virtually always written in C (or occasionally C++), but hardly ever in languages like Java, Python, or Haskell. C is powerful, efficient, and predictable, characteristics critical for writing operating systems. Java, for example, is not predictable because it might run out of storage at a critical moment and need to invoke the garbage collector to reclaim memory at a most inopportune time. This cannot happen in C because there is no garbage collection in C. A quantitative comparison of C, C++, Java, and four other languages is given by Prechelt (2000).

<pre>while (TRUE) { while (turn != 0) { } /* loop */ critical_region(); turn = 1; noncritical_region(); }</pre>	<pre>while (TRUE) { while (turn != 1) { } /* loop */ critical_region(); turn = 0; noncritical_region(); }</pre>
(a)	(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

In Fig. 2-23, 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 **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. A lock that uses busy waiting is called a **spin lock**.

When process 0 leaves the critical region, it sets *turn* to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so that both processes are in their noncritical regions, with *turn* set to 0. Now process 0 executes its whole loop quickly, exiting its critical region and setting *turn* to 1. At this point, *turn* is 1 and both processes are in their noncritical regions.

Suddenly, 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 *turn* is 1 and process 1 is busy with its noncritical region. It hangs in its while loop until process 1 sets *turn* to 0. 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 effectively obsolete. Peterson's algorithm is shown in Fig. 2-24. 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 prototypes here or later.

Before using the shared variables (i.e., before entering its critical region), each process calls *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 *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires.

```

#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 2-24. Peterson's solution for achieving mutual exclusion.

Let us see how this solution works. Initially neither process is in its critical region. Now process 0 calls *enter_region*. It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now makes a call to *enter_region*, it will hang there until *interested*[0] goes to *FALSE*, an event that happens only 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 overwritten and 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 until process 0 exits its critical region.

The TSL Instruction

Now let us look at a proposal that requires a little help from the hardware. Some computers, especially those designed with multiple processors in mind, have an instruction like

TSL RX,LOCK

(Test and Set Lock) that works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address

lock. 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.

It is important to note that locking the memory bus is very different from disabling interrupts. Disabling interrupts then performing a read on a memory word followed by a write does not prevent a second processor on the bus from accessing the word between the read and the write. In fact, disabling interrupts on processor 1 has no effect at all on processor 2. The only way to keep processor 2 out of the memory until processor 1 is finished is to lock the bus, which requires a special hardware facility (basically, a bus line asserting that the bus is locked and not available to processors other than the one that locked it).

To use the TSL instruction, we will use a shared variable, *lock*, to coordinate access to shared memory. When *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 *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. 2-25. It shows a four-instruction subroutine in a fictitious (but typical) assembly language. The first instruction copies the old value of *lock* to the register and then sets *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 very simple. The program just stores a 0 in *lock*. No special synchronization instructions are needed.

```
enter_region:
    TSL REGISTER,LOCK           | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was not zero, lock was set, so loop
    RET                         | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET                         | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

One solution to the critical-region problem is now easy. Before entering its critical region, a process calls *enter_region*, which does busy waiting until the lock is free; then it acquires the lock and returns. After leaving the critical region the process calls *leave_region*, which stores a 0 in *lock*. As with all solutions based on

critical regions, the processes must call *enter_region* and *leave_region* at the correct times for the method to work. If one process cheats, the mutual exclusion will fail. In other words, critical regions work only if the processes cooperate.

An alternative instruction to TSL is XCHG, which exchanges the contents of two locations atomically, for example, a register and a memory word. The code is shown in Fig. 2-26, and, as can be seen, is essentially the same as the solution with TSL. All Intel x86 CPUs use XCHG instruction for low-level synchronization.

```

enter_region:
    MOVE REGISTER,#1           I put a 1 in the register
    XCHG REGISTER,LOCK         I swap contents of register and lock variable
    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

```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction.

2.4.4 Sleep and Wakeup

Both Peterson's solution and the solutions using TSL or XCHG 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 burning the CPU while 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, *H*, with high priority, and *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 a variant of 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 consider only the case of one producer and one consumer because this assumption simplifies the solutions.)

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 awakened, and if so, wakes it up. The code for both producer and consumer is shown in Fig. 2-27.

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 made available on any system that actually had these system calls. The procedures *insert_item* and *remove_item*, which are not shown, handle the book-keeping of putting items into the buffer and taking items out of the buffer.

Now let us get back to the race condition. It can occur because access to *count* is unconstrained. As a consequence, 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 inserts 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

```

#define N 100                                /* number of slots in the buffer */
int count = 0;                               /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        item = produce_item();                /* generate next item */
        if (count == N) sleep();              /* if buffer is full, go to sleep */
        insert_item(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)
{
    int item;

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

```

Figure 2-27. The producer-consumer problem with a fatal race condition.

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 storing wakeup signals. The consumer clears the wakeup waiting bit in every iteration of the loop.

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 32 or 64 of them, but in principle the problem is still there.

2.4.5 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 initial proposal, a new variable type, which he 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 on semaphores, now usually called 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, are 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 atomicity is absolutely essential to solving synchronization problems and avoiding race conditions. Atomic actions, in which a group of related operations are either all performed without interruption or not performed at all, are extremely important in many other areas of computer science as well.

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. After an up on a semaphore with processes sleeping on it, the semaphore will still have the value of 0. However, 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. Since these have no mnemonic significance to people who do not speak Dutch and only marginal significance to those people who do—*Proberen* (try) and *Verhogen* (raise, make higher)—we will use the terms down and up instead. These were first introduced in the Algol 68 programming language.

Solving the Producer-Consumer Problem Using Semaphores

Semaphores solve the lost-wakeup problem, as shown in Fig. 2-28. To make them work correctly, it is essential 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 disabling 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 instructions, 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 or XCHG instructions used to make sure that only one CPU at a time examines the semaphore.

Be sure you understand that using TSL or XCHG to prevent several CPUs from accessing the semaphore at the same time is quite different from the producer or consumer busy waiting for the other to empty or fill the buffer. The semaphore operation will take only a few nanoseconds, whereas the producer or consumer might take arbitrarily long.


```

#define N 100                                /* number of slots in the buffer */
typedef int semaphore;                       /* semaphores are a special kind of int */
semaphore mutex = 1;                         /* controls access to critical region */
semaphore empty = N;                        /* counts empty buffer slots */
semaphore full = 0;                         /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                           /* TRUE is the constant 1 */
        item = produce_item();              /* generate something to put in buffer */
        down(&empty);                       /* decrement empty count */
        down(&mutex);                       /* enter critical region */
        insert_item(item);                  /* put new item in buffer */
        up(&mutex);                         /* leave critical region */
        up(&full);                          /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                           /* infinite loop */
        down(&full);                        /* decrement full count */
        down(&mutex);                       /* enter critical region */
        item = remove_item();               /* take item from buffer */
        up(&mutex);                         /* leave critical region */
        up(&empty);                         /* increment count of empty slots */
        consume_item(item);                 /* do something with the item */
    }
}

```

Figure 2-28. 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 and synchronization primitive at our disposal, let us go back and again look at the interrupt sequence of

Fig. 2-5. 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 5 in Fig. 2-5 consists of doing an up on the device's semaphore, so that in step 6 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 some of the algorithms used for scheduling later on in this chapter.

In the example of Fig. 2-28, 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. We will study mutual exclusion and how to achieve it in the next section.

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 that the consumer stops running when it is empty. This use is different from mutual exclusion.

The Readers and Writers Problem

The producers-consumers problem is useful for modeling two processes (or threads) that exchange blocks of data while sharing a buffer. Another famous problem is the readers and writers problem (Courtois et al., 1971), which models access to a database. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers. The question is how do you program the readers and the writers? One solution is shown in Fig. 2-29.

In this solution, the first reader to get access to the database does a down on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter, and the last to leave does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

The solution presented here implicitly contains a subtle decision worth noting. Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. Additional readers can also be admitted if they come along.

Now suppose a writer shows up. The writer may not be admitted to the database, since writers must have exclusive access, so the writer must be suspended.

```

typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;           /* controls access to rc */
semaphore db = 1;              /* controls access to the database */
int rc = 0;                    /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {              /* repeat forever */
        down(&mutex);           /* get exclusive access to rc */
        rc = rc + 1;            /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);             /* release exclusive access to rc */
        read_data_base();       /* access the data */
        down(&mutex);           /* get exclusive access to rc */
        rc = rc - 1;            /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader ... */
        up(&mutex);             /* release exclusive access to rc */
        use_data_read();        /* noncritical region */
    }
}

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

```

Figure 2-29. A solution to the readers and writers problem.

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 sec, and each reader takes 5 sec to do its work, the writer will never get in. Obviously, this is not a satisfactory situation.

To avoid 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.

2.4.6 Mutexes

When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.

A **mutex** is a shared variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked. Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls *mutex_lock*. If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls *mutex_unlock*. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

Because mutexes are so simple, they can easily be implemented in user space provided that a TSL or XCHG instruction is available. The code for *mutex_lock* and *mutex_unlock* for use with a user-level threads package are shown in Fig. 2-30. The solution with XCHG is essentially the same.

```
mutex_lock:
    TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
    CMP REGISTER,#0         | was mutex zero?
    JZE ok                  | if it was zero, mutex was unlocked, so return
    CALL thread_yield       | mutex is busy; schedule another thread
    JMP mutex_lock          | try again
ok:    RET                  | return to caller; critical region entered

mutex_unlock:
    MOVE MUTEX,#0           | store a 0 in mutex
    RET                     | return to caller
```

Figure 2-30. Implementation of *mutex_lock* and *mutex_unlock*.

The code of *mutex_lock* is similar to the code of *enter_region* of Fig. 2-25 but with a crucial difference. When *enter_region* fails to enter the critical region, it keeps testing the lock repeatedly (busy waiting). Eventually, the clock runs out and some other process is scheduled to run. Sooner or later the process holding the lock gets to run and releases it.

With (user) threads, the situation is different because there is no clock that stops threads that have run too long. Consequently, a thread that tries to acquire a lock by busy waiting will loop forever and never acquire the lock because it never allows any other thread to run and release the lock.

That is where the difference between *enter_region* and *mutex_lock* comes in. When the later fails to acquire a lock, it calls *thread_yield* to give up the CPU to another thread. Consequently there is no busy waiting. When the thread runs the next time, it tests the lock again.

Since *thread_yield* is just a call to the thread scheduler in user space, it is very fast. As a consequence, neither *mutex_lock* nor *mutex_unlock* requires any kernel calls. Using them, user-level threads can synchronize entirely in user space using procedures that require only a handful of instructions.

The mutex system that we have described above is a bare-bones set of calls. With all software, there is always a demand for more features, and synchronization primitives are no exception. For example, sometimes a thread package offers a call *mutex_trylock* that either acquires the lock or returns a code for failure, but does not block. This call gives the thread the flexibility to decide what to do next if there are alternatives to just waiting.

There is a subtle issue that up until now we have glossed over but which is worth making explicit. With a user-space threads package, there is no problem with multiple threads having access to the same mutex, since all the threads operate in a common address space. However, with most of the earlier solutions, such as Peterson's algorithm and semaphores, there is an unspoken assumption that multiple processes have access to at least some shared memory, perhaps only one word, but something. If processes have disjoint address spaces, as we have consistently said, how can they share the *turn* variable in Peterson's algorithm, or semaphores or a common buffer?

There are two answers. First, some of the shared data structures, such as the semaphores, can be stored in the kernel and accessed only by means of system calls. This approach eliminates the problem. Second, most modern operating systems (including UNIX and Windows) offer a way for processes to share some portion of their address space with other processes. In this way, buffers and other data structures can be shared. In the worst case, if nothing else is possible, a shared file can be used.

If two or more processes share most or all of their address spaces, the distinction between processes and threads becomes somewhat blurred but is nevertheless present. Two processes that share a common address space still have different open files, alarm timers, and other per-process properties, whereas the threads within a single process share them. And it is always true that multiple processes sharing a common address space never have the efficiency of user-level threads since the kernel is deeply involved in their management.

Futexes

With increasing parallelism, efficient synchronization and locking is very important for performance. Spin locks (and mutexes implemented by busy waiting in general) are fast if the wait is short, but waste CPU cycles if not. If there is much

contention, it is therefore more efficient to block the process and let the kernel unblock it only when the lock is free. Unfortunately, this has the inverse problem: it works well under heavy contention, but continuously switching to the kernel is expensive if there is very little contention to begin with. To make matters worse, it may not be easy to predict the amount of lock contention. One nice solution that tries to combine the best of both worlds is the **futex**, or “fast user space mutex.”

A futex is a feature of Linux that implements basic locking (much like a mutex) but avoids dropping into the kernel unless it really has to. Since switching to the kernel and back is quite expensive, doing so improves performance considerably. While we focus our discussion on mutex-style locking, futexes are very versatile and used to implement a range of synchronization primitives, from mutexes to condition variables. They are also a very low-level feature of the kernel that most users will never use directly—instead they are wrapped by standard libraries that offer higher-level primitives. It is only when you lift the hood that you see the futex mechanism powering many different kinds of synchronization.

A futex is a construct supported by the kernel to let user space processes synchronize on shared events. It consists of two parts: a kernel service and a user library. The kernel service provides a “wait queue” that allows multiple processes to wait on a lock. They will not run, unless the kernel explicitly unblocks them. For a process to be put on the wait queue requires an (expensive) system call. If possible, it should be avoided. In the absence of any contention, therefore, the futex works entirely in user space. Specifically, the processes or threads share a common lock variable—a fancy name for an integer in shared memory that serves as the lock. Suppose we have a multithreaded program and the lock is initially `-1` which we assume to mean that the lock is free. A thread may grab the lock by performing an atomic “decrement and test” (atomic functions in Linux consist of inline assembly wrapped in C functions and are defined in header files). Next, the thread inspects the result to see whether or not the lock was free. If it was not in the locked state, all is well and our thread has successfully grabbed the lock.

However, if the lock is held by another thread, our thread has to wait. In that case, the futex library does not spin, but uses a system call to put the thread on the wait queue in the kernel. Hopefully, the cost of the switch to the kernel is now justified, because the thread was blocked anyway. When a thread is done with the lock, it releases the lock with an atomic “increment and test” and checks the result to see if any processes are still blocked on the kernel wait queue. If so, it will let the kernel know that it may wake up (unblock) one or more of these processes. In other words, if there is no contention, the kernel is not involved at all.

Mutexes in Pthreads

Pthreads provides a number of functions for synchronizing threads. The basic mechanism uses a mutex variable, which can be locked or unlocked, to guard each critical region. The implementation of a mutex varies from operating system to

operating system, but on Linux it is built on top of futexes. A thread wishing to enter a critical region first tries to lock the associated mutex. If the mutex is unlocked, the thread can enter immediately and the lock is atomically set, preventing other threads from entering. If the mutex is already locked, the calling thread is blocked until it is unlocked. If multiple threads are waiting on the same mutex, when it is unlocked, only one of them is allowed to continue and relock it. These locks are not mandatory. It is up to the programmer to make sure threads use them correctly.

The major calls relating to mutexes are shown in Fig. 2-31. As you might expect, mutexes can be created and destroyed. The calls for performing these operations are *pthread_mutex_init* and *pthread_mutex_destroy*, respectively. They can also be locked—by *pthread_mutex_lock*—which tries to acquire the lock and blocks if it is already locked. There is also an option for trying to lock a mutex and failing with an error code instead of blocking if it is already blocked. This call is *pthread_mutex_trylock*. This call allows a thread to effectively do busy waiting if that is ever needed. Finally, *pthread_mutex_unlock* unlocks a mutex and releases exactly one thread if one or more are waiting on it. Mutexes can also have attributes, but these are used only for specialized purposes.

Thread call	Description
<i>pthread_mutex_init</i>	Create a mutex
<i>pthread_mutex_destroy</i>	Destroy an existing mutex
<i>pthread_mutex_lock</i>	Acquire a lock or block
<i>pthread_mutex_trylock</i>	Acquire a lock or fail
<i>pthread_mutex_unlock</i>	Release a lock

Figure 2-31. Some of the Pthreads' calls relating to mutexes.

In addition to mutexes, Pthreads offers a second synchronization mechanism, condition variables, discussed later. Mutexes are good for allowing or blocking access to a critical region. Condition variables allow threads to block due to some condition not being met. Almost always the two methods are used together. Let us now look at the interaction of threads, mutexes, and condition variables in a bit more detail.

As a simple example, consider the producer-consumer scenario again: one thread puts things in a buffer and another one takes them out. If the producer discovers that there are no more free slots available in the buffer, it has to block until one becomes available. Mutexes make it possible to do the check atomically without interference from other threads, but having discovered that the buffer is full, the producer needs a way to block and be awakened later. This is what condition variables allow.

The most important calls related to condition variables are shown in Fig. 2-32. As you would probably expect, there are calls to create and destroy condition

variables. They can have attributes and there are various calls for managing the attributes (not shown). The most important operations on condition variables are *pthread_cond_wait* and *pthread_cond_signal*. The former blocks the calling thread until some other thread signals it (using the latter call). The reasons for blocking and waiting are not part of the waiting and signaling protocol, of course. The blocking thread often is waiting for the signaling thread to do some work, release some resource, or perform some other activity. Only then can the blocking thread continue. The condition variables allow this waiting and blocking to be done atomically. The *pthread_cond_broadcast* call is used when there are multiple threads potentially all blocked and waiting for the same signal.

Thread call	Description
<code>pthread_cond_init</code>	Create a condition variable
<code>pthread_cond_destroy</code>	Destroy a condition variable
<code>pthread_cond_wait</code>	Block waiting for a signal
<code>pthread_cond_signal</code>	Signal another thread and wake it up
<code>pthread_cond_broadcast</code>	Signal multiple threads and wake all of them

Figure 2-32. Some of the Pthreads calls relating to condition variables.

Condition variables and mutexes are always used together. The pattern is for one thread to lock a mutex, then wait on a conditional variable when it cannot get what it needs. Eventually another thread will signal it and it can continue. The *pthread_cond_wait* call atomically unlocks the mutex it is holding. Then, upon successful return, the mutex shall have been locked again and owned by the calling thread. For this reason, the mutex is one of the parameters.

It is also worth noting that condition variables (unlike semaphores) have no memory. If a signal is sent to a condition variable on which no thread is waiting, the signal is lost. Programmers have to be careful not to lose signals.

As an example of how mutexes and condition variables are used, Fig. 2-33 shows a very simple producer-consumer problem with a single item buffer. When the producer has filled the buffer, it must wait until the consumer empties it before producing the next item. Similarly, when the consumer has removed an item, it must wait until the producer has produced another one. While very simple, this example illustrates the basic mechanisms. The statement that puts a thread to sleep should always check the condition to make sure it is satisfied before continuing, as the thread might have been awakened due to a UNIX signal or some other reason.

2.4.7 Monitors

With semaphores and mutexes interprocess communication looks easy, right? Forget it. Look closely at the order of the downs before inserting or removing items from the buffer in Fig. 2-28. Suppose that the two downs in the producer's code


```

#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                                /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                          /* used for signaling */
int buffer = 0;                                       /* buffer used between producer and consumer */

void *producer(void *ptr)                            /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i;                       /* put item in buffer */
        pthread_cond_signal(&condc);      /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr)                            /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0;                       /* take item from buffer (not shown) and reinitialize */
        pthread_cond_signal(&condp);      /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}

```

Figure 2-33. Using threads to solve the producer-consumer problem.

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 deadlock. We will study deadlocks in detail in Chap. 6.

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, Brinch Hansen (1973) and Hoare (1974) proposed a higher-level synchronization primitive called a **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 2-34 illustrates a monitor written in an imaginary language, Pidgin Pascal. C cannot be used here because monitors are a *language* concept and C does not have them.

```
monitor example
  integer i;
  condition c;

  procedure producer();
  .
  .
  end;

  procedure consumer();
  .
  .
  end;
end monitor;
```

Figure 2-34. 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 mutual exclusion on monitor entries, but a common way is to use a mutex or 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 again 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. We saw condition variables and these operations in the context of Pthreads earlier.

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 finessing 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.

As an aside, there is also a third solution, not proposed by either Hoare or Brinch Hansen. This is to let the signaler continue to run and allow the waiting process to start running only after the signaler has exited the monitor.

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 forever. In other words, 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. 2-35 in Pidgin Pascal. The advantage of using Pidgin Pascal here is that it is pure and simple and follows the Hoare/Brinch Hansen model exactly.

```

monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

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

  count := 0;
end monitor;

procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end;

procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
  end;
end;

```

Figure 2-35. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

You may be thinking that the operations `wait` and `signal` look similar to `sleep` and `wakeup`, which we saw earlier had fatal race conditions. Well, 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 without having to

worry about the possibility that the scheduler may switch to the consumer just before the `wait` completes. The consumer will not even be let into the monitor at all until the `wait` is finished and the producer has been marked as no longer runnable.

Although Pidgin Pascal is an imaginary language, some real programming languages also support monitors, although not always in the form designed by Hoare and Brinch Hansen. One such language is Java. Java is an object-oriented language that supports user-level threads and also allows methods (procedures) to be grouped together into classes. By adding the keyword `synchronized` to a method declaration, Java guarantees that once any thread has started executing that method, no other thread will be allowed to start executing any other `synchronized` method of that object. Without `synchronized`, there are no guarantees about interleaving.

A solution to the producer-consumer problem using monitors in Java is given in Fig. 2-36. Our solution has four classes. The outer class, *ProducerConsumer*, creates and starts two threads, *p* and *c*. The second and third classes, *producer* and *consumer*, respectively, contain the code for the producer and consumer. Finally, the class *our_monitor*, is the monitor. It contains two `synchronized` threads that are used for actually inserting items into the shared buffer and taking them out. Unlike the previous examples, here we have the full code of *insert* and *remove*.

The producer and consumer threads are functionally identical to their counterparts in all our previous examples. The producer has an infinite loop generating data and putting it into the common buffer. The consumer has an equally infinite loop taking data out of the common buffer and doing some fun thing with it.

The interesting part of this program is the class *our_monitor*, which holds the buffer, the administration variables, and two `synchronized` methods. When the producer is active inside *insert*, it knows for sure that the consumer cannot be active inside *remove*, making it safe to update the variables and the buffer without fear of race conditions. The variable *count* keeps track of how many items are in the buffer. It can take on any value from 0 through and including $N - 1$. The variable *lo* is the index of the buffer slot where the next item is to be fetched. Similarly, *hi* is the index of the buffer slot where the next item is to be placed. It is permitted that $lo = hi$, which means that either 0 items or N items are in the buffer. The value of *count* tells which case holds.

`Synchronized` methods in Java differ from classical monitors in an essential way: Java does not have condition variables built in. Instead, it offers two procedures, *wait* and *notify*, which are the equivalent of *sleep* and *wakeup* except that when they are used inside `synchronized` methods, they are not subject to race conditions. In theory, the method *wait* can be interrupted, which is what the code surrounding it is all about. Java requires that the exception handling be made explicit. For our purposes, just imagine that *go_to_sleep* is the way to go to sleep.

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error prone than using semaphores. Nevertheless, they too have some drawbacks. It is not for nothing that our two examples of monitors were in Pidgin Pascal instead of C, as are the other examples in this book.

```

public class ProducerConsumer {
    static final int N = 100;    // constant giving the buffer size
    static producer p = new producer();    // instantiate a new producer thread
    static consumer c = new consumer();    // instantiate a new consumer thread
    static our_monitor mon = new our_monitor();    // instantiate a new monitor

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

    static class producer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) {    // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... }    // actually produce
    }

    static class consumer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) {    // consumer loop
                item = mon.remove();
                consume_item(item);
            }
        }
        private void consume_item(int item) { ... } // actually consume
    }

    static class our_monitor { // this is a monitor
        private int buffer[] = new int[N];
        private int count = 0, lo = 0, hi = 0;    // counters and indices

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep();    // if the buffer is full, go to sleep
            buffer[hi] = val;    // insert an item into the buffer
            hi = (hi + 1) % N;    // slot to place next item in
            count = count + 1;    // one more item in the buffer now
            if (count == 1) notify();    // if consumer was sleeping, wake it up
        }

        public synchronized int remove() {
            int val;
            if (count == 0) go_to_sleep();    // if the buffer is empty, go to sleep
            val = buffer[lo];    // fetch an item from the buffer
            lo = (lo + 1) % N;    // slot to fetch next item from
            count = count - 1;    // one fewer items in the buffer
            if (count == N - 1) notify();    // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
    }
}

```

Fig. 2-36. A solution to the producer-consumer problem in Java.

As we said earlier, monitors are a programming-language concept. The compiler must recognize them and arrange for the mutual exclusion somehow or other. 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 up and 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 assembly language if you are masochistic enough). With monitors, you need a language that has them built in.

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 or XCHG instructions, we can avoid races. When we move to a distributed system consisting of multiple CPUs, each with its own private memory and connected by a local area network, these primitives become inapplicable. The conclusion is that semaphores are too low level and monitors are not usable except in a few programming languages. Also, none of the primitives allow information exchange between machines. Something else is needed.

2.4.8 Message Passing

That something else is **message passing**. This method of interprocess communication uses two primitives, **send** and **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

```
send(destination, &message);
```

and

```
receive(source, &message);
```

The former call sends a message to a given destination and the latter one receives a message from a given source (or from *ANY*, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

Design Issues for Message-Passing Systems

Message-passing systems have many problems and design issues that do not arise with semaphores or with 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 is received correctly, but the acknowledgement back to the sender is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver be able to distinguish a new message from the retransmission of an old one. 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. Successfully communicating in the face of unreliable message passing is a major part of the study of computer networks. For more information, see Tanenbaum et al. (2020).

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 it 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.

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. 2-37. 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 empties waiting for the producer to fill them up; the consumer will be blocked, waiting for a full message.

Many variants are possible with message passing. For starters, let us look at how messages are addressed. One way is 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


```

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

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

    while (TRUE) {
        item = produce_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 */
        item = extract_item(&m);             /* extract item from message */
        send(producer, &m);                  /* send back empty reply */
        consume_item(item);                  /* do something with the item */
    }
}

```

Figure 2-37. The producer-consumer problem with N messages.

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.

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 actual 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 taken, 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 buffering. Similarly, if the `receive` call 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.

Message passing is commonly used in parallel programming systems. One well-known message-passing system, for example, is **MPI (Message-Passing Interface)**. It is widely used for scientific computing. For more information about it, see for example Gropp et al. (1994) and Snir et al. (1996).

2.4.9 Barriers

Our last synchronization mechanism is intended for groups of processes rather than two-process producer-consumer type situations. Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase. This behavior may be achieved by placing a **barrier** at the end of each phase. When a process reaches the barrier, it is blocked until all processes have reached the barrier. This allows groups of processes to synchronize. Barrier operation is illustrated in Fig. 2-38.

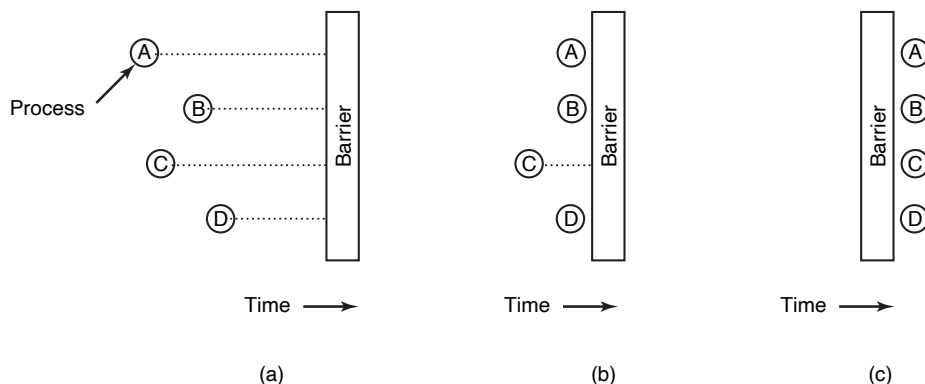


Figure 2-38. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

In Fig. 2-38(a) we see four processes approaching a barrier. What this means is that they are just computing and have not reached the end of the current phase yet. After a while, the first process finishes all the computing required of it during the first phase. It then executes the barrier primitive, generally by calling a library procedure. The process is then suspended. A little later, a second and then a third process finish the first phase and also execute the barrier primitive. This situation is illustrated in Fig. 2-38(b). Finally, when the last process, C, hits the barrier, all the processes are released, as shown in Fig. 2-38(c).

As an example of a problem requiring barriers, consider a common relaxation problem in physics or engineering. There is typically a matrix that contains some

initial values. The values might represent temperatures at various points on a sheet of metal. The idea might be to calculate how long it takes for the effect of a flame placed at one corner to propagate throughout the sheet.

Starting with the current values, a transformation is applied to the matrix to get the second version of the matrix, for example, by applying the laws of thermodynamics to see what all the temperatures are ΔT later. Then the process is repeated over and over, giving the temperatures at the sample points as a function of time as the sheet heats up. The algorithm produces a sequence of matrices over time, each one for a given point in time.

Now imagine that the matrix is very large (for example, 1 million by 1 million), so that parallel processes are needed (possibly on a multiprocessor) to speed up the calculation. Different processes work on different parts of the matrix, calculating the new matrix elements from the old ones according to the laws of physics. However, no process may start on iteration $n + 1$ until iteration n is complete, that is, until all processes have finished their current work. The way to achieve this goal is to program each process to execute a barrier operation after it has finished its part of the current iteration. When all of them are done, the new matrix (the input to the next iteration) will be finished, and all processes will be simultaneously released to start the next iteration.

It is worth mentioning that special low-level barriers are popular also to synchronize memory operations. Such barriers, unimaginatively called **memory barriers** or **memory fences**, enforce an order to guarantee that all memory operations (to read or write memory) started before the barrier instruction will also finish before the memory operations issued after the barrier. They are important because modern CPUs execute instructions out of order and that may cause problems. For instance, if instruction 2 does not depend on the result of instruction 1, the CPU may start executing it ahead of time. After all, modern processors are superscalar and have many execution units to perform calculations and memory accesses in parallel. In fact, if instruction 1 takes a long time, instruction 2 may even complete before it, and the CPU may then start executing instruction 3. Now consider the situation where one thread waits on another using busy waiting:

THREAD 1:

```
while (turn != 1) {} /* loop */  
printf ("%d\n", x);
```

THREAD 2:

```
x = 100;  
turn = 1;
```

If `turn == 0` initially and all instructions execute in order, the program will print the value 100. However, if the instructions in Thread 2 execute out of order, `turn` will be updated before `x` and the printed value could be some older value of `x`. Similarly, the instructions of Thread 1 may be reordered, making it read `x` before performing the check in the line above it. The solution in both cases is to wedge a barrier instruction in between the two lines.

Incidentally, memory barriers often play an important role in the mitigation of a nasty class of CPU vulnerabilities that are commonly referred to as **transient**

execution vulnerabilities. Here, attackers can exploit the fact that CPUs execute instructions out of order. Between the first disclosure of the Meltdown and Spectre issues in 2018, many such vulnerabilities have come to light. Since they generally impact the operating system also, we will briefly look at transient execution attacks in Chap. 9.

2.4.10 Priority Inversion

Earlier in this chapter, we mentioned the priority inversion problem, a truly classic problem that was known already in the 1970s. Now let us look at it in more detail.

A famous example of priority inversion occurred on Mars in 1997. In an impressive engineering effort, NASA had managed to land a little robot rover on the red planet, destined to send a wealth of interesting information back to earth. Except there was a problem. Pathfinder's radio transmissions stopped sending data constantly, requiring system resets to get it going again. It turned out that three threads were getting in each other's hair. Pathfinder used a form of shared memory, which was called the "information bus," for passing information between its different components. A low-priority thread used the bus periodically to pass on the meteorological data (a kind of Mars weather report) it had gathered. Meanwhile, a high priority thread for information bus management would also periodically access it. To prevent both threads from accessing the shared memory at the same time, its access was controlled by a mutex in the rover's software. A third, medium-priority thread was responsible for communications and did not need the mutex at all.

The priority inversion occurred when the low-priority thread for meteorological data gathering had been preempted by the medium priority communications thread, while holding the mutex. After some time, the high-priority thread needed to run but immediately blocked as it could not grab the mutex. The long-running medium-priority thread kept executing, as if it had higher priority than the information bus thread.

There are different ways to solve the priority inversion problem. The simplest one is to disable all interrupts while in the critical region. As mentioned earlier, this is not desirable for user programs: What if they forget to enable them again?

Another solution, known as **priority ceiling** is to associate a priority with the mutex itself and assign that to the process holding it. As long as no process that needs to grab the mutex has a higher priority than the ceiling priority, inversion is no longer possible.

A third way is **priority inheritance**. Here, the low-priority task holding the mutex will temporarily inherit the priority of the high-priority task trying to obtain it. Again, no medium priority task will be able to preempt the task holding the mutex. This was the technique eventually used to fix the Mars Pathfinder problems.

Finally, operating systems such as Microsoft Windows employ **random boosting**, essentially rolling the dice every now and then and giving random mutex-holding threads a high priority until they exit the critical region.

2.4.11 Avoiding Locks: Read-Copy-Update

The fastest locks are no locks at all. And no locks also means no risk of priority inversion. The question is whether we can allow for concurrent read and write accesses to shared data structures without locking. In the general case, the answer is clearly no. Imagine thread A sorting an array of numbers, while thread B is calculating the average. Because A moves the values back and forth across the array, B may encounter some values multiple times and others not at all. The result could be anything, but it would almost certainly be wrong.

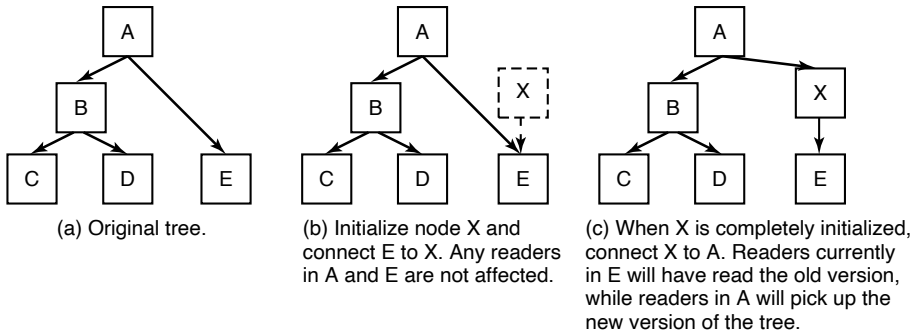
In some cases, however, we can allow a writer to update a data structure even though other processes are still using it. The trick is to ensure that each reader either reads the old version of the data, or the new one, but not some weird combination of old and new. As an illustration, consider the tree shown in Fig. 2-39.

Readers traverse the tree from the root to its leaves. In the top half of the figure, a new node X is added. To do so, we make the node “just right” before making it visible in the tree: we initialize all values in node X, including its child pointers. Then, with one atomic write, we make X a child of A. No reader will ever read an inconsistent version. In the bottom half of the figure, we subsequently remove B and D. First, we make A’s left child pointer point to C. All readers that were in A will continue with node C and never see B or D. In other words, they will see only the new version. Likewise, all readers currently in B or D will continue following the original data structure pointers and see the old version. All is well, and we never need to lock anything. The main reason that the removal of B and D works without locking the data structure, is that **RCU (Read-Copy-Update)** decouples the *removal* and *reclamation* phases of the update.

Of course, there is a problem. As long as we are not sure that there are no more readers of B or D, we cannot really free them. But how long should we wait? One minute? Ten? We have to wait until the last reader has left these nodes. RCU carefully determines the maximum time a reader may hold a reference to the data structure. After that period, it can safely reclaim the memory. Specifically, readers access the data structure in what is known as a **read-side critical section** which may contain any code, as long as it does not block or sleep. In that case, we know the maximum time we need to wait. Specifically, we define a **grace period** as any time period in which we know that each thread to be outside the read-side critical section at least once. All will be well if we wait for a duration that is at least equal to the grace period before reclaiming. As the code in a read-side critical section is not allowed to block or sleep, a simple criterion is to wait until all the threads have executed a context switch.

RCU data structures are not so common in user processes, but quite popular in operating system kernels for data structures that are accessed by multiple threads and require high efficiency. The Linux kernel has thousands of uses of its RCU API, spread across most of its subsystems. The network stack, the file system, drivers, and memory management all use RCU for concurrent reading and writing.

Adding a node:



Removing nodes:

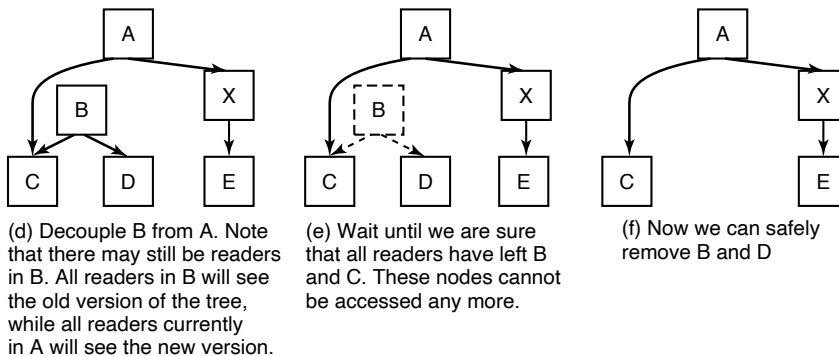


Figure 2-39. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks.

2.5 SCHEDULING

When a computer is multiprogrammed, it frequently has multiple processes or threads competing for the CPU at the same time. This situation occurs whenever two or more of them are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the **scheduler**, and the algorithm it uses is called the **scheduling algorithm**. These topics form the subject matter of the following sections.

Many of the same issues that apply to process scheduling also apply to thread scheduling, although some are different. When the kernel manages threads, scheduling is usually done per thread, with little or no regard to which process the thread belongs. Initially we will focus on scheduling issues that apply to both processes and threads. Later on we will explicitly look at thread scheduling and some of the unique issues it raises. We will deal with multicore chips in Chap. 8.

2.5.1 Introduction to Scheduling

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 multiprogramming, the scheduling algorithm became more complex because there were usually multiple users waiting for service. Some mainframes still combine batch and timesharing service, requiring the scheduler to decide whether a batch job or an interactive user at a terminal should go next. (As an aside, a batch job may be a request to run multiple programs in succession, but for this section, we will just assume it is a request to run a single program.) Because CPU time is a scarce resource on these machines, a good scheduler can make a big difference in perceived performance and user satisfaction. Consequently, a lot of work has gone into devising clever and efficient scheduling algorithms.

With the advent of personal computers, the situation changed in two ways. First, most of the time there is only one active process. A user entering a document on a word processor is unlikely to be simultaneously compiling a program in the background. When the user types a command to the word processor, the scheduler does not have to do much work to figure out which process to run—the word processor is the only candidate.

Second, computers have become so much faster over the years that the CPU is rarely a scarce resource any more. Most programs for personal computers are limited by the rate at which the user can present input (by typing or clicking), not by the rate the CPU can process it. Even compilations, a major sink of CPU cycles in the past, take just a few seconds in most cases nowadays. Even when two programs are actually running at once, such as a word processor and a spreadsheet, it hardly matters which goes first since the user is probably waiting for both of them to finish (except that they generally complete their tasks so quickly that the user will not be doing much waiting much anyway). As a consequence, scheduling does not matter much on simple PCs. Of course, there are applications that practically eat the CPU alive. For instance, rendering one hour of high-resolution video while tweaking the colors in each of the 107,892 frames (in NTSC) or 90,000 frames (in PAL) requires serious computing power. However, similar applications are the exception rather than the rule.

When we turn to networked servers, the situation changes appreciably. Here multiple processes often do compete for the CPU, so scheduling matters again. For example, when the CPU has to choose between running a process that gathers the

daily statistics and one that serves user requests, the users will be a lot happier if the latter gets first crack at the CPU.

The “abundance of resources” argument also does not hold on IoT devices and sensor nodes, and perhaps not even on smartphones. Even if the CPUs on phones have become more powerful and the memory more plentiful, battery lifetime has not. Since battery lifetime is one of the most important constraints on all these devices, some schedulers try to optimize the power consumption.

In addition to picking the right process to run, the scheduler also has to worry about making efficient use of the CPU because process switching is expensive. To start with, a switch from user mode to kernel mode must occur. Then the state of the current process must be saved, including storing its registers in the process table so they can be reloaded later. In some systems, the memory map (e.g., memory reference bits in the page table) must be saved as well. This is called a **context switch**, although people sometimes also use this term to refer to the full **process switch**. Next a new process must be selected by running the scheduling algorithm. After that, the memory management unit (MMU) must be reloaded with the memory map of the new process. Finally, the new process must be started. In addition to all that, the process switch may invalidate the memory cache and related tables, forcing it to be dynamically reloaded from the main memory twice (upon entering the kernel and upon leaving it). All in all, doing too many process switches per second can chew up a substantial amount of CPU time, so caution is advised.

Process Behavior

Nearly all processes alternate bursts of computing with (disk or network) I/O requests, as shown in Fig. 2-40. Often, the CPU runs for a while without stopping, then a system call is made to read from a file or write to a file. When the system call completes, the CPU computes again until it needs more data or has to write more data, and so on. Note that some I/O activities count as computing. For example, when the CPU copies bits to a video RAM to update the screen, it is computing, not doing I/O, because the CPU is in use. I/O in this sense is when a process enters the blocked state waiting for an external device to complete its work.

The important thing to notice about Fig. 2-40 is that some processes, such as the one in Fig. 2-40(a), spend most of their time computing, while other processes, such as the one shown in Fig. 2-40(b), spend most of their time waiting for I/O. The former are called **compute-bound** or **CPU-bound**; the latter are called **I/O-bound**. Compute-bound processes typically have long CPU bursts and thus infrequent I/O waits, whereas I/O-bound processes have short CPU bursts and thus frequent I/O waits. Note that the key factor is the length of the CPU burst, not the length of the I/O burst. I/O-bound processes are I/O bound because they do not compute much between I/O requests, not because they have especially long I/O requests. It takes the same time to issue the hardware request to read a disk block no matter how much or how little time it takes to process the data after they arrive.

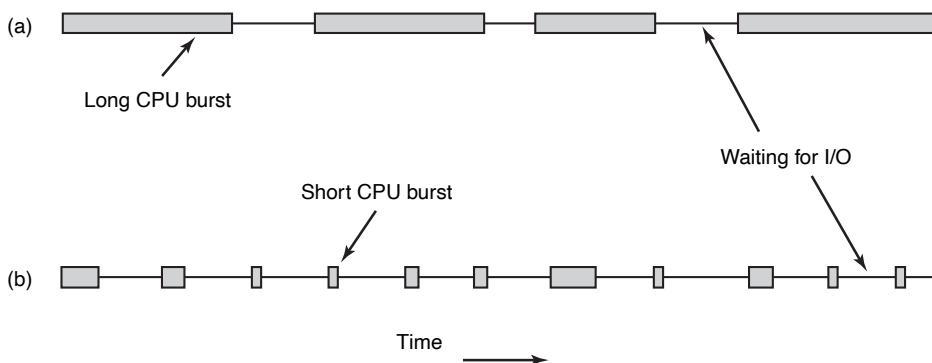


Figure 2-40. Bursts of CPU usage alternate with periods of waiting for I/O.

(a) A CPU-bound process. (b) An I/O-bound process.

It is worth noting that if CPUs get faster, processes tend to get more I/O-bound. This effect occurs because CPUs are improving faster than hard disks. As a consequence, the scheduling of I/O-bound processes could become a more important subject in the future. The basic idea here is that if an I/O-bound process wants to run, it should get a chance quickly so that it can issue its disk request and keep the disk busy. As we saw in Fig. 2-6, when processes are I/O bound, it takes quite a few of them to keep the CPU fully occupied.

On the other hand, CPUs do not seem to be getting much faster these days because making them go faster produces too much heat. Hard disks are not getting any faster either, but SSDs are replacing hard disks in desktop and notebook computers. Then again, in large data centers, hard disks are still widely used due to their lower cost per bit. The consequence of all this is that scheduling depends a lot on the context and an algorithm that works well on a notebook may not work well in a data center. And in 10 years, everything may be different.

When to Schedule

A key issue related to scheduling is when to make scheduling decisions. It turns out that there are a variety of situations in which scheduling is needed. First, when a new process is created, a decision needs to be made whether to run the parent process or the child process. Since both processes are in ready state, it is a normal scheduling decision and can go either way, that is, the scheduler can legitimately choose to run either the parent or the child next.

Second, a scheduling decision must be made when a process exits. That process can no longer run (since it no longer exists), so some other process must be chosen from the set of ready processes. If no process is ready, a system-supplied idle process is normally run.

Third, when a process blocks on I/O, on a semaphore, or for some other reason, another process has to be selected to run. Sometimes the reason for blocking may play a role in the choice. For example, if *A* is an important process and it is waiting for *B* to exit its critical region, letting *B* run next will allow it to exit its critical region and thus let *A* continue. The trouble, however, is that the scheduler generally does not have the necessary information to take this dependency into account.

Fourth, when an I/O interrupt occurs, a scheduling decision may be made. If the interrupt came from an I/O device that has now completed its work, some process that was blocked waiting for the I/O may now be ready to run. It is up to the scheduler to decide whether to run the newly ready process, the process that was running at the time of the interrupt, or some third process.

If a hardware clock provides periodic interrupts at 50 or 60 Hz (or possibly at some other—potentially higher—frequency), a scheduling decision can be made at each clock interrupt or at every *k*th clock interrupt. Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts. A **nonpreemptive** scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or voluntarily releases the CPU. Even if it runs for many hours, it will not be forcibly suspended. In effect, no scheduling decisions are made during clock interrupts. After clock-interrupt processing has been finished, the process that was running before the interrupt is resumed, unless a higher-priority process was waiting for a now-satisfied timeout.

In contrast, a **preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler. If no clock is available, nonpreemptive scheduling is the only option.

Preemption is not just relevant for applications, but also for operating system kernels, especially monolithic ones. Nowadays many of them are preemptive. If they were not, a poorly implemented driver or a very slow system call could hog the CPU. Instead, in a preemptive kernel the scheduler can force the long running driver or system call to context switch.

Categories of Scheduling Algorithms

Not surprisingly, in different environments different scheduling algorithms are needed. This situation arises because different application areas (and different kinds of operating systems) have different goals. In other words, what the scheduler should optimize for is not the same in all systems. Three environments worth distinguishing are:

1. Batch
2. Interactive
3. Real time

Batch systems are still in widespread use in the business world for doing payroll, inventory, accounts receivable, accounts payable, interest calculation (at banks), claims processing (at insurance companies), and other periodic tasks. In batch systems, there are no users impatiently waiting at their terminals for a quick response to a short request. Consequently, nonpreemptive algorithms, or preemptive algorithms with long time periods for each process, are often acceptable. This approach reduces process switches and thus improves performance. The batch algorithms are actually fairly general and often applicable to other situations as well, which makes them worth studying, even for people not involved in corporate mainframe computing.

In an environment with interactive users, preemption is essential to keep one process from hogging the CPU and denying service to the others. Even if no process intentionally ran forever, one process might shut out all the others indefinitely due to a program bug. Preemption is needed to prevent this behavior. Servers also fall into this category, since they normally serve multiple (remote) users, all of whom are in a big hurry. Computer users are always in a big hurry.

In systems with real-time constraints, preemption is, oddly enough, sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are intended to further the application at hand. Interactive systems are general purpose and may run arbitrary programs that are not cooperative and even possibly malicious.

Scheduling Algorithm Goals

In order to design a scheduling algorithm, it is necessary to have some idea of what a good algorithm should do. Some goals depend on the environment (batch, interactive, or real time), but some are desirable in all cases. Some goals are listed in Fig. 2-41. We will discuss these in turn below.

Under all circumstances, fairness is important. Comparable processes should get comparable service. Giving one process much more CPU time than an equivalent one is not fair. Of course, different categories of processes may be treated differently. Think of safety control and doing the payroll on a nuclear reactor's computer.

Somewhat related to fairness is actually enforcing the system's policies. If the local policy is that safety control processes get to run whenever they want to, even if it means the payroll is 30 sec late, the scheduler has to make sure this policy is enforced. That may take some extra effort.

All systems

- Fairness—giving each process a fair share of the CPU
- Policy enforcement—seeing that stated policy is carried out
- Balance—keeping all parts of the system busy

Batch systems

- Throughput—maximize jobs per hour
- Turnaround time—minimize time between submission and termination
- CPU utilization—keep the CPU busy all the time

Interactive systems

- Response time—respond to requests quickly
- Proportionality—meet users' expectations

Real-time systems

- Meeting deadlines—avoid losing data
- Predictability—avoid quality degradation in multimedia systems

Figure 2-41. Some goals of the scheduling algorithm under different circumstances.

Another general goal is keeping all parts of the system busy when possible. If the CPU and all the I/O devices can be kept running all the time, more work gets done per second than if some of the components are idle. In a batch system, for example, the scheduler has control of which jobs are brought into memory to run. Having some CPU-bound processes and some I/O-bound processes in memory together is a better idea than first loading and running all the CPU-bound jobs and then, when they are finished, loading and running all the I/O-bound jobs. If the latter strategy is used, when the CPU-bound processes are running, they will fight for the CPU and the disk will be idle. Later, when the I/O-bound jobs come in, they will fight for the disk and the CPU will be idle. Better to keep the whole system running at once by a careful mix of processes.

The managers of large data centers that run many batch jobs typically look at three metrics to see how well their systems are performing: throughput, turnaround time, and CPU utilization. **Throughput** is the number of jobs per hour that the system completes. All things considered, finishing 50 jobs per hour is better than finishing 40 jobs per hour. **Turnaround time** is the statistically average time from the moment that a batch job is submitted until the moment it is completed. It measures how long the average user has to wait for the output. Here the rule is: Small is Beautiful.

A scheduling algorithm that tries to maximize throughput may not necessarily minimize turnaround time. For example, given a mix of short jobs and long jobs, a scheduler that always ran short jobs and never ran long jobs might achieve an

excellent throughput (many short jobs per hour) but at the expense of a terrible turnaround time for the long jobs. If short jobs kept arriving at a fairly steady rate, the long jobs might never run, making the mean turnaround time infinite while achieving a high throughput.

CPU utilization is often used as a metric on batch systems. Actually though, it is not a good metric. What really matters is how many jobs per hour come out of the system (throughput) and how long it takes to get a job back (turnaround time). Using CPU utilization as a metric is like rating cars based on how many times per hour the engine turns over. However, knowing when the CPU utilization is almost 100% is useful for knowing when it is time to get more computing power.

For interactive systems, different goals apply. The most important one is to minimize **response time**, that is, the time between issuing a command and getting the result. On a personal computer where a background process is running (for example, reading and storing email from the network), a user request to start a program or open a file should take precedence over the background work. Having all interactive requests go first will be perceived as good service.

A somewhat related issue is what might be called **proportionality**. Users have an inherent (but often incorrect) idea of how long things should take. When a request that the user perceives as complex takes a long time, users accept that, but when a request that is perceived as simple takes a long time, users get irritated. For example, if clicking on an icon that starts uploading a 5-GB video to a cloud server takes 60 sec, the user will probably accept that as a fact of life because he does not expect the upload to take 5 sec. He knows it will take time.

On the other hand, when a user clicks on the icon that breaks the connection to the cloud server after the video has been uploaded, he has different expectations. If it has not completed after 30 sec, the user will probably be swearing a blue streak, and after 60 sec he will be foaming at the mouth. This behavior is due to the common user perception that sending a lot of data is *supposed* to take a lot longer than just breaking the connection. In some cases (such as this one), the scheduler cannot do anything about the response time, but in other cases it can, especially when the delay is due to a poor choice of process order.

Real-time systems have different properties than interactive systems, and thus different scheduling goals. They are characterized by having deadlines that must or at least should be met. For example, if a computer is controlling a device that produces data at a regular rate, failure to run the data-collection process on time may result in lost data. Thus, the foremost need in a real-time system is meeting all (or most) deadlines.

In some real-time systems, especially those involving multimedia, predictability is important. Missing an occasional deadline is not fatal, but if the audio process runs too erratically, the sound quality will deteriorate rapidly. Video is also an issue, but the ear is much more sensitive to jitter than the eye. To avoid this problem, process scheduling must be highly predictable and regular. We will study batch and interactive scheduling algorithms in this chapter.

2.5.2 Scheduling in Batch Systems

It is now time to turn from general scheduling issues to specific scheduling algorithms. In this section, we will look at algorithms used in batch systems. In the following ones, we will examine interactive and real-time systems. It is worth pointing out that some algorithms are used in both batch and interactive systems. We will study these later.

First-Come, First-Served

Probably the simplest of all scheduling algorithms ever devised is nonpreemptive **first-come, first-served**. With this algorithm, processes are assigned the CPU in the order they request it. Basically, there is a single queue of ready processes. When the first job enters the system from the outside in the morning, it is started immediately and allowed to run as long as it wants to. It is not interrupted because it has run too long. As other jobs come in, they are put onto the end of the queue. When the running process blocks, the first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue, behind all waiting processes.

The great strength of this algorithm is that it is easy to understand and equally easy to program. It is also fair in the same sense that allocating scarce concert tickets or brand-new iPhones to people who are willing to stand on line starting at 2 A.M. is fair. With this algorithm, a single linked list keeps track of all ready processes. Picking a process to run just requires removing one from the front of the queue. Adding a new job or unblocked process just requires attaching it to the end of the queue. What could be simpler to understand and implement?

Unfortunately, first-come, first-served also has a powerful disadvantage. Suppose there is one compute-bound process that runs for 1 sec at a time and many I/O-bound processes that use little CPU time but each have to perform 1000 disk reads to complete. The compute-bound process runs for 1 sec, then it reads a disk block. All the I/O processes now run and start disk reads. When the compute-bound process gets its disk block, it runs for another 1 sec, followed by all the I/O-bound processes in quick succession.

The net result is that each I/O-bound process gets to read 1 block per second and will take 1000 sec to finish. With a scheduling algorithm that preempted the compute-bound process every 10 msec, the I/O-bound processes would finish in 10 sec instead of 1000 sec, without slowing down the compute-bound process much.

Shortest Job First

Now let us look at another nonpreemptive batch algorithm that assumes the run times are known in advance. In an insurance company, for example, people can predict quite accurately how long it will take to run a batch of 1000 claims, since

similar work is done every day. When several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the **shortest job first**. Look at Fig. 2-42. Here we find four jobs *A*, *B*, *C*, and *D* with run times of 8, 4, 4, and 4 minutes, respectively. By running them in that order, the turnaround time for *A* is 8 minutes, for *B* is 12 minutes, for *C* is 16 minutes, and for *D* is 20 minutes for an average of 14 minutes.



Figure 2-42. An example of shortest-job-first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest-job-first order.

Now let us consider running these four jobs using shortest job first, as shown in Fig. 2-42(b). The turnaround times are now 4, 8, 12, and 20 minutes for an average of 11 minutes. Shortest job first is provably optimal. Consider the case of four jobs, with execution times of a , b , c , and d , respectively. The first job finishes at time a , the second at time $a + b$, and so on. The mean turnaround time is $(4a + 3b + 2c + d)/4$. It is clear that a contributes more to the average than the other times, so it should be the shortest job, with b next, then c , and finally d as the longest since it affects only its own turnaround time. The same argument applies equally well to any number of jobs.

It is worth pointing out that shortest job first is optimal only when all the jobs are available simultaneously. As a counterexample, consider five jobs, *A* through *E*, with run times of 2, 4, 1, 1, and 1, respectively. Their arrival times are 0, 0, 3, 3, and 3. Initially, only *A* or *B* can be chosen, since the other three jobs have not arrived yet. Using shortest job first, we will run the jobs in the order *A*, *B*, *C*, *D*, *E*, for an average wait of 4.6. However, running them in the order *B*, *C*, *D*, *E*, *A* has an average wait of 4.4.

Shortest Remaining Time Next

A preemptive version of shortest job first is **shortest remaining time next**. With this algorithm, the scheduler always chooses the process whose remaining run time is the shortest. Again here, the run time has to be known in advance. When a new job arrives, its total time is compared to the current process' remaining time. If the new job needs less time to finish than the current process, the current process is suspended and the new job started. This scheme allows new short jobs to get good service.

2.5.3 Scheduling in Interactive Systems

We will now look at some algorithms that can be used in interactive systems. These are common on personal computers, servers, and other kinds of systems as well.

Round-Robin Scheduling

One of the oldest, simplest, fairest, and most widely used algorithms is **round robin**. Each process is assigned a time interval, called its **quantum**, during 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. 2-43(a). When the process uses up its quantum, it is put on the end of the list, as shown in Fig. 2-43(b).

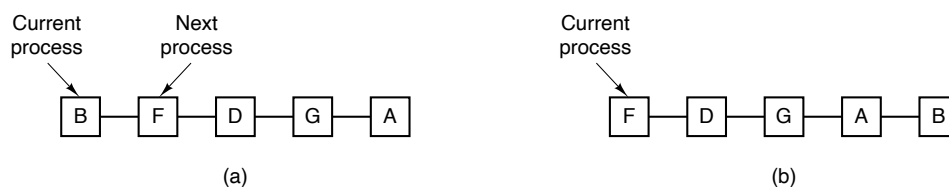


Figure 2-43. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

The only really 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, flushing and reloading the memory cache, and so on. Suppose that this context switch takes 1 msec, including switching memory maps, flushing and reloading the cache, etc. Also suppose that the quantum is set at 4 msec. With these parameters, after doing 4 msec of useful work, the CPU will have to spend (i.e., waste) 1 msec on process switching. Thus, 20% of the CPU time will be thrown away on administrative overhead. Clearly, this is too much.

To improve the CPU efficiency, we could set the quantum to, say, 100 msec. Now the wasted time is only 1%. But consider what happens on a server system if 50 requests come in within a very short time interval and with widely varying CPU requirements. Fifty 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 100 msec 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 sluggish. This situation is especially

bad if some of the requests near the end of the queue required only a few milliseconds of CPU time. With a short quantum, they would have gotten better service.

Another factor is that if the quantum is set longer than the mean CPU burst, preemption will not happen very often. Instead, most processes will perform a blocking operation before the quantum runs out, causing a process switch. Eliminating preemption improves performance because process switches then happen only when they are logically necessary, that is, when a process blocks and cannot continue.

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 20–50 msec is often a reasonable compromise.

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 quite different ideas on that subject. At a university, for example, the pecking order may be the president first, the faculty deans next, 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 of them 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 time quantum that it is allowed to run. When this quantum is used up, the next-highest-priority process is given a chance to run. After a process has been punished long enough, its priority needs to be raised by some algorithm, to let it run again. Otherwise all processes will eventually end up at 0.

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 down the totem pole. Alternatively, at a commercial data center, high-priority jobs might cost \$100 an hour, medium priority \$75 an hour, and low priority \$50 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. Not surprisingly, 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 1 msec of its 50-msec quantum would get priority 50, while a process that ran 25 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 2-44 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.

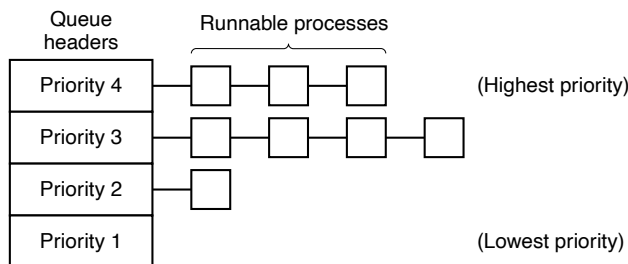


Figure 2-44. A scheduling algorithm with four priority classes.

Multiple Queues

One of the earliest priority schedulers was in CTSS, the M.I.T. Compatible TimeSharing System that ran on the IBM 7094 (Corbató et al., 1962). CTSS had the problem that process switching was 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, rather than giving them small quanta frequently (to reduce swapping). On the other hand, giving all processes a large quantum would mean poor response time, as we

have already seen. Their solution was to set up priority classes. Processes in the highest class were run for one quantum. Processes in the next-highest class were run for two quanta. Processes in the next one were run for four quanta, etc. Whenever a process used up all the quanta allocated to it, it was moved down one class. Thus, the processes in the highest class would run more frequently and with high priority, but for a shorter time—ideal for interactive processes.

As an example, consider a process that needed to compute continuously for 100 quanta. It would initially be given one quantum, then swapped out. Next time it would get two quanta before being swapped out. On succeeding runs it would get 4, 8, 16, 32, and 64 quanta, although it would have used only 37 of the final 64 quanta to complete its work. Only 7 swaps would be needed (including the initial load) instead of 100 with a pure round-robin algorithm. Furthermore, as the process sank deeper and deeper into the priority queues, it would be run less and less frequently, saving the CPU for short, interactive processes.

The following policy was adopted to avoid punishing forever a process that needed to run for a long time when it first started but became interactive later. Whenever a carriage return (*Enter* key) was typed at a terminal, the process belonging to that terminal was moved to the highest-priority class, on the assumption that it was about to become interactive. One fine day, some user with a heavily CPU-bound process discovered that just sitting at the terminal and typing carriage returns at random every few seconds did wonders for his response time. He told all his friends. They told all their friends. Moral of the story: getting it right in practice is much harder than getting it right in principle.

Shortest Process Next

Because shortest job first always produces the minimum average response time for batch systems, it would be nice if it could be used for interactive processes as well. To a certain extent, it can be. Interactive processes generally follow the pattern of wait for command, execute command, wait for command, execute command, etc. If we regard the execution of each command as a separate “job,” then we can minimize overall response time by running the shortest one first. The problem is figuring out which of the currently runnable processes is the shortest one.

One approach is to make estimates based on past behavior and run the process with the shortest estimated running time. Suppose that the estimated time per command for some process is T_0 . Now suppose its next run is measured to be T_1 . We could update our estimate by taking a weighted sum of these two numbers, that is, $aT_0 + (1 - a)T_1$. Through the choice of a we can decide to have the estimation process forget old runs quickly, or remember them for a long time. With $a = 1/2$, we get successive estimates of

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

After three new runs, the weight of T_0 in the new estimate has dropped to $1/8$.

The technique of estimating the next value in a series by taking the weighted average of the current measured value and the previous estimate is sometimes called **aging**. It is applicable to many situations where a prediction must be made based on previous values. Aging is especially easy to implement when $a = 1/2$. All that is needed is to add the new value to the current estimate and divide the sum by 2 (by shifting it right 1 bit).

Guaranteed Scheduling

A completely different approach to scheduling is to make real promises to the users about performance and then live up to those promises. One promise that is realistic to make and easy to live up to is this: If n users are logged in while you are working, you will receive about $1/n$ of the CPU power. Similarly, on a single-user system with n processes running, all things being equal, each one should get $1/n$ of the CPU cycles. That seems fair enough.

To make good on this promise, the system must keep track of how much CPU each process has had since its creation. It then computes the amount of CPU each one is entitled to, namely the time since creation divided by n . Since the amount of CPU time each process has actually had is also known, it is fairly straightforward to compute the ratio of actual CPU time consumed to CPU time entitled. A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to. The algorithm is then to run the process with the lowest ratio until its ratio has moved above that of its closest competitor. Then that one is chosen to run next.

A variant of such a scheduling regime is used in Linux' **CFS (Completely Fair Scheduling)** algorithm which keeps track of the "spent execution time" for processes in an efficient red-black tree. The left-most node in the tree corresponds to the process with the least spent execution time. The scheduler indexes the tree by execution time and selects the left-most node to run. When the process stops running (either because it used up its time slot, or it was blocked or interrupted), the scheduler reinserts it in the tree based on its new spent execution time.

Lottery Scheduling

While making promises to the users and then living up to them is a fine idea, it is difficult to implement. However, another algorithm can be used to give similarly predictable results with a much simpler implementation. It is called **lottery scheduling** (Waldspurger and Weihl, 1994).

The basic idea is to give processes lottery tickets for various system resources, such as CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource. When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

To paraphrase George Orwell: “All processes are equal, but some processes are more equal.” More important processes can be given extra tickets, to increase their odds of winning. If there are 100 tickets outstanding, and one process holds 20 of them, it will have a 20% chance of winning each lottery. In the long run, it will get about 20% of the CPU. In contrast to a priority scheduler, where it is very hard to state what having a priority of 40 actually means, here the rule is clear: a process holding a fraction f of the tickets will get about a fraction f of the resource in question.

Lottery scheduling has several interesting properties. For example, if a new process shows up and is granted some tickets, at the very next lottery it will have a chance of winning in proportion to the number of tickets it holds. In other words, lottery scheduling is highly responsive.

Cooperating processes may exchange tickets if they wish. For example, when a client process sends a message to a server process and then blocks, it may give all of its tickets to the server, to increase the chance of the server running next. When the server is finished, it returns the tickets so that the client can run again. In fact, in the absence of clients, servers need no tickets at all.

Lottery scheduling can be used to solve problems that are difficult to handle with other methods. One example is a video server in which several processes are feeding video streams to their clients, but at different frame rates. Suppose that the processes need frames at 10, 20, and 25 frames/sec. By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in approximately the correct proportion, that is, 10:20:25.

Fair-Share Scheduling

So far we have assumed that each process is scheduled on its own, without regard to who its owner is. As a result, if user 1 starts up nine processes and user 2 starts up one process, with round robin or equal priorities, user 1 will get 90% of the CPU and user 2 only 10% of it.

To prevent this situation, some systems take into account which user owns a process before scheduling it. In this model, each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it. Thus, if two users have each been promised 50% of the CPU, they will each get that, no matter how many processes they have in existence.

As an example, consider a system with two users, each of which has been promised 50% of the CPU. User 1 has four processes, *A*, *B*, *C*, and *D*, and user 2 has only one process, *E*. If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:

A E B E C E D E A E B E C E D E ...

On the other hand, if user 1 is entitled to twice as much CPU time as user 2, we might get:

A B E C D E A B E C D E ...

Numerous other possibilities exist, of course, and can be exploited, depending on what the notion of fairness is.

2.5.4 Scheduling in Real-Time Systems

A **real-time** system is one in which time plays an essential role. Typically, one or more physical devices external to the computer generate stimuli, and the computer must react appropriately to them within a fixed amount of time. For example, the computer in a compact disc player gets the bits as they come off the drive and must convert them into music within a very tight time interval. If the calculation takes too long, the music will sound peculiar. Other real-time systems are patient monitoring in a hospital intensive-care unit, the autopilot in an aircraft, and robot control in an automated factory. In all these cases, having the right answer but having it too late is often just as bad as not having it at all.

Real-time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met—or else!—and **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable. In both cases, real-time behavior is achieved by dividing the program into a number of processes, each of whose behavior is predictable and known in advance. These processes are generally short lived and can run to completion in well under a second. When an external event is detected, it is the job of the scheduler to schedule the processes in such a way that all deadlines are met.

The events that a real-time system may have to respond to can be further categorized as **periodic** (meaning they occur at regular intervals) or **aperiodic** (meaning they occur unpredictably). A system may have to respond to multiple periodic-event streams. Depending on how much time each event requires for processing, handling all of them may not even be possible. For example, if there are m periodic events and event i occurs with period P_i and requires C_i sec of CPU time to handle each event, then the load can be handled only if

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

A real-time system that meets this criterion is said to be **schedulable**. This means it can actually be implemented. A process that fails to meet this test cannot be scheduled because the total amount of CPU time the processes want collectively is more than the CPU can deliver.

As an example, consider a soft real-time system with three periodic events, with periods of 100, 200, and 500 msec, respectively. If these events require 50, 30, and 100 msec of CPU time per event, respectively, the system is schedulable because $0.5 + 0.15 + 0.2 < 1$. If a fourth event with a period of 1 sec is added, the system will remain schedulable as long as this event does not need more than 150 msec of CPU time per event. Implicit in this calculation is the assumption that the context-switching overhead is so small that it can be ignored.

Real-time scheduling algorithms can be static or dynamic. The former make their scheduling decisions before the system starts running. The latter make their scheduling decisions at run time, after execution has started. Static scheduling works only when there is perfect information available in advance about the work to be done and the deadlines that have to be met. Dynamic scheduling algorithms do not have these restrictions.

2.5.5 Policy Versus Mechanism

Up until now, we have tacitly assumed that all the processes in the system belong to different users and are thus competing for the CPU. While this is often true, sometimes it happens that one process has many children running under its control. For example, a database-management-system process may have many children. Each child might be working on a different request, or each might have some specific function to perform (query parsing, disk access, etc.). It is entirely possible that the main process has an excellent idea of which of its children are the most important (or time critical) and which the least. Unfortunately, none of the schedulers discussed earlier accept any input from user processes about scheduling decisions. As a result, the scheduler rarely makes the best choice.

The solution to this problem is to separate the **scheduling mechanism** from the **scheduling policy**, a long-established principle (Levin et al., 1975). What this means is that the scheduling algorithm is parameterized in some way, but the parameters can be filled in by user processes. Let us consider the database example once again. Suppose that the kernel uses a priority-scheduling algorithm but provides a system call by which a process can set (and change) the priorities of its children. In this way, the parent can control how its children are scheduled, even though it itself does not do the scheduling. Here the mechanism is in the kernel but policy is set by a user process. Policy-mechanism separation is a key idea.

2.5.6 Thread Scheduling

When several processes each have multiple threads, we have two levels of parallelism present: processes and threads. Scheduling in such systems differs substantially depending on whether user-level threads or kernel-level threads (or both) are supported.

Let us consider user-level threads first. Since the kernel is not aware of the existence of threads, it operates as it always does, picking a process, say, *A*, and giving *A* control for its quantum. The thread scheduler inside *A* decides which thread to run, say *A1*. Since there are no clock interrupts to multiprogram threads, this thread may continue running as long as it wants to. If it uses up the process' entire quantum, the kernel will select another process to run.

When the process *A* finally runs again, thread *A1* will resume running. It will continue to consume all of *A*'s time until it is finished. However, its antisocial behavior will not affect other processes. They will get whatever the scheduler considers their appropriate share, no matter what is going on inside process *A*.

Now consider the case that *A*'s threads have relatively little work to do per CPU burst, for example, 5 msec of work within a 50-msec quantum. Consequently, each one runs for a little while, then yields the CPU back to the thread scheduler. This might lead to the sequence *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, before the kernel switches to process *B*. This situation is illustrated in Fig. 2-45(a).

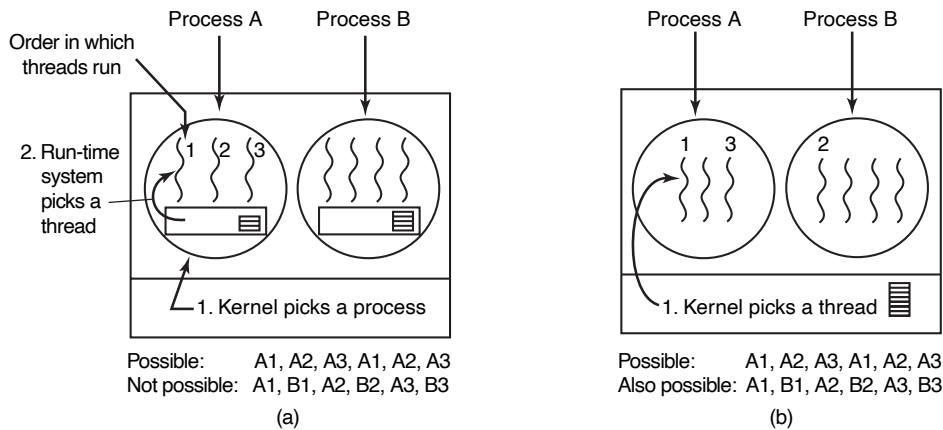


Figure 2-45. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

The scheduling algorithm used by the run-time system can be any of the ones described above. In practice, round-robin scheduling and priority scheduling are most common. The only constraint is the absence of a clock to interrupt a thread that has run too long. Since threads cooperate, this is usually not an issue.

Now consider the situation with kernel-level threads. Here the kernel picks a particular thread to run. It does not have to take into account which process the thread belongs to, but it can if it wants to. The thread is given a quantum and is forcibly suspended if it exceeds the quantum. With a 50-msec quantum and threads that block after 5 msec, the thread order for some period of 30 msec might be *A1*, *B1*, *A2*, *B2*, *A3*, *B3*, something not possible with these parameters and user-level threads. This situation is partially depicted in Fig. 2-45(b).

A major difference between user-level threads and kernel-level threads is the performance. Doing a thread switch with user-level threads takes a handful of machine instructions. With kernel-level threads it requires a full context switch changing the memory map and invalidating the cache, which is several orders of

magnitude slower. On the other hand, with kernel-level threads, having a thread block on I/O does not suspend the entire process as it does with user-level threads.

Since the kernel knows that switching from a thread in process *A* to a thread in process *B* is more expensive than running a second thread in process *A* (due to having to change the memory map and having the memory cache spoiled), it can take this information into account when making a decision. For example, given two threads that are otherwise equally important, with one of them belonging to the same process as a thread that just blocked and one belonging to a different process, preference could be given to the former.

Another important factor is that user-level threads can employ an application-specific thread scheduler. Consider, for example, the Web server of Fig. 2-8. Suppose that a worker thread has just blocked and the dispatcher thread and two worker threads are ready. Who should run next? The run-time system, knowing what all the threads do, can easily pick the dispatcher to run next, so that it can start another worker running. This strategy maximizes the amount of parallelism in an environment where workers frequently block on disk I/O. With kernel-level threads, the kernel would never know what each thread did (although they could be assigned different priorities). In general, however, application-specific thread schedulers can tune an application better than the kernel can.

2.6 RESEARCH ON PROCESSES AND THREADS

In Chap. 1, we looked at some of the current research in operating system structure. In this and subsequent chapters, we will look at more narrowly focused research, starting with processes. As will become clear in time, some subjects are much more settled than others. Most of the research tends to be on the new topics, rather than ones that have been around for decades.

The concept of a process is an example of something that is fairly well settled. Almost every system has some notion of a process as a container for grouping together related resources such as an address space, threads, open files, protection permissions, and so on. Different systems do the grouping slightly differently, but these are just engineering differences. The basic idea is not very controversial any more, and there is little new research on the subject of processes.

Threads are a newer idea than processes, but they, too, have been chewed over quite a bit. Still, the occasional paper about threads appears from time to time, for example, about core-aware thread management (Qin et al., 2019), or on how well modern operating systems like Linux scale with many threads and many cores (Boyd-Wickizer, 2010).

Additionally, there is a lot of work trying to prove things do not break in the presence of concurrency, for instance in file systems (Chajed et al., 2019; and Zou et al., 2019) and other services (Setty et al., 2018; and Li et al., 2019). This is important work as researchers have shown that concurrency bugs are unfortunately

extremely common (Li et al., 2019). As we have seen, locking is not just hard, but also expensive and operating systems have adopted RCU to avoid locking altogether (McKenney et al., 2013).

An active research area deals with recording and replaying a process' execution (Viennot et al., 2013). Replaying helps developers track down hard-to-find bugs and security experts to investigate incidents.

Speaking of security, a major event in 2018 was the disclosure of a series of very serious security vulnerabilities in modern CPUs. They required changes everywhere: hardware, firmware, operating system, and even applications. For this chapter, the scheduling implications are especially relevant. For instance, Windows adopted a scheduling algorithm to prevent code in different security domains from sharing the same processor core (Microsoft, 2018).

Scheduling (both uniprocessor and multiprocessor) in general is still a topic near and dear to the heart of some researchers. Some topics being researched include scheduling in clusters for deep learning (Xiao et al., 2018), scheduling for microservices (Sriraman, 2018), and schedulability (Yang et al., 2018). All in all, processes, threads, and scheduling are not hot topics for research as they once were. The research has moved on to topics like power management, virtualization, clouds, and security.

2.7 SUMMARY

To hide the effects of interrupts, operating systems provide a conceptual model consisting of sequential processes running in parallel. Processes can be created and terminated dynamically. Each process has its own address space.

For some applications, it is useful to have multiple threads of control within a single process. These threads are scheduled independently and each one has its own stack, but all the threads in a process share a common address space. Threads can be implemented in user space or in the kernel.

Alternatively, high-throughput servers may opt for an event-driven model instead. Here, the server operates as a finite state machine that responds to events and interacts with the operating system using non-blocking system calls.

Processes can synchronize with one another using synchronization and interprocess communication primitives, for example, semaphores, monitors, or messages. These primitives are used to ensure that no two processes are ever in their critical regions at the same time, a situation that leads to chaos. A process can be running, runnable, or blocked and can change state when it or another process executes one of the interprocess communication primitives. Interthread communication is similar.

A great many scheduling algorithms have been studied. Some of these are primarily used for batch systems, such as shortest-job-first scheduling. Others are common in both batch systems and interactive systems. These algorithms include

round robin, priority scheduling, multilevel queues, guaranteed scheduling, lottery scheduling, and fair-share scheduling. Some systems make a clean separation between the scheduling mechanism and the scheduling policy, which allows users to have control of the scheduling algorithm.

PROBLEMS

1. In Fig. 2-2, three process states are shown. In theory, with three states, there could be six transitions, two out of each state. However, only four transitions are shown. Are there any circumstances in which either or both of the missing transitions might occur?
2. Suppose that you were to design an advanced computer architecture that did process switching in hardware, instead of having interrupts. What information would the CPU need? Describe how the hardware process switching might work.
3. On all current computers, at least part of the interrupt handlers are written in assembly language. Why?
4. When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?
5. A computer system has enough room to hold four programs in its main memory. These programs are idle waiting for I/O half the time. What fraction of the CPU time is wasted?
6. A computer has 2 GB of RAM of which the operating system occupies 256 MB. The processes are all 128 MB (for simplicity) and have the same characteristics. If the goal is 99% CPU utilization, what is the maximum I/O wait that can be tolerated?
7. Multiple jobs can run in parallel and finish faster than if they had run sequentially. Suppose that two jobs, each needing 10 minutes of CPU time, start simultaneously. How long will the last one take to complete if they run sequentially? How long if they run in parallel? Assume 50% I/O wait.
8. Consider a multiprogrammed system with degree of 5 (i.e., five programs in memory at the same time). Assume that each process spends 40% of its time waiting for I/O. What will be the CPU utilization?
9. Explain how a Web browser can utilize the concept of threads to improve performance.
10. Assume that you are trying to download a large 2-GB file from the Internet. The file is available from a set of mirror servers, each of which can deliver a subset of the file's bytes; assume that a given request specifies the starting and ending bytes of the file. Explain how you might use threads to improve the download time.
11. In the text it was stated that the model of Fig. 2-10(a) was not suited to a file server using a cache in memory. Why not? Could each process have its own cache?

12. In Fig. 2-8, a multithreaded Web server is shown. If the only way to read from a file is the normal blocking read system call, do you think user-level threads or kernel-level threads are being used for the Web server? Why?
13. In the text, we described a multithreaded Web server, showing why it is better than a single-threaded server and a finite-state machine server. Are there any circumstances in which a single-threaded server might be better? Give an example.
14. In Fig. 2-11 the register set is listed as a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.
15. Why would a thread ever voluntarily give up the CPU by calling *thread_yield*? After all, since there is no periodic clock interrupt, it may never get the CPU back.
16. In this problem, you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded?
17. What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?
18. In Fig. 2-14 the thread creations and messages printed by the threads are interleaved at random. Is there a way to force the order to be strictly thread 1 created, thread 1 prints message, thread 1 exits, thread 2 created, thread 2 prints message, thread 2 exits, and so on? If so, how? If not, why not?
19. Suppose that a program has two threads, each executing the *get_account* function, shown below. Identify a race condition in this code.

```
int accounts[LIMIT]; int account_count = 0;

void *get_account(void *tid) {
    char *lineptr = NULL;
    size_t len = 0;

    while (account_count < LIMIT)
    {
        // Read user input from terminal and store it in lineptr
        getline(&lineptr, &len, stdin);

        // Convert user input to integer
        // Assume user entered valid integer value
        int entered_account = atoi(lineptr);

        accounts[account_count] = entered_account;
        account_count++;
    }
}
```

```
// Deallocate memory that was allocated by getline call
free(lineptr);
return NULL; }
```

20. In the discussion on global variables in threads, we used a procedure *create_global* to allocate storage for a pointer to the variable, rather than the variable itself. Is this essential, or could the procedures work with the values themselves just as well?
21. Consider a system in which threads are implemented entirely in user space, with the run-time system getting a clock interrupt once a second. Suppose that a clock interrupt occurs exactly while some thread executing in the run-time system is at the point of blocking or unblocking a thread. What problem might occur? Can you solve it?
22. Suppose that an operating system does not have anything like the *select* system call to see in advance if it is safe to read from a file, pipe, or device, but it does allow alarm clocks (timers) to be set that interrupt blocked system calls. Is it possible to implement in user space a threads package that will not block all threads when one thread performs a system call that may block? Explain your answer.
23. Does Peterson's solution to the mutual-exclusion problem shown in Fig. 2-24 work when process scheduling is preemptive? How about when it is nonpreemptive?
24. Can the priority inversion problem discussed in Sec. 2.3.4 happen with user-level threads? Why or why not?
25. In Sec. 2.3.4, a situation with a high-priority process, *H*, and a low-priority process, *L*, was described, which led to *H* looping forever. Does the same problem occur if round-robin scheduling is used instead of priority scheduling? Discuss.
26. In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used? Explain.
27. What is a race condition?
28. When a computer is being developed, it is usually first simulated by a program that runs one instruction at a time. Even multiprocessors are simulated strictly sequentially like this. Is it possible for a race condition to occur when there are no simultaneous events? Explain.
29. The producer-consumer problem can be extended to a system with multiple producers and consumers that write (or read) to (from) one shared buffer. Assume that each producer and consumer runs in its own thread. Will the solution presented in Fig. 2-28, using semaphores, work for this system?
30. Consider the following solution to the mutual-exclusion problem involving two processes *P0* and *P1*. Assume that the variable *turn* is initialized to 0. Process *P0*'s code is presented below.

```
/* Other code */

while (turn != 0) { /* Do nothing and wait. */
    Critical Section /* . . . */
    turn = 0;

/* Other code */
```

For process *PI*, replace 0 by 1 in above code. Determine if the solution meets *all* the required conditions for a correct mutual-exclusion solution.

31. Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.
32. If a system has only two processes, does it make sense to use a barrier to synchronize them? Why or why not?
33. Can two threads in the same process synchronize using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume that no threads in any other processes have access to the semaphore. Discuss your answers.
34. Suppose that we have a message-passing system using mailboxes. When sending to a full mailbox or trying to receive from an empty one, a process does not block. Instead, it gets an error code back. The process responds to the error code by just trying again, over and over, until it succeeds. Does this scheme lead to race conditions?
35. The CDC 6600 computers could handle up to 10 I/O processes simultaneously using an interesting form of round-robin scheduling called processor sharing. A process switch occurred after each instruction, so instruction 1 came from process 1, instruction 2 came from process 2, etc. The process switching was done by special hardware, and the overhead was zero. If a process needed T sec to complete in the absence of competition, how much time would it need if processor sharing was used with n processes?
36. Consider the following piece of C code:

```
void main() {  
    fork( );  
    fork( );  
    exit( );  
}
```

How many child processes are created upon execution of this program?

37. Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen (scheduling-wise) if a process occurred twice in the list? Can you think of any reason for allowing this?
38. Can a measure of whether a process is likely to be CPU bound or I/O bound be determined by analyzing source code? How can this be determined at run time?
39. In the section “When to Schedule,” it was mentioned that sometimes scheduling could be improved if an important process could play a role in selecting the next process to run when it blocks. Give a situation where this could be used and explain how.
40. Explain how time quantum value and context switching time affect each other, in a round-robin scheduling algorithm.
41. Measurements of a certain system have shown that the average process runs for a time T before blocking on I/O. A process switch requires a time S , which is effectively

wasted (overhead). For round-robin scheduling with quantum Q , give a formula for the CPU efficiency for each of the following:

- (a) $Q = \infty$
- (b) $Q > T$
- (c) $S < Q < T$
- (d) $Q = S$
- (e) Q nearly 0

42. Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and X . In what order should they be run to minimize average response time? (Your answer will depend on X .)
43. Five batch jobs, A through E , arrive at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.
- (a) Round robin.
 - (b) Priority scheduling.
 - (c) First-come, first-served (run in order 10, 6, 2, 4, 8).
 - (d) Shortest job first.

For (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b) through (d), assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

44. A process running on CTSS needs 30 quanta to complete. How many times must it be swapped in, including the very first time (before it has run at all)?
45. Can you think of a way to save the CTSS priority system from being fooled by random carriage returns?
46. Consider a real-time system with two voice calls of periodicity 5 msec each with CPU time per call of 1 msec, and one video stream of periodicity 33 msec with CPU time per call of 11 msec. Is this system schedulable? Show how you derived your answer.
47. For the above problem, can another video stream be added and have the system still be schedulable?
48. The aging algorithm with $a = 1/2$ is being used to predict run times. The previous four runs, from oldest to most recent, are 40, 20, 40, and 15 msec. What is the prediction of the next time?
49. A soft real-time system has four periodic events with periods of 50, 100, 200, and 250 msec each. Suppose that the four events require 35, 20, 10, and x msec of CPU time, respectively. What is the largest value of x for which the system is schedulable?
50. Explain why two-level scheduling is commonly used. What advantages does it have over single-level scheduling?
51. A real-time system needs to handle two voice calls that each run every 5 msec and consume 1 msec of CPU time per burst, plus one video at 25 frames/sec, with each frame

requiring 20 msec of CPU time. Is this system schedulable? Please explain why or why not it is schedulable and how you came to that conclusion.

52. Consider a system in which it is desired to separate policy and mechanism for the scheduling of kernel threads. Propose a means of achieving this goal.
53. The readers and writers problem can be formulated in several ways with regard to which category of processes can be started when. Carefully describe three different variations of the problem, each one favoring (or not favoring) some category of processes (e.g., readers or writers). For each variation, specify what happens when a reader or a writer becomes ready to access the database, and what happens when a process is finished.
54. Write a shell script that produces a file of sequential numbers by reading the last number in the file, adding 1 to it, and then appending it to the file. Run one instance of the script in the background and one in the foreground, each accessing the same file. How long does it take before a race condition manifests itself? What is the critical region? Modify the script to prevent the race. (*Hint*: use
 In file file.lock
to lock the data file.)
55. Assume that you have an operating system that provides semaphores. Implement a message system. Write the procedures for sending and receiving messages.
56. Rewrite the program of Fig. 2-23 to handle more than two processes.
57. Write a producer-consumer problem that uses threads and shares a common buffer. However, do not use semaphores or any other synchronization primitives to guard the shared data structures. Just let each thread access them when it wants to. Use `sleep` and `wakeup` to handle the full and empty conditions. See how long it takes for a fatal race condition to occur. For example, you might have the producer print a number once in a while. Do not print more than one number every minute because the I/O could affect the race conditions.
58. A process can be put into a round-robin queue more than once to give it a higher priority. Running multiple instances of a program each working on a different part of a data pool can have the same effect. First write a program that tests a list of numbers for primality. Then devise a method to allow multiple instances of the program to run at once in such a way that no two instances of the program will work on the same number. Can you in fact get through the list faster by running multiple copies of the program? Note that your results will depend upon what else your computer is doing; on a personal computer running only instances of this program you would not expect an improvement, but on a system with other processes, you should be able to grab a bigger share of the CPU this way.
59. Implement a program to count the frequency of words in a text file. The text file is partitioned into N segments. Each segment is processed by a separate thread that outputs the intermediate frequency count for its segment. The main process waits until all the threads complete; then it computes the consolidated word-frequency data based on the individual threads' output.