The first version will have the child process write the starting time to a region of shared memory before it calls `exec()`. After the child process terminates, the parent will read the starting time from shared memory. Refer to Section 3.7.1 for details using POSIX shared memory. In that section, there are separate programs for the producer and consumer. As the solution to this problem requires only a single program, the region of shared memory can be established before the child process is forked, allowing both the parent and child processes access to the region of shared memory.

The second version will use a pipe. The child will write the starting time to the pipe, and the parent will read from it following the termination of the child process.

You will use the `gettimeofday()` function to record the current timestamp. This function is passed a pointer to a `struct timeval` object, which contains two members: `tv_sec` and `t_usec`. These represent the number of elapsed seconds and microseconds since January 1, 1970 (known as the UNIX EPOCH). The following code sample illustrates how this function can be used:

```
struct timeval current;

gettimeofday(&current,NULL);

// current.tv_sec represents seconds
// current.tv_usec represents microseconds
```

For IPC between the child and parent processes, the contents of the shared memory pointer can be assigned the `struct timeval` representing the starting time. When pipes are used, a pointer to a `struct timeval` can be written to—and read from—the pipe.

3.20 An operating system's **pid manager** is responsible for managing process identifiers. When a process is first created, it is assigned a unique pid by the pid manager. The pid is returned to the pid manager when the process completes execution, and the manager may later reassign this pid. Process identifiers are discussed more fully in Section 3.3.1. What is most important here is to recognize that process identifiers must be unique; no two active processes can have the same pid.

Use the following constants to identify the range of possible pid values:

```
#define MIN_PID 300
#define MAX_PID 5000
```

You may use any data structure of your choice to represent the availability of process identifiers. One strategy is to adopt what Linux has done and use a bitmap in which a value of 0 at position *i* indicates that

a process id of value i is available and a value of 1 indicates that the process id is currently in use.

Implement the following API for obtaining and releasing a pid:

- `int allocate_map(void)`—Creates and initializes a data structure for representing pids; returns −1 if unsuccessful, 1 if successful

- `int allocate_pid(void)`—Allocates and returns a pid; returns −1 if unable to allocate a pid (all pids are in use)

- `void release_pid(int pid)`—Releases a pid

This programming problem will be modified later on in Chapter 4 and in Chapter 6.

3.21 The Collatz conjecture concerns what happens when we take any positive integer n and apply the following algorithm:

$$n = \begin{cases} n/2, & \text{if n is even} \\ 3 \times n + 1, & \text{if n is odd} \end{cases}$$

The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1. For example, if $n = 35$, the sequence is

$$35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

Write a C program using the `fork()` system call that generates this sequence in the child process. The starting number will be provided from the command line. For example, if 8 is passed as a parameter on the command line, the child process will output $8, 4, 2, 1$. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the `wait()` call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a positive integer is passed on the command line.

3.22 In Exercise 3.21, the child process must output the sequence of numbers generated from the algorithm specified by the Collatz conjecture because the parent and child have their own copies of the data. Another approach to designing this program is to establish a shared-memory object between the parent and child processes. This technique allows the child to write the contents of the sequence to the shared-memory object. The parent can then output the sequence when the child completes. Because the memory is shared, any changes the child makes will be reflected in the parent process as well.

This program will be structured using POSIX shared memory as described in Section 3.7.1. The parent process will progress through the following steps:

a. Establish the shared-memory object (`shm_open()`, `ftruncate()`, and `mmap()`).

  b. Create the child process and wait for it to terminate.

  c. Output the contents of shared memory.

  d. Remove the shared-memory object.

  One area of concern with cooperating processes involves synchronization issues. In this exercise, the parent and child processes must be coordinated so that the parent does not output the sequence until the child finishes execution. These two processes will be synchronized using the `wait()` system call: the parent process will invoke `wait()`, which will suspend it until the child process exits.

3.23 Section 3.8.1 describes certain port numbers as being well known—that is, they provide standard services. Port 17 is known as the *quote-of-the-day* service. When a client connects to port 17 on a server, the server responds with a quote for that day.

  Modify the date server shown in Figure 3.27 so that it delivers a quote of the day rather than the current date. The quotes should be printable ASCII characters and should contain fewer than 512 characters, although multiple lines are allowed. Since these well-known ports are reserved and therefore unavailable, have your server listen to port 6017. The date client shown in Figure 3.28 can be used to read the quotes returned by your server.

3.24 A **haiku** is a three-line poem in which the first line contains five syllables, the second line contains seven syllables, and the third line contains five syllables. Write a haiku server that listens to port 5575. When a client connects to this port, the server responds with a haiku. The date client shown in Figure 3.28 can be used to read the quotes returned by your haiku server.

3.25 An echo server echoes back whatever it receives from a client. For example, if a client sends the server the string `Hello there!`, the server will respond with `Hello there!`

  Write an echo server using the Java networking API described in Section 3.8.1. This server will wait for a client connection using the `accept()` method. When a client connection is received, the server will loop, performing the following steps:

- Read data from the socket into a buffer.

- Write the contents of the buffer back to the client.

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

  The date server of Figure 3.27 uses the `java.io.BufferedReader` class. `BufferedReader` extends the `java.io.Reader` class, which is used for reading character streams. However, the echo server cannot guarantee that it will read characters from clients; it may receive binary data as well. The class `java.io.InputStream` deals with data at the byte level rather than the character level. Thus, your echo server must use an object that extends `java.io.InputStream`. The `read()` method in the

java.io.InputStream class returns −1 when the client has closed its end of the socket connection.

3.26 Design a program using ordinary pipes in which one process sends a string message to a second process, and the second process reverses the case of each character in the message and sends it back to the first process. For example, if the first process sends the message Hi There, the second process will return hI tHERE. This will require using two pipes, one for sending the original message from the first to the second process and the other for sending the modified message from the second to the first process. You can write this program using either UNIX or Windows pipes.

3.27 Design a file-copying program named filecopy.c using ordinary pipes. This program will be passed two parameters: the name of the file to be copied and the name of the destination file. The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe. The child process will read this file from the pipe and write it to the destination file. For example, if we invoke the program as follows:

```
./filecopy input.txt copy.txt
```

the file input.txt will be written to the pipe. The child process will read the contents of this file and write it to the destination file copy.txt. You may write this program using either UNIX or Windows pipes.

Programming Projects

Project 1—UNIX Shell

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. Completing this project will involve using the UNIX fork(), exec(), wait(), dup2(), and pipe() system calls and can be completed on any Linux, UNIX, or macOS system.

I. Overview

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt osh> and the user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the UNIX cat command.)

```
osh>cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c) and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 3.9. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

        osh>cat prog.c &

the parent and child processes will run concurrently.

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

A C program that provides the general operations of a command-line shell is supplied in Figure 3.36. The main() function presents the prompt osh-> and outlines the steps to be taken after input from the user has been read. The main() function continually loops as long as should_run equals 1; when the user enters exit at the prompt, your program will set should_run to 0 and terminate.

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

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
char *args[MAX_LINE/2 + 1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */

   while (should_run) {
      printf("osh>");
      fflush(stdout);

      /**
       * After reading user input, the steps are:
       * (1) fork a child process using fork()
       * (2) the child process will invoke execvp()
       * (3) parent will invoke wait() unless command included &
       */
   }

   return 0;
}
```

Figure 3.36 Outline of simple shell.

This project is organized into several parts:

1. Creating the child process and executing the command in the child
2. Providing a history feature
3. Adding support of input and output redirection
4. Allowing the parent and child processes to communicate via a pipe

II. Executing Command in a Child Process

The first task is to modify the `main()` function in Figure 3.36 so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (`args` in Figure 3.36). For example, if the user enters the command `ps  -ael` at the `osh>` prompt, the values stored in the `args` array are:

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

This `args` array will be passed to the `execvp()` function, which has the following prototype:

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

Here, `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as `execvp(args[0], args)`. Be sure to check whether the user included & to determine whether or not the parent process is to wait for the child to exit.

III. Creating a History Feature

The next task is to modify the shell interface program so that it provides a *history* feature to allow a user to execute the most recent command by entering `!!`. For example, if a user enters the command `ls  -l`, she can then execute that command again by entering `!!` at the prompt. Any command executed in this fashion should be echoed on the user's screen, and the command should also be placed in the history buffer as the next command.

Your program should also manage basic error handling. If there is no recent command in the history, entering `!!` should result in a message "No commands in history."

IV. Redirecting Input and Output

Your shell should then be modified to support the '>' and '<' redirection

operators, where '>' redirects the output of a command to a file and '<' redirects the input to a command from a file. For example, if a user enters

```
osh>ls > out.txt
```

the output from the ls command will be redirected to the file out.txt. Similarly, input can be redirected as well. For example, if the user enters

```
osh>sort < in.txt
```

the file in.txt will serve as input to the sort command.

Managing the redirection of both input and output will involve using the dup2() function, which duplicates an existing file descriptor to another file descriptor. For example, if fd is a file descriptor to the file out.txt, the call

```
dup2(fd, STDOUT_FILENO);
```

duplicates fd to standard output (the terminal). This means that any writes to standard output will in fact be sent to the out.txt file.

You can assume that commands will contain either one input or one output redirection and will not contain both. In other words, you do not have to be concerned with command sequences such as sort < in.txt > out.txt.

V. Communication via a Pipe

The final modification to your shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

```
osh>ls -l | less
```

has the output of the command ls -l serve as the input to the less command. Both the ls and less commands will run as separate processes and will communicate using the UNIX pipe() function described in Section 3.7.4. Perhaps the easiest way to create these separate processes is to have the parent process create the child process (which will execute ls -l). This child will also create another child process (which will execute less) and will establish a pipe between itself and the child process it creates. Implementing pipe functionality will also require using the dup2() function as described in the previous section. Finally, although several commands can be chained together using multiple pipes, you can assume that commands will contain only one pipe character and will not be combined with any redirection operators.

Project 2 —Linux Kernel Module for Task Information

In this project, you will write a Linux kernel module that uses the /proc file system for displaying a task's information based on its process identifier value pid. Before beginning this project, be sure you have completed the Linux kernel module programming project in Chapter 2, which involves creating an entry in the /proc file system. This project will involve writing a process identifier to

the file /proc/pid. Once a pid has been written to the /proc file, subsequent reads from /proc/pid will report (1) the command the task is running, (2) the value of the task's pid, and (3) the current state of the task. An example of how your kernel module will be accessed once loaded into the system is as follows:

```
echo "1395" > /proc/pid
cat /proc/pid
command = [bash] pid = [1395] state = [1]
```

The echo command writes the characters "1395" to the /proc/pid file. Your kernel module will read this value and store its integer equivalent as it represents a process identifier. The cat command reads from /proc/pid, where your kernel module will retrieve the three fields from the task_struct associated with the task whose pid value is 1395.

```
ssize_t proc_write(struct file *file, char __user *usr_buf,
   size_t count, loff_t *pos)
{
   int rv = 0;
   char *k_mem;

   /* allocate kernel memory */
   k_mem = kmalloc(count, GFP_KERNEL);

   /* copies user space usr_buf to kernel memory */
   copy_from_user(k_mem, usr_buf, count);

   printk(KERN_INFO "%s\n", k_mem);

   /* return kernel memory */
   kfree(k_mem);

   return count;
}
```

Figure 3.37   The proc_write() function.

I. Writing to the /proc File System

In the kernel module project in Chapter 2, you learned how to read from the /proc file system. We now cover how to write to /proc. Setting the field .write in struct file_operations to

```
.write = proc_write
```

causes the proc_write() function of Figure 3.37 to be called when a write operation is made to /proc/pid

The kmalloc() function is the kernel equivalent of the user-level mal-loc() function for allocating memory, except that kernel memory is being allocated. The GFP_KERNEL flag indicates routine kernel memory allocation. The copy_from_user() function copies the contents of usr_buf (which contains what has been written to /proc/pid) to the recently allocated kernel memory. Your kernel module will have to obtain the integer equivalent of this value using the kernel function kstrtol(), which has the signature

```
int kstrtol(const char *str, unsigned int base, long *res)
```

This stores the character equivalent of str, which is expressed as a base into res.

Finally, note that we return memory that was previously allocated with kmalloc() back to the kernel with the call to kfree(). Careful memory management—which includes releasing memory to prevent *memory leaks*—is crucial when developing kernel-level code.

II. Reading from the /proc File System

Once the process identifier has been stored, any reads from /proc/pid will return the name of the command, its process identifier, and its state. As illustrated in Section 3.1, the PCB in Linux is represented by the structure task_struct, which is found in the <linux/sched.h> include file. Given a process identifier, the function pid_task() returns the associated task_struct. The signature of this function appears as follows:

```
struct task_struct pid_task(struct pid *pid,
    enum pid_type type)
```

The kernel function find_vpid(int pid) can be used to obtain the struct pid, and PIDTYPE_PID can be used as the pid_type.

For a valid pid in the system, pid_task will return its task_struct. You can then display the values of the command, pid, and state. (You will probably have to read through the task_struct structure in <linux/sched.h> to obtain the names of these fields.)

If pid_task() is not passed a valid pid, it returns NULL. Be sure to perform appropriate error checking to check for this condition. If this situation occurs, the kernel module function associated with reading from /proc/pid should return 0.

In the source code download, we give the C program pid.c, which provides some of the basic building blocks for beginning this project.

Project 3—Linux Kernel Module for Listing Tasks

In this project, you will write a kernel module that lists all current tasks in a Linux system. You will iterate through the tasks both linearly and depth first.

Part I—Iterating over Tasks Linearly

In the Linux kernel, the for_each_process() macro easily allows iteration over all current tasks in the system:

```
#include <linux/sched.h>

struct task_struct *task;

for_each_process(task) {
    /* on each iteration task points to the next task */
}
```

The various fields in task_struct can then be displayed as the program loops through the for_each_process() macro.

Assignment

Design a kernel module that iterates through all tasks in the system using the for_each_process() macro. In particular, output the task command, state, and process id of each task. (You will probably have to read through the task_struct structure in <linux/sched.h> to obtain the names of these fields.) Write this code in the module entry point so that its contents will appear in the kernel log buffer, which can be viewed using the dmesg command. To verify that your code is working correctly, compare the contents of the kernel log buffer with the output of the following command, which lists all tasks in the system:

```
ps -el
```

The two values should be very similar. Because tasks are dynamic, however, it is possible that a few tasks may appear in one listing but not the other.

Part II—Iterating over Tasks with a Depth-First Search Tree

The second portion of this project involves iterating over all tasks in the system using a depth-first search (DFS) tree. (As an example: the DFS iteration of the processes in Figure 3.7 is 1, 8415, 8416, 9298, 9204, 2808, 3028, 3610, 4005.)

Linux maintains its process tree as a series of lists. Examining the task_struct in <linux/sched.h>, we see two struct list_head objects:

```
children
```

and

```
sibling
```

These objects are pointers to a list of the task's children, as well as its siblings. Linux also maintains a reference to the initial task in the system — init_task — which is of type task_struct. Using this information as well as macro operations on lists, we can iterate over the children of init_task as follows:

```
struct task_struct *task;
struct list_head *list;
```

```
list_for_each(list, &init_task->children) {
    task = list_entry(list, struct task_struct, sibling);
    /* task points to the next child in the list */
}
```

The `list_for_each()` macro is passed two parameters, both of type `struct list_head`:

- A pointer to the head of the list to be traversed
- A pointer to the head node of the list to be traversed

At each iteration of `list_for_each()`, the first parameter is set to the `list` structure of the next child. We then use this value to obtain each structure in the list using the `list_entry()` macro.

Assignment

Beginning from `init_task` task, design a kernel module that iterates over all tasks in the system using a DFS tree. Just as in the first part of this project, output the name, state, and pid of each task. Perform this iteration in the kernel entry module so that its output appears in the kernel log buffer.

If you output all tasks in the system, you may see many more tasks than appear with the `ps -ael` command. This is because some threads appear as children but do not show up as ordinary processes. Therefore, to check the output of the DFS tree, use the command

```
ps -eLf
```

This command lists all tasks—including threads—in the system. To verify that you have indeed performed an appropriate DFS iteration, you will have to examine the relationships among the various tasks output by the `ps` command.

Project 4—Kernel Data Structures

In Section 1.9, we covered various data structures that are common in operating systems. The Linux kernel provides several of these structures. Here, we explore using the circular, doubly linked list that is available to kernel developers. Much of what we discuss is available in the Linux source code—in this instance, the include file `<linux/list.h>`—and we recommend that you examine this file as you proceed through the following steps.

Initially, you must define a `struct` containing the elements that are to be inserted in the linked list. The following C `struct` defines a color as a mixture of red, blue, and green:

```
struct color {
    int red;
    int blue;
    int green;
```

```
    struct list_head list;
};
```

Notice the member `struct list_head list`. The `list_head` structure is defined in the include file `<linux/types.h>`, and its intention is to embed the linked list within the nodes that comprise the list. This `list_head` structure is quite simple—it merely holds two members, `next` and `prev`, that point to the next and previous entries in the list. By embedding the linked list within the structure, Linux makes it possible to manage the data structure with a series of *macro* functions.

I. Inserting Elements into the Linked List

We can declare a `list_head` object, which we use as a reference to the head of the list by using the LIST_HEAD() macro:

```
        static LIST_HEAD(color_list);
```

This macro defines and initializes the variable `color_list`, which is of type `struct list_head`.

   We create and initialize instances of `struct color` as follows:

```
    struct color *violet;

    violet = kmalloc(sizeof(*violet), GFP_KERNEL);
    violet->red = 138;
    violet->blue = 43;
    violet->green = 226;
    INIT_LIST_HEAD(&violet->list);
```

The `kmalloc()` function is the kernel equivalent of the user-level `malloc()` function for allocating memory, except that kernel memory is being allocated. The GFP_KERNEL flag indicates routine kernel memory allocation. The macro INIT_LIST_HEAD() initializes the `list` member in `struct color`. We can then add this instance to the end of the linked list using the `list_add_tail()` macro:

```
    list_add_tail(&violet->list, &color_list);
```

II. Traversing the Linked List

Traversing the list involves using the `list_for_each_entry()` macro, which accepts three parameters:

- A pointer to the structure being iterated over
- A pointer to the head of the list being iterated over
- The name of the variable containing the `list_head` structure

The following code illustrates this macro:

```
struct color *ptr;

list_for_each_entry(ptr, &color_list, list) {
    /* on each iteration ptr points */
    /* to the next struct color */
}
```

III. Removing Elements from the Linked List

Removing elements from the list involves using the list_del() macro, which is passed a pointer to struct list_head:

```
list_del(struct list_head *element);
```

This removes element from the list while maintaining the structure of the remainder of the list.

Perhaps the simplest approach for removing all elements from a linked list is to remove each element as you traverse the list. The macro list_for_each_entry_safe() behaves much like list_for_each_entry() except that it is passed an additional argument that maintains the value of the next pointer of the item being deleted. (This is necessary for preserving the structure of the list.) The following code example illustrates this macro:

```
struct color *ptr, *next;

list_for_each_entry_safe(ptr,next,&color_list,list) {
    /* on each iteration ptr points */
    /* to the next struct color */
    list_del(&ptr->list);
    kfree(ptr);
}
```

Notice that after deleting each element, we return memory that was previously allocated with kmalloc() back to the kernel with the call to kfree().

Part I—Assignment

In the module entry point, create a linked list containing four struct color elements. Traverse the linked list and output its contents to the kernel log buffer. Invoke the dmesg command to ensure that the list is properly constructed once the kernel module has been loaded.

In the module exit point, delete the elements from the linked list and return the free memory back to the kernel. Again, invoke the dmesg command to check that the list has been removed once the kernel module has been unloaded.

Part II—Parameter Passing

This portion of the project will involve passing a parameter to a kernel module. The module will use this parameter as an initial value and generate the Collatz sequence as described in Exercise 3.21.

Passing a Parameter to a Kernel Module

Parameters may be passed to kernel modules when they are loaded. For example, if the name of the kernel module is collatz, we can pass the initial value of 15 to the kernel parameter start as follows:

```
sudo insmod collatz.ko start=15
```

Within the kernel module, we declare start as a parameter using the following code:

```
#include<linux/moduleparam.h>

static int start = 25;

module_param(start, int, 0);
```

The module_param() macro is used to establish variables as parameters to kernel modules. module_param() is provided three arguments: (1) the name of the parameter, (2) its type, and (3) file permissions. Since we are not using a file system for accessing the parameter, we are not concerned with permissions and use a default value of 0. Note that the name of the parameter used with the insmod command must match the name of the associated kernel parameter. Finally, if we do not provide a value to the module parameter during loading with insmod, the default value (which in this case is 25) is used.

Part II — Assignment

Design a kernel module named collatz that is passed an initial value as a module parameter. Your module will then generate and store the sequence in a kernel linked list when the module is loaded. Once the sequence has been stored, your module will traverse the list and output its contents to the kernel log buffer. Use the dmesg command to ensure that the sequence is properly generated once the module has been loaded.

In the module exit point, delete the contents of the list and return the free memory back to the kernel. Again, use dmesg to check that the list has been removed once the kernel module has been unloaded.

CHAPTER

Threads &
Concurrency

4

The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control. Virtually all modern operating systems, however, provide features enabling a process to contain multiple threads of control. Identifying opportunities for parallelism through the use of threads is becoming increasingly important for modern multicore systems that provide multiple CPUs.

In this chapter, we introduce many concepts, as well as challenges, associated with multithreaded computer systems, including a discussion of the APIs for the Pthreads, Windows, and Java thread libraries. Additionally, we explore several new features that abstract the concept of creating threads, allowing developers to focus on identifying opportunities for parallelism and letting language features and API frameworks manage the details of thread creation and management. We look at a number of issues related to multithreaded programming and its effect on the design of operating systems. Finally, we explore how the Windows and Linux operating systems support threads at the kernel level.

CHAPTER OBJECTIVES

- Identify the basic components of a thread, and contrast threads and processes.
- Describe the major benefits and significant challenges of designing multithreaded processes.
- Illustrate different approaches to implicit threading, including thread pools, fork-join, and Grand Central Dispatch.
- Describe how the Windows and Linux operating systems represent threads.
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs.

4.1   Overview

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

4.1.1   Motivation

Most software applications that run on modern computers and mobile devices
are multithreaded. An application typically is implemented as a separate pro-
cess with several threads of control. Below we highlight a few examples of
multithreaded applications:

- An application that creates photo thumbnails from a collection of images
  may use a separate thread to generate a thumbnail from each separate
  image.

- A web browser might have one thread display images or text while another
  thread retrieves data from the network.

- A word processor may have a thread for displaying graphics, another
  thread for responding to keystrokes from the user, and a third thread for
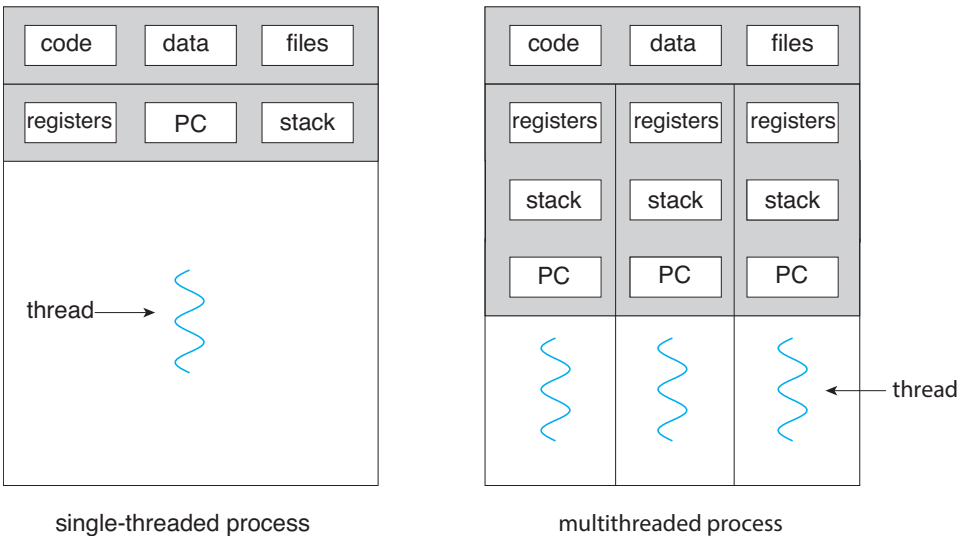 performing spelling and grammar checking in the background.

Figure 4.1   Single-threaded and multithreaded processes.

Applications can also be designed to leverage processing capabilities on multicore systems. Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

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

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resumes listening for additional requests. This is illustrated in Figure 4.2.

Most operating system kernels are also typically multithreaded. As an example, during system boot time on Linux systems, several kernel threads are created. Each thread performs a specific task, such as managing devices, memory management, or interrupt handling. The command `ps -ef` can be used to display the kernel threads on a running Linux system. Examining the output of this command will show the kernel thread `kthreadd` (with `pid = 2`), which serves as the parent of all other kernel threads.

Many applications can also take advantage of multiple threads, including basic sorting, trees, and graph algorithms. In addition, programmers who must solve contemporary CPU-intensive problems in data mining, graphics, and artificial intelligence can leverage the power of modern multicore systems by designing solutions that run in parallel.
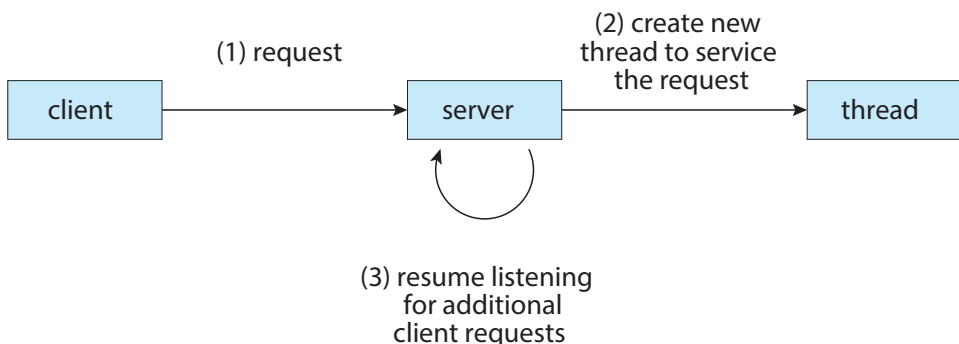

Figure 4.2 Multithreaded server architecture.

4.1.2   Benefits

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

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.

2. **Resource sharing**. Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.

4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

4.2   Multicore Programming

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A later, yet similar, trend in system design is to place multiple computing cores on a single processing chip where each core appears as a separate CPU to the operating system (Section 1.3.2). We refer to such systems as **multicore**, and multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency
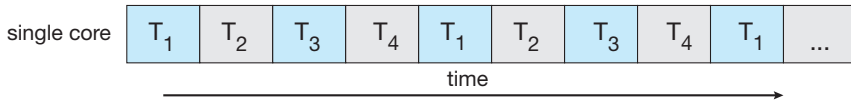
Figure 4.3 Concurrent execution on a single-core system.

means that some threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

Notice the distinction between *concurrency* and *parallelism* in this discussion. A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously. Thus, it is possible to have concurrency without parallelism. Before the advent of multiprocessor and multicore architectures, most computer systems had only a single processor, and CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

4.2.1 Programming Challenges

The trend toward multicore systems continues to place pressure on system designers and application programmers to make better use of the multiple computing cores. Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution shown in Figure 4.4. For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded.

In general, five areas present challenges in programming for multicore systems:

1. **Identifying tasks**. This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.

2. **Balance**. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.
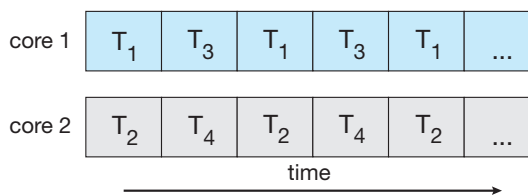

Figure 4.4 Parallel execution on a multicore system.

AMDAHL'S LAW

Amdahl's Law is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial (nonparallel) and parallel components. If S is the portion of the application that must be performed serially on a system with N processing cores, the formula appears as follows:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

As an example, assume we have an application that is 75 percent parallel and 25 percent serial. If we run this application on a system with two processing cores, we can get a speedup of 1.6 times. If we add two additional cores (for a total of four), the speedup is 2.28 times. Below is a graph illustrating Amdahl's Law in several different scenarios.
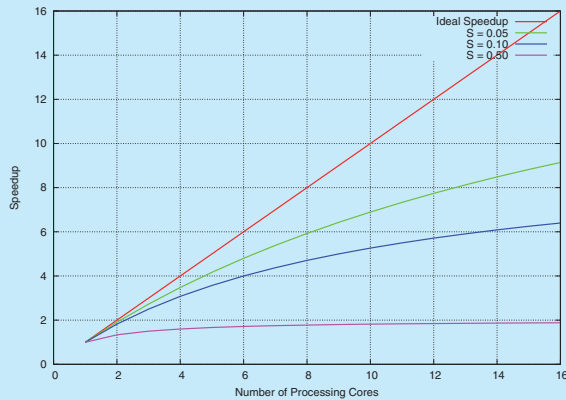
One interesting fact about Amdahl's Law is that as N approaches infinity, the speedup converges to $1/S$. For example, if 50 percent of an application is performed serially, the maximum speedup is 2.0 times, regardless of the number of processing cores we add. This is the fundamental principle behind Amdahl's Law: the serial portion of an application can have a disproportionate effect on the performance we gain by adding additional computing cores.

3. **Data splitting**. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

4. **Data dependency**. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency. We examine such strategies in Chapter 6.
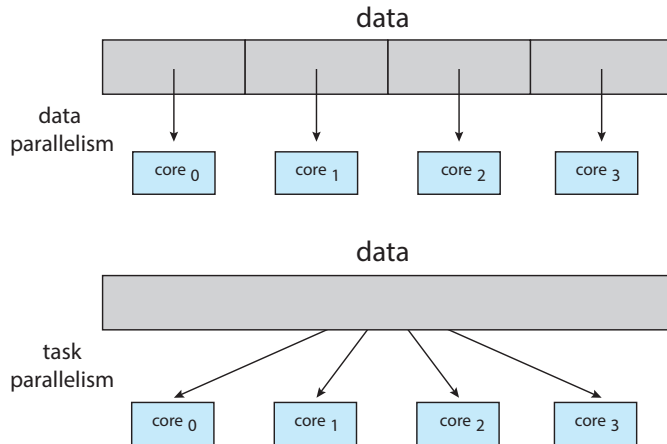
data

Figure 4.5   Data and task parallelism.

5. **Testing and debugging**. When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future. (Similarly, many computer science educators believe that software development must be taught with increased emphasis on parallel programming.)

4.2.2 Types of Parallelism

In general, there are two types of parallelism: data parallelism and task parallelism. **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. Consider, for example, summing the contents of an array of size N. On a single-core system, one thread would simply sum the elements $[0] \dots [N-1]$. On a dual-core system, however, thread A, running on core 0, could sum the elements $[0] \dots [N/2-1]$ while thread B, running on core 1, could sum the elements $[N/2] \dots [N-1]$. The two threads would be running in parallel on separate computing cores.

Task parallelism involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data. Consider again our example above. In contrast to that situation, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.

Fundamentally, then, data parallelism involves the distribution of data across multiple cores, and task parallelism involves the distribution of tasks across multiple cores, as shown in Figure 4.5. However, data and task paral-
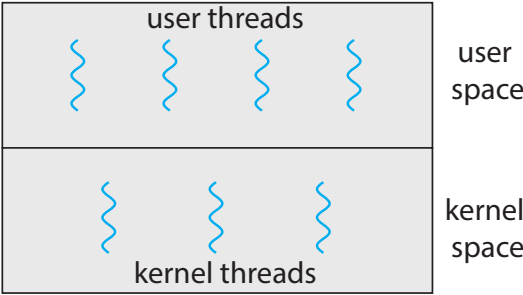
Figure 4.6   User and kernel threads.

lelism are not mutually exclusive, and an application may in fact use a hybrid of these two strategies.

4.3   Multithreading Models

Our discussion so far has treated threads in a generic sense. However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, and macOS— support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads, as illustrated in Figure 4.6. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to-many model.

4.3.1   Many-to-One Model

The many-to-one model (Figure 4.7) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient (we discuss thread libraries in Section 4.4). However, the entire process will block if a thread makes a blocking system call. Also, because only
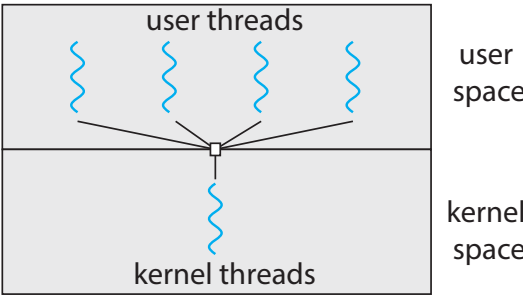

Figure 4.7   Many-to-one model.
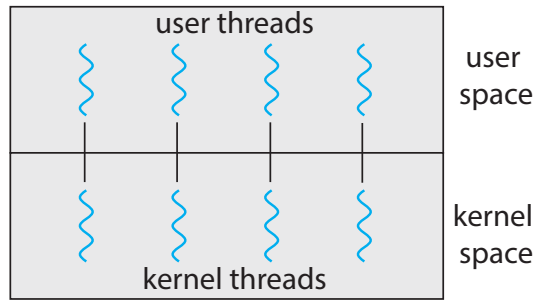
Figure 4.8 One-to-one model.

one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. **Green threads**—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model. However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores, which have now become standard on most computer systems.

4.3.2 One-to-One Model

The one-to-one model (Figure 4.8) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

4.3.3 Many-to-Many Model

The many-to-many model (Figure 4.9) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a system with eight processing cores than a system with four cores).

Figure 4.9 Many-to-many model.

Figure 4.10   Two-level model.

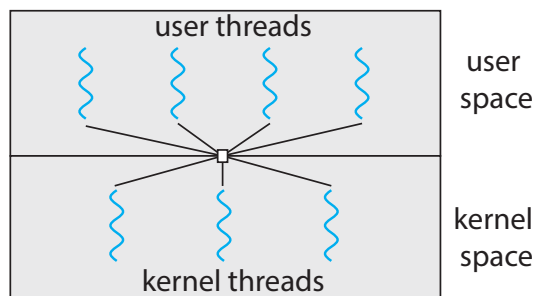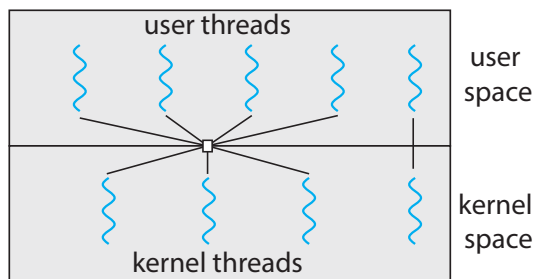Let's consider the effect of this design on concurrency. Whereas the many-to-one model allows the developer to create as many user threads as she wishes, it does not result in parallelism, because the kernel can schedule only one kernel thread at a time. The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application. (In fact, on some systems, she may be limited in the number of threads she can create.) The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model** (Figure 4.10).

Although the many-to-many model appears to be the most flexible of the models discussed, in practice it is difficult to implement. In addition, with an increasing number of processing cores appearing on most systems, limiting the number of kernel threads has become less important. As a result, most operating systems now use the one-to-one model. However, as we shall see in Section 4.5, some contemporary concurrency libraries have developers identify tasks that are then mapped to threads using the many-to-many model.

4.4   Thread Libraries

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

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

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

For POSIX and Windows threading, any data declared globally—that is, declared outside of any function—are shared among all threads belonging to the same process. Because Java has no equivalent notion of global data, access to shared data must be explicitly arranged between threads.

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

$$sum = \sum_{i=1}^{N} i$$

For example, if N were 5, this function would represent the summation of integers from 1 to 5, which is 15. Each of the three programs will be run with the upper bounds of the summation entered on the command line. Thus, if the user enters 8, the summation of the integer values from 1 to 8 will be output.

Before we proceed with our examples of thread creation, we introduce two general strategies for creating multiple threads: **asynchronous threading** and **synchronous threading**. With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of one another. Because the threads are independent, there is typically little data sharing between them. Asynchronous threading is the strategy used in the multithreaded server illustrated in Figure 4.2 and is also commonly used for designing responsive user interfaces.

Synchronous threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes. Here, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed. Once each thread has finished its work, it terminates and joins with its parent. Only after all of the children have joined can the parent resume execution. Typically, synchronous threading involves significant data sharing among threads. For example, the parent thread may combine the results calculated by its various children. All of the following examples use synchronous threading.

4.4.1 Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a *specification* for thread behavior, not an *implementation*. Operating-system designers may implement the specification

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

#include <stdlib.h>

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

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

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

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

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

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

   pthread_exit(0);
}
```

Figure 4.11   Multithreaded C program using the Pthreads API.

in any way they wish. Numerous systems implement the Pthreads specifica-
tion; most are UNIX-type systems, including Linux and macOS. Although Win-
dows doesn't support Pthreads natively, some third-party implementations for
Windows are available.

The C program shown in Figure 4.11 demonstrates the basic Pthreads API
for constructing a multithreaded program that calculates the summation of
a non-negative integer in a separate thread. In a Pthreads program, separate
threads begin execution in a specified function. In Figure 4.11, this is the run-
ner() function. When this program begins, a single thread of control begins in

```
#define NUM_THREADS 10

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

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

Figure 4.12   Pthread code for joining ten threads.

main(). After some initialization, main() creates a second thread that begins control in the runner() function. Both threads share the global data sum.

Let's look more closely at this program. All Pthreads programs must include the pthread.h header file. The statement pthread_t tid declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The pthread_attr_t attr declaration represents the attributes for the thread. We set the attributes in the function call pthread_attr_init(&attr). Because we did not explicitly set any attributes, we use the default attributes provided. (In Chapter 5, we discuss some of the scheduling attributes provided by the Pthreads API.) A separate thread is created with the pthread_create() function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the runner() function. Last, we pass the integer parameter that was provided on the command line, argv[1].

At this point, the program has two threads: the initial (or parent) thread in main() and the summation (or child) thread performing the summation operation in the runner() function. This program follows the thread create/join strategy, whereby after creating the summation thread, the parent thread will wait for it to terminate by calling the pthread_join() function. The summation thread will terminate when it calls the function pthread_exit(). Once the summation thread has returned, the parent thread will output the value of the shared data sum.

This example program creates only a single thread. With the growing dominance of multicore systems, writing programs containing several threads has become increasingly common. A simple method for waiting on several threads using the pthread_join() function is to enclose the operation within a simple for loop. For example, you can join on ten threads using the Pthread code shown in Figure 4.12.

4.4.2   Windows Threads

The technique for creating threads using the Windows thread library is similar to the Pthreads technique in several ways. We illustrate the Windows thread API in the C program shown in Figure 4.13. Notice that we must include the windows.h header file when using the Windows API.

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

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

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

Figure 4.13   Multithreaded C program using the Windows API.

Just as in the Pthreads version shown in Figure 4.11, data shared by the separate threads—in this case, Sum—are declared globally (the DWORD data type is an unsigned 32-bit integer). We also define the Summation() function that is to be performed in a separate thread. This function is passed a pointer to a void, which Windows defines as LPVOID. The thread performing this function sets the global data Sum to the value of the summation from 0 to the parameter passed to Summation().

Threads are created in the Windows API using the `CreateThread()` function, and—just as in Pthreads—a set of attributes for the thread is passed to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state. In this program, we use the default values for these attributes. (The default values do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler.) Once the summation thread is created, the parent must wait for it to complete before outputting the value of `Sum`, as the value is set by the summation thread. Recall that the Pthread program (Figure 4.11) had the parent thread wait for the summation thread using the `pthread_join()` statement. We perform the equivalent of this in the Windows API using the `WaitForSingleObject()` function, which causes the creating thread to block until the summation thread has exited.

In situations that require waiting for multiple threads to complete, the `WaitForMultipleObjects()` function is used. This function is passed four parameters:

1. The number of objects to wait for

2. A pointer to the array of objects

3. A flag indicating whether all objects have been signaled

4. A timeout duration (or `INFINITE`)

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

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

4.4.3 Java Threads

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a `main()` method runs as a single thread in the JVM. Java threads are available on any system that provides a JVM including Windows, Linux, and macOS. The Java thread API is available for Android applications as well.

There are two techniques for explicitly creating threads in a Java program. One approach is to create a new class that is derived from the `Thread` class and to override its `run()` method. An alternative—and more commonly used—technique is to define a class that implements the `Runnable` interface. This interface defines a single abstract method with the signature `public void run()`. The code in the `run()` method of a class that implements `Runnable` is what executes in a separate thread. An example is shown below:

```java
class Task implements Runnable
{
  public void run() {
    System.out.println("I am a thread.");
  }
}
```

LAMBDA EXPRESSIONS IN JAVA

Beginning with Version 1.8 of the language, Java introduced Lambda expressions, which allow a much cleaner syntax for creating threads. Rather than defining a separate class that implements `Runnable`, a Lambda expression can be used instead:

```
Runnable task = () -> {
   System.out.println("I am a thread.");
};

Thread worker = new Thread(task);
worker.start();
```

Lambda expressions—as well as similar functions known as **closures**—are a prominent feature of functional programming languages and have been available in several nonfunctional languages as well including Python, C++, and C#. As we shall see in later examples in this chapter, Lamdba expressions often provide a simple syntax for developing parallel applications.

Thread creation in Java involves creating a `Thread` object and passing it an instance of a class that implements `Runnable`, followed by invoking the `start()` method on the `Thread` object. This appears in the following example:

```
Thread worker = new Thread(new Task());
worker.start();
```

Invoking the `start()` method for the new `Thread` object does two things:

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

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

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

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

If the parent must wait for several threads to finish, the `join()` method can be enclosed in a `for` loop similar to that shown for Pthreads in Figure 4.12.

4.4.3.1   Java Executor Framework

Java has supported thread creation using the approach we have described thus far since its origins. However, beginning with Version 1.5 and its API, Java introduced several new concurrency features that provide developers with much greater control over thread creation and communication. These tools are available in the `java.util.concurrent` package.

Rather than explicitly creating `Thread` objects, thread creation is instead organized around the `Executor` interface:

```
public interface Executor
{
    void execute(Runnable command);
}
```

Classes implementing this interface must define the `execute()` method, which is passed a `Runnable` object. For Java developers, this means using the `Executor` rather than creating a separate `Thread` object and invoking its `start()` method. The `Executor` is used as follows:

```
Executor service = new Executor;
service.execute(new Task());
```

The `Executor` framework is based on the producer-consumer model; tasks implementing the `Runnable` interface are produced, and the threads that execute these tasks consume them. The advantage of this approach is that it not only divides thread creation from execution but also provides a mechanism for communication between concurrent tasks.

Data sharing between threads belonging to the same process occurs easily in Windows and Pthreads, since shared data are simply declared globally. As a pure object-oriented language, Java has no such notion of global data. We can pass parameters to a class that implements `Runnable`, but Java threads cannot return results. To address this need, the `java.util.concurrent` package additionally defines the `Callable` interface, which behaves similarly to `Runnable` except that a result can be returned. Results returned from `Callable` tasks are known as `Future` objects. A result can be retrieved from the `get()` method defined in the `Future` interface. The program shown in Figure 4.14 illustrates the summation program using these Java features.

The `Summation` class implements the `Callable` interface, which specifies the method `V call()`—it is the code in this `call()` method that is executed in a separate thread. To execute this code, we create a `newSingleThreadExecutor` object (provided as a static method in the `Executors` class), which is of type `ExecutorService`, and pass it a `Callable` task using its `submit()` method. (The primary difference between the `execute()` and `submit()` methods is that the former returns no result, whereas the latter returns a result as a `Future`.) Once we submit the `callable` task to the thread, we wait for its result by calling the `get()` method of the `Future` object it returns.

It is quite easy to notice at first that this model of thread creation appears more complicated than simply creating a thread and joining on its termination. However, incurring this modest degree of complication confers benefits. As we have seen, using `Callable` and `Future` allows for threads to return results.

```java
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
  private int upper;
  public Summation(int upper) {
    this.upper = upper;
  }

  /* The thread will execute in this method */
  public Integer call() {
    int sum = 0;
    for (int i = 1; i <= upper; i++)
      sum += i;

    return new Integer(sum);
  }
}

public class Driver
{
 public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
        System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```

Figure 4.14   Illustration of Java Executor framework API.

Additionally, this approach separates the creation of threads from the results they produce: rather than waiting for a thread to terminate before retrieving results, the parent instead only waits for the results to become available. Finally, as we shall see in Section 4.5.1, this framework can be combined with other features to create robust tools for managing a large number of threads.

4.5 Implicit Threading

With the continued growth of multicore processing, applications contain-ing hundreds—or even thousands—of threads are looming on the horizon. Designing such applications is not a trivial undertaking: programmers must

THE JVM AND THE HOST OPERATING SYSTEM

The JVM is typically implemented on top of a host operating system (see Figure 18.10). This setup allows the JVM to hide the implementation details of the underlying operating system and to provide a consistent, abstract environment that allows Java programs to operate on any platform that supports a JVM. The specification for the JVM does not indicate how Java threads are to be mapped to the underlying operating system, instead leaving that decision to the particular implementation of the JVM. For example, the Windows operating system uses the one-to-one model; therefore, each Java thread for a JVM running on Windows maps to a kernel thread. In addition, there may be a relationship between the Java thread library and the thread library on the host operating system. For example, implementations of a JVM for the Windows family of operating systems might use the Windows API when creating Java threads; Linux and macOS systems might use the Pthreads API.

address not only the challenges outlined in Section 4.2 but additional difficulties as well. These difficulties, which relate to program correctness, are covered in Chapter 6 and Chapter 8.

One way to address these difficulties and better support the design of concurrent and parallel applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy, termed **implicit threading**, is an increasingly popular trend. In this section, we explore four alternative approaches to designing applications that can take advantage of multicore processors through implicit threading. As we shall see, these strategies generally require application developers to identify *tasks*—not threads—that can run in parallel. A task is usually written as a function, which the run-time library then maps to a separate thread, typically using the many-to-many model (Section 4.3.3). The advantage of this approach is that developers only need to identify parallel tasks, and the libraries determine the specific details of thread creation and management.

4.5.1 Thread Pools

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

ANDROID THREAD POOLS

In Section 3.8.2.1, we covered RPCs in the Android operating system. You may recall from that section that Android uses the Android Interface Definition Language (AIDL), a tool that specifies the remote interface that clients interact with on the server. AIDL also provides a thread pool. A remote service using the thread pool can handle multiple concurrent requests, servicing each request using a separate thread from the pool.

The general idea behind a thread pool is to create a number of threads at start-up and place them into a pool, where they sit and wait for work. When a server receives a request, rather than creating a thread, it instead submits the request to the thread pool and resumes waiting for additional requests. If there is an available thread in the pool, it is awakened, and the request is serviced immediately. If the pool contains no available thread, the task is queued until one becomes free. Once a thread completes its service, it returns to the pool and awaits more work. Thread pools work well when the tasks submitted to the pool can be executed asynchronously.

Thread pools offer these benefits:

1. Servicing a request with an existing thread is often faster than waiting to create a thread.

2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically.

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

The Windows API provides several functions related to thread pools. Using the thread pool API is similar to creating a thread with the `Thread_Create()` function, as described in Section 4.4.2. Here, a function that is to run as a separate thread is defined. Such a function may appear as follows:

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

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

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

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

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

An example of invoking a function is the following:

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

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

Other members in the Windows thread pool API include utilities that invoke functions at periodic intervals or when an asynchronous I/O request completes.

4.5.1.1 Java Thread Pools

The `java.util.concurrent` package includes an API for several varieties of thread-pool architectures. Here, we focus on the following three models:

1. Single thread executor—`newSingleThreadExecutor()`—creates a pool of size 1.

2. Fixed thread executor—`newFixedThreadPool(int size)`—creates a thread pool with a specified number of threads.

3. Cached thread executor—`newCachedThreadPool()`—creates an unbounded thread pool, reusing threads in many instances.

We have, in fact, already seen the use of a Java thread pool in Section 4.4.3, where we created a `newSingleThreadExecutor` in the program example shown in Figure 4.14. In that section, we noted that the Java executor framework can be used to construct more robust threading tools. We now describe how it can be used to create thread pools.

A thread pool is created using one of the factory methods in the `Executors` class:

- `static ExecutorService newSingleThreadExecutor()`

- `static ExecutorService newFixedThreadPool(int size)`

- `static ExecutorService newCachedThreadPool()`

Each of these factory methods creates and returns an object instance that implements the `ExecutorService` interface. `ExecutorService` extends the `Execu-`

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
public static void main(String[] args) {
   int numTasks = Integer.parseInt(args[0].trim());

   /* Create the thread pool */
   ExecutorService pool = Executors.newCachedThreadPool();

   /* Run each task using a thread in the pool */
   for (int i = 0; i < numTasks; i++)
     pool.execute(new Task());

   /* Shut down the pool once all threads have completed */
   pool.shutdown();
}
```

Figure 4.15   Creating a thread pool in Java.

tor interface, allowing us to invoke the execute() method on this object. In addition, ExecutorService provides methods for managing termination of the thread pool.

The example shown in Figure 4.15 creates a cached thread pool and submits tasks to be executed by a thread in the pool using the execute() method. When the shutdown() method is invoked, the thread pool rejects additional tasks and shuts down once all existing tasks have completed execution.

4.5.2 Fork Join

The strategy for thread creation covered in Section 4.4 is often known as the **fork-join** model. Recall that with this method, the main parent thread creates (*forks*) one or more child threads and then waits for the children to terminate and *join* with it, at which point it can retrieve and combine their results. This synchronous model is often characterized as explicit thread creation, but it is also an excellent candidate for implicit threading. In the latter situation, threads are not constructed directly during the fork stage; rather, parallel tasks are designated. This model is illustrated in Figure 4.16. A library manages the number of threads that are created and is also responsible for assigning tasks to threads. In some ways, this fork-join model is a synchronous version of thread pools in which a library determines the actual number of threads to create— for example, by using the heuristics described in Section 4.5.1.

4.5.2.1   Fork Join in Java

Java introduced a fork-join library in Version 1.7 of the API that is designed to be used with recursive divide-and-conquer algorithms such as Quicksort and Mergesort. When implementing divide-and-conquer algorithms using this
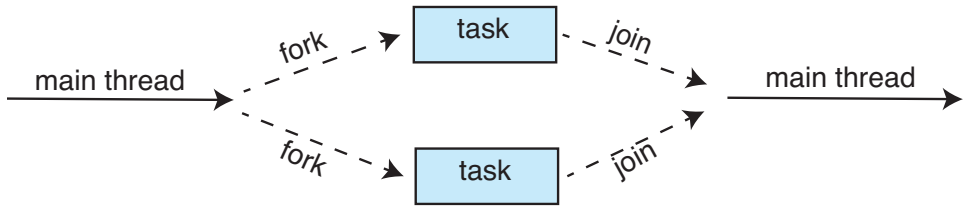
Figure 4.16 Fork-join parallelism.

library, separate tasks are forked during the divide step and assigned smaller subsets of the original problem. Algorithms must be designed so that these separate tasks can execute concurrently. At some point, the size of the problem assigned to a task is small enough that it can be solved directly and requires creating no additional tasks. The general recursive algorithm behind Java's fork-join model is shown below:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

Figure 4.17 depicts the model graphically.

We now illustrate Java's fork-join strategy by designing a divide-and-conquer algorithm that sums all elements in an array of integers. In Version 1.7 of the API Java introduced a new thread pool—the ForkJoinPool—that can be assigned tasks that inherit the abstract base class ForkJoinTask (which for now we will assume is the SumTask class). The following creates a ForkJoin-Pool object and submits the initial task via its invoke() method:

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

Upon completion, the initial call to invoke() returns the summation of array.

The class SumTask—shown in Figure 4.18—implements a divide-and-conquer algorithm that sums the contents of the array using fork-join. New tasks are created using the fork() method, and the compute() method specifies the computation that is performed by each task. The method compute() is invoked until it can directly calculate the sum of the subset it is assigned. The

Figure 4.17 Fork-join in Java.

call to `join()` blocks until the task completes, upon which `join()` returns the results calculated in `compute()`.

Notice that `SumTask` in Figure 4.18 extends `RecursiveTask`. The Java fork-join strategy is organized around the abstract base class `ForkJoinTask`, and the `RecursiveTask` and `RecursiveAction` classes extend this class. The fundamental difference between these two classes is that `RecursiveTask` returns a result (via the return value specified in `compute()`), and `RecursiveAction` does not return a result. The relationship between the three classes is illustrated in the UML class diagram in Figure 4.19.

An important issue to consider is determining when the problem is "small enough" to be solved directly and no longer requires creating additional tasks. In `SumTask`, this occurs when the number of elements being summed is less than the value `THRESHOLD`, which in Figure 4.18 we have arbitrarily set to 1,000. In practice, determining when a problem can be solved directly requires careful timing trials, as the value can vary according to implementation.

What is interesting in Java's fork-join model is the management of tasks wherein the library constructs a pool of worker threads and balances the load of tasks among the available workers. In some situations, there are thousands of tasks, yet only a handful of threads performing the work (for example, a separate thread for each CPU). Additionally, each thread in a `ForkJoinPool` maintains a queue of tasks that it has forked, and if a thread's queue is empty, it can steal a task from another thread's queue using a *work stealing* algorithm, thus balancing the workload of tasks among all threads.

```java
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
   private int[] array;

   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }

   protected Integer compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];

         return sum;
      }
      else {
         int mid = (begin + end) / 2;

         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);

         leftTask.fork();
         rightTask.fork();

         return rightTask.join() + leftTask.join();
      }
   }
}
```

Figure 4.18 Fork-join calculation using the Java API.

4.5.3 OpenMP

OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies **parallel regions** as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-
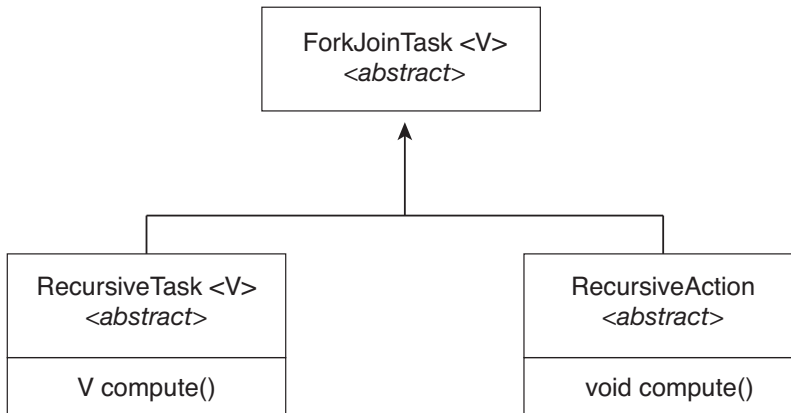
```
┌─────────────────────────┐
│   ForkJoinTask <V>      │
│      <abstract>         │
└─────────────────────────┘
```


Figure 4.19   UML class diagram for Java's fork-join.

time library to execute the region in parallel. The following C program illustrates a compiler directive above the parallel region containing the `printf()` statement:

```c
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

When OpenMP encounters the directive

```
#pragma omp parallel
```

it creates as many threads as there are processing cores in the system. Thus, for a dual-core system, two threads are created; for a quad-core system, four are created; and so forth. All the threads then simultaneously execute the parallel region. As each thread exits the parallel region, it is terminated.

OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops. For example, assume we have two arrays, a and b, of size N. We wish to sum their contents and place the results

in array `c`. We can have this task run in parallel by using the following code segment, which contains the compiler directive for parallelizing `for` loops:

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

OpenMP divides the work contained in the `for` loop among the threads it has created in response to the directive

```
#pragma omp parallel for
```

In addition to providing directives for parallelization, OpenMP allows developers to choose among several levels of parallelism. For example, they can set the number of threads manually. It also allows developers to identify whether data are shared between threads or are private to a thread. OpenMP is available on several open-source and commercial compilers for Linux, Windows, and macOS systems. We encourage readers interested in learning more about OpenMP to consult the bibliography at the end of the chapter.

4.5.4 Grand Central Dispatch

Grand Central Dispatch (GCD) is a technology developed by Apple for its macOS and iOS operating systems. It is a combination of a run-time library, an API, and language extensions that allow developers to identify sections of code (*tasks*) to run in parallel. Like OpenMP, GCD manages most of the details of threading.

GCD schedules tasks for run-time execution by placing them on a **dispatch queue**. When it removes a task from a queue, it assigns the task to an available thread from a pool of threads that it manages. GCD identifies two types of dispatch queues: *serial* and *concurrent*.

Tasks placed on a serial queue are removed in FIFO order. Once a task has been removed from the queue, it must complete execution before another task is removed. Each process has its own serial queue (known as its **main queue**), and developers can create additional serial queues that are local to a particular process. (This is why serial queues are also known as *private dispatch queues*.) Serial queues are useful for ensuring the sequential execution of several tasks.

Tasks placed on a concurrent queue are also removed in FIFO order, but several tasks may be removed at a time, thus allowing multiple tasks to execute in parallel. There are several system-wide concurrent queues (also known as *global dispatch queues*), which are divided into four primary quality-of-service classes:

- `QOS_CLASS_USER_INTERACTIVE`—The **user-interactive** class represents tasks that interact with the user, such as the user interface and event handling, to ensure a responsive user interface. Completing a task belonging to this class should require only a small amount of work.

- `QOS_CLASS_USER_INITIATED`—The **user-initiated** class is similar to the user-interactive class in that tasks are associated with a responsive user interface; however, user-initiated tasks may require longer processing

times. Opening a file or a URL is a user-initiated task, for example. Tasks belonging to this class must be completed for the user to continue inter-acting with the system, but they do not need to be serviced as quickly as tasks in the user-interactive queue.

- QOS_CLASS_UTILITY —The **utility** class represents tasks that require a longer time to complete but do not demand immediate results. This class includes work such as importing data.

- QOS_CLASS_BACKGROUND —Tasks belonging to the **background** class are not visible to the user and are not time sensitive. Examples include index-ing a mailbox system and performing backups.

Tasks submitted to dispatch queues may be expressed in one of two different ways:

1. For the C, C++, and Objective-C languages, GCD identifies a language extension known as a **block**, which is simply a self-contained unit of work. A block is specified by a caret ˆ inserted in front of a pair of braces { }. Code within the braces identifies the unit of work to be performed. A simple example of a block is shown below:

```
ˆ{ printf("I am a block"); }
```

2. For the Swift programming language, a task is defined using a *closure*, which is similar to a block in that it expresses a self-contained unit of functionality. Syntactically, a Swift closure is written in the same way as a block, minus the leading caret.

    The following Swift code segment illustrates obtaining a concurrent queue for the user-initiated class and submitting a task to the queue using the dispatch_async() function:

```
let queue = dispatch_get_global_queue
    (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```

Internally, GCD's thread pool is composed of POSIX threads. GCD actively manages the pool, allowing the number of threads to grow and shrink accord-ing to application demand and system capacity. GCD is implemented by the libdispatch library, which Apple has released under the Apache Commons license. It has since been ported to the FreeBSD operating system.

4.5.5 Intel Thread Building Blocks

Intel threading building blocks (TBB) is a template library that supports design-ing parallel applications in C++. As this is a library, it requires no special compiler or language support. Developers specify tasks that can run in par-

allel, and the TBB task scheduler maps these tasks onto underlying threads. Furthermore, the task scheduler provides load balancing and is cache aware, meaning that it will give precedence to tasks that likely have their data stored in cache memory and thus will execute more quickly. TBB provides a rich set of features, including templates for parallel loop structures, atomic operations, and mutual exclusion locking. In addition, it provides concurrent data structures, including a hash map, queue, and vector, which can serve as equivalent thread-safe versions of the C++ standard template library data structures.

Let's use parallel for loops as an example. Initially, assume there is a function named `apply(float value)` that performs an operation on the parameter `value`. If we had an array `v` of size `n` containing `float` values, we could use the following serial for loop to pass each value in `v` to the `apply()` function:

```
for (int i = 0; i < n; i++) {
    apply(v[i]);
}
```

A developer could manually apply data parallelism (Section 4.2.2) on a multicore system by assigning different regions of the array `v` to each processing core; however, this ties the technique for achieving parallelism closely to the physical hardware, and the algorithm would have to be modified and recompiled for the number of processing cores on each specific architecture.

Alternatively, a developer could use TBB, which provides a `parallel_for` template that expects two values:

    `parallel_for (`*range* *body* `)`

where *range* refers to the range of elements that will be iterated (known as the **iteration space**) and *body* specifies an operation that will be performed on a subrange of elements.

We can now rewrite the above serial for loop using the TBB `parallel_for` template as follows:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

The first two parameters specify that the iteration space is from 0 to $n-1$ (which corresponds to the number of elements in the array v). The second parameter is a C++ lambda function that requires a bit of explanation. The expression `[=](size_t i)` is the parameter i, which assumes each of the values over the iteration space (in this case from 0 to n − 1). Each value of i is used to identify which array element in v is to be passed as a parameter to the `apply(v[i])` function.

The TBB library will divide the loop iterations into separate "chunks" and create a number of tasks that operate on those chunks. (The `parallel_for` function allows developers to manually specify the size of the chunks if they wish to.) TBB will also create a number of threads and assign tasks to available threads. This is quite similar to the fork-join library in Java. The advantage of this approach is that it requires only that developers identify what operations can run in parallel (by specifying a `parallel_for` loop), and the library man-

ages the details involved in dividing the work into separate tasks that run in parallel. Intel TBB has both commercial and open-source versions that run on Windows, Linux, and macOS. Refer to the bibliography for further details on how to develop parallel applications using TBB.

4.6   Threading Issues

In this section, we discuss some of the issues to consider in designing multi-threaded programs.

4.6.1   The fork() and exec() System Calls

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

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

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

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

4.6.2   Signal Handling

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

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

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

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

having a timer expire. Typically, an asynchronous signal is sent to another process.

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

1. A default signal handler
2. A user-defined signal handler

Every signal has a **default signal handler** that the kernel runs when handling that signal. This default action can be overridden by a **user-define signal handler** that is called to handle the signal. Signals are handled in different ways. Some signals may be ignored, while others (for example, an illegal memory access) are handled by terminating the program.

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

In general, the following options exist:

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

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

The standard UNIX function for delivering a signal is

```
kill(pid_t pid, int signal)
```

This function specifies the process (`pid`) to which a particular signal (`signal`) is to be delivered. Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block. Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it. However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it. POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (`tid`):

```
pthread_kill(pthread_t tid, int signal)
```

Although Windows does not explicitly provide support for signals, it allows us to emulate them using **asynchronous procedure calls** (**APCs**). The APC facility enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event. As indicated

by its name, an APC is roughly equivalent to an asynchronous signal in UNIX. However, whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward, since an APC is delivered to a particular thread rather than a process.

4.6.3 Thread Cancellation

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

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

1. **Asynchronous cancellation**. One thread immediately terminates the target thread.

2. **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

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

With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely.

In Pthreads, thread cancellation is initiated using the pthread_cancel() function. The identifier of the target thread is passed as a parameter to the function. The following code illustrates creating—and then canceling—a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

 . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Invoking `pthread_cancel()`indicates only a request to cancel the target thread, however; actual cancellation depends on how the target thread is set up to handle the request. When the target thread is finally canceled, the call to `pthread_join()` in the canceling thread returns. Pthreads supports three cancellation modes. Each mode is defined as a state and a type, as illustrated in the table below. A thread may set its cancellation state and type using an API.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

As the table illustrates, Pthreads allows threads to disable or enable cancellation. Obviously, a thread cannot be canceled if cancellation is disabled. However, cancellation requests remain pending, so the thread can later enable cancellation and respond to the request.

The default cancellation type is deferred cancellation. However, cancellation occurs only when a thread reaches a **cancellation point**. Most of the blocking system calls in the POSIX and standard C library are defined as cancellation points, and these are listed when invoking the command `man pthreads` on a Linux system. For example, the `read()` system call is a cancellation point that allows cancelling a thread that is blocked while awaiting input from `read()`.

One technique for establishing a cancellation point is to invoke the `pthread_testcancel()` function. If a cancellation request is found to be pending, the call to `pthread_testcancel()` will not return, and the thread will terminate; otherwise, the call to the function will return, and the thread will continue to run. Additionally, Pthreads allows a function known as a **cleanup handler** to be invoked if a thread is canceled. This function allows any resources a thread may have acquired to be released before the thread is terminated.

The following code illustrates how a thread may respond to a cancellation request using deferred cancellation:

```
while (1) {
  /* do some work for awhile */

   . . .

  /* check if there is a cancellation request */
  pthread_testcancel();
}
```

Because of the issues described earlier, asynchronous cancellation is not recommended in Pthreads documentation. Thus, we do not cover it here. An interesting note is that on Linux systems, thread cancellation using the Pthreads API is handled through signals (Section 4.6.2).

Thread cancellation in Java uses a policy similar to deferred cancellation in Pthreads. To cancel a Java thread, you invoke the `interrupt()` method, which sets the interruption status of the target thread to true:

```
Thread worker;

 . . .

/* set the interruption status of the thread */
worker.interrupt()
```

A thread can check its interruption status by invoking the isInter-rupted() method, which returns a boolean value of a thread's interruption status:

```
while (!Thread.currentThread().isInterrupted()) {
   . . .
}
```

4.6.4 Thread-Local Storage

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

It is easy to confuse TLS with local variables. However, local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations. Additionally, when the developer has no control over the thread creation process—for example, when using an implicit technique such as a thread pool—then an alternative approach is necessary.

In some ways, TLS is similar to static data; the difference is that TLS data are unique to each thread. (In fact, TLS is usually declared as static.) Most thread libraries and compilers provide support for TLS. For example, Java provides a ThreadLocal<T> class with set() and get() methods for ThreadLocal<T> objects. Pthreads includes the type pthread_key_t, which provides a key that is specific to each thread. This key can then be used to access TLS data. Microsoft's C# language simply requires adding the storage attribute [ThreadStatic] to declare thread-local data. The gcc compiler provides the storage class keyword __thread for declaring TLS data. For example, if we wished to assign a unique identifier for each thread, we would declare it as follows:

```
static __thread int threadID;
```

4.6.5 Scheduler Activations

A final issue to be considered with multithreaded programs concerns commu-nication between the kernel and the thread library, which may be required

by the many-to-many and two-level models discussed in Section 4.3.3. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a **lightweight process**, or **LWP**—is shown in Figure 4.20. To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.

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

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor.

One event that triggers an upcall occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the
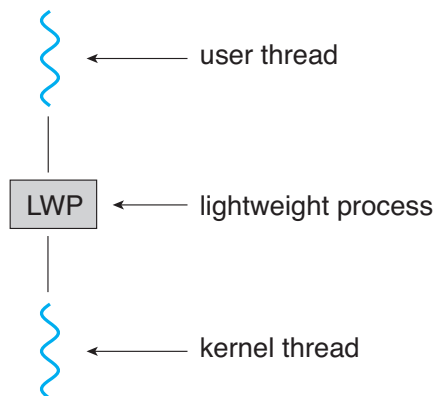

Figure 4.20   Lightweight process (LWP).

state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor. After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

4.7 Operating-System Examples

At this point, we have examined a number of concepts and issues related to threads. We conclude the chapter by exploring how threads are implemented in Windows and Linux systems.

4.7.1 Windows Threads

A Windows application runs as a separate process, and each process may contain one or more threads. The Windows API for creating threads is covered in Section 4.4.2. Additionally, Windows uses the one-to-one mapping described in Section 4.3.2, where each user-level thread maps to an associated kernel thread.

The general components of a thread include:

- A thread ID uniquely identifying the thread

- A register set representing the status of the processor

- A program counter

- A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode

- A private storage area used by various run-time libraries and dynamic link libraries (DLLs)

The register set, stacks, and private storage area are known as the **context** of the thread.

The primary data structures of a thread include:

- ETHREAD—executive thread block

- KTHREAD—kernel thread block

- TEB—thread environment block

The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.
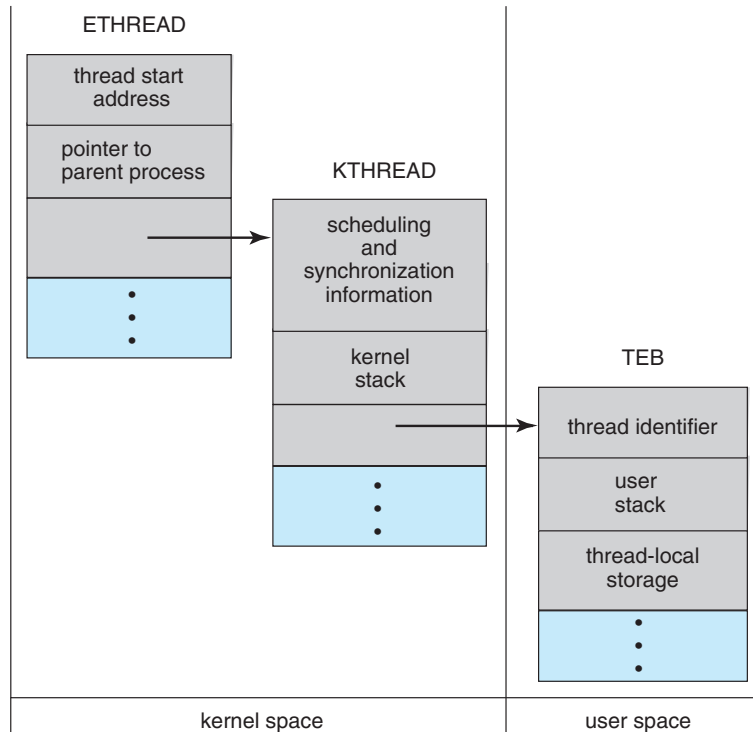
Figure 4.21   Data structures of a Windows thread.

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

The ETHREAD and the KTHREAD exist entirely in kernel space; this means that only the kernel can access them. The TEB is a user-space data structure that is accessed when the thread is running in user mode. Among other fields, the TEB contains the thread identifier, a user-mode stack, and an array for thread-local storage. The structure of a Windows thread is illustrated in Figure 4.21.

4.7.2 Linux Threads

Linux provides the `fork()` system call with the traditional functionality of duplicating a process, as described in Chapter 3. Linux also provides the ability to create threads using the `clone()` system call. However, Linux does not distinguish between processes and threads. In fact, Linux uses the term *task* —rather than *process* or *thread*— when referring to a flow of control within a program.

When `clone()` is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks. Some of these flags are listed in Figure 4.22. For example, suppose that `clone()` is passed the flags CLONE_FS, CLONE_VM, CLONE_SIGHAND, and CLONE_FILES. The parent and child tasks will then share the same file-system information (such as the current working directory), the same memory space, the same signal handlers,

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

Figure 4.22   Some of the flags passed when `clone()` is invoked.

and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread as described in this chapter, since the parent task shares most of its resources with its child task. However, if none of these flags is set when `clone()` is invoked, no sharing takes place, resulting in functionality similar to that provided by the `fork()` system call.

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

Finally, the flexibility of the `clone()` system call can be extended to the concept of containers, a virtualization topic which was introduced in Chapter 1. Recall from that chapter that a container is a virtualization technique provided by the operating system that allows creating multiple Linux systems (containers) under a single Linux kernel that run in isolation to one another. Just as certain flags passed to `clone()` can distinguish between creating a task that behaves more like a process or a thread based upon the amount of sharing between the parent and child tasks, there are other flags that can be passed to `clone()` that allow a Linux container to be created. Containers will be covered more fully in Chapter 18.

4.8   Summary

- A thread represents a basic unit of CPU utilization, and threads belonging to the same process share many of the process resources, including code and data.

- There are four primary benefits to multithreaded applications: (1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.

- Concurrency exists when multiple threads are making progress, whereas parallelism exists when multiple threads are making progress simulta-

neously. On a system with a single CPU, only concurrency is possible; parallelism requires a multicore system that provides multiple CPUs.

- There are several challenges in designing multithreaded applications. They include dividing and balancing the work, dividing the data between the different threads, and identifying any data dependencies. Finally, multithreaded programs are especially challenging to test and debug.

- Data parallelism distributes subsets of the same data across different computing cores and performs the same operation on each core. Task parallelism distributes not data but tasks across multiple cores. Each task is running a unique operation.

- User applications create user-level threads, which must ultimately be mapped to kernel threads to execute on a CPU. The many-to-one model maps many user-level threads to one kernel thread. Other approaches include the one-to-one and many-to-many models.

- A thread library provides an API for creating and managing threads. Three common thread libraries include Windows, Pthreads, and Java threading. Windows is for the Windows system only, while Pthreads is available for POSIX-compatible systems such as UNIX, Linux, and macOS. Java threads will run on any system that supports a Java virtual machine.

- Implicit threading involves identifying tasks—not threads—and allowing languages or API frameworks to create and manage threads. There are several approaches to implicit threading, including thread pools, fork-join frameworks, and Grand Central Dispatch. Implicit threading is becoming an increasingly common technique for programmers to use in developing concurrent and parallel applications.

- Threads may be terminated using either asynchronous or deferred cancellation. Asynchronous cancellation stops a thread immediately, even if it is in the middle of performing an update. Deferred cancellation informs a thread that it should terminate but allows the thread to terminate in an orderly fashion. In most circumstances, deferred cancellation is preferred to asynchronous termination.

- Unlike many other operating systems, Linux does not distinguish between processes and threads; instead, it refers to each as a task. The Linux `clone()` system call can be used to create tasks that behave either more like processes or more like threads.

Practice Exercises

4.1   Provide three programming examples in which multithreading provides better performance than a single-threaded solution.

4.2   Using Amdahl's Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.

4.3 Does the multithreaded web server described in Section 4.1 exhibit task or data parallelism?

4.4 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

4.5 Describe the actions taken by a kernel to context-switch between kernel-level threads.

4.6 What resources are used when a thread is created? How do they differ from those used when a process is created?

4.7 Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.

Further Reading

[Vahalia (1996)] covers threading in several versions of UNIX. [McDougall and Mauro (2007)] describes developments in threading the Solaris kernel. [Russinovich et al. (2017)] discuss threading in the Windows operating system family. [Mauerer (2008)] and [Love (2010)] explain how Linux handles threading, and [Levin (2013)] covers threads in macOS and iOS. [Herlihy and Shavit (2012)] covers parallelism issues on multicore systems. [Aubanel (2017)] covers parallelism of several different algorithms.

Bibliography

[Aubanel (2017)] E. Aubanel, *Elements of Parallel Computing*, CRC Press (2017).

[Herlihy and Shavit (2012)] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming,* Revised First Edition, Morgan Kaufmann Publishers Inc. (2012).

[Levin (2013)] J. Levin, *Mac OS X and iOS Internals to the Apple's Core*, Wiley (2013).

[Love (2010)] R. Love, *Linux Kernel Development,* Third Edition, Developer's Library (2010).

[Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

[McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals,* Second Edition, Prentice Hall (2007).

[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals−Part 1,* Seventh Edition, Microsoft Press (2017).

[Vahalia (1996)] U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall (1996).

Chapter 4 Exercises

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

4.9 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

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

   a.   Register values

   b.   Heap memory

   c.   Global variables

   d.   Stack memory

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

4.12 In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new tab in a separate process. Would the same benefits have been achieved if, instead, Chrome had been designed to open each new tab in a separate thread? Explain.

4.13 Is it possible to have concurrency but not parallelism? Explain.

4.14 Using Amdahl's Law, calculate the speedup gain for the following applications:

   • 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores

   • 67 percent parallel with (a) two processing cores and (b) four processing cores

   • 90 percent parallel with (a) four processing cores and (b) eight processing cores

4.15 Determine if the following problems exhibit task or data parallelism:

   • Using a separate thread to generate a thumbnail for each photo in a collection

   • Transposing a matrix in parallel

   • A networked application where one thread reads from the network and another writes to the network

   • The fork-join array summation application described in Section 4.5.2

   • The Grand Central Dispatch system

4.16 A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be

opened. Similarly, all output is performed just before the program termi-
nates, when the program results must be written to a single file. Between
start-up and termination, the program is entirely CPU-bound. Your task
is to improve the performance of this application by multithreading it.
The application runs on a system that uses the one-to-one threading
model (each user thread maps to a kernel thread).

- How many threads will you create to perform the input and output?
  Explain.

- How many threads will you create for the CPU-intensive portion of
  the application? Explain.

4.17 Consider the following code segment:

```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
    fork();
    thread_create( . . .);
}
fork();
```

    a.   How many unique processes are created?

    b.   How many unique threads are created?

4.18 As described in Section 4.7.2, Linux does not distinguish between pro-
cesses and threads. Instead, Linux treats both in the same way, allowing a
task to be more akin to a process or a thread depending on the set of flags
passed to the `clone()` system call. However, other operating systems,
such as Windows, treat processes and threads differently. Typically, such
systems use a notation in which the data structure for a process contains
pointers to the separate threads belonging to the process. Contrast these
two approaches for modeling processes and threads within the kernel.

4.19 The program shown in Figure 4.23 uses the Pthreads API. What would
be the output from the program at LINE C and LINE P?

4.20 Consider a multicore system and a multithreaded program written
using the many-to-many threading model. Let the number of user-level
threads in the program be greater than the number of processing cores
in the system. Discuss the performance implications of the following
scenarios.

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

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

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

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

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

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

   pid = fork();

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

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

Figure 4.22   C program for Exercise 4.19.

4.21 Pthreads provides an API for managing thread cancellation. The pthread_setcancelstate() function is used to set the cancellation state. Its prototype appears as follows:

pthread_setcancelstate(int state, int *oldstate)

The two possible values for the state are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE.

   Using the code segment shown in Figure 4.24, provide examples of two operations that would be suitable to perform between the calls to disable and enable thread cancellation.

```
int oldstate;

pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);

/* What operations would be performed here? */

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

Figure 4.23   C program for Exercise 4.21.

Programming Problems

4.22 Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

90 81 78 95 79 72 85

The program will report

```
The average value is 82
The minimum value is 72
The maximum value is 95
```

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited. (We could obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

4.23 Write a multithreaded program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

4.24 An interesting way of calculating π is to use a technique known as *Monte Carlo,* which involves randomization. This technique works as follows:
    Suppose you have a circle inscribed within a square, as shown in Figure 4.25. (Assume that the radius of this circle is 1.)

- First, generate a series of random points as simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle.

- Next, estimate π by performing the following calculation:

$$\pi = 4 \times \textit{(number of points in circle) / (total number of points)}$$

Write a multithreaded version of this algorithm that creates a separate thread to generate a number of random points. The thread will count the number of points that occur within the circle and store that result in a global variable. When this thread has exited, the parent thread will calculate and output the estimated value of π. It is worth experimenting with the number of random points generated. As a general rule, the greater the number of points, the closer the approximation to π.
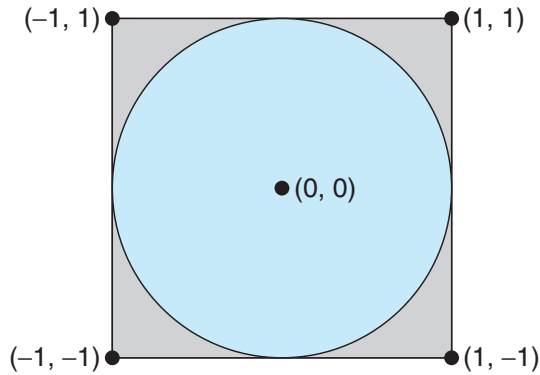
Figure 4.25   Monte Carlo technique for calculating π.

In the source-code download for this text, you will find a sample program that provides a technique for generating random numbers, as well as determining if the random (x, y) point occurs within the circle.

Readers interested in the details of the Monte Carlo method for estimating π should consult the bibliography at the end of this chapter. In Chapter 6, we modify this exercise using relevant material from that chapter.

4.25 Repeat Exercise 4.24, but instead of using a separate thread to generate random points, use OpenMP to parallelize the generation of points. Be careful not to place the calculation of π in the parallel region, since you want to calculate π only once.

4.26 Modify the socket-based date server (Figure 3.27) in Chapter 3 so that the server services each client request in a separate thread.

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

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

Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish. Use the techniques described in Section 4.4 to meet this requirement.

4.28 Modify programming problem Exercise 3.20 from Chapter 3, which asks you to design a pid manager. This modification will consist of writing a

multithreaded program that tests your solution to Exercise 3.20. You will create a number of threads—for example, 100—and each thread will request a pid, sleep for a random period of time, and then release the pid. (Sleeping for a random period of time approximates the typical pid usage in which a pid is assigned to a new process, the process executes and then terminates, and the pid is released on the process's termination.) On UNIX and Linux systems, sleeping is accomplished through the sleep() function, which is passed an integer value representing the number of seconds to sleep. This problem will be modified in Chapter 7.

4.29 Exercise 3.25 in Chapter 3 involves designing an echo server using the Java threading API. This server is single-threaded, meaning that the server cannot respond to concurrent echo clients until the current client exits. Modify the solution to Exercise 3.25 so that the echo server services each client in a separate request.

Programming Projects

Project 1—Sudoku Solution Validator

A *Sudoku* puzzle uses a 9×9 grid in which each column and row, as well as each of the nine 3×3 subgrids, must contain all of the digits $1 \cdots 9$. Figure 4.26 presents an example of a valid Sudoku puzzle. This project consists of designing a multithreaded application that determines whether the solution to a Sudoku puzzle is valid.

There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria:

- A thread to check that each column contains the digits 1 through 9

- A thread to check that each row contains the digits 1 through 9

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figure 4.26   Solution to a 9×9 Sudoku puzzle.

- Nine threads to check that each of the 3×3 subgrids contains the digits 1 through 9

This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome to create even more threads for this project. For example, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column.

I. Passing Parameters to Each Thread

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. The easiest approach is to create a data structure using a `struct`. For example, a structure to pass the row and column where a thread must begin validating would appear as follows:

```
/* structure for passing data to threads */
typedef struct
{
   int row;
   int column;
} parameters;
```

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

```
parameters *data = (parameters *) malloc(sizeof(parameters));
data->row = 1;
data->column = 1;
/* Now create the thread passing it data as a parameter */
```

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

II. Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The ith index in this array corresponds to the ith worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 indicates otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

Project 2—Multithreaded Sorting Application

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we

will term *sorting threads*) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread—a *merging thread* —which merges the two sublists into a single sorted list.

Because global data are shared across all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured as in Figure 4.27.

This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting. Refer to the instructions in Project 1 for details on passing parameters to a thread.

The parent thread will output the sorted array once all sorting threads have exited.

Project 3—Fork-Join Sorting Application

Implement the preceding project (Multithreaded Sorting Application) using Java's fork-join parallelism API. This project will be developed in two different versions. Each version will implement a different divide-and-conquer sorting algorithm:

1. Quicksort

2. Mergesort

The Quicksort implementation will use the Quicksort algorithm for dividing the list of elements to be sorted into a *left half* and a *right half* based on the
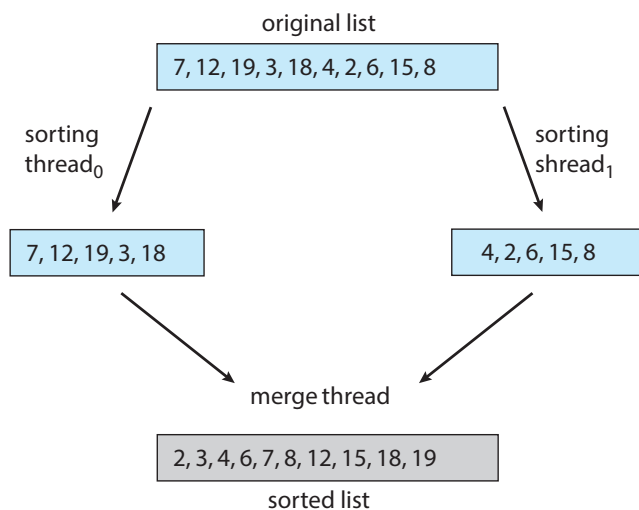
original list

7, 12, 19, 3, 18, 4, 2, 6, 15, 8

sorting
thread$_0$

sorting
shread$_1$

7, 12, 19, 3, 18

4, 2, 6, 15, 8

merge thread

2, 3, 4, 6, 7, 8, 12, 15, 18, 19

sorted list

Figure 4.27   Multithreaded sorting.

position of the pivot value. The Mergesort algorithm will divide the list into two evenly sized halves. For both the Quicksort and Mergesort algorithms, when the list to be sorted falls within some threshold value (for example, the list is size 100 or fewer), directly apply a simple algorithm such as the Selection or Insertion sort. Most data structures texts describe these two well-known, divide-and-conquer sorting algorithms.

The class `SumTask` shown in Section 4.5.2.1 extends `RecursiveTask`, which is a result-bearing `ForkJoinTask`. As this assignment will involve sorting the array that is passed to the task, but not returning any values, you will instead create a class that extends `RecursiveAction`, a non result-bearing `ForkJoinTask` (see Figure 4.19).

The objects passed to each sorting algorithm are required to implement Java's `Comparable` interface, and this will need to be reflected in the class definition for each sorting algorithm. The source code download for this text includes Java code that provides the foundations for beginning this project.

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms, including real-time systems. We also consider the problem of selecting an algorithm for a particular system.

In Chapter 4, we introduced threads to the process model. On modern operating systems it is kernel-level threads—not processes—that are in fact being scheduled by the operating system. However, the terms "process scheduling" and "thread scheduling" are often used interchangeably. In this chapter, we use *process scheduling* when discussing general scheduling concepts and *thread scheduling* to refer to thread-specific ideas.

Similarly, in Chapter 1 we describe how a *core* is the basic computational unit of a CPU, and that a process executes on a CPU's core. However, in many instances in this chapter, when we use the general terminology of scheduling a process to "run on a CPU", we are implying that the process is running on a CPU's core.

CHAPTER OBJECTIVES

- Describe various CPU scheduling algorithms.
- Assess CPU scheduling algorithms based on scheduling criteria.
- Explain the issues related to multiprocessor and multicore scheduling.
- Describe various real-time scheduling algorithms.
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems.
- Apply modeling and simulations to evaluate CPU scheduling algorithms.
- Design a program that implements several different CPU scheduling algorithms.

5.1   Basic Concepts

In a system with a single CPU core, only one process can run at a time. Others must wait until the CPU's core is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.
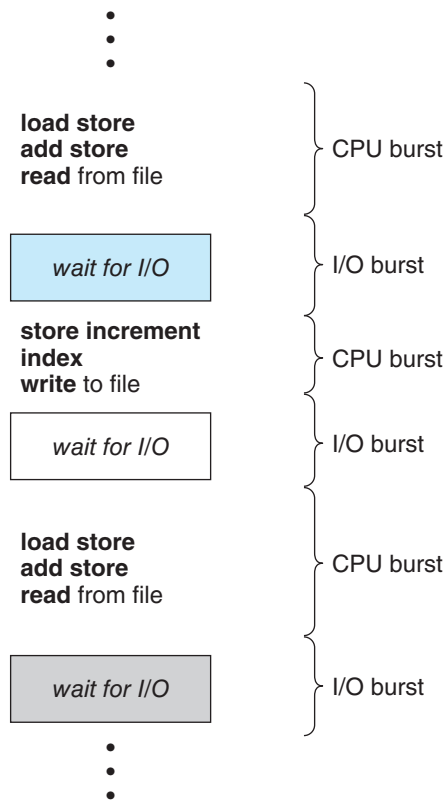

Figure 5.1   Alternating sequence of CPU and I/O bursts.

5.1.1 CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 5.1).

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 5.2. The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts. An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important when implementing a CPU-scheduling algorithm.

5.1.2 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **CPU scheduler**, which selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.
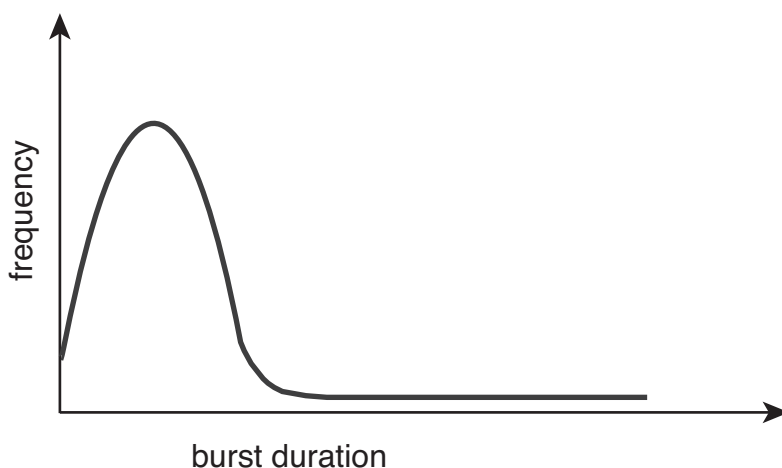

Figure 5.2 Histogram of CPU-burst durations.

5.1.3 Preemptive and Nonpreemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of `wait()` for the termination of a child process)

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)

3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)

4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state. Virtually all modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.

Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. This issue will be explored in detail in Chapter 6.

Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues. As will be discussed in Section 6.2, operating-system kernels can be designed as either nonpreemptive or preemptive. A nonpreemptive kernel will wait for a system call to complete or for a process to block while waiting for I/O to complete to take place before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting real-time computing, where tasks must complete execution within a given time frame. In Section 5.6, we explore scheduling demands of real-time systems. A preemptive kernel requires mechanisms such as mutex locks to prevent race conditions when accessing shared kernel data structures. Most modern operating systems are now fully preemptive when running in kernel mode.
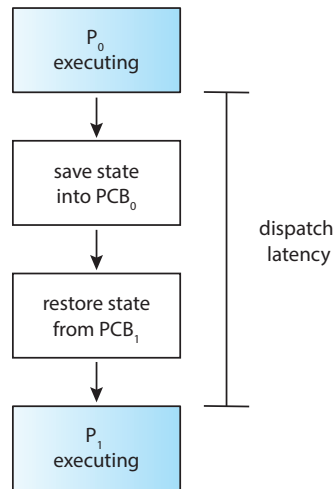
Figure 5.3 The role of the dispatcher.

Because interrupts can, by definition, occur at any time, and because they cannot always be ignored by the kernel, the sections of code affected by interrupts must be guarded from simultaneous use. The operating system needs to accept interrupts at almost all times. Otherwise, input might be lost or output overwritten. So that these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit. It is important to note that sections of code that disable interrupts do not occur very often and typically contain few instructions.

5.1.4 Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler. This function involves the following:

- Switching context from one process to another
- Switching to user mode
- Jumping to the proper location in the user program to resume that program

The dispatcher should be as fast as possible, since it is invoked during every context switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency** and is illustrated in Figure 5.3.

An interesting question to consider is, how often do context switches occur? On a system-wide level, the number of context switches can be obtained by using the vmstat command that is available on Linux systems. Below is the output (which has been trimmed) from the command

```
vmstat 1 3
```

This command provides 3 lines of output over a 1-second delay:

```
------cpu-----
24
225
339
```

The first line gives the average number of context switches over 1 second since the system booted, and the next two lines give the number of context switches over two 1-second intervals. Since this machine booted, it has averaged 24 context switches per second. And in the past second, 225 context switches were made, with 339 context switches in the second prior to that.

We can also use the /proc file system to determine the number of context switches for a given process. For example, the contents of the file /proc/2166/status will list various statistics for the process with pid = 2166. The command

```
cat /proc/2166/status
```

provides the following trimmed output:

```
voluntary_ctxt_switches          150
nonvoluntary_ctxt_switches       8
```

This output shows the number of context switches over the lifetime of the process. Notice the distinction between *voluntary* and *nonvoluntary* context switches. A voluntary context switch occurs when a process has given up control of the CPU because it requires a resource that is currently unavailable (such as blocking for I/O.) A nonvoluntary context switch occurs when the CPU has been taken away from a process, such as when its time slice has expired or it has been preempted by a higher-priority process.

5.2 Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system). (CPU utilization can be obtained by using the top command on Linux, macOS, and UNIX systems.)

- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed

per time unit, called throughput. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

- **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.

- **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

- **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Investigators have suggested that, for interactive systems (such as a PC desktop or laptop system), it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance.

As we discuss various CPU-scheduling algorithms in the following section, we illustrate their operation. An accurate illustration should involve many processes, each a sequence of several hundred CPU bursts and I/O bursts. For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples. Our measure of comparison is the average waiting time. More elaborate evaluation mechanisms are discussed in Section 5.8.

5.3 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core. There are many different CPU-scheduling algorithms. In this section, we describe several of them. Although most modern CPU architectures have multiple processing cores, we describe these scheduling algorithms in the context of only one processing core available. That is, a single CPU that has a single processing core, thus the system is

capable of only running one process at a time. In Section 5.5 we discuss CPU scheduling in the context of multiprocessor systems.

5.3.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come first-serve** (**FCFS**) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:

P_1	P_2	P_3
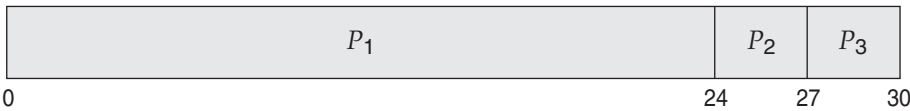
0   24   27   30

The waiting time is 0 milliseconds for process P_1, 24 milliseconds for process P_2, and 27 milliseconds for process P_3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P_2, P_3, P_1, however, the results will be as shown in the following Gantt chart:

P_2	P_3	P_1

0   3   6   30
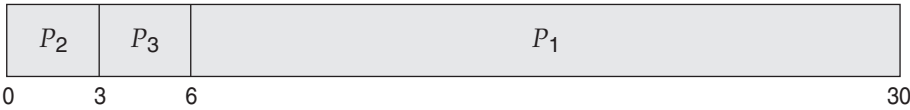
The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O

devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Note also that the FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

5.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the **shortest-job-firs** (**SJF**) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the *shortest-next-CPU-burst* algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:

P_4	P_1	P_3	P_2

0   3   9   16   24
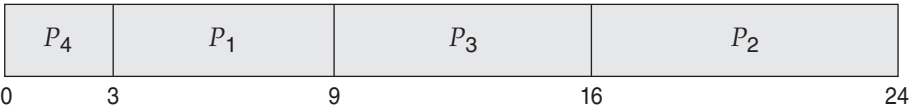
The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process

before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts. We can define the exponential average with the following formula. Let t_n be the length of the nth CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

The value of t_n contains our most recent information, while τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient). If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. The initial τ_0 can be defined as a constant or as an overall system average. Figure 5.4 shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.

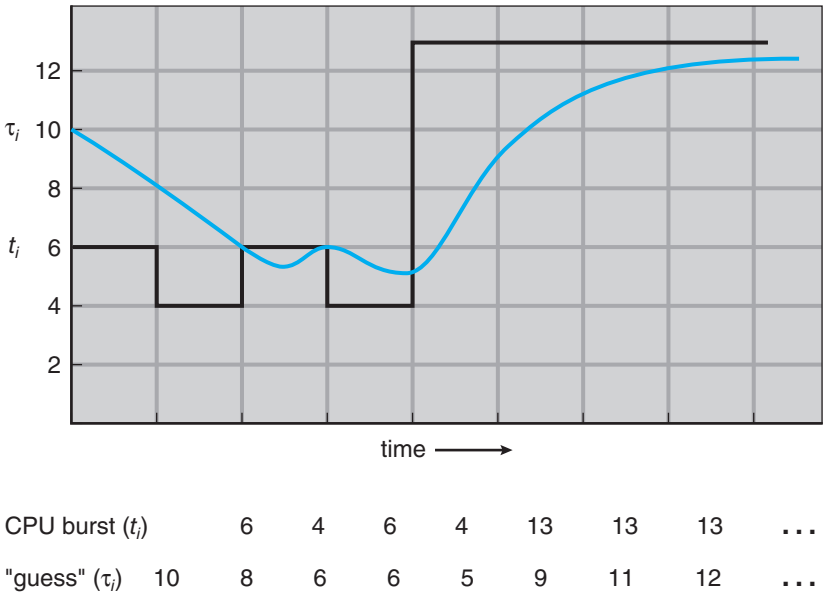| CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Figure 5.4 Prediction of the length of the next CPU burst.

To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n to find

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0.$$

Typically, α is less than 1. As a result, $(1 - \alpha)$ is also less than 1, and each successive term has less weight than its predecessor.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-firs** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:

P_1	P_2	P_4	P_1	P_3
0 1	5	10	17	26
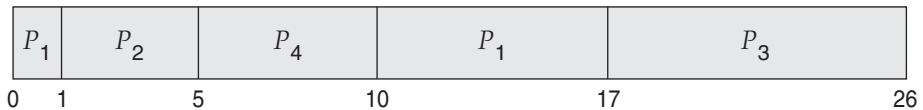
Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5$ milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

5.3.3 Round-Robin Scheduling

The **round-robin** (**RR**) scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.

The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2. Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is as follows:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

```
0       4     7    10        14        18        22        26        30
```

Let's calculate the average waiting time for this schedule. P_1 waits for 6 milliseconds (10 − 4), P_2 waits for 4 milliseconds, and P_3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n-1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large
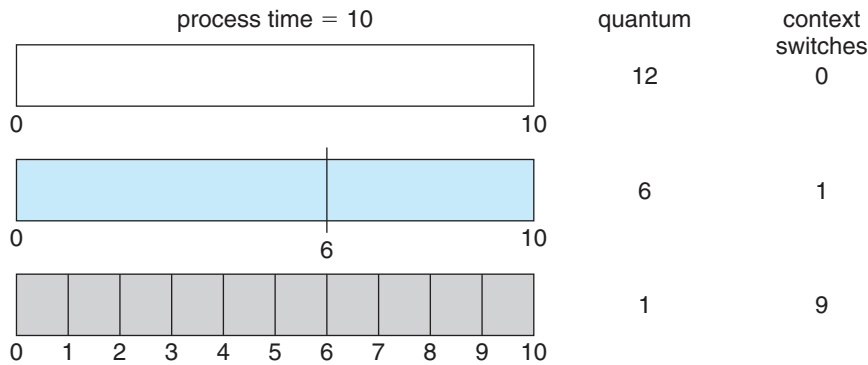
Figure 5.5 How a smaller time quantum increases context switches.

number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 5.5).

Thus, we want the time quantum to be large with respect to the context-switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

Turnaround time also depends on the size of the time quantum. As we can see from Figure 5.6, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context-switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

5.3.4 Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling** algorithm. A priority is associated with each process, and the CPU is allocated to the
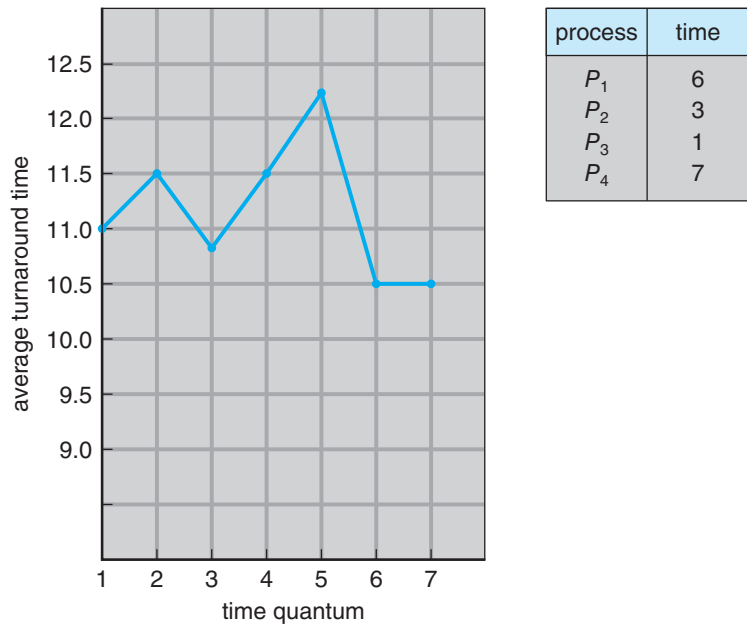
process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 5.6   How turnaround time varies with the time quantum.

process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of *high* priority and *low* priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P_1, P_2, $\cdots$, P_5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:

P_2	P_5	P_1	P_3	P_4

0   1   6   16   18 19
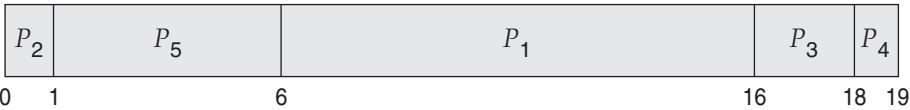
The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
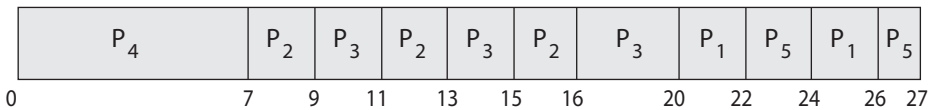
A major problem with priority scheduling algorithms is **indefinit block-ing**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes. (Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of a waiting process by 1. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take a little over 2 minutes for a priority-127 process to age to a priority-0 process.

Another option is to combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling. Let's illustrate with an example using the following set of processes, with the burst time in milliseconds:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Using priority scheduling with round-robin for processes with equal priority, we would schedule these processes according to the following Gantt chart using a time quantum of 2 milliseconds:

| P_4 | | | | P_2 | P_3 | P_2 | P_3 | P_2 | P_3 | | P_1 | P_5 | P_1 | P_5 |

```
0                         7   9   11  13  15  16      20  22  24  26 27
```

In this example, process P_4 has the highest priority, so it will run to completion. Processes P_2 and P_3 have the next-highest priority, and they will execute in a round-robin fashion. Notice that when process P_2 finishes at time 16, process P_3 is the highest-priority process, so it will run until it completes execution. Now, only processes P_1 and P_5 remain, and as they have equal priority, they will execute in round-robin order until they complete.

5.3.5 Multilevel Queue Scheduling

With both priority and round-robin scheduling, all processes may be placed in a single queue, and the scheduler then selects the process with the highest priority to run. Depending on how the queues are managed, an $O(n)$ search may be necessary to determine the highest-priority process. In practice, it is often easier to have separate queues for each distinct priority, and priority scheduling simply schedules the process in the highest-priority queue. This is illustrated in Figure 5.7. This approach—known as **multilevel queue**—also works well when priority scheduling is combined with round-robin: if there are multiple processes in the highest-priority queue, they are executed in round-robin order. In the most generalized form of this approach, a priority is assigned statically to each process, and a process remains in the same queue for the duration of its runtime.
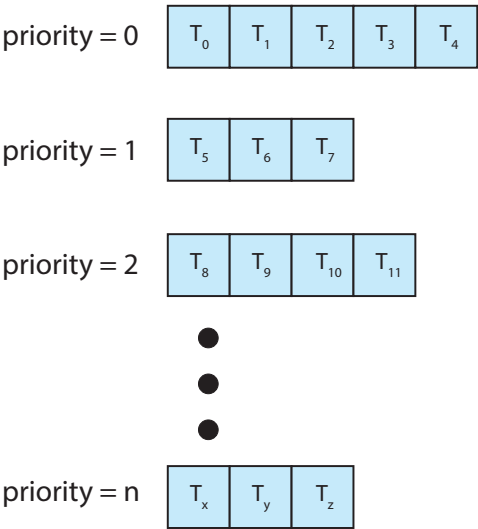

Figure 5.7 Separate queues for each priority.
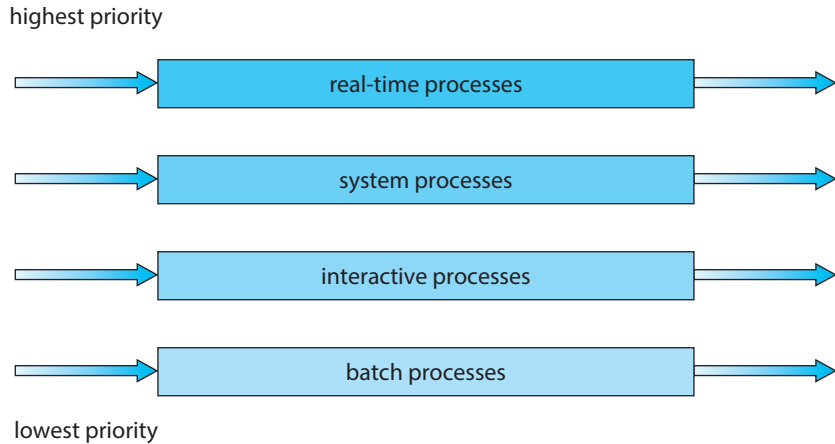
highest priority

lowest priority

Figure 5.8 Multilevel queue scheduling.

A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process type (Figure 5.8). For example, a common division is made between **foreground** (interactive) processes and **background** (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes. Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm. The foreground queue might be scheduled by an RR algorithm, for example, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the real-time queue may have absolute priority over the interactive queue.

Let's look at an example of a multilevel queue scheduling algorithm with four queues, listed below in order of priority:

1. Real-time processes
2. System processes
3. Interactive processes
4. Batch processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for real-time processes, system processes, and interactive processes were all empty. If an interactive process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling

among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

5.3.6 Multilevel Feedback Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes—which are typically characterized by short CPU bursts—in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 5.9). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

An entering process is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty. To prevent starvation, a process that waits too long in a lower-priority queue may gradually be moved to a higher-priority queue.
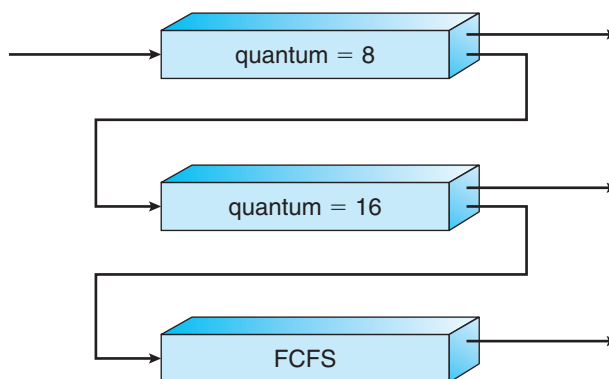

Figure 5.9 Multilevel feedback queues.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues

- The scheduling algorithm for each queue

- The method used to determine when to upgrade a process to a higher-priority queue

- The method used to determine when to demote a process to a lower-priority queue

- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

5.4   Thread Scheduling

In Chapter 4, we introduced threads to the process model, distinguishing between *user-level* and *kernel-level* threads. On most modern operating systems it is kernel-level threads—not processes—that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP). In this section, we explore scheduling issues involving user-level and kernel-level threads and offer specific examples of scheduling for Pthreads.

5.4.1   Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one (Section 4.3.1) and many-to-many (Section 4.3.3) models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as **process-contention scope** (**PCS**), since competition for the CPU takes place among threads belonging to the same process. (When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are actually *running* on a CPU as that further requires the operating system to schedule the LWP's kernel thread onto a physical CPU core.) To decide which

kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope** (SCS). Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model (Section 4.3.2), such as Windows and Linux schedule threads using only SCS.

Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing (Section 5.3.3) among threads of equal priority.

5.4.2   Pthread Scheduling

We provided a sample POSIX Pthread program in Section 4.4.1, along with an introduction to thread creation with Pthreads. Now, we highlight the POSIX Pthread API that allows specifying PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

- PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
- PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations (Section 4.6.5). The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

The Pthread IPC (Interprocess Communication) provides two functions for setting—and getting—the contention scope policy:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the `pthread_attr_setscope()` function is passed either the PTHREAD_SCOPE_SYSTEM or the PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set. In the case of `pthread_attr_getscope()`, this second parameter contains a pointer to an `int` value that is set to the current value of the contention scope. If an error occurs, each of these functions returns a nonzero value.

In Figure 5.10, we illustrate a Pthread scheduling API. The program first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM. It then creates five separate threads that will run using the SCS scheduling policy. Note that on some systems, only certain contention scope values are allowed. For example, Linux and macOS systems allow only PTHREAD_SCOPE_SYSTEM.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

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

  /* get the default attributes */
  pthread_attr_init(&attr);

  /* first inquire on the current scope */
  if (pthread_attr_getscope(&attr, &scope) != 0)
    fprintf(stderr, "Unable to get scheduling scope\n");
  else {
    if (scope == PTHREAD_SCOPE_PROCESS)
     printf("PTHREAD_SCOPE_PROCESS");
    else if (scope == PTHREAD_SCOPE_SYSTEM)
     printf("PTHREAD_SCOPE_SYSTEM");
    else
     fprintf(stderr, "Illegal scope value.\n");
   }

  /* set the scheduling algorithm to PCS or SCS */
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* create the threads */
  for (i = 0; i < NUM_THREADS; i++)
     pthread_create(&tid[i],&attr,runner,NULL);

  /* now join on each thread */
  for (i = 0; i < NUM_THREADS; i++)
     pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
  /* do some work ... */

  pthread_exit(0);
}
```

Figure 5.10   Pthread scheduling API.

5.5 Multi-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processing core. If multiple CPUs are available, **load sharing**, where multiple threads may run in parallel, becomes possible, however scheduling issues become correspondingly more complex. Many possibilities have been tried; and as we saw with CPU scheduling with a single-core CPU, there is no one best solution.

Traditionally, the term **multiprocessor** referred to systems that provided multiple physical processors, where each processor contained one single-core CPU. However, the definition of multiprocessor has evolved significantly, and on modern computing systems, *multiprocessor* now applies to the following system architectures:

- Multicore CPUs
- Multithreaded cores
- NUMA systems
- Heterogeneous multiprocessing

Here, we discuss several concerns in multiprocessor scheduling in the context of these different architectures. In the first three examples we concentrate on systems in which the processors are identical—homogeneous—in terms of their functionality. We can then use any available CPU to run any process in the queue. In the last example we explore a system where the processors are not identical in their capabilities.

5.5.1 Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor — the master server. The other processors execute only user code. This **asymmetric multiprocessing** is simple because only one core accesses the system data structures, reducing the need for data sharing. The downfall of this approach is the master server becomes a potential bottleneck where overall system performance may be reduced.

The standard approach for supporting multiprocessors is **symmetric multiprocessing** (**SMP**), where each processor is self-scheduling. Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a thread to run. Note that this provides two possible strategies for organizing the threads eligible to be scheduled:

1. All threads may be in a common ready queue.
2. Each processor may have its own private queue of threads.

These two strategies are contrasted in Figure 5.11. If we select the first option, we have a possible race condition on the shared ready queue and therefore must ensure that two separate processors do not choose to schedule the same thread and that threads are not lost from the queue. As discussed in
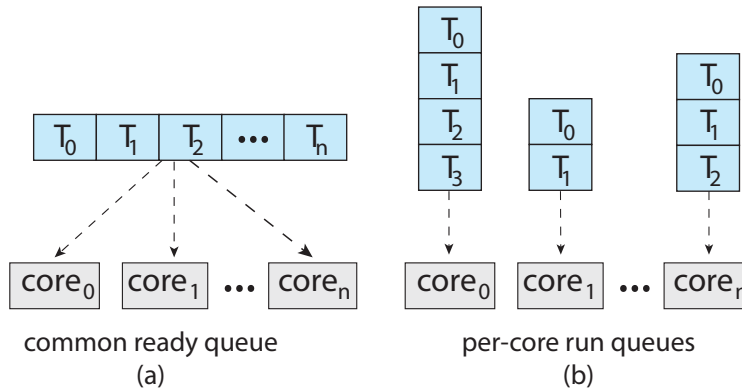
Figure 5.11 Organization of ready queues.

Chapter 6, we could use some form of locking to protect the common ready queue from this race condition. Locking would be highly contended, however, as all accesses to the queue would require lock ownership, and accessing the shared queue would likely be a performance bottleneck. The second option permits each processor to schedule threads from its private run queue and therefore does not suffer from the possible performance problems associated with a shared run queue. Thus, it is the most common approach on systems supporting SMP. Additionally, as described in Section 5.5.4, having private, per-processor run queues in fact may lead to more efficient use of cache memory. There are issues with per-processor run queues—most notably, workloads of varying sizes. However, as we shall see, balancing algorithms can be used to equalize workloads among all processors.

Virtually all modern operating systems support SMP, including Windows, Linux, and macOS as well as mobile systems including Android and iOS. In the remainder of this section, we discuss issues concerning SMP systems when designing CPU scheduling algorithms.

5.5.2 Multicore Processors

Traditionally, SMP systems have allowed several processes to run in parallel by providing multiple physical processors. However, most contemporary computer hardware now places multiple computing cores on the same physical chip, resulting in a **multicore processor**. Each core maintains its architectural state and thus appears to the operating system to be a separate logical CPU. SMP systems that use multicore processors are faster and consume less power than systems in which each CPU has its own physical chip.

Multicore processors may complicate scheduling issues. Let's consider how this can happen. Researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a **memory stall**, occurs primarily because modern processors operate at much faster speeds than memory. However, a memory stall can also occur because of a cache miss (accessing data that are not in cache memory). Figure 5.12 illustrates a memory stall. In this scenario, the processor can spend up to 50 percent of its time waiting for data to become available from memory.
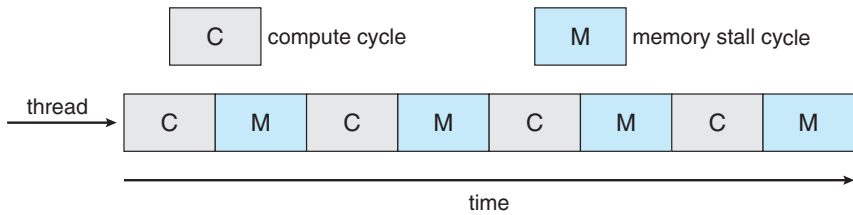
| C | compute cycle | M | memory stall cycle |

thread → | C | M | C | M | C | M | C | M |

time

Figure 5.12   Memory stall.

To remedy this situation, many recent hardware designs have implemented multithreaded processing cores in which two (or more) **hardware threads** are assigned to each core. That way, if one hardware thread stalls while waiting for memory, the core can switch to another thread. Figure 5.13 illustrates a dual-threaded processing core on which the execution of thread 0 and the execution of thread 1 are interleaved. From an operating system perspective, each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread. This technique—known as **chip multithreading** (CMT) —is illustrated in Figure 5.14. Here, the processor contains four computing cores, with each core containing two hardware threads. From the perspective of the operating system, there are eight logical CPUs.

Intel processors use the term **hyper-threading** (also known as **simultaneous multithreading** or SMT) to describe assigning multiple hardware threads to a single processing core. Contemporary Intel processors—such as the i7—support two threads per core, while the Oracle Sparc M7 processor supports eight threads per core, with eight cores per processor, thus providing the operating system with 64 logical CPUs.

In general, there are two ways to multithread a processing core: **coarse-grained** and **fine-graine** multithreading. With coarse-grained multithreading, a thread executes on a core until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the core must switch to another thread to begin execution. However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions. Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity—typically at the boundary of an instruction

thread$_1$ → | C | M | C | M | C | M | C |

thread$_0$ → | C | M | C | M | C | M | C |

time
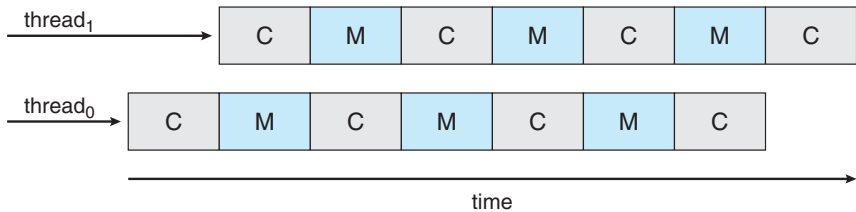
Figure 5.13   Multithreaded multicore system.
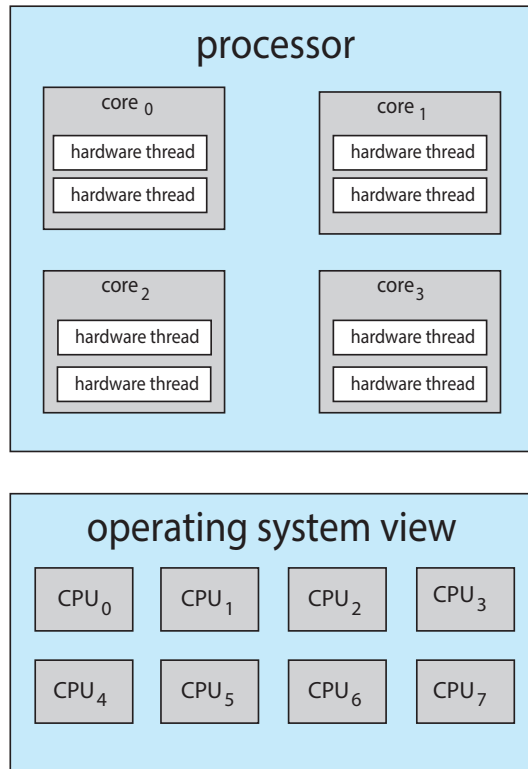
Figure 5.14   Chip multithreading.

cycle. However, the architectural design of fine-grained systems includes logic for thread switching. As a result, the cost of switching between threads is small.

It is important to note that the resources of the physical core (such as caches and pipelines) must be shared among its hardware threads, and therefore a processing core can only execute one hardware thread at a time. Consequently, a multithreaded, multicore processor actually requires two different levels of scheduling, as shown in Figure 5.15, which illustrates a dual-threaded processing core.

On one level are the scheduling decisions that must be made by the operating system as it chooses which software thread to run on each hardware thread (logical CPU). For all practical purposes, such decisions have been the primary focus of this chapter. Therefore, for this level of scheduling, the operating system may choose any scheduling algorithm, including those described in Section 5.3.

A second level of scheduling specifies how each core decides which hardware thread to run. There are several strategies to adopt in this situation. One approach is to use a simple round-robin algorithm to schedule a hardware thread to the processing core. This is the approach adopted by the UltraSPARC T3. Another approach is used by the Intel Itanium, a dual-core processor with two hardware-managed threads per core. Assigned to each hardware thread is a dynamic *urgency* value ranging from 0 to 7, with 0 representing the lowest urgency and 7 the highest. The Itanium identifies five different events that may

Figure 5.15   Two levels of scheduling.

trigger a thread switch. When one of these events occurs, the thread-switching logic compares the urgency of the two threads and selects the thread with the highest urgency value to execute on the processor core.

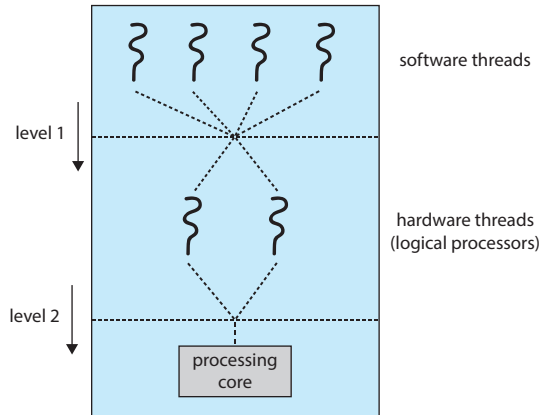Note that the two different levels of scheduling shown in Figure 5.15 are not necessarily mutually exclusive. In fact, if the operating system scheduler (the first level) is made aware of the sharing of processor resources, it can make more effective scheduling decisions. As an example, assume that a CPU has two processing cores, and each core has two hardware threads. If two software threads are running on this system, they can be running either on the same core or on separate cores. If they are both scheduled to run on the same core, they have to share processor resources and thus are likely to proceed more slowly than if they were scheduled on separate cores. If the operating system is aware of the level of processor resource sharing, it can schedule software threads onto logical processors that do not share resources.

5.5.3   Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads, along with ready queues of threads awaiting the CPU. **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private ready queue of eligible threads to execute. On systems with a common run queue, load balancing is unnecessary, because once a processor becomes idle, it immediately extracts a runnable thread from the common ready queue.

There are two general approaches to load balancing: push migration and pull migration. With **push migration**, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors. **Pull migration** occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are, in fact, often implemented in parallel on load-balancing systems. For

example, the Linux CFS scheduler (described in Section 5.7.1) and the ULE scheduler available for FreeBSD systems implement both techniques.

The concept of a "balanced load" may have different meanings. One view of a balanced load may require simply that all queues have approximately the same number of threads. Alternatively, balance may require an equal distribution of thread priorities across all queues. In addition, in certain situations, neither of these strategies may be sufficient. Indeed, they may work against the goals of the scheduling algorithm. (We leave further consideration of this as an exercise.)

5.5.4   Processor Affinity

Consider what happens to cache memory when a thread has been running on a specific processor. The data most recently accessed by the thread populate the cache for the processor. As a result, successive memory accesses by the thread are often satisfied in cache memory (known as a "warm cache"). Now consider what happens if the thread migrates to another processor—say, due to load balancing. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most operating systems with SMP support try to avoid migrating a thread from one processor to another and instead attempt to keep a thread running on the same processor and take advantage of a warm cache. This is known as **processor affinit** —that is, a process has an affinity for the processor on which it is currently running.

The two strategies described in Section 5.5.1 for organizing the queue of threads available for scheduling have implications for processor affinity. If we adopt the approach of a common ready queue, a thread may be selected for execution by any processor. Thus, if a thread is scheduled on a new processor, that processor's cache must be repopulated. With private, per-processor ready queues, a thread is always scheduled on the same processor and can therefore benefit from the contents of a warm cache. Essentially, per-processor ready queues provide processor affinity for free!

Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—we have a situation known as **soft affinit** . Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors during load balancing. In contrast, some systems provide system calls that support **hard affinit** , thereby allowing a process to specify a subset of processors on which it can run. Many systems provide both soft and hard affinity. For example, Linux implements soft affinity, but it also provides the sched_setaffinity() system call, which supports hard affinity by allowing a thread to specify the set of CPUs on which it is eligible to run.

The main-memory architecture of a system can affect processor affinity issues as well. Figure 5.16 illustrates an architecture featuring non-uniform memory access (NUMA) where there are two physical processor chips each with their own CPU and local memory. Although a system interconnect allows all CPUs in a NUMA system to share one physical address space, a CPU has faster access to its local memory than to memory local to another CPU. If the operating system's CPU scheduler and memory-placement algorithms are *NUMA-aware*
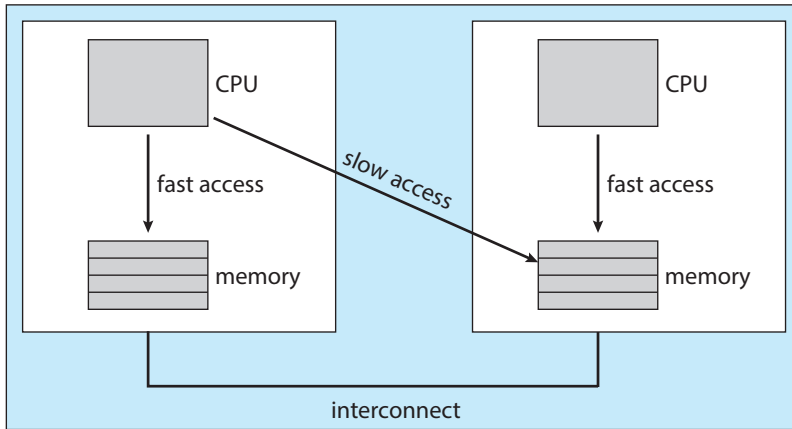
Figure 5.16   NUMA and CPU scheduling.

and work together, then a thread that has been scheduled onto a particular CPU can be allocated memory closest to where the CPU resides, thus providing the thread the fastest possible memory access.

Interestingly, load balancing often counteracts the benefits of processor affinity. That is, the benefit of keeping a thread running on the same processor is that the thread can take advantage of its data being in that processor's cache memory. Balancing loads by moving a thread from one processor to another removes this benefit. Similarly, migrating a thread between processors may incur a penalty on NUMA systems, where a thread may be moved to a processor that requires longer memory access times. In other words, there is a natural tension between load balancing and minimizing memory access times. Thus, scheduling algorithms for modern multicore NUMA systems have become quite complex. In Section 5.7.1, we examine the Linux CFS scheduling algorithm and explore how it balances these competing goals.

5.5.5   Heterogeneous Multiprocessing

In the examples we have discussed so far, all processors are identical in terms of their capabilities, thus allowing any thread to run on any processing core. The only difference being that memory access times may vary based upon load balancing and processor affinity policies, as well as on NUMA systems.

Although mobile systems now include multicore architectures, some systems are now designed using cores that run the same instruction set, yet vary in terms of their clock speed and power management, including the ability to adjust the power consumption of a core to the point of idling the core. Such systems are known as **heterogeneous multiprocessing** (HMP). Note this is not a form of asymmetric multiprocessing as described in Section 5.5.1 as both system and user tasks can run on any core. Rather, the intention behind HMP is to better manage power consumption by assigning tasks to certain cores based upon the specific demands of the task.

For ARM processors that support it, this type of architecture is known as **big.LITTLE** where higher-peformance *big* cores are combined with energy efficient *LITTLE* cores. *Big* cores consume greater energy and therefore should

only be used for short periods of time. Likewise, *little* cores use less energy and can therefore be used for longer periods.

There are several advantages to this approach. By combining a number of slower cores with faster ones, a CPU scheduler can assign tasks that do not require high performance, but may need to run for longer periods, (such as background tasks) to little cores, thereby helping to preserve a battery charge. Similarly, interactive applications which require more processing power, but may run for shorter durations, can be assigned to big cores. Additionally, if the mobile device is in a power-saving mode, energy-intensive big cores can be disabled and the system can rely solely on energy-efficient little cores. Windows 10 supports HMP scheduling by allowing a thread to select a scheduling policy that best supports its power management demands.

5.6 Real-Time CPU Scheduling

CPU scheduling for real-time operating systems involves special issues. In general, we can distinguish between soft real-time systems and hard real-time systems. **Soft real-time systems** provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes. **Hard real-time systems** have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all. In this section, we explore several issues related to process scheduling in both soft and hard real-time operating systems.

5.6.1 Minimizing Latency

Consider the event-driven nature of a real-time system. The system is typically waiting for an event in real time to occur. Events may arise either in software —as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction. When an event occurs, the system must respond to and service it as quickly as possible. We refer to **event latency** as the amount of time that elapses from when an event occurs to when it is serviced (Figure 5.17).

Figure 5.17 Event latency.

Usually, different events have different latency requirements. For example, the latency requirement for an antilock brake system might be 3 to 5 milliseconds. That is, from the time a wheel first detects that it is sliding, the system controlling the antilock brakes has 3 to 5 milliseconds to respond to and control the situation. Any response that takes longer might result in the automobile's veering out of control. In contrast, an embedded system controlling radar in an airliner might tolerate a latency period of several seconds.

Two types of latencies affect the performance of real-time systems:

1. Interrupt latency

2. Dispatch latency

Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR). The total time required to perform these tasks is the interrupt latency (Figure 5.18).

Obviously, it is crucial for real-time operating systems to minimize interrupt latency to ensure that real-time tasks receive immediate attention. Indeed, for hard real-time systems, interrupt latency must not simply be minimized, it must be bounded to meet the strict requirements of these systems.

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts be disabled for only very short periods of time.

The amount of time required for the scheduling dispatcher to stop one process and start another is known as **dispatch latency**. Providing real-time

Figure 5.18 Interrupt latency.
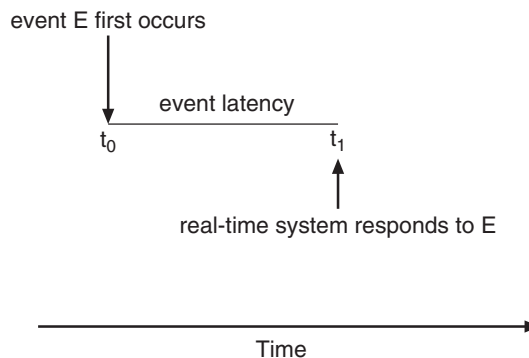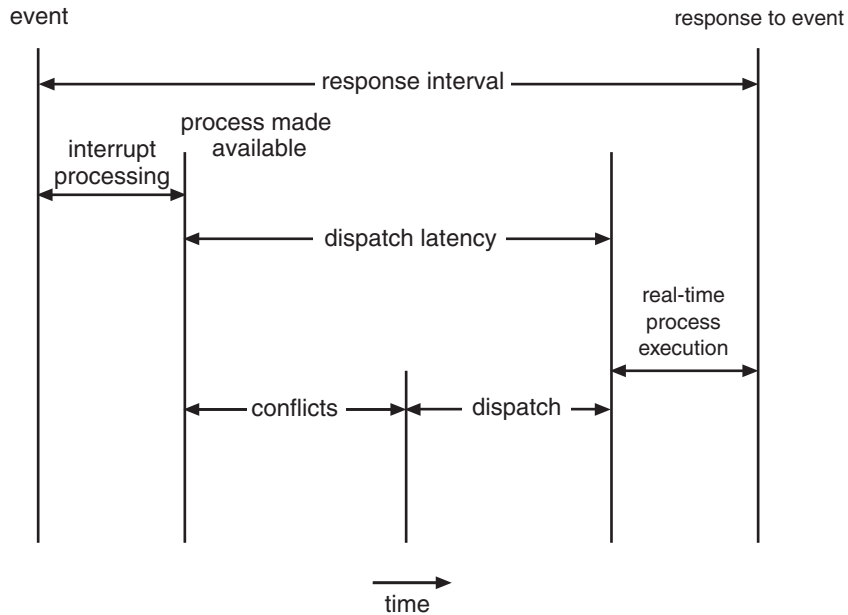
Figure 5.19 Dispatch latency.

tasks with immediate access to the CPU mandates that real-time operating systems minimize this latency as well. The most effective technique for keeping dispatch latency low is to provide preemptive kernels. For hard real-time systems, dispatch latency is typically measured in several microseconds.

In Figure 5.19, we diagram the makeup of dispatch latency. The **conflic phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel

2. Release by low-priority processes of resources needed by a high-priority process

Following the conflict phase, the dispatch phase schedules the high-priority process onto an available CPU.

5.6.2 Priority-Based Scheduling

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. As a result, the scheduler for a real-time operating system must support a priority-based algorithm with preemption. Recall that priority-based scheduling algorithms assign each process a priority based on its importance; more important tasks are assigned higher priorities than those deemed less important. If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run.

Preemptive, priority-based scheduling algorithms are discussed in detail in Section 5.3.4, and Section 5.7 presents examples of the soft real-time scheduling features of the Linux, Windows, and Solaris operating systems. Each of these systems assigns real-time processes the highest scheduling priority. For
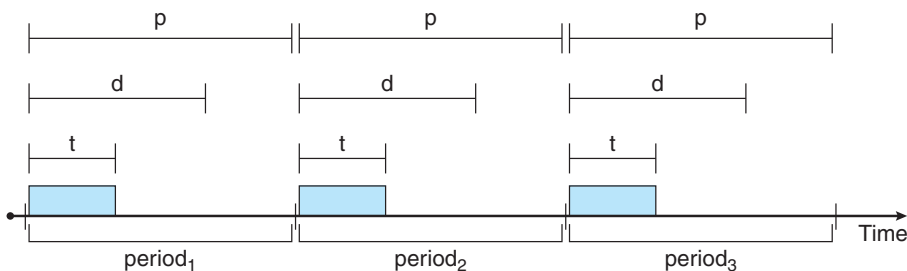
Figure 5.20   Periodic task.

example, Windows has 32 different priority levels. The highest levels—priority values 16 to 31—are reserved for real-time processes. Solaris and Linux have similar prioritization schemes.

Note that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees requires additional scheduling features. In the remainder of this section, we cover scheduling algorithms appropriate for hard real-time systems.

Before we proceed with the details of the individual schedulers, however, we must define certain characteristics of the processes that are to be scheduled. First, the processes are considered **periodic**. That is, they require the CPU at constant intervals (periods). Once a periodic process has acquired the CPU, it has a fixed processing time t, a deadline d by which it must be serviced by the CPU, and a period p. The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The **rate** of a periodic task is $1/p$. Figure 5.20 illustrates the execution of a periodic process over time. Schedulers can take advantage of these characteristics and assign priorities according to a process's deadline or rate requirements.

What is unusual about this form of scheduling is that a process may have to announce its deadline requirements to the scheduler. Then, using a technique known as an **admission-control** algorithm, the scheduler does one of two things. It either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

5.6.3   Rate-Monotonic Scheduling

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the process-
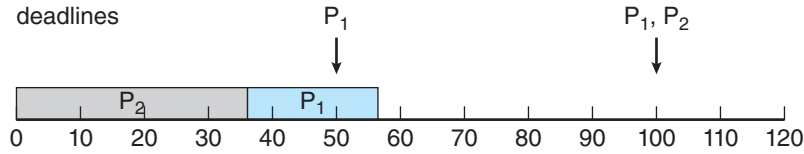
Figure 5.21 Scheduling of tasks when P_2 has a higher priority than P_1.

ing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Let's consider an example. We have two processes, P_1 and P_2. The periods for P_1 and P_2 are 50 and 100, respectively—that is, $p_1 = 50$ and $p_2 = 100$. The processing times are $t_1 = 20$ for P_1 and $t_2 = 35$ for P_2. The deadline for each process requires that it complete its CPU burst by the start of its next period.

We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines. If we measure the CPU utilization of a process P_i as the ratio of its burst to its period—t_i/p_i—the CPU utilization of P_1 is $20/50 = 0.40$ and that of P_2 is $35/100 = 0.35$, for a total CPU utilization of 75 percent. Therefore, it seems we can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

Suppose we assign P_2 a higher priority than P_1. The execution of P_1 and P_2 in this situation is shown in Figure 5.21. As we can see, P_2 starts execution first and completes at time 35. At this point, P_1 starts; it completes its CPU burst at time 55. However, the first deadline for P_1 was at time 50, so the scheduler has caused P_1 to miss its deadline.

Now suppose we use rate-monotonic scheduling, in which we assign P_1 a higher priority than P_2 because the period of P_1 is shorter than that of P_2. The execution of these processes in this situation is shown in Figure 5.22. P_1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline. P_2 starts running at this point and runs until time 50. At this time, it is preempted by P_1, although it still has 5 milliseconds remaining in its CPU burst. P_1 completes its CPU burst at time 70, at which point the scheduler resumes P_2. P_2 completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when P_1 is scheduled again.

Rate-monotonic scheduling is considered optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities. Let's next examine a set of processes that cannot be scheduled using the rate-monotonic algorithm.

Assume that process P_1 has a period of $p_1 = 50$ and a CPU burst of $t_1 = 25$. For P_2, the corresponding values are $p_2 = 80$ and $t_2 = 35$. Rate-monotonic
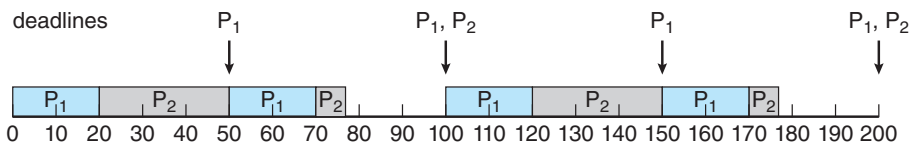

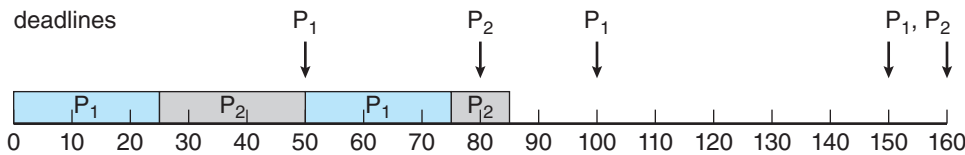**Figure 5.22** Rate-monotonic scheduling.

Figure 5.23   Missing deadlines with rate-monotonic scheduling.

scheduling would assign process P_1 a higher priority, as it has the shorter period. The total CPU utilization of the two processes is $(25/50) + (35/80) = 0.94$, and it therefore seems logical that the two processes could be scheduled and still leave the CPU with 6 percent available time. Figure 5.23 shows the scheduling of processes P_1 and P_2. Initially, P_1 runs until it completes its CPU burst at time 25. Process P_2 then begins running and runs until time 50, when it is preempted by P_1. At this point, P_2 still has 10 milliseconds remaining in its CPU burst. Process P_1 runs until time 75; consequently, P2 finishes its burst at time 85, after the deadline for completion of its CPU burst at time 80.

Despite being optimal, then, rate-monotonic scheduling has a limitation: CPU utilization is bounded, and it is not always possible to maximize CPU resources fully. The worst-case CPU utilization for scheduling N processes is

$$N(2^{1/N} - 1).$$

With one process in the system, CPU utilization is 100 percent, but it falls to approximately 69 percent as the number of processes approaches infinity. With two processes, CPU utilization is bounded at about 83 percent. Combined CPU utilization for the two processes scheduled in Figure 5.21 and Figure 5.22 is 75 percent; therefore, the rate-monotonic scheduling algorithm is guaranteed to schedule them so that they can meet their deadlines. For the two processes scheduled in Figure 5.23, combined CPU utilization is approximately 94 percent; therefore, rate-monotonic scheduling cannot guarantee that they can be scheduled so that they meet their deadlines.

5.6.4 Earliest-Deadline-First Scheduling

Earliest-deadline-firs (**EDF**) scheduling assigns priorities dynamically according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

To illustrate EDF scheduling, we again schedule the processes shown in Figure 5.23, which failed to meet deadline requirements under rate-monotonic scheduling. Recall that P_1 has values of $p_1 = 50$ and $t_1 = 25$ and that P_2 has values of $p_2 = 80$ and $t_2 = 35$. The EDF scheduling of these processes is shown in Figure 5.24. Process P_1 has the earliest deadline, so its initial priority is higher than that of process P_2. Process P_2 begins running at the end of the CPU burst for P_1. However, whereas rate-monotonic scheduling allows P_1 to preempt P_2
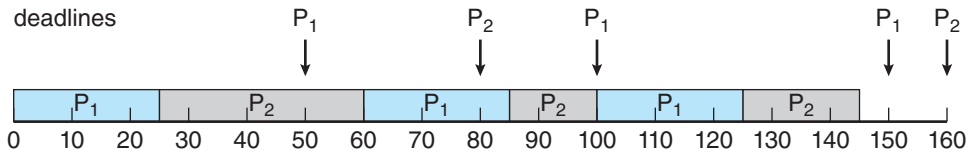
Figure 5.24   Earliest-deadline-first scheduling.

at the beginning of its next period at time 50, EDF scheduling allows process P_2 to continue running. P_2 now has a higher priority than P_1 because its next deadline (at time 80) is earlier than that of P_1 (at time 100). Thus, both P_1 and P_2 meet their first deadlines. Process P_1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100. P_2 begins running at this point, only to be preempted by P_1 at the start of its next period at time 100. P_2 is preempted because P_1 has an earlier deadline (time 150) than P_2 (time 160). At time 125, P_1 completes its CPU burst and P_2 resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when P_1 is scheduled to run once again.

Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable. The appeal of EDF scheduling is that it is theoretically optimal—theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

5.6.5   Proportional Share Scheduling

Proportional share schedulers operate by allocating T shares among all applications. An application can receive N shares of time, thus ensuring that the application will have N/T of the total processor time. As an example, assume that a total of $T = 100$ shares is to be divided among three processes, A, B, and C. A is assigned 50 shares, B is assigned 15 shares, and C is assigned 20 shares. This scheme ensures that A will have 50 percent of total processor time, B will have 15 percent, and C will have 20 percent.

Proportional share schedulers must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time. An admission-control policy will admit a client requesting a particular number of shares only if sufficient shares are available. In our current example, we have allocated $50 + 15 + 20 = 85$ shares of the total of 100 shares. If a new process D requested 30 shares, the admission controller would deny D entry into the system.

5.6.6   POSIX Real-Time Scheduling

The POSIX standard also provides extensions for real-time computing— POSIX.1b. Here, we cover some of the POSIX API related to scheduling real-time threads. POSIX defines two scheduling classes for real-time threads:

- SCHED_FIFO

- SCHED_RR

SCHED_FIFO schedules threads according to a first-come, first-served policy using a FIFO queue as outlined in Section 5.3.1. However, there is no time slicing among threads of equal priority. Therefore, the highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks. SCHED_RR uses a round-robin policy. It is similar to SCHED_FIFO except that it provides time slicing among threads of equal priority. POSIX provides an additional scheduling class—SCHED_OTHER—but its implementation is undefined and system specific; it may behave differently on different systems.

The POSIX API specifies the following two functions for getting and setting the scheduling policy:

- `pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)`

- `pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)`

The first parameter to both functions is a pointer to the set of attributes for the thread. The second parameter is either (1) a pointer to an integer that is set to the current scheduling policy (for `pthread_attr_getsched_policy()`) or (2) an integer value (SCHED_FIFO, SCHED_RR, or SCHED_OTHER) for the `pthread_attr_setsched_policy()` function. Both functions return nonzero values if an error occurs.

In Figure 5.25, we illustrate a POSIX Pthread program using this API. This program first determines the current scheduling policy and then sets the scheduling algorithm to SCHED_FIFO.

5.7 Operating-System Examples

We turn next to a description of the scheduling policies of the Linux, Windows, and Solaris operating systems. It is important to note that we use the term *process scheduling* in a general sense here. In fact, we are describing the scheduling of *kernel threads* with Solaris and Windows systems and of *tasks* with the Linux scheduler.

5.7.1 Example: Linux Scheduling

Process scheduling in Linux has had an interesting history. Prior to Version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm. However, as this algorithm was not designed with SMP systems in mind, it did not adequately support systems with multiple processors. In addition, it resulted in poor performance for systems with a large number of runnable processes. With Version 2.5 of the kernel, the scheduler was overhauled to include a scheduling algorithm—known as $O(1)$—that ran in constant time regardless of the number of tasks in the system. The $O(1)$ scheduler also provided

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

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

  /* get the default attributes */
  pthread_attr_init(&attr);

  /* get the current scheduling policy */
  if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    fprintf(stderr, "Unable to get policy.\n");
  else {
    if (policy == SCHED_OTHER)
     printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR)
     printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO)
     printf("SCHED_FIFO\n");
  }

  /* set the scheduling policy - FIFO, RR, or OTHER */
  if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");

  /* create the threads */
  for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);

  /* now join on each thread */
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
  /* do some work ... */

  pthread_exit(0);
}
```

Figure 5.25 POSIX real-time scheduling API.

increased support for SMP systems, including processor affinity and load balancing between processors. However, in practice, although the $O(1)$ scheduler delivered excellent performance on SMP systems, it led to poor response times for the interactive processes that are common on many desktop computer systems. During development of the 2.6 kernel, the scheduler was again revised; and in release 2.6.23 of the kernel, the *Completely Fair Scheduler* (CFS) became the default Linux scheduling algorithm.

Scheduling in the Linux system is based on **scheduling classes**. Each class is assigned a specific priority. By using different scheduling classes, the kernel can accommodate different scheduling algorithms based on the needs of the system and its processes. The scheduling criteria for a Linux server, for example, may be different from those for a mobile device running Linux. To decide which task to run next, the scheduler selects the highest-priority task belonging to the highest-priority scheduling class. Standard Linux kernels implement two scheduling classes: (1) a default scheduling class using the CFS scheduling algorithm and (2) a real-time scheduling class. We discuss each of these classes here. New scheduling classes can, of course, be added.

Rather than using strict rules that associate a relative priority value with the length of a time quantum, the CFS scheduler assigns a proportion of CPU processing time to each task. This proportion is calculated based on the **nice value** assigned to each task. Nice values range from −20 to +19, where a numerically lower nice value indicates a higher relative priority. Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values. The default nice value is 0. (The term *nice* comes from the idea that if a task increases its nice value from, say, 0 to +10, it is being nice to other tasks in the system by lowering its relative priority. In other words, nice processes finish last!) CFS doesn't use discrete values of time slices and instead identifies a **targeted latency**, which is an interval of time during which every runnable task should run at least once. Proportions of CPU time are allocated from the value of targeted latency. In addition to having default and minimum values, targeted latency can increase if the number of active tasks in the system grows beyond a certain threshold.

The CFS scheduler doesn't directly assign priorities. Rather, it records how long each task has run by maintaining the **virtual run time** of each task using the per-task variable `vruntime`. The virtual run time is associated with a decay factor based on the priority of a task: lower-priority tasks have higher rates of decay than higher-priority tasks. For tasks at normal priority (nice values of 0), virtual run time is identical to actual physical run time. Thus, if a task with default priority runs for 200 milliseconds, its `vruntime` will also be 200 milliseconds. However, if a lower-priority task runs for 200 milliseconds, its `vruntime` will be higher than 200 milliseconds. Similarly, if a higher-priority task runs for 200 milliseconds, its `vruntime` will be less than 200 milliseconds. To decide which task to run next, the scheduler simply selects the task that has the smallest `vruntime` value. In addition, a higher-priority task that becomes available to run can preempt a lower-priority task.

Let's examine the CFS scheduler in action: Assume that two tasks have the same nice values. One task is I/O-bound, and the other is CPU-bound. Typically, the I/O-bound task will run only for short periods before blocking for additional I/O, and the CPU-bound task will exhaust its time period whenever it has an opportunity to run on a processor. Therefore, the value of `vruntime` will

CFS PERFORMANCE

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Rather than using a standard queue data structure, each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below.
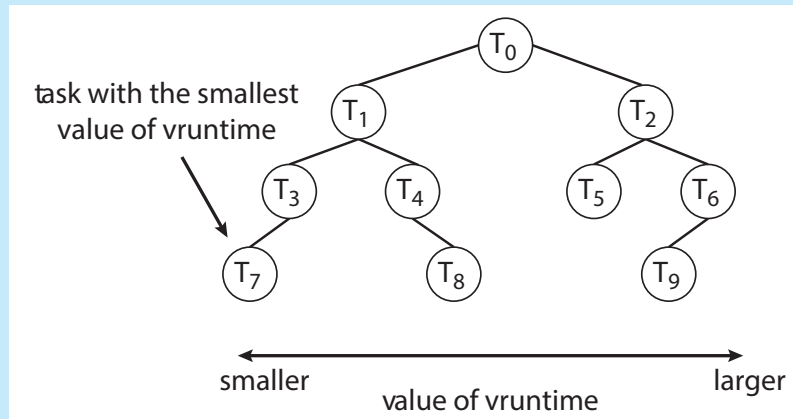
When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\log N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

eventually be lower for the I/O-bound task than for the CPU-bound task, giving the I/O-bound task higher priority than the CPU-bound task. At that point, if the CPU-bound task is executing when the I/O-bound task becomes eligible to run (for example, when I/O the task is waiting for becomes available), the I/O-bound task will preempt the CPU-bound task.

Linux also implements real-time scheduling using the POSIX standard as described in Section 5.6.6. Any task scheduled using either the SCHED_FIFO or the SCHED_RR real-time policy runs at a higher priority than normal (non-real-time) tasks. Linux uses two separate priority ranges, one for real-time tasks and a second for normal tasks. Real-time tasks are assigned static priorities within the range of 0 to 99, and normal tasks are assigned priorities from 100 to 139. These two ranges map into a global priority scheme wherein numerically lower values indicate higher relative priorities. Normal tasks are assigned a priority

Figure 5.26   Scheduling priorities on a Linux system.

based on their nice values, where a value of -20 maps to priority 100 and a nice value of $+19$ maps to 139. This scheme is shown in Figure 5.26.

The CFS scheduler also supports load balancing, using a sophisticated technique that equalizes the load among processing cores yet is also NUMA-aware and minimizes the migration of threads. CFS defines the load of each thread as a combination of the thread's priority and its average rate of CPU utilization. Therefore, a thread that has a high priority, yet is mostly I/O-bound and requires little CPU usage, has a generally low load, similar to the load of a low-priority thread that has high CPU utilization. Using this metric, the load of a queue is the sum of the loads of all threads in the queue, and balancing is simply ensuring that all queues have approximately the same load.

As highlighted in Section 5.5.4, however, migrating a thread may result in a memory access penalty due to either having to invalidate cache contents or, on NUMA systems, incurring longer memory access times. To address this problem, Linux identifies a hierarchical system of scheduling domains. A **scheduling domain** is a set of CPU cores that can be balanced against one another. This idea is illustrated in Figure 5.27. The cores in each scheduling domain are grouped according to how they share the resources of the system. For example, although each core shown in Figure 5.27 may have its own level 1 (L1) cache, pairs of cores share a level 2 (L2) cache and are thus organized into separate $domain_0$ and $domain_1$. Likewise, these two domains may share a level 3 (L3) cache, and are therefore organized into a processor-level domain (also known as a **NUMA node**). Taking this one-step further, on a NUMA system,

Figure 5.27   NUMA-aware load balancing with Linux CFS scheduler.

a larger system-level domain would combine separate processor-level NUMA nodes.

The general strategy behind CFS is to balance loads within domains, beginning at the lowest level of the hierarchy. Using Figure 5.27 as an example, initially a thread would only migrate between cores on the same domain (i.e. within $domain_0$ or $domain_1$.) Load balancing at the next level would occur between $domain_0$ and $domain_1$. CFS is reluctant to migrate threads between separate NUMA nodes if a thread would be moved farther from its local memory, and such migration would only occur under severe load imbalances. As a general rule, if the overall system is busy, CFS will not load-balance beyond the domain local to each core to avoid the memory latency penalties of NUMA systems.

5.7.2   Example: Windows Scheduling

Windows schedules threads using a priority-based, preemptive scheduling algorithm. The Windows scheduler ensures that the highest-priority thread will always run. The portion of the Windows kernel that handles scheduling is called the **dispatcher**. A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O. If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes. The **variable class** contains threads having priorities from 1 to 15, and the **real-time class** contains threads with priorities ranging from 16 to 31. (There is also a thread running at priority 0 that is used for memory management.) The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the **idle thread**.

There is a relationship between the numeric priorities of the Windows kernel and the Windows API. The Windows API identifies the following six priority classes to which a process can belong:

- IDLE_PRIORITY_CLASS

- BELOW_NORMAL_PRIORITY_CLASS

- NORMAL_PRIORITY_CLASS

- ABOVE_NORMAL_PRIORITY_CLASS

- HIGH_PRIORITY_CLASS

- REALTIME_PRIORITY_CLASS

Processes are typically members of the NORMAL_PRIORITY_CLASS. A process belongs to this class unless the parent of the process was a member of the IDLE_PRIORITY_CLASS or unless another class was specified when the process was created. Additionally, the priority class of a process can be altered with

the `SetPriorityClass()` function in the Windows API. Priorities in all classes except the REALTIME_PRIORITY_CLASS are variable, meaning that the priority of a thread belonging to one of these classes can change.

A thread within a given priority class also has a relative priority. The values for relative priorities include:

- IDLE

- LOWEST

- BELOW_NORMAL

- NORMAL

- ABOVE_NORMAL

- HIGHEST

- TIME_CRITICAL

The priority of each thread is based on both the priority class it belongs to and its relative priority within that class. This relationship is shown in Figure 5.28. The values of the priority classes appear in the top row. The left column contains the values for the relative priorities. For example, if the relative priority of a thread in the ABOVE_NORMAL_PRIORITY_CLASS is NORMAL, the numeric priority of that thread is 10.

Furthermore, each thread has a base priority representing a value in the priority range for the class to which the thread belongs. By default, the base priority is the value of the NORMAL relative priority for that class. The base priorities for each priority class are as follows:

- REALTIME_PRIORITY_CLASS—24

- HIGH_PRIORITY_CLASS—13

- ABOVE_NORMAL_PRIORITY_CLASS—10

- NORMAL_PRIORITY_CLASS—8

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figure 5.28   Windows thread priorities.

- BELOW_NORMAL_PRIORITY_CLASS—6

- IDLE_PRIORITY_CLASS—4

The initial priority of a thread is typically the base priority of the process the thread belongs to, although the `SetThreadPriority()` function in the Windows API can also be used to modify a thread's base priority.

When a thread's time quantum runs out, that thread is interrupted. If the thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority, however. Lowering the priority tends to limit the CPU consumption of compute-bound threads. When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority. The amount of the boost depends on what the thread was waiting for. For example, a thread waiting for keyboard I/O would get a large increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy tends to give good response times to interactive threads that are using the mouse and windows. It also enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background. This strategy is used by several operating systems, including UNIX. In addition, the window with which the user is currently interacting receives a priority boost to enhance its response time.

When a user is running an interactive program, the system needs to provide especially good performance. For this reason, Windows has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS. Windows distinguishes between the **foreground process** that is currently selected on the screen and the **background processes** that are not currently selected. When a process moves into the foreground, Windows increases the scheduling quantum by some factor—typically by 3. This increase gives the foreground process three times longer to run before a time-sharing preemption occurs.

Windows 7 introduced **user-mode scheduling** (UMS), which allows applications to create and manage threads independently of the kernel. Thus, an application can create and schedule multiple threads without involving the Windows kernel scheduler. For applications that create a large number of threads, scheduling threads in user mode is much more efficient than kernel-mode thread scheduling, as no kernel intervention is necessary.

Earlier versions of Windows provided a similar feature known as **fiber**, which allowed several user-mode threads (fibers) to be mapped to a single kernel thread. However, fibers were of limited practical use. A fiber was unable to make calls to the Windows API because all fibers had to share the thread environment block (TEB) of the thread on which they were running. This presented a problem if a Windows API function placed state information into the TEB for one fiber, only to have the information overwritten by a different fiber. UMS overcomes this obstacle by providing each user-mode thread with its own thread context.

In addition, unlike fibers, UMS is not intended to be used directly by the programmer. The details of writing user-mode schedulers can be very challenging, and UMS does not include such a scheduler. Rather, the schedulers come from programming language libraries that build on top of UMS. For example, Microsoft provides **Concurrency Runtime** (ConcRT), a concurrent programming framework for C++ that is designed for task-based parallelism

(Section 4.2) on multicore processors. ConcRT provides a user-mode scheduler together with facilities for decomposing programs into tasks, which can then be scheduled on the available processing cores.

Windows also supports scheduling on multiprocessor systems as described in Section 5.5 by attempting to schedule a thread on the most optimal processing core for that thread, which includes maintaining a thread's preferred as well as most recent processor. One technique used by Windows is to create sets of logical processors (known as **SMT sets**). On a hyper-threaded SMT system, hardware threads belonging to the same CPU core would also belong to the same SMT set. Logical processors are numbered, beginning from 0. As an example, a dual-threaded/quad-core system would contain eight logical processors, consisting of the four SMT sets: {0, 1}, {2, 3}, {4, 5}, and {6, 7}. To avoid cache memory access penalites highlighted in Section 5.5.4, the scheduler attempts to maintain a thread running on logical processors within the same SMT set.

To distribute loads across different logical processors, each thread is assigned an **ideal processor**, which is a number representing a thread's preferred processor. Each process has an initial seed value identifying the ideal CPU for a thread belonging to that process. This seed is incremented for each new thread created by that process, thereby distributing the load across different logical processors. On SMT systems, the increment for the next ideal processor is in the next SMT set. For example, on a dual-threaded/quad-core system, the ideal processors for threads in a specific process would be assigned 0, 2, 4, 6, 0, 2, ... To avoid the situation wherby the first thread for each process is assigned processor 0, processes are assigned different seed values, thereby distributing the load of threads across all physical processing cores in the system. Continuing our example from above, if the seed for a second process were 1, the ideal processors would be assigned in the order 1, 3, 5, 7, 1, 3, and so forth.

5.7.3 Example: Solaris Scheduling

Solaris uses priority-based thread scheduling. Each thread belongs to one of six classes:

1. Time sharing (TS)

2. Interactive (IA)

3. Real time (RT)

4. System (SYS)

5. Fair share (FSS)

6. Fixed priority (FP)

Within each class there are different priorities and different scheduling algorithms.

The default scheduling class for a process is time sharing. The scheduling policy for the time-sharing class dynamically alters priorities and assigns time slices of different lengths using a multilevel feedback queue. By default, there is an inverse relationship between priorities and time slices. The higher the

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figure 5.29 Solaris dispatch table for time-sharing and interactive threads.

priority, the smaller the time slice; and the lower the priority, the larger the time slice. Interactive processes typically have a higher priority; CPU-bound processes, a lower priority. This scheduling policy gives good response time for interactive processes and good throughput for CPU-bound processes. The interactive class uses the same scheduling policy as the time-sharing class, but it gives windowing applications—such as those created by the KDE or GNOME window managers—a higher priority for better performance.

Figure 5.29 shows the simplified dispatch table for scheduling time-sharing and interactive threads. These two scheduling classes include 60 priority levels, but for brevity, we display only a handful. (To see the full dispatch table on a Solaris system or VM, run dispadmin -c TS -g.) The dispatch table shown in Figure 5.29 contains the following fields:

- **Priority**. The class-dependent priority for the time-sharing and interactive classes. A higher number indicates a higher priority.

- **Time quantum**. The time quantum for the associated priority. This illustrates the inverse relationship between priorities and time quanta: the lowest priority (priority 0) has the highest time quantum (200 milliseconds), and the highest priority (priority 59) has the lowest time quantum (20 milliseconds).

- **Time quantum expired**. The new priority of a thread that has used its entire time quantum without blocking. Such threads are considered CPU-intensive. As shown in the table, these threads have their priorities lowered.

- **Return from sleep**. The priority of a thread that is returning from sleeping (such as from waiting for I/O). As the table illustrates, when I/O is available for a waiting thread, its priority is boosted to between 50 and 59, supporting the scheduling policy of providing good response time for interactive processes.

Threads in the real-time class are given the highest priority. A real-time process will run before a process in any other class. This assignment allows a real-time process to have a guaranteed response from the system within a bounded period of time. In general, however, few processes belong to the real-time class.

Solaris uses the system class to run kernel threads, such as the scheduler and paging daemon. Once the priority of a system thread is established, it does not change. The system class is reserved for kernel use (user processes running in kernel mode are not in the system class).

The fixed-priority and fair-share classes were introduced with Solaris 9. Threads in the fixed-priority class have the same priority range as those in the time-sharing class; however, their priorities are not dynamically adjusted. The fair-share class uses CPU **shares** instead of priorities to make scheduling decisions. CPU shares indicate entitlement to available CPU resources and are allocated to a set of processes (known as a **project**).

Each scheduling class includes a set of priorities. However, the scheduler converts the class-specific priorities into global priorities and selects the thread with the highest global priority to run. The selected thread runs on the CPU until it (1) blocks, (2) uses its time slice, or (3) is preempted by a higher-priority thread. If there are multiple threads with the same priority, the scheduler uses a round-robin queue. Figure 5.30 illustrates how the six scheduling classes relate to one another and how they map to global priorities. Notice that the kernel maintains ten threads for servicing interrupts. These threads do not belong to any scheduling class and execute at the highest priority (160–169). As mentioned, Solaris has traditionally used the many-to-many model (Section 4.3.3) but switched to the one-to-one model (Section 4.3.2) beginning with Solaris 9.

5.8 Algorithm Evaluation

How do we select a CPU-scheduling algorithm for a particular system? As we saw in Section 5.3, there are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult.

The first problem is defining the criteria to be used in selecting an algorithm. As we saw in Section 5.2, criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these elements. Our criteria may include several measures, such as these:

- Maximizing CPU utilization under the constraint that the maximum response time is 300 milliseconds
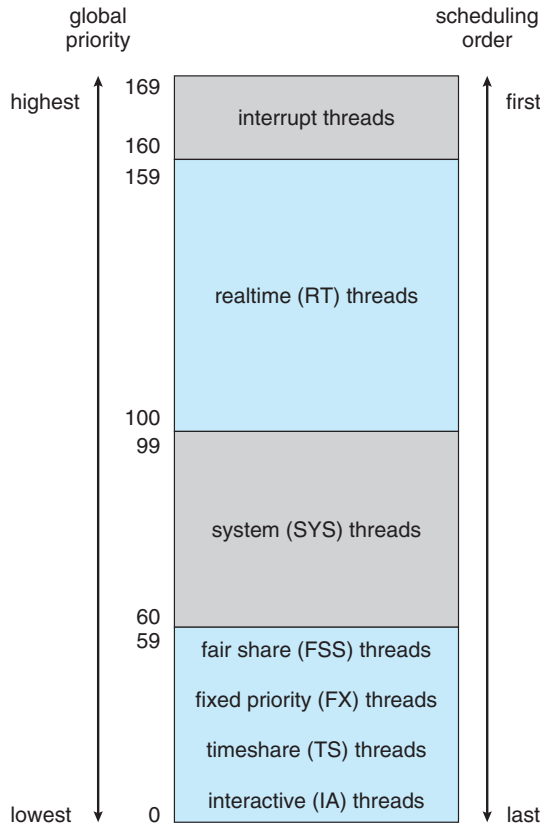
Figure 5.30   Solaris scheduling.

- Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Once the selection criteria have been defined, we want to evaluate the algorithms under consideration. We next describe the various evaluation methods we can use.

5.8.1 Deterministic Modeling

One major class of evaluation methods is **analytic evaluation**. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number to evaluate the performance of the algorithm for that workload.

Deterministic modeling is one type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

| Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as

P_1	P_2	P_3	P_4	P_5

0   10   39   42   49   61

The waiting time is 0 milliseconds for process P_1, 10 milliseconds for process P_2, 39 milliseconds for process P_3, 42 milliseconds for process P_4, and 49 milliseconds for process P_5. Thus, the average waiting time is $(0 + 10 + 39 + 42 + 49)/5 = 28$ milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as

P_3	P_4	P_1	P_5	P_2

0   3   10   20   32   61
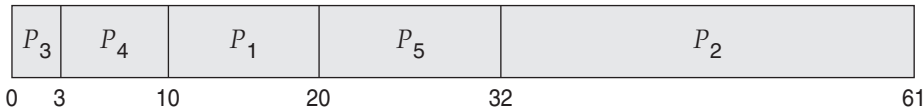
The waiting time is 10 milliseconds for process P_1, 32 milliseconds for process P_2, 0 milliseconds for process P_3, 3 milliseconds for process P_4, and 20 milliseconds for process P_5. Thus, the average waiting time is $(10 + 32 + 0 + 3 + 20)/5 = 13$ milliseconds.

With the RR algorithm, we execute the processes as

P_1	P_2	P_3	P_4	P_5	P_2	P_5	P_2

0   10   20   23   30   40   50 52   61
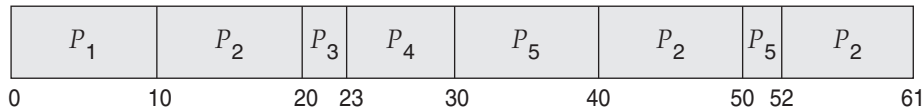
The waiting time is 0 milliseconds for process P_1, 32 milliseconds for process P_2, 20 milliseconds for process P_3, 23 milliseconds for process P_4, and 40 milliseconds for process P_5. Thus, the average waiting time is $(0 + 32 + 20 + 23 + 40)/5 = 23$ milliseconds.

We can see that, in this case, the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. However, it requires exact numbers for input, and its answers apply only to those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we are running the same program over and over again and can

measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately. For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

5.8.2 Queueing Models

On many systems, the processes that are run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling. What can be determined, however, is the distribution of CPU and I/O bursts. These distributions can be measured and then approximated or simply estimated. The result is a mathematical formula describing the probability of a particular CPU burst. Commonly, this distribution is exponential and is described by its mean. Similarly, we can describe the distribution of times when processes arrive in the system (the arrival-time distribution). From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms.

The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called **queueing-network analysis**.

As an example, let n be the average long-term queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). We expect that during the time W that a process waits, $\lambda \times W$ new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

$$n = \lambda \times W.$$

This equation, known as **Little's formula**, is particularly useful because it is valid for any scheduling algorithm and arrival distribution. For example n could be the number of customers in a store.

We can use Little's formula to compute one of the three variables if we know the other two. For example, if we know that 7 processes arrive every second (on average) and that there are normally 14 processes in the queue, then we can compute the average waiting time per process as 2 seconds.

Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. At the moment, the classes of algorithms and distributions that can be handled are fairly limited. The mathematics of complicated algorithms and distributions can be difficult to work with. Thus, arrival and service distributions are often defined in mathematically tractable—but unrealistic—ways. It is also generally necessary to make a number of independent assumptions, which may not be accurate. As a result of these difficulties, queueing models are often only approximations of real systems, and the accuracy of the computed results may be questionable.

5.8.3 Simulations

To get a more accurate evaluation of scheduling algorithms, we can use simulations. Running simulations involves programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock. As this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator that is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions. The distributions can be defined mathematically (uniform, exponential, Poisson) or empirically. If a distribution is to be defined empirically, measurements of the actual system under study are taken. The results define the distribution of events in the real system; this distribution can then be used to drive the simulation.

A distribution-driven simulation may be inaccurate, however, because of relationships between successive events in the real system. The frequency distribution indicates only how many instances of each event occur; it does not indicate anything about the order of their occurrence. To correct this problem, we can use **trace files**. We create a trace by monitoring the real system and recording the sequence of actual events (Figure 5.31). We then use this sequence to drive the simulation. Trace files provide an excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs.

Simulations can be expensive, often requiring many hours of computer time. A more detailed simulation provides more accurate results, but it also
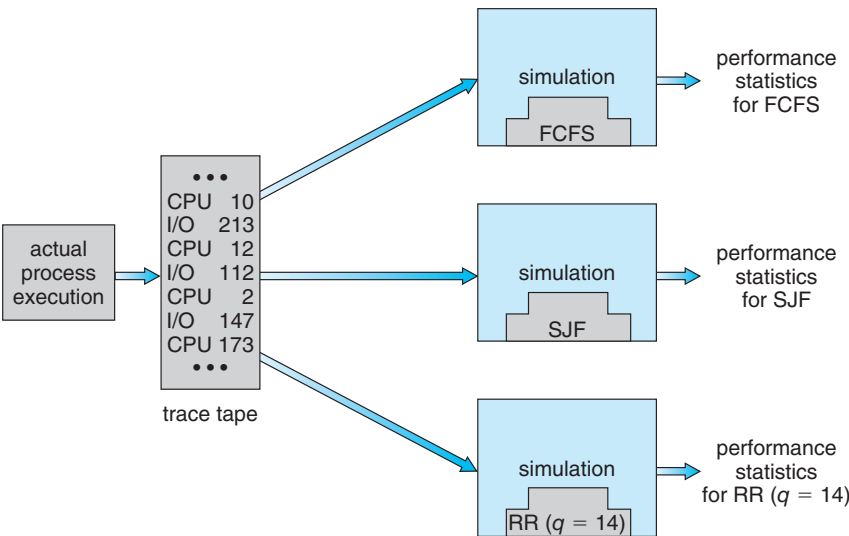

Figure 5.31 Evaluation of CPU schedulers by simulation.

takes more computer time. In addition, trace files can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.

5.8.4 Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.

This method is not without expense. The expense is incurred in coding the algorithm and modifying the operating system to support it (along with its required data structures). There is also cost in testing the changes, usually in virtual machines rather than on dedicated hardware. **Regression testing** confirms that the changes haven't made anything worse, and haven't caused new bugs or caused old bugs to be recreated (for example because the algorithm being replaced solved some bug and changing it caused that bug to reoccur).

Another difficulty is that the environment in which the algorithm is used will change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive processes are given priority over noninteractive processes, then users may switch to interactive use. This problem is usually addressed by using tools or scripts that encapsulate complete sets of actions, repeatedly using those tools, and using those tools while measuring the results (and detecting any problems they cause in the new environment).

Of course human or program behavior can attempt to circumvent scheduling algorithms. For example, researchers designed one system that classified interactive and noninteractive processes automatically by looking at the amount of terminal I/O. If a process did not input or output to the terminal in a 1-second interval, the process was classified as noninteractive and was moved to a lower-priority queue. In response to this policy, one programmer modified his programs to write an arbitrary character to the terminal at regular intervals of less than 1 second. The system gave his programs a high priority, even though the terminal output was completely meaningless.

In general, most flexible scheduling algorithms are those that can be altered by the system managers or by the users so that they can be tuned for a specific application or set of applications. A workstation that performs high-end graphical applications, for instance, may have scheduling needs different from those of a web server or file server. Some operating systems—particularly several versions of UNIX—allow the system manager to fine-tune the scheduling parameters for a particular system configuration. For example, Solaris provides the `dispadmin` command to allow the system administrator to modify the parameters of the scheduling classes described in Section 5.7.3.

Another approach is to use APIs that can modify the priority of a process or thread. The Java, POSIX, and Windows APIs provide such functions. The downfall of this approach is that performance-tuning a system or application most often does not result in improved performance in more general situations.

5.9 Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.

- Scheduling algorithms may be either preemptive (where the CPU can be taken away from a process) or nonpreemptive (where a process must voluntarily relinquish control of the CPU). Almost all modern operating systems are preemptive.

- Scheduling algorithms can be evaluated according to the following five criteria: (1) CPU utilization, (2) throughput, (3) turnaround time, (4) waiting time, and (5) response time.

- First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes.

- Shortest-job-first (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult, however, because predicting the length of the next CPU burst is difficult.

- Round-robin (RR) scheduling allocates the CPU to each process for a time quantum. If the process does not relinquish the CPU before its time quantum expires, the process is preempted, and another process is scheduled to run for a time quantum.

- Priority scheduling assigns each process a priority, and the CPU is allocated to the process with the highest priority. Processes with the same priority can be scheduled in FCFS order or using RR scheduling.

- Multilevel queue scheduling partitions processes into several separate queues arranged by priority, and the scheduler executes the processes in the highest-priority queue. Different scheduling algorithms may be used in each queue.

- Multilevel feedback queues are similar to multilevel queues, except that a process may migrate between different queues.

- Multicore processors place one or more CPUs on the same physical chip, and each CPU may have more than one hardware thread. From the perspective of the operating system, each hardware thread appears to be a logical CPU.

- Load balancing on multicore systems equalizes loads between CPU cores, although migrating threads between cores to balance loads may invalidate cache contents and therefore may increase memory access times.

- Soft real-time scheduling gives priority to real-time tasks over non-real-time tasks. Hard real-time scheduling provides timing guarantees for real-time tasks,

- Rate-monotonic real-time scheduling schedules periodic tasks using a static priority policy with preemption.

- Earliest-deadline-first (EDF) scheduling assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.

- Proportional share scheduling allocates T shares among all applications. If an application is allocated N shares of time, it is ensured of having N/T of the total processor time.

- Linux uses the completely fair scheduler (CFS), which assigns a proportion of CPU processing time to each task. The proportion is based on the virtual runtime (`vruntime`) value associated with each task.

- Windows scheduling uses a preemptive, 32-level priority scheme to determine the order of thread scheduling.

- Solaris identifies six unique scheduling classes that are mapped to a global priority. CPU-intensive threads are generally assigned lower priorities (and longer time quantums), and I/O-bound threads are usually assigned higher priorities (with shorter time quantums.)

- Modeling and simulations can be used to evaluate a CPU scheduling algorithm.

Practice Exercises

5.1 A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of n.

5.2 Explain the difference between preemptive and nonpreemptive scheduling.

5.3 Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the

average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as *future-knowledge scheduling*.

5.4 Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

5.5 The following processes are being scheduled using a preemptive, round-robin scheduling algorithm.

Process	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an **idle task** (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a

time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

    a.   Show the scheduling order of the processes using a Gantt chart.

    b.   What is the turnaround time for each process?

    c.   What is the waiting time for each process?

    d.   What is the CPU utilization rate?

5.6 What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?

5.7 Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on.

These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?

    a.   Priority and SJF

    b.   Multilevel feedback queues and FCFS

    c.   Priority and FCFS

    d.   RR and SJF

5.8 Suppose that a CPU scheduling algorithm favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

5.9 Distinguish between PCS and SCS scheduling.

5.10 The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

$$\text{Priority} = (\text{recent CPU usage} / 2) + \text{base}$$

where base = 60 and *recent CPU usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage for process P_1 is 40, for process P_2 is 18, and for process P_3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

Further Reading

Scheduling policies used in the UNIX FreeBSD 5.2 are presented by [McKusick et al. (2015)]; The Linux CFS scheduler is further described in https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/. Solaris scheduling is described by [Mauro and McDougall (2007)]. [Russinovich et al. (2017)] discusses scheduling in Windows internals. [Butenhof (1997)] and [Lewis and Berg (1998)] describe scheduling in Pthreads systems. Multicore scheduling is examined in [McNairy and Bhatia (2005)], [Kongetira et al. (2005)], and [Siddha et al. (2007)] .

Bibliography

[Butenhof (1997)]   D. Butenhof, *Programming with POSIX Threads*, Addison-Wesley (1997).

[Kongetira et al. (2005)]   P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded SPARC Processor", *IEEE Micro Magazine*, Volume 25, Number 2 (2005), pages 21–29.

[Lewis and Berg (1998)]   B. Lewis and D. Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press (1998).

[Mauro and McDougall (2007)]   J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall (2007).

[McKusick et al. (2015)]   M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System − Second Edition*, Pearson (2015).

[McNairy and Bhatia (2005)]   C. McNairy and R. Bhatia, "Montecito: A Dual–Core, Dual-Threaded Itanium Processor", *IEEE Micro Magazine*, Volume 25, Number 2 (2005), pages 10–20.

[Russinovich et al. (2017)]   M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals − Part 1,* Seventh Edition, Microsoft Press (2017).

[Siddha et al. (2007)]   S. Siddha, V. Pallipadi, and A. Mallick, "Process Scheduling Challenges in the Era of Multi-Core Processors", *Intel Technology Journal*, Volume 11, Number 4 (2007).

Chapter 5 Exercises

5.11 Of these two types of programs:

  a. I/O-bound

  b. CPU-bound

which is more likely to have voluntary context switches, and which is more likely to have nonvoluntary context switches? Explain your answer.

5.12 Discuss how the following pairs of scheduling criteria conflict in certain settings.

  a. CPU utilization and response time

  b. Average turnaround time and maximum waiting time

  c. I/O device utilization and CPU utilization

5.13 One technique for implementing **lottery scheduling** works by assigning processes lottery tickets, which are used for allocating 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 CPU. The BTV operating system implements lottery scheduling by holding a lottery 50 times each second, with each lottery winner getting 20 milliseconds of CPU time (20 milliseconds × 50 = 1 second). Describe how the BTV scheduler can ensure that higher-priority threads receive more attention from the CPU than lower-priority threads.

5.14 Most scheduling algorithms maintain a **run queue**, which lists processes eligible to run on a processor. On multicore systems, there are two general options: (1) each processing core has its own run queue, or (2) a single run queue is shared by all processing cores. What are the advantages and disadvantages of each of these approaches?

5.15 Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

  a. $\alpha = 0$ and $\tau_0 = 100$ milliseconds

  b. $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds

5.16 A variation of the round-robin scheduler is the **regressive round-robin** scheduler. This scheduler assigns each process a time quantum and a priority. The initial value of a time quantum is 50 milliseconds. However, every time a process has been allocated the CPU and uses its entire time quantum (does not block for I/O), 10 milliseconds is added to its time quantum, and its priority level is boosted. (The time quantum for a process can be increased to a maximum of 100 milliseconds.) When a process blocks before using its entire time quantum, its time quantum is reduced by 5 milliseconds, but its priority remains the same. What type of process (CPU-bound or I/O-bound) does the regressive round-robin scheduler favor? Explain.

5.17 Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	5	4
P_2	3	1
P_3	1	2
P_4	7	2
P_5	4	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

5.18 The following processes are being scheduled using a preemptive, priority-based, round-robin scheduling algorithm.

Process	Priority	Burst	Arrival
P_1	8	15	0
P_2	3	20	0
P_3	4	20	20
P_4	4	20	25
P_5	5	5	45
P_6	5	15	55

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. The scheduler will execute the highest-priority process. For processes with the same priority, a round-robin scheduler will be used with a time quantum of 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.

b. What is the turnaround time for each process?

c. What is the waiting time for each process?

5.19 The `nice` command is used to set the nice value of a process on Linux, as well as on other UNIX systems. Explain why some systems may allow any user to assign a process a nice value >= 0 yet allow only the root (or administrator) user to assign nice values < 0.

5.20   Which of the following scheduling algorithms could result in starvation?

   a.   First-come, first-served

   b.   Shortest job first

   c.   Round robin

   d.   Priority

5.21   Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.

   a.   What would be the effect of putting two pointers to the same process in the ready queue?

   b.   What would be two major advantages and two disadvantages of this scheme?

   c.   How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

5.22   Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. Describe the CPU utilization for a round-robin scheduler when:

   a.   The time quantum is 1 millisecond

   b.   The time quantum is 10 milliseconds

5.23   Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?

5.24   Consider a preemptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α. When it is running, its priority changes at a rate β. All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.

   a.   What is the algorithm that results from $\beta > \alpha > 0$?

   b.   What is the algorithm that results from $\alpha < \beta < 0$?

5.25   Explain the how the following scheduling algorithms discriminate either in favor of or against short processes:

   a.   FCFS

   b.   RR

   c.   Multilevel feedback queues

5.26   Describe why a shared ready queue might suffer from performance problems in an SMP environment.

5.27   Consider a load-balancing algorithm that ensures that each queue has approximately the same number of threads, independent of priority. How effectively would a priority-based scheduling algorithm handle this situation if one run queue had all high-priority threads and a second queue had all low-priority threads?

5.28   Assume that an SMP system has private, per-processor run queues. When a new process is created, it can be placed in either the same queue as the parent process or a separate queue.

   a.   What are the benefits of placing the new process in the same queue as its parent?

   b.   What are the benefits of placing the new process in a different queue?

5.29   Assume that a thread has blocked for network I/O and is eligible to run again. Describe why a NUMA-aware scheduling algorithm should reschedule the thread on the same CPU on which it previously ran.

5.30   Using the Windows scheduling algorithm, determine the numeric priority of each of the following threads.

   a.   A thread in the REALTIME_PRIORITY_CLASS with a relative priority of NORMAL

   b.   A thread in the ABOVE_NORMAL_PRIORITY_CLASS with a relative priority of HIGHEST

   c.   A thread in the BELOW_NORMAL_PRIORITY_CLASS with a relative priority of ABOVE_NORMAL

5.31   Assuming that no threads belong to the REALTIME_PRIORITY_CLASS and that none may be assigned a TIME_CRITICAL priority, what combination of priority class and priority corresponds to the highest possible relative priority in Windows scheduling?

5.32   Consider the scheduling algorithm in the Solaris operating system for time-sharing threads.

   a.   What is the time quantum (in milliseconds) for a thread with priority 15? With priority 40?

   b.   Assume that a thread with priority 50 has used its entire time quantum without blocking. What new priority will the scheduler assign this thread?

   c.   Assume that a thread with priority 20 blocks for I/O before its time quantum has expired. What new priority will the scheduler assign this thread?

5.33   Assume that two tasks, *A* and *B*, are running on a Linux system. The nice values of *A* and *B* are −5 and +5, respectively. Using the CFS scheduler as a guide, describe how the respective values of `vruntime` vary between the two processes given each of the following scenarios:

- Both *A* and *B* are CPU-bound.

- *A* is I/O-bound, and *B* is CPU-bound.

- *A* is CPU-bound, and *B* is I/O-bound.

5.34   Provide a specific circumstance that illustrates where rate-monotonic scheduling is inferior to earliest-deadline-first scheduling in meeting real-time process deadlines?

5.35   Consider two processes, P_1 and P_2, where $p_1 = 50$, $t_1 = 25$, $p_2 = 75$, and $t_2 = 30$.

a.   Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart such as the ones in Figure 5.21–Figure 5.24.

b.   Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.

5.36   Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.

5.37   Describe the advantages of using heterogeneous multiprocessing in a mobile system.

Programming Projects

Scheduling Algorithms

This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:

- First-come, first-served (FCFS), which schedules tasks in the order in which they request the CPU.

- Shortest-job-first (SJF), which schedules tasks in order of the length of the tasks' next CPU burst.

- Priority scheduling, which schedules tasks based on priority.

- Round-robin (RR) scheduling, where each task is run for a time quantum (or for the remainder of its CPU burst).

- Priority with round-robin, which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority.

Priorities range from 1 to 10, where a higher numeric value indicates a higher relative priority. For round-robin scheduling, the length of a time quantum is 10 milliseconds.

I. Implementation

The implementation of this project may be completed in either C or Java, and program files supporting both of these languages are provided in the source code download for the text. These supporting files read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler.

The schedule of tasks has the form [*task name*] [*priority*] [*CPU burst*], with the following example format:

```
T1, 4, 20
T2, 2, 25
T3, 3, 25
T4, 3, 15
T5, 10, 10
```

Thus, task T1 has priority 4 and a CPU burst of 20 milliseconds, and so forth. It is assumed that all tasks arrive at the same time, so your scheduler algorithms do not have to support higher-priority processes preempting processes with lower priorities. In addition, tasks do not have to be placed into a queue or list in any particular order.

There are a few different strategies for organizing the list of tasks, as first presented in Section 5.1.2. One approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling

algorithm. For example, SJF scheduling would search the list to find the task with the shortest next CPU burst. Alternatively, a list could be ordered according to scheduling criteria (that is, by priority). One other strategy involves having a separate queue for each unique priority, as shown in Figure 5.7. These approaches are briefly discussed in Section 5.3.6. It is also worth highlighting that we are using the terms *list* and *queue* somewhat interchangeably. However, a queue has very specific FIFO functionality, whereas a list does not have such strict insertion and deletion requirements. You are likely to find the functionality of a general list to be more suitable when completing this project.

II. C Implementation Details

The file `driver.c` reads in the schedule of tasks, inserts each task into a linked list, and invokes the process scheduler by calling the `schedule()` function. The `schedule()` function executes each task according to the specified scheduling algorithm. Tasks selected for execution on the CPU are determined by the `pickNextTask()` function and are executed by invoking the `run()` function defined in the `CPU.c` file. A `Makefile` is used to determine the specific scheduling algorithm that will be invoked by `driver`. For example, to build the FCFS scheduler, we would enter

```
make fcfs
```

and would execute the scheduler (using the schedule of tasks `schedule.txt`) as follows:

```
./fcfs schedule.txt
```

Refer to the `README` file in the source code download for further details. Before proceeding, be sure to familiarize yourself with the source code provided as well as the `Makefile`.

III. Java Implementation Details

The file `Driver.java` reads in the schedule of tasks, inserts each task into a Java `ArrayList`, and invokes the process scheduler by calling the `schedule()` method. The following interface identifies a generic scheduling algorithm, which the five different scheduling algorithms will implement:

```java
public interface Algorithm
{
   // Implementation of scheduling algorithm
   public void schedule();

   // Selects the next task to be scheduled
   public Task pickNetTask();
}
```

The `schedule()` method obtains the next task to be run on the CPU by invoking the `pickNextTask()` method and then executes this `Task` by calling the `static run()` method in the `CPU.java` class.

   The program is run as follows:

```
java Driver fcfs schedule.txt
```

Refer to the README file in the source code download for further details. Before proceeding, be sure to familiarize yourself with all Java source files provided in the source code download.

IV. Further Challenges

Two additional challenges are presented for this project:

1. Each task provided to the scheduler is assigned a unique task (tid). If a scheduler is running in an SMP environment where each CPU is separately running its own scheduler, there is a possible race condition on the variable that is used to assign task identifiers. Fix this race condition using an atomic integer.

   On Linux and macOS systems, the ⎵⎵sync⎵fetch⎵and⎵add() function can be used to atomically increment an integer value. As an example, the following code sample atomically increments value by 1:

   ```
    int value = 0;
   __sync_fetch_and_add(&value,1);
   ```

   Refer to the Java API for details on how to use the AtomicInteger class for Java programs.

2. Calculate the average turnaround time, waiting time, and response time for each of the scheduling algorithms.

Part Three

Process Synchronization

A system typically consists of several (perhaps hundreds or even thousands) of threads running either concurrently or in parallel. Threads often share user data. Meanwhile, the operating system continuously updates various data structures to support multiple threads. A *race condition* exists when access to shared data is not controlled, possibly resulting in corrupt data values.

Process synchronization involves using tools that control access to shared data to avoid race conditions. These tools must be used carefully, as their incorrect use can result in poor system performance, including deadlock.

Synchronization Tools

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through shared memory or message passing. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

CHAPTER OBJECTIVES

- Describe the critical-section problem and illustrate a race condition.
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables.
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem in low-, moderate-, and high-contention scenarios.

6.1 Background

We've already seen that processes can execute concurrently or in parallel. Section 3.2.2 introduced the role of process scheduling and described how the CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process. Additionally, Section 4.2 introduced parallel execution, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores. In this chapter, we explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

Let's consider an example of how this can happen. In Chapter 3, we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the producer–consumer problem, which is a representative paradigm of many operating system functions. Specifically, in Section 3.5, we described how a bounded buffer could be used to enable processes to share memory.

We now return to our consideration of the bounded buffer. As we pointed out, our original solution allowed at most BUFFER_SIZE − 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable, count, initialized to 0. count is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```
while (true) {
      /* produce an item in next_produced */

      while (count == BUFFER_SIZE)
        ; /* do nothing */

      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      count++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
      while (count == 0)
        ; /* do nothing */

      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      count--;

      /* consume the item in next_consumed */
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable count is currently 5 and that the producer and consumer processes concurrently execute the statements "count++" and "count--". Following the execution of these two statements, the value of the variable count may be 4, 5, or 6! The only correct result, though, is count == 5, which is generated correctly if the producer and consumer execute separately.

   We can show that the value of count may be incorrect as follows. Note that the statement "count++" may be implemented in machine language (on a typical machine) as follows:

$$register_1 = \texttt{count}$$
$$register_1 = register_1 + 1$$
$$\texttt{count} = register_1$$

where $register_1$ is one of the local CPU registers. Similarly, the statement "count--" is implemented as follows:

$$register_2 = \texttt{count}$$
$$register_2 = register_2 - 1$$
$$\texttt{count} = register_2$$

where again $register_2$ is one of the local CPU registers. Even though $register_1$ and $register_2$ may be the same physical register, remember that the contents of this register will be saved and restored by the interrupt handler (Section 1.2.3).

   The concurrent execution of "count++" and "count--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is the following:

T_0:	*producer*	execute	$register_1 = \texttt{count}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{count}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{count} = register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	$\texttt{count} = register_2$	$\{count = 4\}$

Notice that we have arrived at the incorrect state "count == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T_4 and T_5, we would arrive at the incorrect state "count == 6".

   We would arrive at this incorrect state because we allowed both processes to manipulate the variable count concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable count. To make such a guarantee, we require that the processes be synchronized in some way.

   Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, as we have emphasized in earlier chapters, the prominence of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads—which are quite possibly sharing data—are running in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one

another. Because of the importance of this issue, we devote a major portion of this chapter to **process synchronization** and **coordination** among cooperating processes.

6.2 The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so-called critical-section problem. Consider a system consisting of n processes $\{P_0, P_1, ..., P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Figure 6.1 General structure of a typical process.

3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the n processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (**kernel code**) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Another example is illustrated in Figure 6.2. In this situation, two processes, P_0 and P_1, are creating child processes using the `fork()` system call. Recall from Section 3.3.1 that `fork()` returns the process identifier of the newly created process to the parent process. In this example, there is a race condition on the variable kernel variable `next_available_pid` which represents the value of the next available process identifier. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence
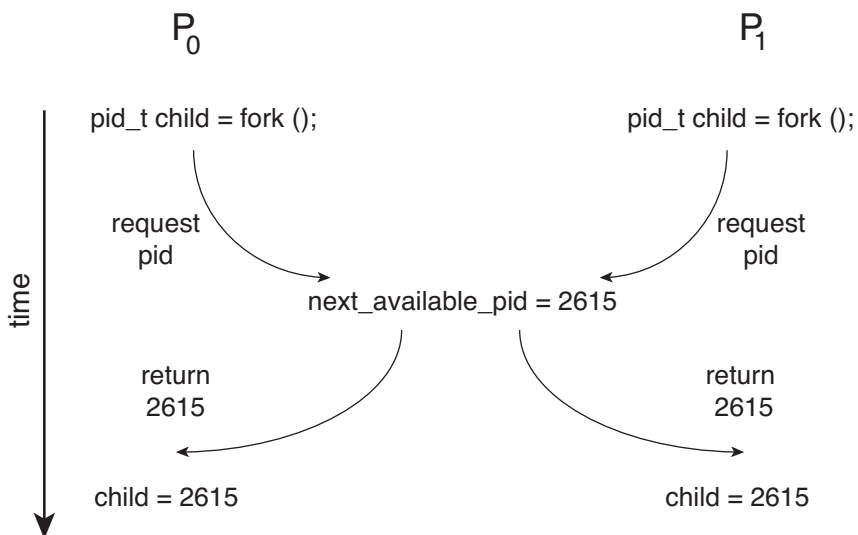

Figure 6.2 Race condition when assigning a pid.

of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different CPU cores.

Why, then, would anyone favor a preemptive kernel over a nonpreemptive one? A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. (Of course, this risk can also be minimized by designing kernel code that does not behave in this way.) Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

6.3   Peterson's Solution

Next, we illustrate a classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as `load` and `store`, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1. For convenience, when presenting P_i, we use P_j to denote the other process; that is, j equals $1 - i$.

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Figure 6.3   The structure of process P_i in Peterson's solution.

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process P_i is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 6.3.

To enter the critical section, process P_i first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.

2. The progress requirement is satisfied.

3. The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes—say, P_j—must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition `flag[j] == true` and `turn == j`; this loop is the only one possible. If P_j is not ready to enter the critical section, then `flag[j] == false`, and P_i can enter its critical section. If P_j has set `flag[j]` to `true` and is also executing in its `while` statement, then either `turn == i` or `turn == j`. If `turn == i`, then P_i will enter the critical section. If `turn == j`, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset `flag[j]` to `false`, allowing P_i to enter its critical section. If P_j resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus, since P_i does not change the value of the variable `turn` while executing the `while` statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

As mentioned at the beginning of this section, Peterson's solution is not guaranteed to work on modern computer architectures for the primary reason that, to improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies. For a single-threaded application, this reordering is immaterial as far as program correctness is concerned, as the final values are consistent with what is expected. (This is similar to balancing a checkbook—the actual order in which credit and debit operations are performed is unimportant, because the final balance will still be the same.) But for a multithreaded application with shared data, the reordering of instructions may render inconsistent or unexpected results.

As an example, consider the following data that are shared between two threads:

```
boolean flag = false;
int x = 0;
```

where Thread 1 performs the statements

```
while (!flag)
    ;
print x;
```

and Thread 2 performs

```
x = 100;
flag = true;
```

The expected behavior is, of course, that Thread 1 outputs the value 100 for variable x. However, as there are no data dependencies between the variables `flag` and `x`, it is possible that a processor may reorder the instructions for Thread 2 so that `flag` is assigned `true` before assignment of `x = 100`. In this situation, it is possible that Thread 1 would output 0 for variable x. Less obvious is that the processor may also reorder the statements issued by Thread 1 and load the variable x before loading the value of `flag`. If this were to occur, Thread 1 would output 0 for variable x even if the instructions issued by Thread 2 were not reordered.
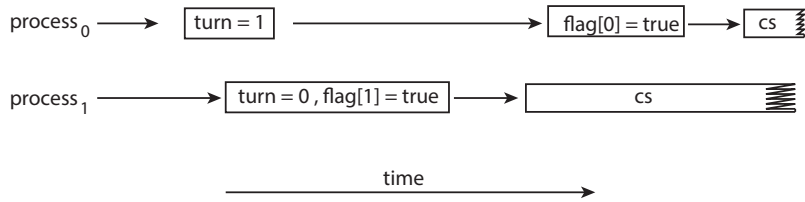
Figure 6.4   The effects of instruction reordering in Peterson's solution.

How does this affect Peterson's solution? Consider what happens if the assignments of the first two statements that appear in the entry section of Peterson's solution in Figure 6.3 are reordered; it is possible that both threads may be active in their critical sections at the same time, as shown in Figure 6.4.

As you will see in the following sections, the only way to preserve mutual exclusion is by using proper synchronization tools. Our discussion of these tools begins with primitive support in hardware and proceeds through abstract, high-level, software-based APIs available to both kernel developers and application programmers.

6.4   Hardware Support for Synchronization

We have just described one software-based solution to the critical-section problem. (We refer to it as a *software-based* solution because the algorithm involves no special support from the operating system or specific hardware instructions to ensure mutual exclusion.) However, as discussed, software-based solutions are not guaranteed to work on modern computer architectures. In this section, we present three hardware instructions that provide support for solving the critical-section problem. These primitive operations can be used directly as synchronization tools, or they can be used to form the foundation of more abstract synchronization mechanisms.

6.4.1   Memory Barriers

In Section 6.3, we saw that a system may reorder instructions, a policy that can lead to unreliable data states. How a computer architecture determines what memory guarantees it will provide to an application program is known as its **memory model**. In general, a memory model falls into one of two categories:

1. **Strongly ordered**, where a memory modification on one processor is immediately visible to all other processors.

2. **Weakly ordered**, where modifications to memory on one processor may not be immediately visible to other processors.

Memory models vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor. To address this issue, computer architectures provide instructions that can *force* any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to

threads running on other processors. Such instructions are known as **memory barriers** or **memory fences**. When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed. Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Let's return to our most recent example, in which reordering of instructions could have resulted in the wrong output, and use a memory barrier to ensure that we obtain the expected output.

If we add a memory barrier operation to Thread 1

```
while (!flag)
   memory_barrier();
print x;
```

we guarantee that the value of flag is loaded before the value of x.

Similarly, if we place a memory barrier between the assignments performed by Thread 2

```
x = 100;
memory_barrier();
flag = true;
```

we ensure that the assignment to x occurs before the assignment to flag.

With respect to Peterson's solution, we could place a memory barrier between the first two assignment statements in the entry section to avoid the reordering of operations shown in Figure 6.4. Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion.

6.4.2 Hardware Instructions

Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the test_and_set() and compare_and_swap() instructions.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 6.5 The definition of the atomic test_and_set() instruction.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 6.6   Mutual-exclusion implementation with `test_and_set()`.

The `test_and_set()` instruction can be defined as shown in Figure 6.5. The important characteristic of this instruction is that it is executed atomically. Thus, if two `test_and_set()` instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order. If the machine supports the `test_and_set()` instruction, then we can implement mutual exclusion by declaring a boolean variable `lock`, initialized to `false`. The structure of process P_i is shown in Figure 6.6.

The `compare_and_swap()` instruction (CAS), just like the `test_and_set()` instruction, operates on two words atomically, but uses a different mechanism that is based on swapping the content of two words.

The CAS instruction operates on three operands and is defined in Figure 6.7. The operand `value` is set to `new_value` only if the expression (`*value == expected`) is true. Regardless, CAS always returns the original value of the variable `value`. The important characteristic of this instruction is that it is executed atomically. Thus, if two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.

Mutual exclusion using CAS can be provided as follows: A global variable (`lock`) is declared and is initialized to 0. The first process that invokes `compare_and_swap()` will set `lock` to 1. It will then enter its critical section,

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 6.7   The definition of the atomic `compare_and_swap()` instruction.

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Figure 6.8   Mutual exclusion with the compare_and_swap() instruction.

because the original value of lock was equal to the expected value of 0. Subsequent calls to compare_and_swap() will not succeed, because lock now is not equal to the expected value of 0. When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section. The structure of process P_i is shown in Figure 6.8.

Although this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement. In Figure 6.9, we present

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = 0;
   else
      waiting[j] = false;

      /* remainder section */
}
```

Figure 6.9   Bounded-waiting mutual exclusion with compare_and_swap().

MAKING COMPARE-AND-SWAP ATOMIC

On Intel x86 architectures, the assembly language statement `cmpxchg` is used to implement the `compare_and_swap()` instruction. To enforce atomic execution, the `lock` prefix is used to lock the bus while the destination operand is being updated. The general form of this instruction appears as:

```
lock cmpxchg <destination operand>, <source operand>
```

---

another algorithm using the `compare_and_swap()` instruction that satisfies all the critical-section requirements. The common data structures are

```
boolean waiting[n];
int lock;
```

The elements in the `waiting` array are initialized to `false`, and `lock` is initialized to 0. To prove that the mutual-exclusion requirement is met, we note that process P_i can enter its critical section only if either `waiting[i] == false` or `key == 0`. The value of `key` can become 0 only if the `compare_and_swap()` is executed. The first process to execute the `compare_and_swap()` will find `key == 0`; all others must wait. The variable `waiting[i]` can become `false` only if another process leaves its critical section; only one `waiting[i]` is set to `false`, maintaining the mutual-exclusion requirement.

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets `lock` to 0 or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array `waiting` in the cyclic ordering ($i + 1$, $i + 2$, ..., $n - 1$, 0, ..., $i - 1$). It designates the first process in this ordering that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

Details describing the implementation of the atomic `test_and_set()` and `compare_and_swap()` instructions are discussed more fully in books on computer architecture.

6.4.3 Atomic Variables

Typically, the `compare_and_swap()` instruction is not used directly to provide mutual exclusion. Rather, it is used as a basic building block for constructing other tools that solve the critical-section problem. One such tool is an **atomic variable**, which provides atomic operations on basic data types such as integers and booleans. We know from Section 6.1 that incrementing or decrementing an integer value may produce a race condition. Atomic variables can be used in to ensure mutual exclusion in situations where there may be a data race on a single variable while it is being updated, as when a counter is incremented.

Most systems that support atomic variables provide special atomic data types as well as functions for accessing and manipulating atomic variables.

These functions are often implemented using `compare_and_swap()` operations. As an example, the following increments the atomic integer `sequence`:

    increment(&sequence);

where the `increment()` function is implemented using the CAS instruction:

```
void increment(atomic_int *v)
{
  int temp;

  do {
     temp = *v;
  }
  while (temp != compare_and_swap(v, temp, temp+1));
}
```

It is important to note that although atomic variables provide atomic updates, they do not entirely solve race conditions in all circumstances. For example, in the bounded-buffer problem described in Section 6.1, we could use an atomic integer for `count`. This would ensure that the updates to `count` were atomic. However, the producer and consumer processes also have `while` loops whose condition depends on the value of `count`. Consider a situation in which the buffer is currently empty and two consumers are looping while waiting for `count > 0`. If a producer entered one item in the buffer, both consumers could exit their `while` loops (as `count` would no longer be equal to 0) and proceed to consume, even though the value of `count` was only set to 1.

Atomic variables are commonly used in operating systems as well as concurrent applications, although their use is often limited to single updates of shared data such as counters and sequence generators. In the following sections, we explore more robust tools that address race conditions in more generalized situations.

6.5 Mutex Locks

The hardware-based solutions to the critical-section problem presented in Section 6.4 are complicated as well as generally inaccessible to application programmers. Instead, operating-system designers build higher-level software tools to solve the critical-section problem. The simplest of these tools is the **mutex lock**. (In fact, the term *mutex* is short for *mut*ual *ex*clusion.) We use the mutex lock to protect critical sections and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The `acquire()` function acquires the lock, and the `release()` function releases the lock, as illustrated in Figure 6.10.

A mutex lock has a boolean variable `available` whose value indicates if the lock is available or not. If the lock is available, a call to `acquire()` succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

```
while (true) {

    acquire lock

        critical section

    release lock

        remainder section

}
```

Figure 6.10 Solution to the critical-section problem using mutex locks.

The definition of `acquire()` is as follows:

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}
```

The definition of `release()` is as follows:

```
release() {
    available = true;
}
```

Calls to either `acquire()` or `release()` must be performed atomically. Thus, mutex locks can be implemented using the CAS operation described in Section 6.4, and we leave the description of this technique as an exercise.

LOCK CONTENTION

Locks are either contended or uncontended. A lock is considered **contended** if a thread blocks while trying to acquire the lock. If a lock is available when a thread attempts to acquire it, the lock is considered **uncontended**. Contended locks can experience either *high contention* (a relatively large number of threads attempting to acquire the lock) or *low contention* (a relatively small number of threads attempting to acquire the lock.) Unsurprisingly, highly contended locks tend to decrease overall performance of concurrent applications.

> ### *WHAT IS MEANT BY "SHORT DURATION"?*
>
> Spinlocks are often identified as the locking mechanism of choice on multi-processor systems when the lock is to be held for a short duration. But what exactly constitutes a *short duration*? Given that waiting on a lock requires two context switches—a context switch to move the thread to the waiting state and a second context switch to restore the waiting thread once the lock becomes available—the general rule is to use a spinlock if the lock will be held for a duration of less than two context switches.

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to `acquire()`. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU core is shared among many processes. Busy waiting also wastes CPU cycles that some other process might be able to use productively. (In Section 6.6, we examine a strategy that avoids busy waiting by temporarily putting the waiting process to sleep and then awakening it once the lock becomes available.)

The type of mutex lock we have been describing is also called a **spinlock** because the process "spins" while waiting for the lock to become available. (We see the same issue with the code examples illustrating the `compare_and_swap()` instruction.) Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. In certain circumstances on multicore systems, spinlocks are in fact the preferable choice for locking. If a lock is to be held for a short duration, one thread can "spin" on one processing core while another thread performs its critical section on another core. On modern multicore computing systems, spinlocks are widely used in many operating systems.

In Chapter 7 we examine how mutex locks can be used to solve classical synchronization problems. We also discuss how mutex locks and spinlocks are used in several operating systems, as well as in Pthreads.

6.6 Semaphores

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`. Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra, and such, the `wait()` operation was originally termed P (from the Dutch

proberen, "to test"); `signal()` was originally called V (from *verhogen,* "to increment"). The definition of `wait()` is as follows:

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

The definition of `signal()` is as follows:

```
signal(S) {
    S++;
}
```

All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed atomically. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of S (S $\leq$ 0), as well as its possible modification (S--), must be executed without interruption. We shall see how these operations can be implemented in Section 6.6.2. First, let's see how semaphores can be used.

6.6.1 Semaphore Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore `synch`, initialized to 0. In process P_1, we insert the statements

$$S_1;$$
```
signal(synch);
```

In process P_2, we insert the statements

```
wait(synch);
S₂;
```

Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after statement S_1 has been executed.

6.6.2 Semaphore Implementation

Recall that the implementation of mutex locks discussed in Section 6.5 suffers from busy waiting. The definitions of the wait() and signal() semaphore operations just described present the same problem. To overcome this problem, we can modify the definition of the wait() and signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can suspend itself. The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is suspended, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

Now, the wait() semaphore operation can be defined as

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                sleep();
        }
}
```

and the `signal()` semaphore operation can be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

The `sleep()` operation suspends the process that invokes it. The `wakeup(P)` operation resumes the execution of a suspended process P. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the `wait()` operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use any queuing strategy. Correct usage of semaphores does not depend on a particular queuing strategy for the semaphore lists.

As mentioned, it is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute `wait()` and `signal()` operations on the same semaphore at the same time. This is a critical-section problem, and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the `wait()` and `signal()` operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

In a multicore environment, interrupts must be disabled on every processing core. Otherwise, instructions from different processes (running on different cores) may be interleaved in some arbitrary way. Disabling interrupts on every core can be a difficult task and can seriously diminish performance. Therefore, SMP systems must provide alternative techniques—such as `compare_and_swap()` or spinlocks—to ensure that `wait()` and `signal()` are performed atomically.

It is important to admit that we have not completely eliminated busy waiting with this definition of the `wait()` and `signal()` operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the `wait()` and `signal()` operations, and these sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied, and busy waiting occurs

rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

6.7   Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place, and these sequences do not always occur.

We have seen an example of such errors in the use of a count in our solution to the producer–consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then the count value appeared to be reasonable—off by only 1. Nevertheless, the solution is obviously not an acceptable one. It is for this reason that mutex locks and semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur when either mutex locks or semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a binary semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we list several difficulties that may result. Note that these difficulties will arise even if a *single* process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a program interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

  signal(mutex);
  ...
  critical section
  ...
  wait(mutex);

  In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a program replaces signal(mutex) with wait(mutex). That is, it executes

  wait(mutex);
  ...
  critical section
  ...
  wait(mutex);

In this case, the process will permanently block on the second call to `wait()`, as the semaphore is now unavailable.

- Suppose that a process omits the `wait(mutex)`, or the `signal(mutex)`, or both. In this case, either mutual exclusion is violated or the process will permanently block.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores or mutex locks incorrectly to solve the critical-section problem. One strategy for dealing with such errors is to incorporate simple synchronization tools as high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the **monitor** type.

6.7.1 Monitor Usage

An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. A *monitor type* is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an

```
monitor monitor name
{
    /* shared variable declarations */

    function P1 ( . . . ) {
        . . .
    }

    function P2 ( . . . ) {
        . . .
    }


        .
        .
        .
    function Pn ( . . . ) {
        . . .
    }

    initialization_code ( . . . ) {
        . . .
    }
}
```

Figure 6.11 Pseudocode syntax of a monitor.

instance of that type, along with the bodies of functions that operate on those variables. The syntax of a monitor type is shown in Figure 6.11. The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 6.12). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the `condition` construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

```
condition x, y;
```

The only operations that can be invoked on a condition variable are `wait()` and `signal()`. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes
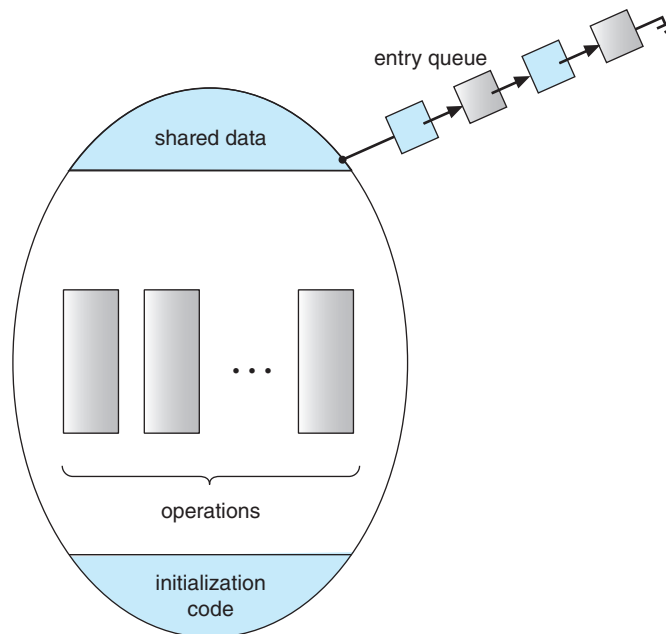
```
x.signal();
```


Figure 6.12   Schematic view of a monitor.

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed (Figure 6.13). Contrast this operation with the signal() operation associated with semaphores, which always affects the state of the semaphore.

Now suppose that, when the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that conceptually both processes can continue with their execution. Two possibilities exist:

1. **Signal and wait**. P either waits until Q leaves the monitor or waits for another condition.

2. **Signal and continue**. Q either waits until P leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since P was already executing in the monitor, the *signal-and-continue* method seems more reasonable. On the other, if we allow thread P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold. A compromise between these two choices exists as well: when thread P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.
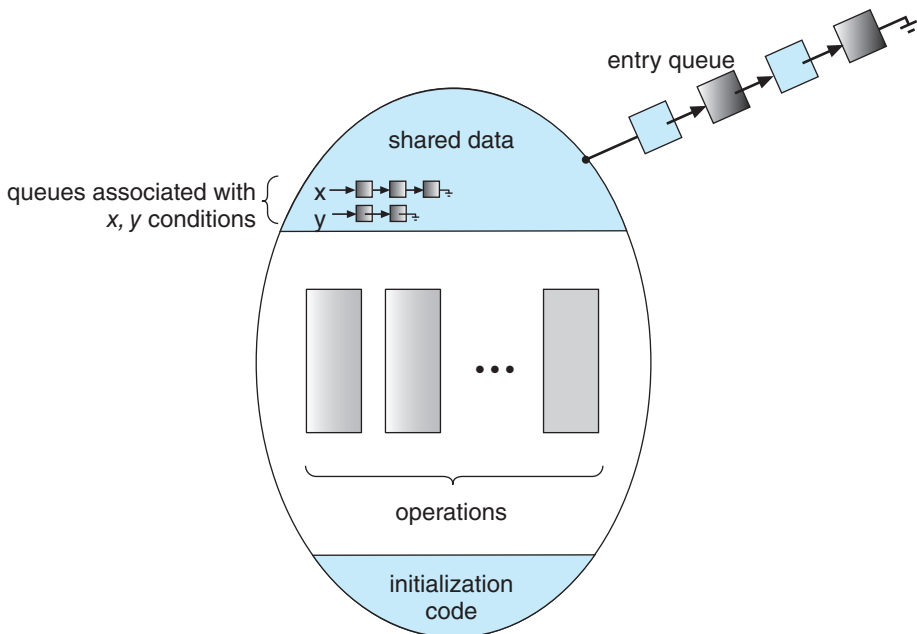

Figure 6.13 Monitor with condition variables.

Many programming languages have incorporated the idea of the monitor as described in this section, including Java and C#. Other languages—such as Erlang—provide concurrency support using a similar mechanism.

6.7.2 Implementing a Monitor Using Semaphores

We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a binary semaphore `mutex` (initialized to 1) is provided to ensure mutual exclusion. A process must execute `wait(mutex)` before entering the monitor and must execute `signal(mutex)` after leaving the monitor.

We will use the signal-and-wait scheme in our implementation. Since a signaling process must wait until the resumed process either leaves or waits, an additional binary semaphore, `next`, is introduced, initialized to 0. The signaling processes can use `next` to suspend themselves. An integer variable `next_count` is also provided to count the number of processes suspended on `next`. Thus, each external function F is replaced by

```
wait(mutex);
    ...
    body of F
    ...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented as well. For each condition x, we introduce a binary semaphore `x_sem` and an integer variable `x_count`, both initialized to 0. The operation `x.wait()` can now be implemented as

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

The operation `x.signal()` can be implemented as

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen (see the bibliographical notes at the end of the chapter). In some cases, however, the generality of the implementation is

```
monitor ResourceAllocator
{
  boolean busy;
  condition x;

  void acquire(int time) {
    if (busy)
      x.wait(time);
    busy = true;
  }

  void release() {
    busy = false;
    x.signal();
  }

  initialization_code() {
    busy = false;
  }
}
```

Figure 6.14   A monitor to allocate a single resource.

unnecessary, and a significant improvement in efficiency is possible. We leave this problem to you in Exercise 6.27.

6.7.3   Resuming Processes within a Monitor

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. In these circumstances, the **conditional-wait** construct can be used. This construct has the form

$$x.wait(c);$$

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

To illustrate this new mechanism, consider the ResourceAllocator monitor shown in Figure 6.14, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation

request. A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
    ...
  access the resource;
    ...
R.release();
```

where R is an instance of type `ResourceAllocator`.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.

- A process might never release a resource once it has been granted access to the resource.

- A process might attempt to release a resource that it never requested.

- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource-access operations within the `ResourceAllocator` monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the `ResourceAllocator` monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.

Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. This access-control problem can be solved only through the use of the additional mechanisms that are described in Chapter 17.

6.8 Liveness

One consequence of using synchronization tools to coordinate access to critical sections is the possibility that a process attempting to enter its critical section will wait indefinitely. Recall that in Section 6.2, we outlined three criteria that solutions to the critical-section problem must satisfy. Indefinite waiting violates two of these—the progress and bounded-waiting criteria.

Liveness refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle. A process waiting indefinitely under the circumstances just described is an example of a "liveness failure."

There are many different forms of liveness failure; however, all are generally characterized by poor performance and responsiveness. A very simple example of a liveness failure is an infinite loop. A busy wait loop presents the *possibility* of a liveness failure, especially if a process may loop an arbitrarily long period of time. Efforts at providing mutual exclusion using tools such as mutex locks and semaphores can often lead to such failures in concurrent programming. In this section, we explore two situations that can lead to liveness failures.

6.8.1 Deadlock

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

$$P_0 \qquad\qquad P_1$$

```
wait(S);        wait(Q);
wait(Q);        wait(S);
    .               .
    .               .
    .               .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Suppose that P_0 executes `wait(S)` and then P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked.

We say that a set of processes is in a deadlocked state when *every process in the set is waiting for an event that can be caused only by another process in the set*. The "events" with which we are mainly concerned here are the acquisition and release of resources such as mutex locks and semaphores. Other types of events may result in deadlocks, as we show in more detail in Chapter 8. In

that chapter, we describe various mechanisms for dealing with the deadlock problem, as well as other forms of liveness failures.

6.8.2 Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

As an example, assume we have three processes—L, M, and H—whose priorities follow the order $L < M < H$. Assume that process H requires a semaphore S, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource S. However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority—process M—has affected how long process H must wait for L to relinquish resource S.

This liveness problem is known as **priority inversion**, and it can occur only in systems with more than two priorities. Typically, priority inversion is avoided by implementing a **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L had finished using resource S, it would relinquish its inherited priority from H and assume its original priority. Because resource S would now be available, process H—not M—would run next.

6.9 Evaluation

We have described several different synchronization tools that can be used to solve the critical-section problem. Given correct implementation and usage, these tools can be used effectively to ensure mutual exclusion as well as address liveness issues. With the growth of concurrent programs that leverage the power of modern multicore computer systems, increasing attention is being paid to the performance of synchronization tools. Trying to identify when to use which tool, however, can be a daunting challenge. In this section, we present some simple strategies for determining when to use specific synchronization tools.

The hardware solutions outlined in Section 6.4 are considered very low level and are typically used as the foundations for constructing other synchronization tools, such as mutex locks. However, there has been a recent focus on using the CAS instruction to construct **lock-free** algorithms that provide protection from race conditions without requiring the overhead of locking. Although these lock-free solutions are gaining popularity due to low overhead

> ### *PRIORITY INVERSION AND THE MARS PATHFINDER*
>
> Priority inversion can be more than a scheduling inconvenience. On systems with tight time constraints—such as real-time systems—priority inversion can cause a process to take longer than it should to accomplish a task. When that happens, other failures can cascade, resulting in system failure.
>
> Consider the Mars Pathfinder, a NASA space probe that landed a robot, the Sojourner rover, on Mars in 1997 to conduct experiments. Shortly after the Sojourner began operating, it started to experience frequent computer resets. Each reset reinitialized all hardware and software, including communications. If the problem had not been solved, the Sojourner would have failed in its mission.
>
> The problem was caused by the fact that one high-priority task, "bc_dist," was taking longer than expected to complete its work. This task was being forced to wait for a shared resource that was held by the lower-priority "ASI/MET" task, which in turn was preempted by multiple medium-priority tasks. The "bc_dist" task would stall waiting for the shared resource, and ultimately the "bc_sched" task would discover the problem and perform the reset. The Sojourner was suffering from a typical case of priority inversion.
>
> The operating system on the Sojourner was the VxWorks real-time operating system, which had a global variable to enable priority inheritance on all semaphores. After testing, the variable was set on the Sojourner (on Mars!), and the problem was solved.
>
> A full description of the problem, its detection, and its solution was written by the software team lead and is available at http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html.

and ability to scale, the algorithms themselves are often difficult to develop and test. (In the exercises at the end of this chapter, we ask you to evaluate the correctness of a lock-free stack.)

CAS-based approaches are considered an *optimistic* approach—you optimistically first update a variable and then use collision detection to see if another thread is updating the variable concurrently. If so, you repeatedly retry the operation until it is successfully updated without conflict. Mutual-exclusion locking, in contrast, is considered a *pessimistic* strategy; you assume another thread is concurrently updating the variable, so you pessimistically acquire the lock before making any updates.

The following guidelines identify general rules concerning performance differences between CAS-based synchronization and traditional synchronization (such as mutex locks and semaphores) under varying contention loads:

- **Uncontended**. Although both options are generally fast, CAS protection will be somewhat faster than traditional synchronization.

- **Moderate contention**. CAS protection will be faster—possibly much faster—than traditional synchronization.

- **High contention**. Under very highly contended loads, traditional synchronization will ultimately be faster than CAS-based synchronization.

Moderate contention is particularly interesting to examine. In this scenario, the CAS operation succeeds most of the time, and when it fails, it will iterate through the loop shown in Figure 6.8 only a few times before ultimately succeeding. By comparison, with mutual-exclusion locking, *any* attempt to acquire a contended lock will result in a more complicated—and time-intensive—code path that suspends a thread and places it on a wait queue, requiring a context switch to another thread.

The choice of a mechanism that addresses race conditions can also greatly affect system performance. For example, atomic integers are much lighter weight than traditional locks, and are generally more appropriate than mutex locks or semaphores for single updates to shared variables such as counters. We also see this in the design of operating systems where spinlocks are used on multiprocessor systems when locks are held for short durations. In general, mutex locks are simpler and require less overhead than semaphores and are preferable to binary semaphores for protecting access to a critical section. However, for some uses—such as controlling access to a finite number of resources—a counting semaphore is generally more appropriate than a mutex lock. Similarly, in some instances, a reader–writer lock may be preferred over a mutex lock, as it allows a higher degree of concurrency (that is, multiple readers).

The appeal of higher-level tools such as monitors and condition variables is based on their simplicity and ease of use. However, such tools may have significant overhead and, depending on their implementation, may be less likely to scale in highly contended situations.

Fortunately, there is much ongoing research toward developing scalable, efficient tools that address the demands of concurrent programming. Some examples include:

- Designing compilers that generate more efficient code.

- Developing languages that provide support for concurrent programming.

- Improving the performance of existing libraries and APIs.

In the next chapter, we examine how various operating systems and APIs available to developers implement the synchronization tools presented in this chapter.

6.10 Summary

- A race condition occurs when processes have concurrent access to shared data and the final result depends on the particular order in which concurrent accesses occur. Race conditions can result in corrupted values of shared data.

- A critical section is a section of code where shared data may be manipulated and a possible race condition may occur. The critical-section problem

is to design a protocol whereby processes can synchronize their activity to cooperatively share data.

- A solution to the critical-section problem must satisfy the following three requirements: (1) mutual exclusion, (2) progress, and (3) bounded waiting. Mutual exclusion ensures that only one process at a time is active in its critical section. Progress ensures that programs will cooperatively determine what process will next enter its critical section. Bounded waiting limits how much time a program will wait before it can enter its critical section.

- Software solutions to the critical-section problem, such as Peterson's solution, do not work well on modern computer architectures.

- Hardware support for the critical-section problem includes memory barriers; hardware instructions, such as the compare-and-swap instruction; and atomic variables.

- A mutex lock provides mutual exclusion by requiring that a process acquire a lock before entering a critical section and release the lock on exiting the critical section.

- Semaphores, like mutex locks, can be used to provide mutual exclusion. However, whereas a mutex lock has a binary value that indicates if the lock is available or not, a semaphore has an integer value and can therefore be used to solve a variety of synchronization problems.

- A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.

- Solutions to the critical-section problem may suffer from liveness problems, including deadlock.

- The various tools that can be used to solve the critical-section problem as well as to synchronize the activity of processes can be evaluated under varying levels of contention. Some tools work better under certain contention loads than others.

Practice Exercises

6.1 In Section 6.4, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur and how such effects can be minimized.

6.2 What is the meaning of the term *busy waiting*? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

6.3 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

6.4 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

6.5   Illustrate how a binary semaphore can be used to implement mutual exclusion among *n* processes.

6.6   Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the `amount` that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function, and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Further Reading

The mutual-exclusion problem was first discussed in a classic paper by [Dijkstra (1965)]. The semaphore concept was suggested by [Dijkstra (1965)]. The monitor concept was developed by [Brinch-Hansen (1973)]. [Hoare (1974)] gave a complete description of the monitor.

For more on the Mars Pathfinder problem see http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

A thorough discussion of memory barriers and cache memory is presented in [Mckenney (2010)]. [Herlihy and Shavit (2012)] presents details on several issues related to multiprocessor programming, including memory models and compare-and-swap instructions. [Bahra (2013)] examines nonblocking algorithms on modern multicore systems.

Bibliography

[Bahra (2013)]   S. A. Bahra, "Nonblocking Algorithms and Scalable Multicore Programming", *ACM queue*, Volume 11, Number 5 (2013).

[Brinch-Hansen (1973)]   P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).

[Dijkstra (1965)]   E. W. Dijkstra, "Cooperating Sequential Processes", Technical report, Technological University, Eindhoven, the Netherlands (1965).

[Herlihy and Shavit (2012)]   M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming,* Revised First Edition, Morgan Kaufmann Publishers Inc. (2012).

[Hoare (1974)]   C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Volume 17, Number 10 (1974), pages 549–557.

[Mckenney (2010)]   P. E. Mckenney, "Memory Barriers: a Hardware View for Software Hackers" (2010).

Chapter 6 Exercises

6.7 The pseudocode of Figure 6.15 illustrates the basic push() and pop() operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:

   a. What data have a race condition?

   b. How could the race condition be fixed?

6.8 Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
   if (amount > highestBid)
      highestBid = amount;
}
```

```
push(item) {
   if (top < SIZE) {
      stack[top] = item;
      top++;
   }
   else
      ERROR
}

pop() {
   if (!is_empty()) {
      top--;
      return stack[top];
   }
   else
      ERROR
}

is_empty() {
   if (top == 0)
      return true;
   else
      return false;
}
```

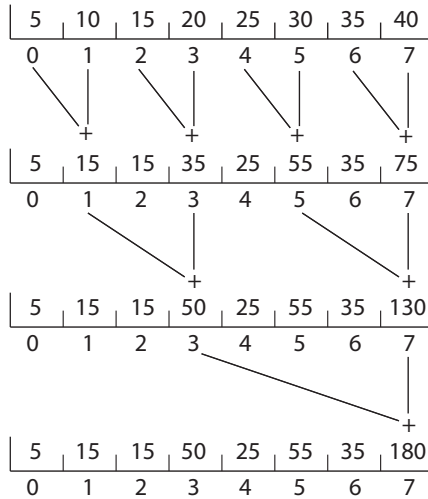**Figure 6.16** Array-based stack for Exercise 6.12.

Figure 6.17 Summing an array as a series of partial sums for Exercise 6.14.

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

6.9 The following program example can be used to sum the array `values` of size N elements in parallel on a system containing N computing cores (there is a separate processor for each array element):

```
for j = 1 to log_2(N) {
    for k = 1 to N {
        if ((k + 1) % pow(2,j) == 0) {
            values[k] += values[k - pow(2,(j-1))]
        }
    }
}
```

This has the effect of summing the elements in the array as a series of partial sums, as shown in Figure 6.16. After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and illustrate with an example. If not, demonstrate why this algorithm is free from race conditions.

6.10 The `compare_and_swap()` instruction can be used to design lock-free data structures such as stacks, queues, and lists. The program example shown in Figure 6.17 presents a possible solution to a lock-free stack using CAS instructions, where the stack is represented as a linked list of `Node` elements with `top` representing the top of the stack. Is this implementation free from race conditions?

```
typedef struct node {
   value_t data;
   struct node *next;
} Node;

Node *top; // top of stack

void push(value_t item) {
   Node *old_node;
   Node *new_node;

   new_node = malloc(sizeof(Node));
   new_node->data = item;

   do {
      old_node = top;
      new_node->next = old_node;
   }
   while (compare_and_swap(top,old_node,new_node) != old_node);
}

value_t pop() {
   Node *old_node;
   Node *new_node;

   do {
      old_node = top;
      if (old_node == NULL)
         return NULL;
      new_node = old_node->next;
   }
   while (compare_and_swap(top,old_node,new_node) != old_node);

   return old_node->data;
}
```

Figure 6.18 Lock-free stack for Exercise 6.15.

6.11 One approach for using `compare_and_swap()` for implementing a spin-lock is as follows:

```
void lock_spinlock(int *lock) {
   while (compare_and_swap(lock, 0, 1) != 0)
      ; /* spin */
}
```

A suggested alternative approach is to use the "compare and compare-and-swap" idiom, which checks the status of the lock before invoking the

compare_and_swap() operation. (The rationale behind this approach is to invoke compare_and_swap()only if the lock is currently available.) This strategy is shown below:

```
void lock_spinlock(int *lock) {
{
   while (true) {
      if (*lock == 0) {
         /* lock appears to be available */

         if (!compare_and_swap(lock, 0, 1))
            break;
      }
   }
}
```

Does this "compare and compare-and-swap" idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

6.12 Some semaphore implementations provide a function getValue() that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling wait() so that a process will only call wait() if the value of the semaphore is > 0, thereby preventing blocking while waiting for the semaphore. For example:

```
if (getValue(&sem) > 0)
   wait(&sem);
```

Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function getValue() in this scenario.

6.13 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i (i == 0 or 1) is shown in Figure 6.18. The other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.14 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

```
                  while (true) {
                    flag[i] = true;

                    while (flag[j]) {
                      if (turn == j) {
                        flag[i] = false;
                        while (turn == j)
                          ; /* do nothing */
                        flag[i] = true;
                      }
                    }

                      /* critical section */

                    turn = j;
                    flag[i] = false;

                      /* remainder section */
                  }
```

Figure 6.19   The structure of process P_i in Dekker's algorithm.

All the elements of flag are initially idle. The initial value of turn is immaterial (between 0 and n−1). The structure of process P_i is shown in Figure 6.19. Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.15 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

6.16 Consider how to implement a mutex lock using the compare_and_swap() instruction. Assume that the following structure defining the mutex lock is available:

```
                    typedef struct {
                        int available;
                    } lock;
```

The value (available == 0) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this struct, illustrate how the following functions can be implemented using the compare_and_swap() instruction:

- void acquire(lock *mutex)
- void release(lock *mutex)

Be sure to include any initialization that may be necessary.

```
while (true) {
  while (true) {
    flag[i] = want_in;
    j = turn;

    while (j != i) {
      if (flag[j] != idle) {
        j = turn;
      else
        j = (j + 1) % n;
    }

    flag[i] = in_cs;
    j = 0;

    while ( (j < n) && (j == i || flag[j] != in_cs))
      j++;

    if ( (j >= n) && (turn == i || flag[turn] == idle))
      break;
  }

    /* critical section */

  j = (turn + 1) % n;

  while (flag[j] == idle)
    j = (j + 1) % n;

  turn = j;
  flag[i] = idle;

    /* remainder section */
}
```

Figure 6.20   The structure of process P_i in Eisenberg and McGuire's algorithm.

6.17 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

6.18 The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

6.19 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a

spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock.

6.20 Assume that a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

6.21 A multithreaded web server wishes to keep track of the number of requests it services (known as *hits*). Consider the two following strategies to prevent a race condition on the variable `hits`. The first strategy is to use a basic mutex lock when updating `hits`:

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

6.22 Consider the code example for allocating and releasing processes shown in Figure 6.20.

a. Identify the race condition(s).

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

```
int number_of_processes = 0
```

with the atomic integer

```
atomic_t number_of_processes = 0
```

to prevent the race condition(s)?

6.23 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;

   if (number_of_processes == MAX_PROCESSES)
      return -1;
   else {
      /* allocate necessary process resources */
      ++number_of_processes;

      return new_pid;
   }
}

/* the implementation of exit() calls this function */
void release_process() {
   /* release process resources */
   --number_of_processes;
}
```

Figure 6.21   Allocating and releasing processes for Exercise 6.27.

released. Illustrate how semaphores can be used by a server to limit the number of concurrent connections.

6.24 In Section 6.7, we use the following illustration as an incorrect use of semaphores to solve the critical-section problem:

<div align="center">

wait(mutex);

...

critical section

...

wait(mutex);

</div>

Explain why this is an example of a liveness failure.

6.25 Demonstrate that monitors and semaphores are equivalent to the degree that they can be used to implement solutions to the same types of synchronization problems.

6.26 Describe how the signal() operation associated with monitors differs from the corresponding operation defined for semaphores.

6.27 Suppose the signal() statement can appear only as the last statement in a monitor function. Suggest how the implementation described in Section 6.7 can be simplified in this situation.

6.28 Consider a system consisting of processes $P_1, P_2, ..., P_n$, each of which has a unique priority number. Write a monitor that allocates three identical printers to these processes, using the priority numbers for deciding the order of allocation.

6.29 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than n. Write a monitor to coordinate access to the file.

6.30 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways in which signaling can be performed?

6.31 Design an algorithm for a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a function `tick()` in your monitor at regular intervals.

6.32 Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

Programming Problems

6.33   Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and will return them once finished. As an example, many commercial software packages provide a given number of *licenses*, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such a request will be granted only when an existing license holder terminates the application and a license is returned.

The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the decrease_count() function:

```
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
  if (available_resources < count)
    return -1;
  else {
    available_resources -= count;

    return 0;
  }
}
```

When a process wants to return a number of resources, it calls the increase_count() function:

```
/* increase available_resources by count */
int increase_count(int count) {
  available_resources += count;

  return 0;
}
```

The preceding program segment produces a race condition. Do the following:

a.   Identify the data involved in the race condition.

    b.   Identify the location (or locations) in the code where the race condition occurs.

    c.   Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the `decrease_count()` function so that the calling process is blocked until sufficient resources are available.

6.34 The `decrease_count()` function in the previous exercise currently returns 0 if sufficient resources are available and −1 otherwise. This leads to awkward programming for a process that wishes to obtain a number of resources:

```
while (decrease_count(count) == -1)
    ;
```

Rewrite the resource-manager code segment using a monitor and condition variables so that the `decrease_count()` function suspends the process until sufficient resources are available. This will allow a process to invoke `decrease_count()` by simply calling

```
decrease_count(count);
```

The process will return from this function call only when sufficient resources are available.

Synchronization Examples

In Chapter 6, we presented the critical-section problem and focused on how race conditions can occur when multiple concurrent processes share data. We went on to examine several tools that address the critical-section problem by preventing race conditions from occurring. These tools ranged from low-level hardware solutions (such as memory barriers and the compare-and-swap operation) to increasingly higher-level tools (from mutex locks to semaphores to monitors). We also discussed various challenges in designing applications that are free from race conditions, including liveness hazards such as deadlocks. In this chapter, we apply the tools presented in Chapter 6 to several classic synchronization problems. We also explore the synchronization mechanisms used by the Linux, UNIX, and Windows operating systems, and we describe API details for both Java and POSIX systems.

CHAPTER OBJECTIVES

- Explain the bounded-buffer, readers–writers, and dining–philosophers synchronization problems.
- Describe specific tools used by Linux and Windows to solve process synchronization problems.
- Illustrate how POSIX and Java can be used to solve process synchronization problems.
- Design and develop solutions to process synchronization problems using POSIX and Java APIs.

7.1 Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization, since that is the

```
while (true) {
    . . .
    /* produce an item in next_produced */
    . . .
    wait(empty);
    wait(mutex);
    . . .
    /* add next_produced to the buffer */
    . . .
    signal(mutex);
    signal(full);
}
```

Figure 7.1   The structure of the producer process.

traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

7.1.1 The Bounded-Buffer Problem

The *bounded-buffer problem* was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation. We provide a related programming project in the exercises at the end of the chapter.

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of n buffers, each capable of holding one item. The mutex binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is shown in Figure 7.1, and the code for the consumer process is shown in Figure 7.2. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

7.1.2 The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, read and write) the database. We distinguish

```
while (true) {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
}
```

Figure 7.2   The structure of the consumer process.

between these two types of processes by referring to the former as *readers* and to the latter as *writers*. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem**. Since it was originally stated, it has been used to test nearly every new synchronization primitive.

The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers–writers problem. See the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers–writers problem.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

The binary semaphores mutex and rw_mutex are initialized to 1; read_count is a counting semaphore initialized to 0. The semaphore rw_mutex

```
while (true) {
  wait(rw_mutex);
    . . .
  /* writing is performed */
    . . .
  signal(rw_mutex);
}
```

Figure 7.3 The structure of a writer process.

is common to both reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `read_count` is updated. The `read_count` variable keeps track of how many processes are currently reading the object. The semaphore `rw_mutex` functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers that enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 7.3; the code for a reader process is shown in Figure 7.4. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on `rw_mutex`, and $n-1$ readers are queued on `mutex`. Also observe that, when a writer executes `signal(rw_mutex)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers–writers problem and its solutions have been generalized to provide **reader–writer** locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either *read* or *write* access. When a

```
while (true) {
  wait(mutex);
  read_count++;
  if (read_count == 1)
    wait(rw_mutex);
  signal(mutex);
    . . .
  /* reading is performed */
    . . .
  wait(mutex);
  read_count--;
  if (read_count == 0)
    signal(rw_mutex);
  signal(mutex);
}
```

Figure 7.4 The structure of a reader process.

process wishes only to read shared data, it requests the reader–writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.

- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock.

7.1.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 7.5). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

The **dining-philosophers problem** is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need
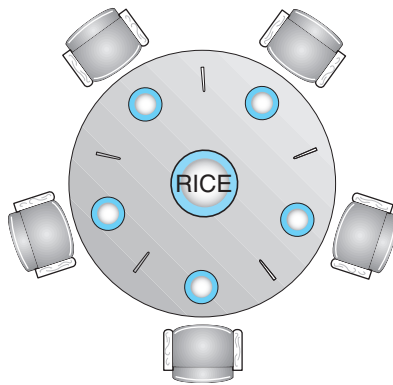

Figure 7.5 The situation of the dining philosophers.

```
while (true) {
  wait(chopstick[i]);
  wait(chopstick[(i+1) % 5]);

    . . .
  /* eat for a while */
    . . .
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);

    . . .
  /* think for awhile */
    . . .
}
```

Figure 7.6 The structure of philosopher i.

to allocate several resources among several processes in a deadlock-free and starvation-free manner.

7.1.3.1 Semaphore Solution

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore. She releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

                   semaphore chopstick[5];

where all the elements of `chopstick` are initialized to 1. The structure of philosopher i is shown in Figure 7.6.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of `chopstick` will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are the following:

- Allow at most four philosophers to be sitting simultaneously at the table.

- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).

- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

In Section 6.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks. Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility

that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

7.1.3.2 Monitor Solution

Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher i can set the variable `state[i]` = `EATING` only if her two neighbors are not eating: (`state[(i+4) % 5]` != `EATING`) and (`state[(i+1) % 5]` != `EATING`).

We also need to declare

```
condition self[5];
```

This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor `DiningPhilosophers`, whose definition is shown in Figure 7.7. Each philosopher, before starting to eat, must invoke the operation `pickup()`. This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the `putdown()` operation. Thus, philosopher i must invoke the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);
            ...
           eat
            ...
DiningPhilosophers.putdown(i);
```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. As we already noted, however, it is possible for a philosopher to starve to death. We do not present a solution to this problem but rather leave it as an exercise for you.

7.2 Synchronization within the Kernel

We next describe the synchronization mechanisms provided by the Windows and Linux operating systems. These two operating systems provide good examples of different approaches to synchronizing the kernel, and as you will

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
   }
}
```

Figure 7.7   A monitor solution to the dining-philosophers problem.

see, the synchronization mechanisms available in these systems differ in subtle
yet significant ways.

7.2.1 Synchronization in Windows

The Windows operating system is a multithreaded kernel that provides sup-
port for real-time applications and multiple processors. When the Windows
kernel accesses a global resource on a single-processor system, it temporar-
ily masks interrupts for all interrupt handlers that may also access the global
resource. On a multiprocessor system, Windows protects access to global
resources using spinlocks, although the kernel uses spinlocks only to protect
short code segments. Furthermore, for reasons of efficiency, the kernel ensures
that a thread will never be preempted while holding a spinlock.

For thread synchronization outside the kernel, Windows provides **dispatcher objects**. Using a dispatcher object, threads synchronize according to several different mechanisms, including mutex locks, semaphores, events, and timers. The system protects shared data by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished. Semaphores behave as described in Section 6.6. **Events** are similar to condition variables; that is, they may notify a waiting thread when a desired condition occurs. Finally, timers are used to notify one (or more than one) thread that a specified amount of time has expired.

Dispatcher objects may be in either a signaled state or a nonsignaled state. An object in a **signaled state** is available, and a thread will not block when acquiring the object. An object in a **nonsignaled state** is not available, and a thread will block when attempting to acquire the object. We illustrate the state transitions of a mutex lock dispatcher object in Figure 7.8.

A relationship exists between the state of a dispatcher object and the state of a thread. When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting, and the thread is placed in a waiting queue for that object. When the state for the dispatcher object moves to signaled, the kernel checks whether any threads are waiting on the object. If so, the kernel moves one thread—or possibly more—from the waiting state to the ready state, where they can resume executing. The number of threads the kernel selects from the waiting queue depends on the type of dispatcher object for which each thread is waiting. The kernel will select only one thread from the waiting queue for a mutex, since a mutex object may be "owned" by only a single thread. For an event object, the kernel will select all threads that are waiting for the event.

We can use a mutex lock as an illustration of dispatcher objects and thread states. If a thread tries to acquire a mutex dispatcher object that is in a nonsignaled state, that thread will be suspended and placed in a waiting queue for the mutex object. When the mutex moves to the signaled state (because another thread has released the lock on the mutex), the thread waiting at the front of the queue will be moved from the waiting state to the ready state and will acquire the mutex lock.

A **critical-section object** is a user-mode mutex that can often be acquired and released without kernel intervention. On a multiprocessor system, a critical-section object first uses a spinlock while waiting for the other thread to release the object. If it spins too long, the acquiring thread will then allocate a kernel mutex and yield its CPU. Critical-section objects are particularly efficient because the kernel mutex is allocated only when there is contention for the object. In practice, there is very little contention, so the savings are significant.
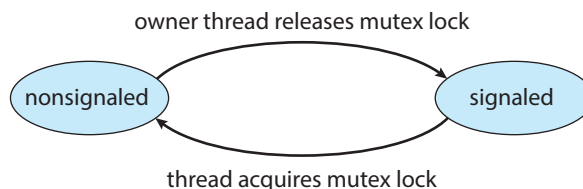
owner thread releases mutex lock

nonsignaled   signaled

thread acquires mutex lock

Figure 7.8 Mutex dispatcher object.

We provide a programming project at the end of this chapter that uses mutex locks and semaphores in the Windows API.

7.2.2 Synchronization in Linux

Prior to Version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. Now, however, the Linux kernel is fully preemptive, so a task can be preempted when it is running in the kernel.

Linux provides several different mechanisms for synchronization in the kernel. As most computer architectures provide instructions for atomic versions of simple math operations, the simplest synchronization technique within the Linux kernel is an atomic integer, which is represented using the opaque data type `atomic_t`. As the name implies, all math operations using atomic integers are performed without interruption. To illustrate, consider a program that consists of an atomic integer `counter` and an integer `value`.

```
atomic_t counter;
int value;
```

The following code illustrates the effect of performing various atomic operations:

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Atomic integers are particularly efficient in situations where an integer variable—such as a counter—needs to be updated, since atomic operations do not require the overhead of locking mechanisms. However, their use is limited to these sorts of scenarios. In situations where there are several variables contributing to a possible race condition, more sophisticated locking tools must be used.

Mutex locks are available in Linux for protecting critical sections within the kernel. Here, a task must invoke the `mutex_lock()` function prior to entering a critical section and the `mutex_unlock()` function after exiting the critical section. If the mutex lock is unavailable, a task calling `mutex_lock()` is put into a sleep state and is awakened when the lock's owner invokes `mutex_unlock()`.

Linux also provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, such as embedded systems with only a single processing core, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on systems with a single processing core, rather than holding a spinlock, the kernel disables kernel preemption; and rather than releasing the spinlock, it enables kernel preemption. This is summarized below:

Single Processor	Multiple Processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

In the Linux kernel, both spinlocks and mutex locks are *nonrecursive*, which means that if a thread has acquired one of these locks, it cannot acquire the same lock a second time without first releasing the lock. Otherwise, the second attempt at acquiring the lock will block.

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple system calls—preempt_disable() and preempt_enable()—for disabling and enabling kernel preemption. The kernel is not preemptible, however, if a task running in the kernel is holding a lock. To enforce this rule, each task in the system has a thread-info structure containing a counter, preempt_count, to indicate the number of locks being held by the task. When a lock is acquired, preempt_count is incremented. It is decremented when a lock is released. If the value of preempt_count for the task currently running in the kernel is greater than 0, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is 0, the kernel can safely be interrupted (assuming there are no outstanding calls to preempt_disable()).

Spinlocks—along with enabling and disabling kernel preemption—are used in the kernel only when a lock (or disabling kernel preemption) is held for a short duration. When a lock must be held for a longer period, semaphores or mutex locks are appropriate for use.

7.3 POSIX Synchronization

The synchronization methods discussed in the preceding section pertain to synchronization within the kernel and are therefore available only to kernel developers. In contrast, the POSIX API is available for programmers at the user level and is not part of any particular operating-system kernel. (Of course, it must ultimately be implemented using tools provided by the host operating system.)

In this section, we cover mutex locks, semaphores, and condition variables that are available in the Pthreads and POSIX APIs. These APIs are widely used for thread creation and synchronization by developers on UNIX, Linux, and macOS systems.

7.3.1 POSIX Mutex Locks

Mutex locks represent the fundamental synchronization technique used with Pthreads. A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section. Pthreads uses the pthread_mutex_t data type for mutex locks. A mutex is created with the pthread_mutex_init() function. The first parameter is a pointer to the mutex. By passing NULL as a second parameter, we initialize the mutex to its default attributes. This is illustrated below:

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

The mutex is acquired and released with the pthread_mutex_lock() and pthread_mutex_unlock() functions. If the mutex lock is unavailable when pthread_mutex_lock() is invoked, the calling thread is blocked until the owner invokes pthread_mutex_unlock(). The following code illustrates protecting a critical section with mutex locks:

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

7.3.2 POSIX Semaphores

Many systems that implement Pthreads also provide semaphores, although semaphores are not part of the POSIX standard and instead belong to the POSIX SEM extension. POSIX specifies two types of semaphores—**named** and **unnamed**. Fundamentally, the two are quite similar, but they differ in terms of how they are created and shared between processes. Because both techniques are common, we discuss both here. Beginning with Version 2.6 of the kernel, Linux systems provide support for both named and unnamed semaphores.

7.3.2.1 POSIX Named Semaphores

The function sem_open() is used to create and open a POSIX named sempahore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

In this instance, we are naming the semaphore SEM. The O_CREAT flag indicates that the semaphore will be created if it does not already exist. Additionally, the semaphore has read and write access for other processes (via the parameter 0666) and is initialized to 1.

The advantage of named semaphores is that multiple unrelated processes can easily use a common semaphore as a synchronization mechanism by

simply referring to the semaphore's name. In the example above, once the semaphore SEM has been created, subsequent calls to `sem_open()` (with the same parameters) by other processes return a descriptor to the existing semaphore.

In Section 6.6, we described the classic `wait()` and `signal()` semaphore operations. POSIX declares these operations `sem_wait()` and `sem_post()`, respectively. The following code sample illustrates protecting a critical section using the named semaphore created above:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

Both Linux and macOS systems provide POSIX named semaphores.

7.3.2.2 POSIX Unnamed Semaphores

An unnamed semaphore is created and initialized using the `sem_init()` function, which is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

and is illusrated in the following programming example:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

In this example, by passing the flag 0, we are indicating that this semaphore can be shared only by threads belonging to the process that created the semaphore. (If we supplied a nonzero value, we could allow the semaphore to be shared between separate processes by placing it in a region of shared memory.) In addition, we initialize the semaphore to the value 1.

POSIX unnamed semaphores use the same `sem_wait()` and `sem_post()` operations as named semaphores. The following code sample illustrates protecting a critical section using the unnamed semaphore created above:

```
                    /* acquire the semaphore */
                    sem_wait(&sem);

                    /* critical section */

                    /* release the semaphore */
                    sem_post(&sem);
```

Just like mutex locks, all semaphore functions return 0 when successful and nonzero when an error condition occurs.

7.3.3 POSIX Condition Variables

Condition variables in Pthreads behave similarly to those described in Section 6.7. However, in that section, condition variables are used within the context of a monitor, which provides a locking mechanism to ensure data integrity. Since Pthreads is typically used in C programs—and since C does not have a monitor— we accomplish locking by associating a condition variable with a mutex lock.

Condition variables in Pthreads use the pthread_cond_t data type and are initialized using the pthread_cond_init() function. The following code creates and initializes a condition variable as well as its associated mutex lock:

```
        pthread_mutex_t mutex;
        pthread_cond_t cond_var;

        pthread_mutex_init(&mutex,NULL);
        pthread_cond_init(&cond_var,NULL);
```

The pthread_cond_wait() function is used for waiting on a condition variable. The following code illustrates how a thread can wait for the condition a == b to become true using a Pthread condition variable:

```
        pthread_mutex_lock(&mutex);
        while (a != b)
            pthread_cond_wait(&cond_var, &mutex);

        pthread_mutex_unlock(&mutex);
```

The mutex lock associated with the condition variable must be locked before the pthread_cond_wait() function is called, since it is used to protect the data in the conditional clause from a possible race condition. Once this lock is acquired, the thread can check the condition. If the condition is not true, the thread then invokes pthread_cond_wait(), passing the mutex lock and the condition variable as parameters. Calling pthread_cond_wait() releases the mutex lock, thereby allowing another thread to access the shared data and possibly update its value so that the condition clause evaluates to true. (To protect against program errors, it is important to place the conditional clause within a loop so that the condition is rechecked after being signaled.)

A thread that modifies the shared data can invoke the `pthread_cond_signal()` function, thereby signaling one thread waiting on the condition variable. This is illustrated below:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

It is important to note that the call to `pthread_cond_signal()` does not release the mutex lock. It is the subsequent call to `pthread_mutex_unlock()` that releases the mutex. Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to `pthread_cond_wait()`.

We provide several programming problems and projects at the end of this chapter that use Pthreads mutex locks and condition variables, as well as POSIX semaphores.

7.4   Synchronization in Java

The Java language and its API have provided rich support for thread synchronization since the origins of the language. In this section, we first cover Java monitors, Java's original synchronization mechanism. We then cover three additional mechanisms that were introduced in Release 1.5: reentrant locks, semaphores, and condition variables. We include these because they represent the most common locking and synchronization mechanisms. However, the Java API provides many features that we do not cover in this text—for example, support for atomic variables and the CAS instruction—and we encourage interested readers to consult the bibliography for more information.

7.4.1   Java Monitors

Java provides a monitor-like concurrency mechanism for thread synchronization. We illustrate this mechanism with the `BoundedBuffer` class (Figure 7.9), which implements a solution to the bounded-buffer problem wherein the producer and consumer invoke the `insert()` and `remove()` methods, respectively.

Every object in Java has associated with it a single lock. When a method is declared to be `synchronized`, calling the method requires owning the lock for the object. We declare a `synchronized` method by placing the `synchronized` keyword in the method definition, such as with the `insert()` and `remove()` methods in the `BoundedBuffer` class.

Invoking a `synchronized` method requires owning the lock on an object instance of `BoundedBuffer`. If the lock is already owned by another thread, the thread calling the `synchronized` method blocks and is placed in the **entry set** for the object's lock. The entry set represents the set of threads waiting for the lock to become available. If the lock is available when a `synchronized` method is called, the calling thread becomes the owner of the object's lock and can enter the method. The lock is released when the thread exits the method. If the entry set for the lock is not empty when the lock is released, the JVM

```
public class BoundedBuffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */
   }
}
```

Figure 7.9 Bounded buffer using Java synchronization.

arbitrarily selects a thread from this set to be the owner of the lock. (When we say "arbitrarily," we mean that the specification does not require that threads in this set be organized in any particular order. However, in practice, most virtual machines order threads in the entry set according to a FIFO policy.) Figure 7.10 illustrates how the entry set operates.

In addition to having a lock, every object also has associated with it a **wait set** consisting of a set of threads. This wait set is initially empty. When a thread enters a synchronized method, it owns the lock for the object. However, this thread may determine that it is unable to continue because a certain condition
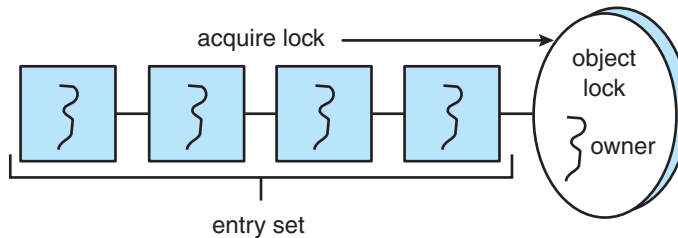

Figure 7.10 Entry set for a lock.

BLOCK SYNCHRONIZATION

The amount of time between when a lock is acquired and when it is released is defined as the **scope** of the lock. A `synchronized` method that has only a small percentage of its code manipulating shared data may yield a scope that is too large. In such an instance, it may be better to synchronize only the block of code that manipulates shared data than to synchronize the entire method. Such a design results in a smaller lock scope. Thus, in addition to declaring `synchronized` methods, Java also allows block synchronization, as illustrated below. Only the access to the critical-section code requires ownership of the object lock for the `this` object.

```
public void someMethod() {
   /* non-critical section */

   synchronized(this) {
      /* critical section */
   }

   /* remainder section */
}
```

---

has not been met. That will happen, for example, if the producer calls the `insert()` method and the buffer is full. The thread then will release the lock and wait until the condition that will allow it to continue is met.

When a thread calls the `wait()` method, the following happens:

1. The thread releases the lock for the object.

2. The state of the thread is set to blocked.

3. The thread is placed in the wait set for the object.

Consider the example in Figure 7.11. If the producer calls the `insert()` method and sees that the buffer is full, it calls the `wait()` method. This call releases the lock, blocks the producer, and puts the producer in the wait set for the object. Because the producer has released the lock, the consumer ultimately enters the `remove()` method, where it frees space in the buffer for the producer. Figure 7.12 illustrates the entry and wait sets for a lock. (Note that although `wait()` can throw an `InterruptedException`, we choose to ignore it for code clarity and simplicity.)

How does the consumer thread signal that the producer may now proceed? Ordinarily, when a thread exits a `synchronized` method, the departing thread releases only the lock associated with the object, possibly removing a thread from the entry set and giving it ownership of the lock. However, at the end of the `insert()` and `remove()` methods, we have a call to the method `notify()`. The call to `notify()`:

1. Picks an arbitrary thread T from the list of threads in the wait set

```
   /* Producers call this method */
   public synchronized void insert(E item) {
      while (count == BUFFER_SIZE) {
         try {
            wait();
         }
         catch (InterruptedException ie) { }
      }

      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      count++;

      notify();
   }

   /* Consumers call this method */
   public synchronized E remove() {
      E item;

      while (count == 0) {
         try {
            wait();
         }
         catch (InterruptedException ie) { }
      }

      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      count--;

      notify();

      return item;
   }
```

Figure 7.11 `insert()` and `remove()` methods using `wait()` and `notify()`.

2. Moves T from the wait set to the entry set
3. Sets the state of T from blocked to runnable

T is now eligible to compete for the lock with the other threads. Once T has regained control of the lock, it returns from calling `wait()`, where it may check the value of count again. (Again, the selection of an *arbitrary* thread is according to the Java specification; in practice, most Java virtual machines order threads in the wait set according to a FIFO policy.)
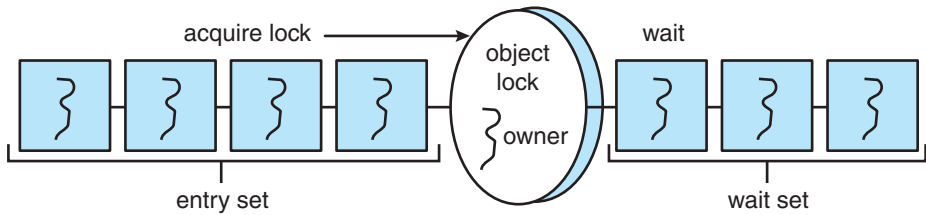
Figure 7.12   Entry and wait sets.

Next, we describe the `wait()` and `notify()` methods in terms of the methods shown in Figure 7.11. We assume that the buffer is full and the lock for the object is available.

- The producer calls the `insert()` method, sees that the lock is available, and enters the method. Once in the method, the producer determines that the buffer is full and calls `wait()`. The call to `wait()` releases the lock for the object, sets the state of the producer to blocked, and puts the producer in the wait set for the object.

- The consumer ultimately calls and enters the `remove()` method, as the lock for the object is now available. The consumer removes an item from the buffer and calls `notify()`. Note that the consumer still owns the lock for the object.

- The call to `notify()` removes the producer from the wait set for the object, moves the producer to the entry set, and sets the producer's state to runnable.

- The consumer exits the `remove()` method. Exiting this method releases the lock for the object.

- The producer tries to reacquire the lock and is successful. It resumes execution from the call to `wait()`. The producer tests the `while` loop, determines that room is available in the buffer, and proceeds with the remainder of the `insert()` method. If no thread is in the wait set for the object, the call to `notify()` is ignored. When the producer exits the method, it releases the lock for the object.

The `synchronized`, `wait()`, and `notify()` mechanisms have been part of Java since its origins. However, later revisions of the Java API introduced much more flexible and robust locking mechanisms, some of which we examine in the following sections.

7.4.2 Reentrant Locks

Perhaps the simplest locking mechanism available in the API is the `Reentrant-Lock`. In many ways, a `ReentrantLock` acts like the `synchronized` statement described in Section 7.4.1: a `ReentrantLock` is owned by a single thread and is used to provide mutually exclusive access to a shared resource. However, the `ReentrantLock` provides several additional features, such as setting a *fairness* parameter, which favors granting the lock to the longest-waiting thread. (Recall

that the specification for the JVM does not indicate that threads in the wait set for an object lock are to be ordered in any specific fashion.)

A thread acquires a `ReentrantLock` lock by invoking its `lock()` method. If the lock is available—or if the thread invoking `lock()` already owns it, which is why it is termed *reentrant*—`lock()` assigns the invoking thread lock ownership and returns control. If the lock is unavailable, the invoking thread blocks until it is ultimately assigned the lock when its owner invokes `unlock()`. `ReentrantLock` implements the `Lock` interface; it is used as follows:

```
Lock key = new ReentrantLock();

key.lock();
try {
   /* critical section */
}
finally {
   key.unlock();
}
```

The programming idiom of using `try` and `finally` requires a bit of explanation. If the lock is acquired via the `lock()` method, it is important that the lock be similarly released. By enclosing `unlock()` in a `finally` clause, we ensure that the lock is released once the critical section completes or if an exception occurs within the `try` block. Notice that we do not place the call to `lock()` within the `try` clause, as `lock()` does not throw any checked exceptions. Consider what happens if we place `lock()` within the `try` clause and an unchecked exception occurs when `lock()` is invoked (such as `OutofMemoryError`): The `finally` clause triggers the call to `unlock()`, which then throws the unchecked `IllegalMonitorStateException`, as the lock was never acquired. This `IllegalMonitorStateException` replaces the unchecked exception that occurred when `lock()` was invoked, thereby obscuring the reason why the program initially failed.

Whereas a `ReentrantLock` provides mutual exclusion, it may be too conservative a strategy if multiple threads only read, but do not write, shared data. (We described this scenario in Section 7.1.2.) To address this need, the Java API also provides a `ReentrantReadWriteLock`, which is a lock that allows multiple concurrent readers but only one writer.

7.4.3 Semaphores

The Java API also provides a counting semaphore, as described in Section 6.6. The constructor for the semaphore appears as

```
Semaphore(int value);
```

where `value` specifies the initial value of the semaphore (a negative value is allowed). The `acquire()` method throws an `InterruptedException` if the acquiring thread is interrupted. The following example illustrates using a semaphore for mutual exclusion:

```
Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```

Notice that we place the call to `release()` in the `finally` clause to ensure that the semaphore is released.

7.4.4 Condition Variables

The last utility we cover in the Java API is the condition variable. Just as the `ReentrantLock` is similar to Java's `synchronized` statement, condition variables provide functionality similar to the `wait()` and `notify()` methods. Therefore, to provide mutual exclusion, a condition variable must be associated with a reentrant lock.

We create a condition variable by first creating a `ReentrantLock` and invoking its `newCondition()` method, which returns a `Condition` object representing the condition variable for the associated `ReentrantLock`. This is illustrated in the following statements:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

Once the condition variable has been obtained, we can invoke its `await()` and `signal()` methods, which function in the same way as the `wait()` and `signal()` commands described in Section 6.7.

Recall that with monitors as described in Section 6.7, the `wait()` and `signal()` operations can be applied to *named* condition variables, allowing a thread to wait for a specific condition or to be notified when a specific condition has been met. At the language level, Java does not provide support for named condition variables. Each Java monitor is associated with just one unnamed condition variable, and the `wait()` and `notify()` operations described in Section 7.4.1 apply only to this single condition variable. When a Java thread is awakened via `notify()`, it receives no information as to why it was awakened; it is up to the reactivated thread to check for itself whether the condition for which it was waiting has been met. Condition variables remedy this by allowing a specific thread to be notified.

We illustrate with the following example: Suppose we have five threads, numbered 0 through 4, and a shared variable `turn` indicating which thread's turn it is. When a thread wishes to do work, it calls the `doWork()` method in Figure 7.13, passing its thread number. Only the thread whose value of `threadNumber` matches the value of `turn` can proceed; other threads must wait their turn.

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
   lock.lock();

   try {
      /**
       * If it's not my turn, then wait
       * until I'm signaled.
       */
      if (threadNumber != turn)
        condVars[threadNumber].await();

      /**
       * Do some work for awhile ...
       */

      /**
       * Now signal to the next thread.
       */
      turn = (turn + 1) % 5;
      condVars[turn].signal();
   }
   catch (InterruptedException ie) { }
   finally {
      lock.unlock();
   }
}
```

Figure 7.13   Example using Java condition variables.

We also must create a `ReentrantLock` and five condition variables (representing the conditions the threads are waiting for) to signal the thread whose turn is next. This is shown below:

```
        Lock lock = new ReentrantLock();
        Condition[] condVars = new Condition[5];

        for (int i = 0; i < 5; i++)
           condVars[i] = lock.newCondition();
```

When a thread enters `doWork()`, it invokes the `await()` method on its associated condition variable if its `threadNumber` is not equal to `turn`, only to resume when it is signaled by another thread. After a thread has completed its work, it signals the condition variable associated with the thread whose turn follows.

It is important to note that `doWork()` does not need to be declared `synchronized`, as the `ReentrantLock` provides mutual exclusion. When a thread

invokes `await()` on the condition variable, it releases the associated `Reen-trantLock`, allowing another thread to acquire the mutual exclusion lock. Similarly, when `signal()` is invoked, only the condition variable is signaled; the lock is released by invoking `unlock()`.

7.5   Alternative Approaches

With the emergence of multicore systems has come increased pressure to develop concurrent applications that take advantage of multiple processing cores. However, concurrent applications present an increased risk of race conditions and liveness hazards such as deadlock. Traditionally, techniques such as mutex locks, semaphores, and monitors have been used to address these issues, but as the number of processing cores increases, it becomes increasingly difficult to design multithreaded applications that are free from race conditions and deadlock. In this section, we explore various features provided in both programming languages and hardware that support the design of thread-safe concurrent applications.

7.5.1 Transactional Memory

Quite often in computer science, ideas from one area of study can be used to solve problems in other areas. The concept of **transactional memory** originated in database theory, for example, yet it provides a strategy for process synchronization. A **memory transaction** is a sequence of memory read–write operations that are atomic. If all operations in a transaction are completed, the memory transaction is committed. Otherwise, the operations must be aborted and rolled back. The benefits of transactional memory can be obtained through features added to a programming language.

Consider an example. Suppose we have a function `update()` that modifies shared data. Traditionally, this function would be written using mutex locks (or semaphores) such as the following:

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

However, using synchronization mechanisms such as mutex locks and semaphores involves many potential problems, including deadlock. Additionally, as the number of threads increases, traditional locking doesn't scale as well, because the level of contention among threads for lock ownership becomes very high.

As an alternative to traditional locking methods, new features that take advantage of transactional memory can be added to a programming language. In our example, suppose we add the construct `atomic{S}`, which ensures that

the operations in S execute as a transaction. This allows us to rewrite the
update() function as follows:

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```

The advantage of using such a mechanism rather than locks is that the
transactional memory system—not the developer—is responsible for guar-
anteeing atomicity. Additionally, because no locks are involved, deadlock is
not possible. Furthermore, a transactional memory system can identify which
statements in atomic blocks can be executed concurrently, such as concurrent
read access to a shared variable. It is, of course, possible for a programmer
to identify these situations and use reader–writer locks, but the task becomes
increasingly difficult as the number of threads within an application grows.

Transactional memory can be implemented in either software or hardware.
Software transactional memory (STM), as the name suggests, implements
transactional memory exclusively in software—no special hardware is needed.
STM works by inserting instrumentation code inside transaction blocks. The
code is inserted by a compiler and manages each transaction by examining
where statements may run concurrently and where specific low-level locking is
required. **Hardware transactional memory** (HTM) uses hardware cache hierar-
chies and cache coherency protocols to manage and resolve conflicts involving
shared data residing in separate processors' caches. HTM requires no special
code instrumentation and thus has less overhead than STM. However, HTM
does require that existing cache hierarchies and cache coherency protocols be
modified to support transactional memory.

Transactional memory has existed for several years without widespread
implementation. However, the growth of multicore systems and the associ-
ated emphasis on concurrent and parallel programming have prompted a
significant amount of research in this area on the part of both academics and
commercial software and hardware vendors.

7.5.2 OpenMP

In Section 4.5.2, we provided an overview of OpenMP and its support of parallel
programming in a shared-memory environment. Recall that OpenMP includes
a set of compiler directives and an API. Any code following the compiler direc-
tive #pragma omp parallel is identified as a parallel region and is performed
by a number of threads equal to the number of processing cores in the system.
The advantage of OpenMP (and similar tools) is that thread creation and man-
agement are handled by the OpenMP library and are not the responsibility of
application developers.

Along with its #pragma omp parallel compiler directive, OpenMP pro-
vides the compiler directive #pragma omp critical, which specifies the code
region following the directive as a critical section in which only one thread may
be active at a time. In this way, OpenMP provides support for ensuring that
threads do not generate race conditions.

As an example of the use of the critical-section compiler directive, first assume that the shared variable `counter` can be modified in the `update()` function as follows:

```
void update(int value)
{
    counter += value;
}
```

If the `update()` function can be part of—or invoked from—a parallel region, a race condition is possible on the variable `counter`.

The critical-section compiler directive can be used to remedy this race condition and is coded as follows:

```
void update(int value)
{
    #pragma omp critical
    {
        counter += value;
    }
}
```

The critical-section compiler directive behaves much like a binary semaphore or mutex lock, ensuring that only one thread at a time is active in the critical section. If a thread attempts to enter a critical section when another thread is currently active in that section (that is, *owns* the section), the calling thread is blocked until the owner thread exits. If multiple critical sections must be used, each critical section can be assigned a separate name, and a rule can specify that no more than one thread may be active in a critical section of the same name simultaneously.

An advantage of using the critical-section compiler directive in OpenMP is that it is generally considered easier to use than standard mutex locks. However, a disadvantage is that application developers must still identify possible race conditions and adequately protect shared data using the compiler directive. Additionally, because the critical-section compiler directive behaves much like a mutex lock, deadlock is still possible when two or more critical sections are identified.

7.5.3 Functional Programming Languages

Most well-known programming languages—such as C, C++, Java, and C#— are known as **imperative** (or **procedural**) languages. Imperative languages are used for implementing algorithms that are state-based. In these languages, the flow of the algorithm is crucial to its correct operation, and state is represented with variables and other data structures. Of course, program state is mutable, as variables may be assigned different values over time.

With the current emphasis on concurrent and parallel programming for multicore systems, there has been greater focus on **functional** programming languages, which follow a programming paradigm much different from that offered by imperative languages. The fundamental difference between imperative and functional languages is that functional languages do not maintain state. That is, once a variable has been defined and assigned a value, its value

is immutable—it cannot change. Because functional languages disallow mutable state, they need not be concerned with issues such as race conditions and deadlocks. Essentially, most of the problems addressed in this chapter are nonexistent in functional languages.

Several functional languages are presently in use, and we briefly mention two of them here: Erlang and Scala. The Erlang language has gained significant attention because of its support for concurrency and the ease with which it can be used to develop applications that run on parallel systems. Scala is a functional language that is also object-oriented. In fact, much of the syntax of Scala is similar to the popular object-oriented languages Java and C#. Readers interested in Erlang and Scala, and in further details about functional languages in general, are encouraged to consult the bibliography at the end of this chapter for additional references.

7.6 Summary

- Classic problems of process synchronization include the bounded-buffer, readers–writers, and dining-philosophers problems. Solutions to these problems can be developed using the tools presented in Chapter 6, including mutex locks, semaphores, monitors, and condition variables.

- Windows uses dispatcher objects as well as events to implement process synchronization tools.

- Linux uses a variety of approaches to protect against race conditions, including atomic variables, spinlocks, and mutex locks.

- The POSIX API provides mutex locks, semaphores, and condition variables. POSIX provides two forms of semaphores: named and unnamed. Several unrelated processes can easily access the same named semaphore by simply referring to its name. Unnamed semaphores cannot be shared as easily, and require placing the semaphore in a region of shared memory.

- Java has a rich library and API for synchronization. Available tools include monitors (which are provided at the language level) as well as reentrant locks, semaphores, and condition variables (which are supported by the API).

- Alternative approaches to solving the critical-section problem include transactional memory, OpenMP, and functional languages. Functional languages are particularly intriguing, as they offer a different programming paradigm from procedural languages. Unlike procedural languages, functional languages do not maintain state and therefore are generally immune from race conditions and critical sections.

Practice Exercises

7.1 Explain why Windows and Linux implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutex locks, semaphores, and condition variables. In each case, explain why the mechanism is needed.