An Introduction to Parallel Programming

Peter S. Pacheco
Matthew Malensek
University of San Francisco

April 12, 2021

Chapter 1

Shared-Memory Programming with Pthreads

Recall that from a programmer's point of view a shared-memory system is one in which all the cores can access all the memory locations (see Figure 1.1). Thus, an obvious approach to the problem of coordinating the work of the cores is to specify that certain memory locations are "shared." This is a very natural approach to parallel programming. Indeed, we might well wonder why all parallel programs don't use this shared-memory approach. However, we'll see in this chapter that there are problems that arise with programming shared-memory systems; problems that are often different from the problems encountered in distributed memory programming.

For example, in Chapter **??** we saw that if different cores attempt to update a single shared-memory location, then the contents of the shared location can be unpredictable. The code that updates the shared location is an example of a *critical section.* We'll see some other examples of critical sections, and we'll learn several methods for controlling access to a critical section.

We'll also learn about other issues and techniques in shared-memory programming. In shared-memory programming, an instance of a program running on a processor is usually called a **thread** (unlike MPI, where it's called a process). We'll learn how to synchronize threads so that each thread will wait to execute a block of statements until another thread has completed some work. We'll learn how to put a thread "to sleep" until a condition has occurred. We'll see that there are some circumstances in which it may at first seem that a critical section must be quite large. However, we'll also see that there are tools that can allow us to "fine-tune" access to these large blocks of code so that more of the program can truly be executed in parallel. We'll see that the use of cache memories can actually cause a shared-memory program to run more slowly. Finally, we'll observe that functions that "maintain state" between successive calls can cause inconsistent or even incorrect results.

In this chapter we'll be using POSIX® Threads for most of our shared-memory functions. In the next chapter we'll look at an alternative approach to shared-memory programming called OpenMP.

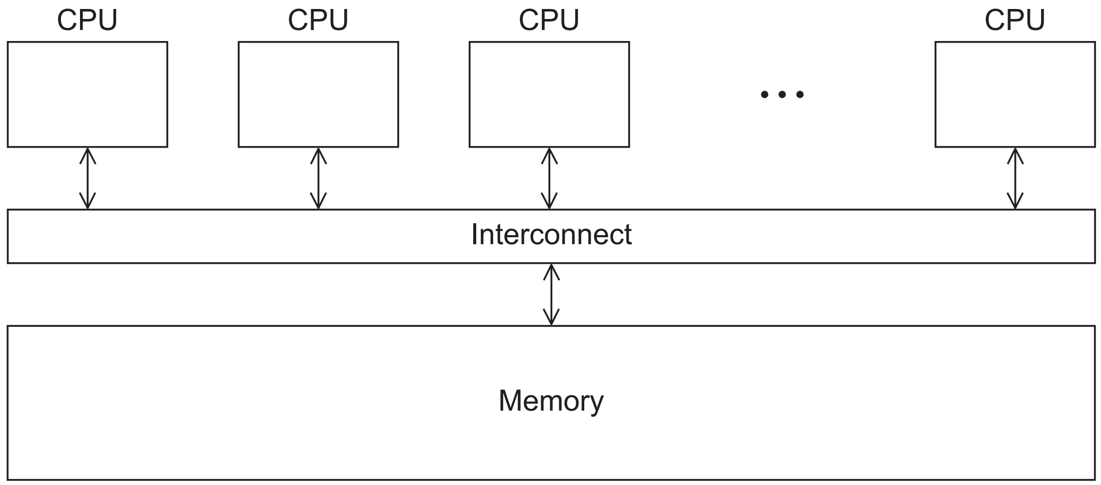

Figure 1.1: A Shared-Memory System

1.1 Processes, Threads and Pthreads

Recall from Chapter **??** that in shared-memory programming, a thread is somewhat analogous to a process in MPI programming. However, it can, in principle, be “lighter-weight.” A process is an instance of a running (or suspended) program. In addition to its executable, it consists of the following:

- A block of memory for the stack

- A block of memory for the heap

- Descriptors of resources that the system has allocated for the process—for example, file descriptors (including `stdout`, `stdin`, and `stderr`)

- Security information—for example, information about which hardware and software resources the process can access

- Information about the state of the process, such as whether the process is ready to run or is waiting on a resource, the content of the registers including the program counter, and so on

In most systems, by default, a process’ memory blocks are private: another process can’t directly access the memory of a process unless the operating system intervenes. This makes sense. If you're using a text editor to write a program (one process—the running text editor), you don't want your browser (another process) overwriting your text editor's memory. This is even more crucial in a multiuser environment. Ordinarily, one user's processes shouldn't be allowed access to the memory of another user's processes.

However, this isn't desirable when we're running shared-memory programs. At a minimum, we'd like certain variables to be available to multiple processes, allowing much easier memory access. It is also convenient for the processes to share access to things like `stdout` and all other process-specific resources, except for their stacks and program counters. This can be arranged by starting a single process and then having the process start these additional "lighter-weight" processes. For this reason, they're often called **light-weight processes**.

The more commonly used term, **thread**, comes from the concept of "thread of control." A thread of control is just a sequence of statements in a program. The term suggests a stream of control in a single process, and in a shared-memory program a single *process* may have multiple *threads* of control.

As we noted earlier, in this chapter the particular implementation of threads that we'll be using is called POSIX® threads or, more often, **Pthreads**. POSIX®[**?**] is a standard for Unix-like operating systems—for example, Linux and macOS. It specifies a variety of facilities that should be available in such systems. In particular, it specifies an application programming interface (API) for *multithreaded* programming.

Pthreads is not a programming language (like C or Java). Rather, like MPI, Pthreads specifies a *library* that can be linked with C programs. Unlike MPI, the Pthreads API is only available on POSIX® systems — Linux, macOS, Solaris, HPUX, and so on. Also unlike MPI, there are a number of other widely used specifications for multithreaded programming: Java threads, Windows threads, Solaris threads. However, all of the thread specifications support the same basic ideas, so once you've learned how to program in Pthreads, it won't be difficult to learn how to program with another thread API.

Since Pthreads is a C library, it can, in principle, be used in C++ programs. However, the recent C++11 standard includes its own shared-memory programming model with support for threads (`std::thread`), so it may make sense to use it instead if you're writing C++ programs.

1.2 Hello, World

Let's take a look at a Pthreads program. In Program 1.1, the main function starts up several threads. Each thread prints a message and then quits.

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

/* Global variable:  accessible to all threads */
int thread_count;

void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Hello, (void*) thread);

   printf("Hello from the main thread\n");

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   free(thread_handles);
   return 0;
}  /* main */

void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */

   printf("Hello from thread %ld of %d\n", my_rank, thread_count);

   return NULL;
}  /* Hello */
```

Program 1.1: A Pthreads “hello, world” program

1.2.1 Execution

The program is compiled like an ordinary C program, with the possible exception that we may need to link in the Pthreads library:[1]

```
$ gcc −g −Wall −o pth_hello pth_hello.c −lpthread
```

The −lpthread tells the compiler that we want to link in the Pthreads library. Note that it's −lpthread, *not* −lpthreads. On some systems the compiler will automatically link in the library, and −lpthread won't be needed.

To run the program, we just type

```
$ ./pth_hello <number of threads>
```

For example, to run the program with 1 thread, we type

```
$ ./pth_hello 1
```

and the output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 1
```

To run the program with four threads, we type

```
$ ./pth_hello 4
```

and the output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

If your output appears out of order, don't worry. As we will discuss later, we usually do not have direct control of the order in which threads execute.

1.2.2 Preliminaries

Let's take a closer look at the source code in Program 1.1. First notice that this *is* just a C program with a main function and one other function. The program includes the familiar stdio.h and stdlib.h header files. However, there's a lot that's new and different.

In Line 3 we include pthread.h, the Pthreads header file, which declares the various Pthreads functions, constants, types, and so on.

[1] Recall that the dollar sign ($) is the shell prompt, so it shouldn't be typed in. Also recall that for the sake of explicitness, we assume that we're using the Gnu C compiler, gcc, and we always use the options -g, -Wall, and -o. See **??** for further information.

In Line 6 we define a *global* variable thread_count. In Pthreads programs, global variables are shared by all the threads. Local variables and function arguments—that is, variables declared in functions—are (ordinarily) private to the thread executing the function. If several threads are executing the same function, each thread will have its own private copies of the local variables and function arguments. This makes sense if you recall that each thread has its own stack.

We should keep in mind that global variables can introduce subtle and confusing bugs. For example, suppose we write a program in which we declare a global variable **int** x. Then we write a function f in which we intend to use a local variable called x, but we forget to declare it. The program will compile with no warnings, since f has access to the global x. But when we run the program, it produces very strange output, which we eventually determine to have been caused by the fact that the global variable x has a strange value. Days later, we finally discover that the strange value came from f. As a rule of thumb, we should try to limit our use of global variables to situations in which they're really needed—for example, for a shared variable.

In Line 15 the program gets the number of threads it should start from the command line. Unlike MPI programs, Pthreads programs are typically compiled and run just like serial programs, and one relatively simple way to specify the number of threads that should be started is to use a command-line argument. This isn't a requirement, it's simply a convenient convention we'll be using.

The strtol function converts a string into a **long int**. It's declared in stdlib.h, and its syntax is

```
long strtol(
         const char*   number_p   /* in   */,
         char**        end_p      /* out  */,
         int           base       /* in   */);
```

It returns a **long int** corresponding to the string referred to by number_p. The base of the representation of the number is given by the base argument. If end_p isn't NULL, it will point to the first invalid (that is, nonnumeric) character in number_p.

1.2.3 Starting the Threads

As we already noted, unlike MPI programs, in which the processes are usually started by a script, in Pthreads the threads are started by the program executable. This introduces a bit of additional complexity, as we need to include code in our program to explicitly start the threads, and we need data structures to store information on the threads.

In Line 17 we allocate storage for one pthread_t object for each thread. The pthread_t data structure is used for storing thread-specific information. It's declared in pthread.h.

The pthread_t objects are examples of **opaque** objects. The actual data that they store is system specific, and their data members aren't directly accessible to user code. However, the

Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it's associated. So, for example, there is a Pthreads function that a thread can use to retrieve its associated pthread_t object, and there is a Pthreads function that can determine whether two threads are in fact the same by examining their associated pthread_t objects.

In Lines 19–21, we use the pthread_create function to start the threads. Like most Pthreads functions, its name starts with the string pthread_. The syntax of pthread_create is

```
int pthread_create(
      pthread_t*             thread_p                  /* out */,
      const pthread_attr_t*  attr_p                    /* in  */,
      void*                  (*start_routine)(void*)   /* in  */,
      void*                  args_p                     /* in  */);
```

The first argument is a pointer to the appropriate pthread_t object. Note that the object is not allocated by the call to pthread_create; it must be allocated *before* the call. We won't be using the second argument, so we just pass NULL in our function call.[2] The third argument is the function that the thread is to run, and the last argument is a pointer to the argument that should be passed to the function start_routine. The return value for most Pthreads functions indicates if there's been an error in the function call. In order to reduce the clutter in our examples, in this chapter (as in most of the rest of the book) we'll generally ignore the return values of Pthreads functions.

Let's take a closer look at the last two arguments. The function that's started by pthread_create should have a prototype that looks something like this:

```
 void* thread_function(void* args_p);
```

Recall that the type **void*** can be cast to any pointer type in C, so args_p can point to a list containing one or more values needed by thread_function. Similarly, the return value of thread_function can point to a list of one or more values.

In our call to pthread_create, the final argument is a fairly common kluge: we're effectively assigning each thread a unique integer *rank*. Let's first look at why we are doing this; then we'll worry about the details of how to do it.

Consider the following problem: We start a Pthreads program that uses two threads, but one of the threads encounters an error. How do we, the users, know which thread encountered the error? We can't just print out the pthread_t object, since it's opaque. However, if when we start the threads, we assign the first thread rank 0, and the second thread rank 1, we can easily determine which thread ran into trouble by just including the thread's rank in the error message.

Since the thread function takes a **void*** argument, we could allocate one **int** in main for each thread and assign each allocated **int** a unique value. When we start a thread, we could then pass a pointer to the appropriate **int** in the call to pthread_create. However, most programmers resort to

[2] Passing NULL here uses the default set of Pthread *attributes*—settings that specify a variety of properties including operating system scheduling parameters and the stack size of the new thread.

some trickery with casts. Instead of creating an **int** in main for the "rank," we cast the loop variable thread to have type **void***. Then in the thread function, hello, we cast the argument back to a **long** (Line 33).

The result of carrying out these casts is "system-defined," but most C compilers do allow this. However, if the size of pointer types is different from the size of the integer type you use for the rank, you may get a warning. On the machines we used, pointers are 64 bits, and **int**s are only 32 bits, so we use **long** instead of **int**.

Note that our method of assigning thread ranks and, indeed, the thread ranks themselves are just a convenient convention that we'll use. There is no requirement that a thread rank be passed in the call to pthread_create, nor a requirement that a thread be assigned a rank. The following thread procedure expects a pointer to a **struct** to be passed in for args_p. The **struct** contains both a rank and the name of the task. (Imagine distinguishing between different requests in a web server, for instance.)

```
struct thread_args {
    long my_rank;
    char *task_name;
};

void *Hello(void *args) {
    struct thread_args* t_args = (struct thread_args *) args;
    printf("Thread %ld is working on task '%s'\n",
            t_args->my_rank, t_args->task_name);
    return NULL;
}
```

When we create the thread, a pointer to the appropriate **struct** is passed to pthread_create. We can add the logic to do this at Line 19 (in this case, each thread has the same "task name"):

```
struct thread_args *t_args = malloc(sizeof(struct thread_args));
t_args->my_rank = thread;
t_args->task_name = "Hello task";
pthread_create(&thread_handles[thread], NULL, Hello, (void*) t_args);
```

Also note that there is no technical reason for each thread to run the same function; we could have one thread run hello, another run goodbye, and so on. However, as with the MPI programs, we'll typically use "single program, multiple data" style parallelism with our Pthreads programs. That is, each thread will run the same thread function, but we'll obtain the effect of different thread functions by branching within a thread.

1.2.4 Running the Threads

The thread that's running the main function is sometimes called the **main thread**. Hence, after starting the threads, it prints the message

```
Hello from the main thread
```

In the meantime, the threads started by the calls to pthread_create are also running. They get their ranks by casting in Line 33, and then print their messages. Note that when a thread is done, since the type of its function has a return value, the thread should return something. In this example, the threads don't actually need to return anything, so they return NULL.

As we hinted earlier, in Pthreads the programmer doesn't directly control where the threads are run.[3] There's no argument in pthread_create saying which core should run which thread; thread placement is controlled by the operating system. Indeed, on a heavily loaded system, the threads may all be run on the same core. In fact, if a program starts more threads than cores, we should expect multiple threads to be run on a single core. However, if there is a core that isn't being used, operating systems will typically place a new thread on such a core.

1.2.5 Stopping the Threads

In Lines 25 and 26, we call the function pthread_join once for each thread. A single call to pthread_join will wait for the thread associated with the pthread_t object to complete. The syntax of pthread_join is

```
int pthread_join(
      pthread_t  thread     /* in  */,
      void**     ret_val_p  /* out */);
```

The second argument can be used to receive any return value computed by the thread. In the example, each thread returns NULL and eventually the main thread will call pthread_join on that thread to complete its termination.

This function is called pthread_join because of a diagramming style that is often used to describe the threads in a multithreaded process. If we think of the main thread as a single line in our diagram, then, when we call pthread_create, we can create a *branch* or *fork* off the main thread. Multiple calls to pthread_create will result in multiple branches or forks. Then, when the threads started by pthread_create terminate, the diagram shows the branches *joining* the main thread. See Figure 1.2.

As noted previously, every thread requires a variety of resources to be allocated, including stacks and local variables. The pthread_join function not only allows us to wait for a particular thread to finish its execution, but also frees the resources associated with the thread. In fact, not

[3] Some systems (for example, some implementations of Linux) do allow the programmer to specify where a thread is run. However, these constructions will not be portable.

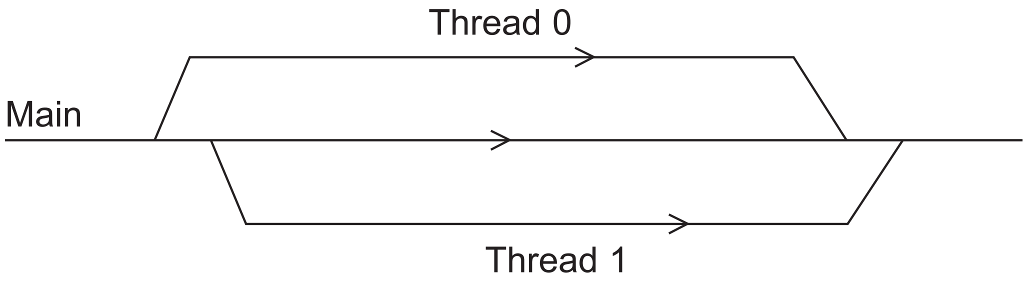

Figure 1.2: Main thread forks and joins two threads

joining threads that have finished execution produces *zombie threads* that waste resources and may even prevent the creation of new threads if left unchecked. If your program does not need to wait for a particular thread to finish, it can be *detached* with the `pthread_detach` function to indicate that its resources should be freed automatically upon termination. See Exercise 7 for an example of using `pthread_detach`.

1.2.6 Error Checking

In the interest of keeping the program compact and easy to read, we have resisted the temptation to include many details that would be important in a "real" program. The most likely source of problems in this example (and in many programs) is the user input (or lack thereof). Therefore, it would be a very good idea to check that the program was started with command line arguments, and, if it was, to check the actual value of the number of threads to see if it's reasonable. If you visit the book's website, you can download a version of the program that includes this basic error checking.

In general, it is good practice to always check the error codes returned by the Pthreads functions. This can be especially useful when you're just starting to use Pthreads and some of the details of function use aren't completely clear. We'd suggest getting in the habit of consulting the "RETURN VALUE" sections of the man pages for Pthreads functions (for instance, see `man pthread_create`; you will note several return values that indicate a variety of errors).

1.2.7 Other Approaches to Thread Startup

In our example, the user specifies the number of threads to start by typing in a command-line argument. The main thread then creates all of the "subsidiary" threads. While the threads are running, the main thread prints a message, and then waits for the other threads to terminate. This approach to threaded programming is very similar to our approach to MPI programming, in which the MPI system starts a collection of processes and waits for them to complete.

There is, however, a very different approach to the design of multithreaded programs. In this approach, subsidiary threads are only started as the need arises. As an example, imagine a Web server that handles requests for information about highway traffic in the San Francisco Bay Area. Suppose that the main thread receives the requests and subsidiary threads fulfill the requests. At 1 o'clock on a typical Tuesday morning, there will probably be very few requests, while at 5 o'clock on a typical Tuesday evening, there will probably be thousands. Thus, a natural approach to the design of this Web server is to have the main thread start subsidiary threads when it receives requests.

Intuitively, thread startup involves some overhead. The time required to start a thread will be much greater than, for instance, a floating point arithmetic operation, so in applications that need maximum performance the "start threads as needed" approach may not be ideal. In such a case, it is usually more performant to employ a scheme that leverages the strengths of both approaches: our main thread will start all the threads it anticipates needing at the beginning of the program, but the threads will sit idle instead of terminating when they finish their work. Once another request arrives, an idle thread can fulfill it without incurring thread creation overhead. This approach is called a *thread pool*, which we'll cover in Programming Assignment 5.

1.3 Matrix-Vector Multiplication

Let's take a look at writing a Pthreads matrix-vector multiplication program. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^T$ is an n-dimensional column vector,[4] then the matrix-vector product $A\mathbf{x} = \mathbf{y}$ is an m-dimensional column vector, $\mathbf{y} = (y_0, y_1, \ldots, y_{m-1})^T$ in which the ith component y_i is obtained by finding the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j.$$

See Figure 1.3.

Thus, pseudocode for a *serial* program for matrix-vector multiplication might look like this:

```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
```

[4] Recall that we use the convention that matrix and vector subscripts start with 0. Also recall that if **b** is a matrix or a vector, then $\mathbf{b}^T$ denotes its transpose.

$$\begin{array}{|c|c|c|c|}\hline a_{00} & a_{01} & \cdots & a_{0,n-1} \\ \hline a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline \end{array} \quad \begin{array}{|c|}\hline x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{n-1} \\ \hline \end{array} = \begin{array}{|c|}\hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline \end{array}$$

Figure 1.3: Matrix-vector multiplication

```
      y[i] += A[i][j]* x[j];
}
```

We want to parallelize this by dividing the work among the threads. One possibility is to divide the iterations of the outer loop among the threads. If we do this, each thread will compute some of the components of y. For example, suppose that $m = n = 6$ and the number of threads, thread_count or t, is three. Then the computation could be divided among the threads as follows:

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

To compute y[0], thread 0 will need to execute the code

```
y[0] = 0.0;
for (j = 0; j < n; j++)
   y[0] += A[0][j]* x[j];
```

Therefore, thread 0 will need to access every element of row 0 of A and every element of x. More generally, the thread that has been assigned y[i] will need to execute the code

```
y[i] = 0.0;
for (j = 0; j < n; j++)
   y[i] += A[i][j]*x[j];
```

Thus, this thread will need to access every element of row i of A and every element of x. We see that each thread needs to access every component of x, while each thread only needs to access its assigned rows of A and assigned components of y. This suggests that, at a minimum, x should be shared. Let's also make A and y shared. This might seem to violate our principle that we should only make variables global that need to be global. However, in the exercises, we'll take a closer look at some of the issues involved in making the A and y variables local to the thread function, and we'll see that making them global can make good sense. At this point, we'll just observe that if they are global, the main thread can easily initialize all of A by just reading its entries from stdin, and the product vector y can be easily printed by the main thread.

Having made these decisions, we only need to write the code that each thread will use for deciding which components of y it will compute. In order to simplify the code, let's assume that both m and n are evenly divisible by t. Our example with $m = 6$ and $t = 3$ suggests that each thread gets m/t components. Furthermore, thread 0 gets the first m/t, thread 1 gets the next m/t, and so on. Thus, the formulas for the components assigned to thread q might be

$$\text{first component: } q \times \frac{m}{t}$$

and

$$\text{last component: } (q+1) \times \frac{m}{t} - 1.$$

With these formulas, we can write the thread function that carries out matrix-vector multiplication. See Program 1.2. Note that in this code, we're assuming that A, x, y, m, and n are all global and shared.

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```

Program 1.2: Pthreads matrix-vector multiplication

If you have already read the MPI chapter, you may recall that it took more work to write a matrix-vector multiplication program using MPI. This was because of the fact that the data structures were necessarily distributed, that is, each MPI process only has direct access to its own local memory. Thus, for the MPI code, we need to explicitly *gather* all of x into each process' memory.

We see from this example that there are instances in which writing shared-memory programs is easier than writing distributed-memory programs. However, we'll shortly see that there are situations in which shared-memory programs can be more complex.

1.4 Critical Sections

Matrix-vector multiplication was very easy to code because the shared-memory locations were accessed in a highly desirable way. After initialization, all of the variables—except `y`—are only *read* by the threads. That is, except for `y`, none of the shared variables are changed after they've been initialized by the main thread. Furthermore, although the threads do make changes to `y`, only one thread makes changes to any individual component, so there are no attempts by two (or more) threads to modify any single component. What happens if this isn't the case? That is, what happens when multiple threads update a single memory location? We also discuss this in Chapters **??** and **??**, so if you've read one of these chapters, you already know the answer. But let's look at an example.

Let's try to estimate the value of π. There are lots of different formulas we could use. One of the simplest is

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right).$$

This isn't the best formula for computing π, because it takes *a lot* of terms on the right-hand side before it is very accurate. However, for our purposes, lots of terms will be better to demonstrate the effects of parallelism.

The following *serial* code uses this formula:

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
   sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

We can try to parallelize this in the same way we parallelized the matrix-vector multiplication program: divide up the iterations in the **for** loop among the threads and make `sum` a shared variable. To simplify the computations, let's assume that the number of threads, `thread_count` or t, evenly divides the number of terms in the sum, n. Then, if $\bar{n} = n/t$, thread 0 can add the first $\bar{n}$ terms. Therefore, for thread 0, the loop variable `i` will range from 0 to $\bar{n}-1$. Thread 1 will add the next $\bar{n}$ terms, so for thread 1, the loop variable will range from $\bar{n}$ to $2\bar{n}-1$. More generally, for thread q the loop variable will range over

$$q\bar{n}, q\bar{n}+1, q\bar{n}+2, \ldots, (q+1)\bar{n}-1.$$

Furthermore, the sign of the first term, term $q\bar{n}$, will be positive if $q\bar{n}$ is even and negative if $q\bar{n}$ is odd. The thread function might use the code shown in Program 1.3.

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

   return NULL;
}  /* Thread_sum */
```

Program 1.3: An attempt at a thread function for computing π

If we run the Pthreads program with two threads and n is relatively small, we find that the results of the Pthreads program are in agreement with the serial sum program. However, as n gets larger, we start getting some peculiar results. For example, with a dual-core processor we get the following results:

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Notice that as we increase n, the estimate with one thread gets better and better. In fact, with each factor of 10 increase in n we get another correct digit. With $n = 10^5$, the result as computed by a single thread has five correct digits. With $n = 10^6$, it has six correct digits, and so on. The result computed by two threads agrees with the result computed by one thread when $n = 10^5$. However, for larger values of n, the result computed by two threads actually gets worse. In fact, if we ran the program several times with two threads and the same value of n, we would see that the result computed by two threads *changes* from run to run. The answer to our original question must clearly be, "Yes, it matters if multiple threads try to update a single shared variable."

Let's recall why this is the case. Remember that the addition of two values is typically *not* a single machine instruction. For example, although we can add the contents of a memory location y to a memory location x with a single C statement,

```
x = x + y;
```

what the machine does is typically more complicated. The current values stored in x and y will, in general, be stored in the computer's main memory, which has no circuitry for carrying out arithmetic operations. Before the addition can be carried out, the values stored in x and y may therefore have to be transferred from main memory to registers in the CPU. Once the values are in registers, the addition can be carried out. After the addition is completed, the result may have to be transferred from a register back to memory.

Suppose that we have two threads, and each computes a value that is stored in its private variable y. Also suppose that we want to add these private values together into a shared variable x that has been initialized to 0 by the main thread. Each thread will execute the following code:

```
y = Compute(my_rank);
x = x + y;
```

Let's also suppose that thread 0 computes y = 1 and thread 1 computes y = 2. The "correct" result should then be x = 3. Here's one possible scenario:

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put x=0 and y=1 into registers	Assign `y = 2`
5	Add 0 and 1	Put x=0 and y=2 into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

So we see that if thread 1 copies x from memory to a register *before* thread 0 stores its result, the computation carried out by thread 0 will be *overwritten* by thread 1. The problem could be reversed: if thread 1 *races* ahead of thread 0, then its result may be overwritten by thread 0. In fact, unless one of the threads stores its result *before* the other thread starts reading x from memory, the "winner's" result will be overwritten by the "loser."

This example illustrates a fundamental problem in shared-memory programming: when multiple threads attempt to update a shared resource—in our case a shared variable—the result may be unpredictable. Recall that more generally, when multiple threads attempt to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a **race condition**. In our example, in order for our code to produce the correct result, we need to make sure that once one of the threads starts executing the statement `x = x + y`, it finishes executing the statement *before* the other thread starts executing the statement. Therefore the code `x = x + y` is a **critical section**. That is, it's a block of code that updates a shared resource that can only be updated by one thread at a time.

To further illustrate the concept of a race condition, imagine a bank wants to improve the performance of its checking account system. An obvious first step would be to make the system multithreaded; rather than processing a single transaction at a time, banking operations should be spread across multiple threads to take advantage of parallelism. This works well — until multiple transactions modify an account at the same time. Consider two pending transactions on a checking account with an initial balance of $1000:

- A $100 utility bill payment
- A $500 salary deposit

After the transactions complete, the new account balance should be $1400. The salary deposit will require an addition operation and the utility payment will require a subtraction. However, as mentioned previously, these simple math operations will be broken into more than one machine instruction. One possible outcome is:

Time	Thread 0 (Bill Payment)	Thread 1 (Salary Deposit)
1		Read Balance ($1000)
2	Read Balance ($1000)	Calculate Balance + $500
3	Calculate Balance - $100	Write Balance ($1500)
4	Write Balance ($900)	

Rather than the expected ending balance of $1400, we get $900 instead because the transaction processed by thread 1 was overwritten by thread 0.

These types of issues are particularly difficult to debug because the outcome is non-deterministic. It is entirely possible that the error shown above occurs less than 1% of the time and could be influenced by external factors including the hardware, operating system, or process scheduling algorithm. Even worse, attaching a debugger or adding `printf` statements to the code may change the relative timing of the threads and seemingly "correct" the issue temporarily. Such bugs that disappear when inspected are known as *Heisenbugs* (the act of observing the system alters its state).

1.5 Busy-Waiting

To avoid race conditions, threads need exclusive access to shared memory regions. When, say, thread 0 wants to execute the statement `x = x + y`, it needs to first make sure that thread 1 is not already executing the statement. Once thread 0 makes sure of this, it needs to provide some way for thread 1 to determine that it, thread 0, is executing the statement, so that thread 1 won't attempt to start executing the statement until thread 0 is done. Finally, after thread 0 has completed execution of the statement, it needs to provide some way for thread 1 to determine that it is done, so that thread 1 can safely start executing the statement.

A simple approach that doesn't involve any new concepts is the use of a flag variable. Suppose `flag` is a shared **int** that is set to 0 by the main thread. Further, suppose we add the following code to our example:

```
1    y = Compute(my_rank);
2    while (flag != my_rank);
3    x = x + y;
4    flag++;
```

Let's suppose that thread 1 finishes the assignment in Line 1 before thread 0. What happens when it reaches the **while** statement in Line 2? If you look at the **while** statement for a minute, you'll see that it has the somewhat peculiar property that its body is empty. So if the test `flag` != `my_rank` is true, then thread 1 will just execute the test a second time. In fact, it will keep re-executing the test until the test is false. When the test is false, thread 1 will go on to execute the code in the critical section `x` = `x` + `y`.

Since we're assuming that the main thread has initialized `flag` to 0, thread 1 won't proceed to the critical section in Line 3 until thread 0 executes the statement `flag`++. In fact, we see that unless some catastrophe befalls thread 0, it will eventually catch up to thread 1. However, when thread 0 executes its first test of `flag` != `my_rank`, the condition is false, and it will go on to execute the code in the critical section `x` = `x` + `y`. When it's done with this, we see that it will execute `flag`++, and thread 1 can finally enter the critical section.

The key here is that thread 1 *cannot enter the critical section until thread 0 has completed the execution of* `flag`++. And, provided the statements are executed exactly as they're written, this means that thread 1 cannot enter the critical section until thread 0 has completed it.

The **while** loop is an example of **busy-waiting**. In busy-waiting, a thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value (false in our example).

Note that we said that the busy-wait solution would work "provided the statements are executed exactly as they're written." If compiler optimization is turned on, it *is* possible that the compiler will make changes that will affect the correctness of busy-waiting. The reason for this is that the compiler is unaware that the program is multithreaded, so it doesn't "know" that the variables `x` and `flag` can be modified by another thread. For example, if our code

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

is run by just one thread, the order of the statements **while** (flag != my_rank) and x = x + y is unimportant. An optimizing compiler might therefore determine that the program would make better use of registers if the order of the statements were switched. Of course, this will result in the code

```
y = Compute(my_rank);
x = x + y;
while (flag != my_rank);
flag++;
```

which defeats the purpose of the busy-wait loop. The simplest solution to this problem is to turn compiler optimizations off when we use busy-waiting. For an alternative to completely turning off optimizations, see Exercise 3.

We can immediately see that busy-waiting is not an ideal solution to the problem of controlling access to a critical section. Since thread 1 will execute the test over and over until thread 0 executes flag++, if thread 0 is delayed (for example, if the operating system preempts it to run something else), thread 1 will simply "spin" on the test, eating up CPU cycles. This approach — often called a *spinlock* — can be positively disastrous for performance. Turning off compiler optimizations can also seriously degrade performance.

Before going on, though, let's return to our π calculation program in Figure 1.3 and correct it by using busy-waiting. The critical section in this function is Line 15. We can therefore precede this with a busy-wait loop. However, when a thread is done with the critical section, if it simply increments flag, eventually flag will be greater than t, the number of threads, and none of the threads will be able to return to the critical section. That is, after executing the critical section once, all the threads will be stuck forever in the busy-wait loop. Thus, in this instance, we don't want to simply increment flag. Rather, the last thread, thread $t-1$, should reset flag to zero. This can be accomplished by replacing flag++ with

```
flag = (flag + 1) % thread_count;
```

With this change, we get the thread function shown in Program 1.4. If we compile the program and run it with two threads, we see that it is computing the correct results. However, if we add in code for computing elapsed time, we see that when $n = 10^8$, the serial sum is consistently faster than the parallel sum. For example, on the dual-core system, the elapsed time for the sum as computed by two threads is about 19.5 seconds, while the elapsed time for the serial sum is about 2.8 seconds!

Why is this? Of course, there's overhead associated with starting up and joining the threads. However, we can estimate this overhead by writing a Pthreads program in which the thread function simply returns:

```
void* Thread_function(void* ignore) {
   return NULL;
}  /* Thread_function */
```

```c
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

Program 1.4: Pthreads global sum with busy-waiting

When we find the time that's elapsed between starting the first thread and joining the second thread, we see that on this particular system, the overhead is less than 0.3 milliseconds, so the slowdown isn't due to thread overhead. If we look closely at the thread function that uses busy-waiting, we see that the threads alternate between executing the critical section code in Line 16. Initially `flag` is 0, so thread 1 must wait until thread 0 executes the critical section and increments `flag`. Then, thread 0 must wait until thread 1 executes and increments. The threads will alternate between waiting and executing, and evidently the waiting and the incrementing increase the overall run time by a factor of seven.

As we'll see, busy-waiting isn't the only solution to protecting a critical section. In fact, there are much better solutions. However, since the code in a critical section can only be executed by one thread at a time, no matter how we limit access to the critical section, we'll effectively serialize the code in the critical section. Therefore, if it's at all possible, we should minimize the number of times we execute critical section code. One way to greatly improve the performance of the sum function is to have each thread use a *private* variable to store its total contribution to the sum. Then, each thread can add in its contribution to the global sum once, *after* the **for** loop. See Program 1.5. When we run this on the dual core system with $n = 10^8$, the elapsed time is reduced to 1.5 seconds for two threads, a *substantial* improvement.

1.6 Mutexes

Since a thread that is busy-waiting may continually use the CPU, busy-waiting is generally not an ideal solution to the problem of limiting access to a critical section. Two better solutions are mutexes and semaphores. **Mutex** is an abbreviation of *mutual exclusion,* and a mutex is a special type of variable that, together with a couple of special functions, can be used to restrict access to a critical section to a single thread at a time. Thus, a mutex can be used to guarantee that one thread "excludes" all other threads while it executes the critical section. Hence, the mutex guarantees mutually exclusive access to the critical section.

The Pthreads standard includes a special type for mutexes: `pthread_mutex_t`. A variable of type `pthread_mutex_t` needs to be initialized by the system before it's used. This can be done with a call to

```
int pthread_mutex_init(
      pthread_mutex_t*            mutex_p   /* out */,
      const pthread_mutexattr_t*  attr_p    /* in  */);
```

We won't make use of the second argument, so we'll just pass in `NULL` to use the default attributes. You may also occasionally encounter the following *static* mutex initialization that declares a mutex and initializes it in a single line of code:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```c
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

Program 1.5: Global sum function with critical section after loop

Although in general pthread_mutex_init is more flexible, this initialization is fine in many, if not most, cases.

When a Pthreads program finishes using a mutex (regardless of how they are initalized), it should call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p  /* in/out */);
```

The point of a mutex is to protect a critical section from being entered by more than one thread at a time. In order to gain access to a critical section, a thread will lock the mutex, do its work, and then unlock the mutex to let other threads execute the critical section. To lock the mutex and gain exclusive access to the critical section, a thread calls

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p  /* in/out */);
```

When a thread is finished executing the code in a critical section, it should call

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p  /* in/out */);
```

The call to pthread_mutex_lock will cause the thread to wait until no other thread is in the critical section, and the call to pthread_mutex_unlock notifies the system that the calling thread has completed execution of the code in the critical section.

We can use mutexes instead of busy-waiting in our global sum program by declaring a global mutex variable, having the main thread initialize it, and then, instead of busy-waiting and incrementing a flag, the threads call pthread_mutex_lock before entering the critical section, and they call pthread_mutex_unlock when they're done with the critical section. See Program 1.6. The first thread to call pthread_mutex_lock will, effectively, "lock the door" to the critical section: any other thread that attempts to execute the critical section code must first also call pthread_mutex_lock, and until the first thread calls pthread_mutex_unlock, all the threads that have called pthread_mutex_lock will **block** in their calls—they'll just wait until the first thread is done. After the first thread calls pthread_mutex_unlock, the system will choose one of the blocked threads and allow it to execute the code in the critical section. This process will be repeated until all the threads have completed executing the critical section.

"Locking" and "unlocking" the door to the critical section isn't the only metaphor that's used in connection with mutexes. Programmers often say that the thread that has returned from a call to pthread_mutex_lock has "obtained the mutex" or "obtained the lock." When this terminology is used, a thread that calls pthread_mutex_unlock relinquishes the mutex or lock. (You may also encounter terminology referring to this as "acquiring" and "releasing" the lock).

Notice that with mutexes (unlike our busy-waiting solution), the order in which the threads execute the code in the critical section is more or less random: the first thread to call pthread_mutex_lock will be the first to execute the code in the critical section. Subsequent accesses will be scheduled by the system. Pthreads doesn't guarantee that the threads will obtain the lock in the order in which they called Pthread_mutex_lock. However, in our setting, a finite number of threads will try to acquire the lock and they are guaranteed to eventually obtain it.

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

Program 1.6: Global sum function that uses a mutex

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

Table 1.1: Run-times (in seconds) of π programs using $n = 10^8$ terms on a system with two four-core processors

If we look at the (unoptimized) performance of the busy-wait π program (with the critical section after the loop) and the mutex program, we see that for both versions the ratio of the run time of the single-threaded program with the multithreaded program is equal to the number of threads, as long as the number of threads is no greater than the number of cores. That is,

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \texttt{thread\_count},$$

provided `thread_count` is less than or equal to the number of cores. Recall that $T_{\text{serial}}/T_{\text{parallel}}$ is called the *speedup*, and when the speedup is equal to the number of threads, we have achieved more or less “ideal” performance or *linear speedup*.

If we compare the performance of the version that uses busy-waiting with the version that uses mutexes, we don’t see much difference in the overall run time when the programs are run with fewer threads than cores. This shouldn’t be surprising, as each thread only enters the critical section once; unless the critical section is very long, or the Pthreads functions are very slow, we wouldn’t expect the threads to be delayed very much by waiting to enter the critical section. However, if we start increasing the number of threads beyond the number of cores, the performance of the version that uses mutexes remains largely unchanged, while the performance of the busy-wait version degrades. See Table 1.1.

We see that when we use busy-waiting, performance can degrade if there are more threads than cores.[5] This should make sense. For example, suppose we have two cores and five threads. Also suppose that thread 0 is in the critical section, thread 1 is in the busy-wait loop, and threads 2, 3, and 4 have been descheduled by the operating system. After thread 0 completes the critical section and sets `flag` = 1, it will be terminated, and thread 1 can enter the critical section so the operating

[5] These are typical run-times. When using busy-waiting and the number of threads is greater than the number of cores, the run-times vary considerably.

Time	flag	Thread				
		0	1	2	3	4
0	0	crit sect	busy wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy wait	susp
2	2	—	terminate	susp	busy wait	busy wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy wait

Table 1.2: Possible sequence of events with busy-waiting and more threads than cores

system can schedule thread 2, thread 3, or thread 4. Suppose it schedules thread 3, which will spin in the **while** loop. When thread 1 finishes the critical section and sets `flag` = 2, the operating system can schedule thread 2 or thread 4. If it schedules thread 4, then both thread 3 and thread 4, will be busily spinning in the busy-wait loop until the operating system deschedules one of them and schedules thread 2. See Table 1.2.

1.7 Producer-Consumer Synchronization and Semaphores

Although busy-waiting is generally wasteful of CPU resources, it does have the property that we know, in advance, the order in which the threads will execute the code in the critical section: thread 0 is first, then thread 1, then thread 2, and so on. With mutexes, the order in which the threads execute the critical section is left to chance and the system. Since addition is commutative, this doesn't matter in our program for estimating π. However, it's not difficult to think of situations in which we also want to control the order in which the threads execute the code in the critical section. For example, suppose each thread generates an $n \times n$ matrix, and we want to multiply the matrices together in thread-rank order. Since matrix multiplication isn't commutative, our mutex solution would have problems:

```
/* n and product_matrix are shared and initialized by the main thread */
/* product_matrix is initialized to be the identity matrix             */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);
   Free_matrix(&my_mat);
   return NULL;
}  /* Thread_work */
```

A somewhat more complicated example involves having each thread "send a message" to another thread. For example, suppose we have `thread_count` or t threads and we want thread 0 to send a message to thread 1, thread 1 to send a message to thread 2, ..., thread $t-2$ to send a message to thread $t-1$ and thread $t-1$ to send a message to thread 0. After a thread "receives" a message, it can print the message and terminate. In order to implement the message transfer, we can allocate a shared array of **char***. Then each thread can allocate storage for the message it's sending, and, after it has initialized the message, set a pointer in the shared array to refer to it. In order to avoid dereferencing undefined pointers, the main thread can set the individual entries in the shared array to `NULL`. See Program 1.7. When we run the program with more than a couple of threads on a dual core system, we see that some of the messages are never received. For example, thread 0, which is started first, will typically finish before thread $t-1$ has copied the message into the `messages` array.

```
 1  /* messages has type char**. It's allocated in main. */
 2  /* Each entry is set to NULL in main.                 */
 3  void *Send_msg(void* rank) {
 4     long my_rank = (long) rank;
 5     long dest = (my_rank + 1) % thread_count;
 6     long source = (my_rank + thread_count - 1) % thread_count;
 7     char* my_msg = malloc(MSG_MAX*sizeof(char));
 8
 9     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10     messages[dest] = my_msg;
11
12     if (messages[my_rank] != NULL)
13        printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
14     else
15        printf("Thread %ld > No message from %ld\n", my_rank, source);
16
17     return NULL;
18  }  /* Send_msg */
```

Program 1.7: A first attempt at sending messages using pthreads

This isn't surprising, and we could fix the problem by replacing the **if** statement in Line 12 with a busy-wait **while** statement:

```
   while (messages[my_rank] == NULL);
   printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
```

Of course, this solution would have the same problems that any busy-waiting solution has, so we'd prefer a different approach.

After executing the assignment in Line 10, we'd like to "notify" the thread with rank `dest` that it can proceed to print the message. We'd like to do something like this:

```
. . .
messages[dest] = my_msg;
Notify thread dest that it can proceed;

Await notification from thread source
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
. . .
```

It's not at all clear how mutexes can help here. We might try calling `pthread_mutex_unlock` to "notify" the thread `dest`. However, mutexes are initialized to be *unlocked*, so we'd need to add a call *before* initializing `messages[dest]` to lock the mutex. This will be a problem since we don't know when the threads will reach the calls to `pthread_mutex_lock`.

To make this a little clearer, suppose that the main thread creates and initializes an array of mutexes, one for each thread. Then we're trying to do something like this:

```
1     . . .
2     pthread_mutex_lock(&mutex[dest]);
3     . . .
4     messages[dest] = my_msg;
5     pthread_mutex_unlock(&mutex[dest]);
6     . . .
7     pthread_mutex_lock(&mutex[my_rank]);
8     printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
9     . . .
```

Now suppose we have two threads, and thread 0 gets so far ahead of thread 1 that it reaches the second call to `pthread_mutex_lock` in Line 7 before thread 1 reaches the first in Line 2. Then, of course, it will acquire the lock and continue to the `printf` statement. This will result in thread 0 dereferencing a null pointer and it will crash.

There *are* other approaches to solving this problem with mutexes. See, for example, Exercise 8. However, POSIX® also provides a somewhat different means of controlling access to critical sections: **semaphores.** Let's take a look at them.

A semaphore can be thought of as a special type of **unsigned int**, so they take on the values 0, 1, 2, ... In many cases, we'll only be interested in using them when they take on the values 0 and 1. A semaphore that only takes on these values is called a *binary* semaphore. Very roughly speaking, 0 corresponds to a locked mutex, and 1 corresponds to an unlocked mutex. To use a binary semaphore as a mutex, you *initialize* it to 1: "unlocked." Before the critical section you want to protect, you place a call to the function `sem_wait`. A thread that executes `sem_wait` will block if the semaphore is 0. If the semaphore is nonzero, it will *decrement* the semaphore and proceed. After executing the code in the critical section, a thread calls `sem_post`, which *increments* the semaphore and a thread waiting in `sem_wait` can proceed.

Semaphores were first defined by the computer scientist Edsger Dijkstra in [**?**]. The name is taken from the mechanical device that railroads use to control which train can use a track. The device consists of an arm attached by a pivot to a post. When the arm points down, approaching trains can proceed, and when the arm is perpendicular to the post, approaching trains must stop and wait. The track corresponds to the critical section: when the arm is down corresponds to a semaphore of 1, and when the arm is up corresponds to a semaphore of 0. The `sem_wait` and `sem_post` calls correspond to signals sent by the train to the semaphore controller.

For our current purposes, the crucial difference between semaphores and mutexes is that there is no ownership associated with a semaphore. The main thread can initialize all of the semaphores to 0—that is, “locked”—and then any thread can execute a `sem_post` on any of the semaphores. Similarly, any thread can execute `sem_wait` on any of the semaphores. Thus, if we use semaphores, our `Send_msg` function can be written as shown in Program 1.8.

```
/* messages is allocated and initialized to NULL in main          */
/* semaphores is allocated and initialized to 0 (locked) in main */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;
   sem_post(&semaphores[dest]);  /* ‘‘Unlock’’ the semaphore of dest */

   /* Wait for our semaphore to be unlocked */
   sem_wait(&semaphores[my_rank]);
   printf("Thread %ld > %s\n", my_rank, messages[my_rank]);

   return NULL;
}  /* Send_msg */
```

Program 1.8: Using semaphores so that threads can send messages

The syntax of the various semaphore functions is

```
int sem_init(
      sem_t*     semaphore_p   /* out */,
      int        shared        /* in  */,
        unsigned  initial_val   /* in   */);

int sem_destroy(sem_t*  semaphore_p  /* in/out */);
int sem_post(sem_t*     semaphore_p  /* in/out */);
int sem_wait(sem_t*     semaphore_p  /* in/out */);
```

The second argument to sem_init controls whether the semaphore is shared among threads or processes. In our examples, we'll be sharing the semaphore among threads, so the constant 0 can be passed in.

Note that semaphores are part of the POSIX® standard, but *not* part of Pthreads. Hence it is necessary to ensure your operating system does indeed support semaphores, and then add the following preprocessor directive to any program that uses them:[6]

```
#include <semaphore.h>
```

Finally, note that the message-sending problem didn't involve a critical section. The problem wasn't that there was a block of code that could only be executed by one thread at a time. Rather, thread my_rank couldn't proceed until thread source had finished creating the message. This type of synchronization, when a thread can't proceed until another thread has taken some action, is sometimes called **producer-consumer synchronization**. For example, imagine a *producer* thread that generates tasks and places them in a fixed-size queue (or *bounded buffer*) for a *consumer* thread to execute. In this case, the consumer blocks until at least one task is ready, at which point it will be signaled by the producer. Once signaled, the work is carried out by the thread in isolation; no critical section is involved. This paradigm is seen in stream processing, web servers, and so on; in the case of a web server, the producer thread could listen for incoming request URIs and place them in the queue, while the consumer would be responsible for reading the corresponding file from disk (e.g., http://server/file.txt might be located at /www/file.txt on the web server's file system) and sending data back to the client that requested the URI.

As mentioned earlier, binary semaphores (those that only take on the values 0 and 1) are fairly typical. However, *counting* semaphores can also be useful in scenarios where we wish to restrict access to a finite resource. One common example is an application design pattern that involves limiting the number of threads used by a program to be no more than the number of cores available on a given machine. Consider a program with a workload of N tasks, where N is much greater than the available cores. In this case, the main thread is responsible for distributing the workload and would initialize its semaphore with the number of cores available, and then call sem_wait before starting each worker thread with pthread_create. Once the counter reaches 0, the main thread will block; the machine has a task running for each core and the program must wait for a thread to finish before starting more. When a thread does finish its task, it will call sem_post to signal that

[6] Some systems, including macOS, don't support this version of semaphores. However, they may support something called "named" semaphores. The functions sem_wait and sem_post can be used in the same way. However, sem_init should be replaced by sem_open, and sem_destroy should be replaced by sem_close and sem_unlink. See the book's website for an example.

the main thread can create another worker thread. For this approach to be efficient, the amount of time spent on each task much be longer than the thread creation overhead because *N* total threads will be started during the program's execution. For an approach that reuses existing threads in a *thread pool*, see Programming Assignment 5.

1.8 Barriers and Condition Variables

Let's take a look at another problem in shared-memory programming: synchronizing the threads by making sure that they all are at the same point in a program. Such a point of synchronization is called a **barrier** because no thread can proceed beyond the barrier until all the threads have reached it.

Barriers have numerous applications. As we discussed in Chapter **??** if we're timing some part of a multithreaded program, we'd like for all the threads to start the timed code at the same instant, and then report the time taken by the last thread to finish, i.e., the "slowest" thread. So we'd like to do something like this:

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish − my_start;

elapsed = Maximum of my_elapsed values;
```

Using this approach, we're sure that all of the threads will record `my_start` at approximately the same time.

Another very important use of barriers is in debugging. As you've probably already seen, it can be very difficult to determine *where* an error is occurring in a parallel program. We can, of course, have each thread print a message indicating which point it's reached in the program, but it doesn't take long for the volume of the output to become overwhelming. Barriers provide an alternative:

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
```

```
      fflush(stdout);
   }
```

Many implementations of Pthreads don't provide barriers, so if our code is to be portable, we need to develop our own implementation. There are a number of options; we'll look at three. The first two only use constructs that we've already studied. The third uses a new type of Pthreads object: a *condition variable.*

1.8.1 Busy-waiting and a Mutex

Implementing a barrier using busy-waiting and a mutex is straightforward: we use a shared counter protected by the mutex. When the counter indicates that every thread has entered the critical section, threads can leave the busy-wait loop.

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   . . .
}
```

Of course, this implementation will have the same problems that our other busy-wait codes had: we'll waste CPU cycles when threads are in the busy-wait loop, and, if we run the program with more threads than cores, we may find that the performance of the program seriously degrades.

Another issue is the shared variable `counter`. What happens if we want to implement a second barrier and we try to reuse the counter? When the first barrier is completed, `counter` will have the value `thread_count`. Unless we can somehow reset `counter`, the **while** condition we used for our first barrier `counter` < `thread_count` will be false, and the barrier won't cause the threads to block. Furthermore any attempt to reset `counter` to zero is almost certainly doomed to failure. If the last thread to enter the loop tries to reset it, some thread in the busy-wait may never see the fact that `counter` == `thread_count`, and that thread may hang in the busy-wait. If some thread tries to reset the counter after the barrier, some other thread may enter the second barrier before the counter is reset and its increment to the counter will be lost. This will have the unfortunate effect of causing all the threads to hang in the second busy-wait loop. So if we want to use this barrier, we need one counter variable for each instance of the barrier.

1.8.2 Semaphores

A natural question is whether we can implement a barrier with semaphores, and, if so, whether we can reduce the number of problems we encountered with busy-waiting. The answer to the first question is yes:

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
}
```

As with the busy-wait barrier, we have a counter that we use to determine how many threads have entered the barrier. We use two semaphores: `count_sem` protects the counter, and `barrier_sem` is used to block threads that have entered the barrier. The `count_sem` semaphore is initialized to 1 (that is, "unlocked"), so the first thread to reach the barrier will be able to proceed past the call to `sem_wait`. Subsequent threads, however, will block until they can have exclusive access to the counter. When a thread has exclusive access to the counter, it checks to see if `counter` < `thread_count`-1. If it is, the thread increments `counter`, relinquishes the lock (`sem_post(&count_sem)`), and blocks in `sem_wait(&barrier_sem)`. On the other hand, if `counter` == `thread_count`-1, the thread is the last to enter the barrier, so it can reset `counter` to zero and "unlock" `count_sem` by calling `sem_post(&count_sem)`. Now, it wants to notify all the other threads that they can proceed, so it executes `sem_post(&barrier_sem)` for each of the `thread_count`-1 threads that are blocked in sem_wait(&barrier_sem).

Note that it doesn't matter if the thread executing the loop of calls to sem_post(&barrier_sem) races ahead and executes multiple calls to sem_post before a thread can be unblocked from sem_wait(&barrier_se Recall that a semaphore is an **unsigned int**, and the calls to sem_post increment it, while the calls to sem_wait decrement it—unless it's already 0. If it's 0, the calling threads will block until it's positive again. Therefore, it doesn't matter if the thread executing the loop of calls to sem_post(&barrier_sem) gets ahead of the threads blocked in the calls to sem_wait(&barrier_sem), because eventually the blocked threads will see that barrier_sem is positive, and they'll decrement it and proceed.

It should be clear that this implementation of a barrier is superior to the busy-wait barrier, since the threads don't need to consume CPU cycles when they're blocked in sem_wait. Can we reuse the data structures from the first barrier if we want to execute a second barrier?

The counter can be reused, since we were careful to reset it before releasing any of the threads from the barrier. Also, count_sem can be reused, since it is reset to 1 before any threads can leave the barrier. This leaves barrier_sem. Since there's exactly one sem_post for each sem_wait, it might appear that the value of barrier_sem will be 0 when the threads start executing a second barrier. However, suppose we have two threads, and thread 0 is blocked in sem_wait(&barrier_sem) in the first barrier, while thread 1 is executing the loop of calls to sem_post. Also suppose that the operating system has seen that thread 0 is idle, and descheduled it out. Then thread 1 can go on to the second barrier. Since counter == 0, it will execute the **else** clause. After incrementing counter, it executes sem_post(&count_sem), and then executes sem_wait(&barrier_sem).

However, if thread 0 is still descheduled, it will not have decremented barrier_sem. Thus when thread 1 reaches sem_wait(&barrier_sem), barrier_sem will still be 1, so it will simply decrement barrier_sem and proceed. This will have the unfortunate consequence that when thread 0 starts executing again, it will still be blocked in the *first* sem_wait(&barrier_sem), and thread 1 will proceed through the second barrier before thread 0 has entered it. Reusing barrier_sem therefore results in a race condition.

1.8.3 Condition Variables

A somewhat better approach to creating a barrier in Pthreads is provided by *condition variables.* A **condition variable** is a data object that allows a thread to suspend execution until a certain event or *condition* occurs. When the event or condition occurs another thread can *signal* the thread to "wake up." A condition variable is *always* associated with a mutex.

Typically, condition variables are used in constructs similar to this pseudocode:

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

Condition variables in Pthreads have type `pthread_cond_t`. The function

```
int pthread_cond_signal(pthread_cond_t*  cond_var_p   /* in/out */);
```

will unblock *one* of the blocked threads, and

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p  /* in/out */);
```

will unblock *all* of the blocked threads. This is one advantage of condition variables; recall that we needed a **for** loop calling `sem_post` to achieve similar functionality with semaphores. The function

```
int pthread_cond_wait(
      pthread_cond_t*     cond_var_p    /* in/out */,
      pthread_mutex_t*    mutex_p       /* in/out */);
```

will unlock the mutex referred to by `mutex_p` and cause the executing thread to block until it is unblocked by another thread's call to `pthread_cond_signal` or `pthread_cond_broadcast`. When the thread is unblocked, it reacquires the mutex. So in effect, `pthread_cond_wait` implements the following sequence of functions:

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

The following code implements a barrier with a condition variable:

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
```

```
      pthread_mutex_unlock(&mutex);
      . . .
   }
```

Note that it is possible that events other than the call to pthread_cond_broadcast can cause a suspended thread to unblock (see, for example, Butenhof [**?**], page 80). This is called a *spurious wake-up*. Hence, the call to pthread_cond_wait should usually be placed in a **while** loop. If the thread is unblocked by some event other than a call to pthread_cond_signal or pthread_cond_broadcast, then the return value of pthread_cond_wait will be nonzero, and the unblocked thread will call pthread_cond_wait again.

If a single thread is being awakened, it's also a good idea to check that the condition has, in fact, been satisfied before proceeding. In our example, if a single thread were being released from the barrier with a call to pthread_cond_signal, then that thread should verify that counter == 0 before proceeding. This can be dangerous with the broadcast, though. After being awakened, some thread may race ahead and change the condition, and if each thread is checking the condition, a thread that awakened later may find the condition is no longer satisfied and go back to sleep.

Note that in order for our barrier to function correctly, it's essential that the call to pthread_cond_wait unlock the mutex. If it didn't unlock the mutex, then only one thread could enter the barrier; all of the other threads would block in the call to pthread_mutex_lock, the first thread to enter the barrier would block in the call to pthread_cond_wait, and our program would hang.

Also note that the semantics of mutexes require that the mutex be relocked before we return from the call to pthread_cond_wait. We obtained the lock when we returned from the call to pthread_mutex_lock. Hence, we should at some point relinquish the lock through a call to pthread_mutex_unlock.

Like mutexes and semaphores, condition variables should be initialized and destroyed. In this case, the functions are

```
int pthread_cond_init(
      pthread_cond_t*              cond_p       /* out */,
      const pthread_condattr_t*    cond_attr_p  /* in  */);

int pthread_cond_destroy(pthread_cond_t*  cond_p  /* in/out */);
```

We won't be using the second argument to pthread_cond_init — as with mutexes, the default the attributes are fine for our purposes — so we'll call it with second argument set to NULL. As usual, there is also a *static* version of the initializer if we are planning to use the default attributes:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Condition variables are often quite useful whenever a thread needs to wait for something. When protected application state cannot be represented by an unsigned integer counter, condition variables may be preferable to semaphores.

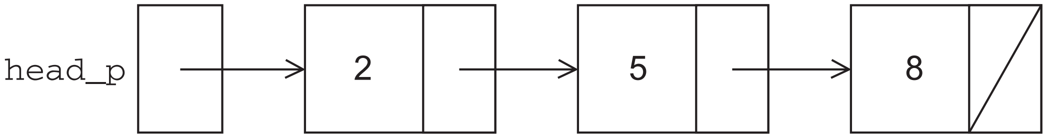

Figure 1.4: A linked list

1.8.4 Pthreads Barriers

Before proceeding we should note that the Open Group, the standards group that is continuing to develop the POSIX® standard, does define a barrier interface for Pthreads. However, as we noted earlier, it is not universally available, so we haven't discussed it in the text. See Exercise 10 for some of the details of the API.

1.9 Read-Write Locks

Let's take a look at the problem of controlling access to a large, shared data structure, which can be either simply searched or updated by the threads. For the sake of explicitness, let's suppose the shared data structure is a sorted, singly-linked list of **int**s, and the operations of interest are `Member`, `Insert`, and `Delete`.

1.9.1 Sorted linked list functions

The list itself is composed of a collection of list *nodes*, each of which is a struct with two members: an **int** and a pointer to the next node. We can define such a struct with the definition

```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```

A typical list is shown in Figure 1.4. A pointer, `head_p`, with type **struct** `list_node_s*` refers to the first node in the list. The `next` member of the last node is `NULL` (which is indicated by a slash (/) in the `next` member).

The `Member` function (Program 1.9) uses a pointer to traverse the list until it either finds the desired value or determines that the desired value cannot be in the list. Since the list is sorted, the latter condition occurs when the curr_p pointer is NULL or when the data member of the current node is larger than the desired value.

```
int  Member(int value, struct list_node_s* head_p) {
   struct list_node_s* curr_p = head_p;

   while (curr_p != NULL && curr_p->data < value)
      curr_p = curr_p->next;

   if (curr_p == NULL || curr_p->data > value) {
      return 0;
   } else {
      return 1;
   }
}  /* Member */
```

Program 1.9: The Member function

The Insert function (Program 1.10) begins by searching for the correct position in which to insert the new node. Since the list is sorted, it must search until it finds a node whose data member is greater than the value to be inserted. When it finds this node, it needs to insert the new node in the position *preceding* the node that's been found. Since the list is singly-linked, we can't "back up" to this position without traversing the list a second time. There are several approaches to dealing with this; the approach we use is to define a second pointer pred_p, which, in general, refers to the predecessor of the current node. When we exit the loop that searches for the position to insert, the next member of the node referred to by pred_p can be updated so that it refers to the new node. See Figure 1.5.

The Delete function (Program 1.11) is similar to the Insert function in that it also needs to keep track of the predecessor of the current node while it's searching for the node to be deleted. The predecessor node's next member can then be updated after the search is completed. See Figure 1.6.

1.9.2 A Multithreaded Linked List

Now let's try to use these functions in a Pthreads program. In order to share access to the list, we can define head_p to be a global variable. This will simplify the function headers for Member, Insert, and Delete, since we won't need to pass in either head_p or a pointer to head_p, we'll only need to pass in the value of interest. What now are the consequences of having multiple threads simultaneously execute the three functions?

```c
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node_s));
      temp_p->data = value;
      temp_p->next = curr_p;
      if (pred_p == NULL)  /* New first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
}  /* Insert */
```

Program 1.10: The `Insert` function

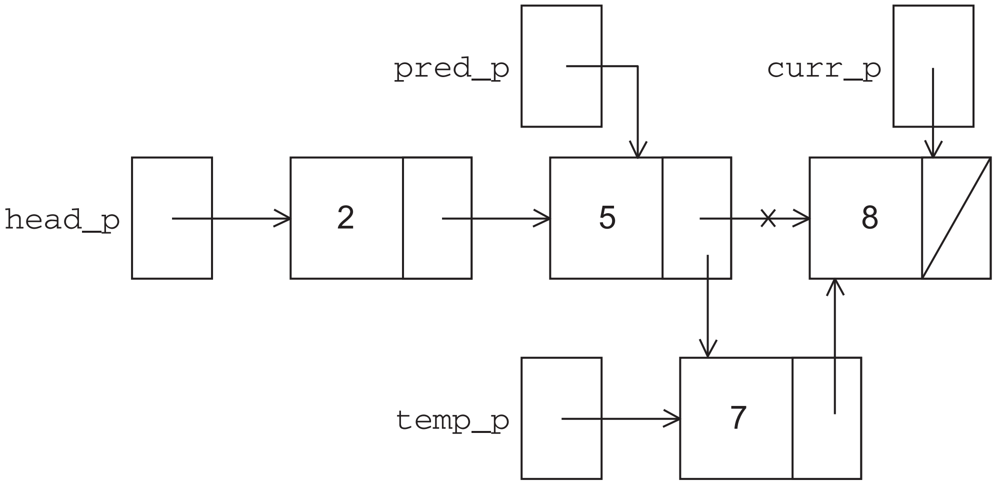

Figure 1.5: Inserting a new node into a list

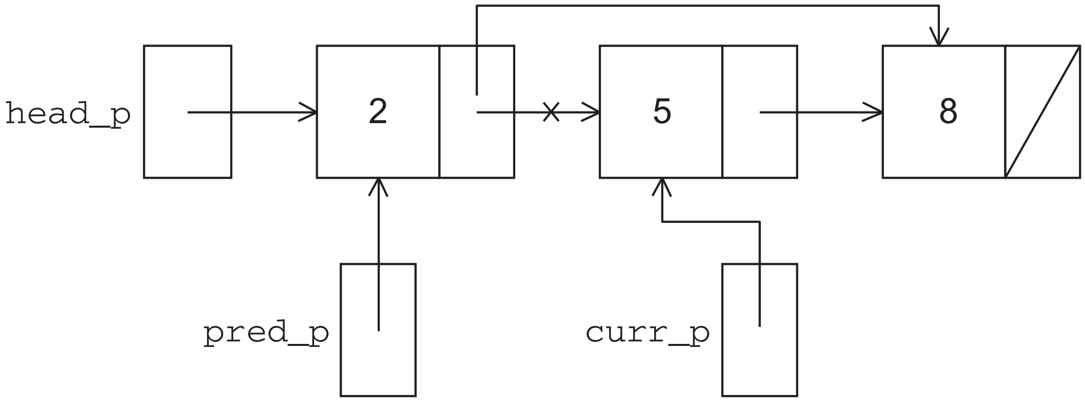

Figure 1.6: Deleting a node from the list

```
int Delete(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p != NULL && curr_p->data == value) {
      if (pred_p == NULL) { /* Deleting first node in list */
         *head_pp = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else { /* Value isn't in list */
      return 0;
   }
}  /* Delete */
```

Program 1.11: The `Delete` function

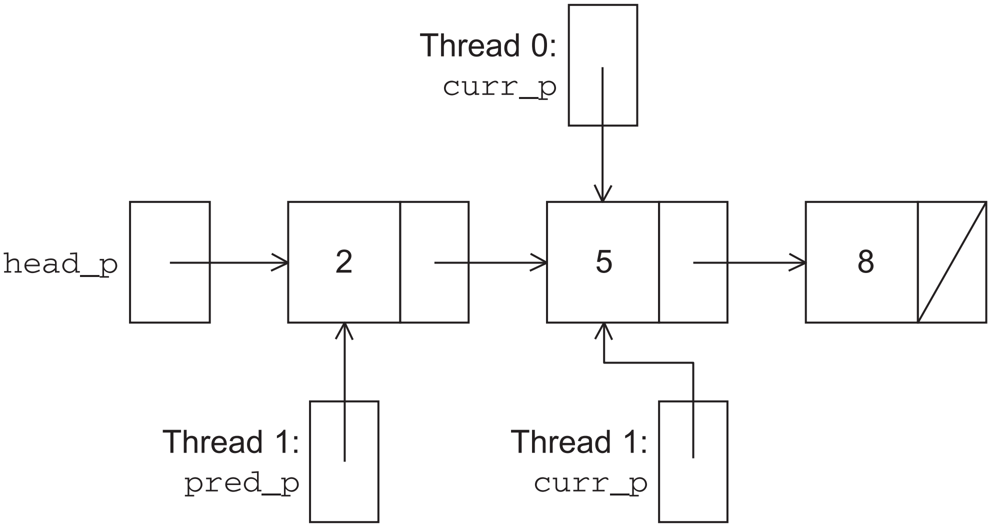

Figure 1.7: Simultaneous access by two threads

Since multiple threads can simultaneously *read* a memory location without conflict, it should be clear that multiple threads can simultaneously execute `Member`. On the other hand, `Delete` and `Insert` also *write* to memory locations, so there may be problems if we try to execute either of these operations at the same time as another operation. As an example, suppose that thread 0 is executing `Member`(5) at the same time that thread 1 is executing `Delete`(5), and the current state of the list is shown in Figure 1.7. An obvious problem is that if thread 0 is executing `Member`(5), it is going to report that 5 is in the list, when, in fact, it may be deleted even before thread 0 returns. A second obvious problem is if thread 0 is executing `Member`(8), thread 1 may free the memory used for the node storing 5 before thread 0 can advance to the node storing 8. Although typical implementations of `free` don't overwrite the freed memory, if the memory is reallocated before thread 0 advances, there can be serious problems. For example, if the memory is reallocated for use in something other than a list node, what thread 0 "thinks" is the `next` member may be set to utter garbage, and after it executes

```
curr_p = curr_p->next;
```

dereferencing `curr_p` may result in a segmentation violation.

More generally, we can run into problems if we try to simultaneously execute another operation while we're executing an `Insert` or a `Delete`. It's OK for multiple threads to simultaneously execute `Member`—that is, *read* the list nodes—but it's unsafe for multiple threads to access the list if at least one of the threads is executing an `Insert` or a `Delete`—that is, is *writing* to the list nodes (see Exercise 12).

How can we deal with this problem? An obvious solution is to simply lock the list any time that a thread attempts to access it. For example, a call to each of the three functions can be protected by a mutex, so we might execute

```
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```

instead of simply calling `Member(value)`.

An equally obvious problem with this solution is that we are serializing access to the list, and if the vast majority of our operations are calls to `Member`, we'll fail to exploit this opportunity for parallelism. On the other hand, if most of our operations are calls to `Insert` and `Delete`, then this may be the best solution, since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.

An alternative to this approach involves "finer-grained" locking. Instead of locking the entire list, we could try to lock individual nodes. We would add, for example, a mutex to the list node struct:

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```

Now each time we try to access a node we must first lock the mutex associated with the node. Note that this will also require that we have a mutex associated with the `head_p` pointer. So, for example, we might implement `Member` as shown in Program 1.12. Admittedly this implementation is *much* more complex than the original `Member` function. It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked. At a minimum it will add two function calls to the node access, but it can also add a substantial delay if a thread has to wait for a lock. A further problem is that the addition of a mutex field to each node will substantially increase the amount of storage needed for the list. On the other hand, the finer-grained locking might be a closer approximation to what we want. Since we're only locking the nodes of current interest, multiple threads can simultaneously access different parts of the list, regardless of which operations they're executing.

1.9.3 Pthreads Read-Write Locks

Neither of our multithreaded linked lists exploits the potential for simultaneous access to *any* node by threads that are executing `Member`. The first solution only allows one thread to access the entire list at any instant, and the second only allows one thread to access any given node at any instant. An alternative is provided by Pthreads' **read-write locks**. A read-write lock is somewhat like a mutex except that it provides *two* lock functions. The first lock function locks the read-write lock

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

Program 1.12: Implementation of `Member` with one mutex per list node for *reading*, while the second locks it for *writing*. Multiple threads can thereby simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function. Thus, if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function. Furthermore, if any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.

Using Pthreads read-write locks, we can protect our linked list functions with the following code (we're ignoring function return values):

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

The syntax for the new Pthreads functions is

```
int pthread_rwlock_rdlock(pthread_rwlock_t*  rwlock_p  /* in/out */);
int pthread_rwlock_wrlock(pthread_rwlock_t*  rwlock_p  /* in/out */);
int pthread_rwlock_unlock(pthread_rwlock_t*  rwlock_p  /* in/out */);
```

As their names suggest, the first function locks the read-write lock for reading, the second locks it for writing, and the last unlocks it.

As with mutexes, read-write locks should be initialized before use and destroyed after use. The following function can be used for initialization:

```
int pthread_rwlock_init(
       pthread_rwlock_t*            rwlock_p  /* out */,
       const pthread_rwlockattr_t*  attr_p    /* in  */);

/* And, the static version: */
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Also as with mutexes, we'll not use the second argument, so we'll just pass `NULL`. The following function can be used for destruction of a read-write lock:

```
int pthread_rwlock_destroy(pthread_rwlock_t*  rwlock_p  /* in/out*/);
```

	Number of Threads			
Implementation	1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

Table 1.3: Linked list times: 100,000 ops/thread, 99.9% `Member,` 0.05% `Insert`, 0.05% `Delete`

	Number of Threads			
Implementation	1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

Table 1.4: Linked List Times: 100,000 ops/thread, 80% `Member,` 10% `Insert`, 10% `Delete`

1.9.4 Performance of the Various Implementations

Of course, we really want to know which of the three implementations is “best,” so we included our implementations in a small program in which the main thread first inserts a user-specified number of randomly generated keys into an empty list. After being started by the main thread, each thread carries out a user-specified number of operations on the list. The user also specifies the percentages of each type of operation (`Member`, `Insert`, `Delete`). However, which operation occurs when and on which key is determined by a random number generator. Thus, for example, the user might specify that 1,000 keys should be inserted into an initially empty list and a total of 100,000 operations are to be carried out by the threads. Further, she might specify that 80% of the operations should be `Member`, 15% should be `Insert`, and the remaining 5% should be `Delete`. However, since the operations are randomly generated, it might happen that the threads execute a total of, say, 79,000 calls to `Member`, 15,500 calls to `Insert`, and 5500 calls to `Delete`.

Tables 1.3 and 1.4 show the times (in seconds) that it took for 100,000 operations on a list that was initialized to contain 1000 keys. Both sets of data were taken on a system containing four dual-core processors.

Table 1.3 shows the times when 99.9% of the operations are `Member` and the remaining 0.1% are divided equally between `Insert` and `Delete`. Table 1.4 shows the times when 80% of the operations are `Member`, 10% are `Insert`, and 10% are Delete. Note that in both tables when one thread is used, the run times for the read-write locks and the single-mutex implementations are about the same. This makes sense: the operations are serialized, and since there is no contention for the read-write lock or the mutex, the overhead associated with both implementations should consist of a function call before the list operation and a function call after the operation. On the other hand, the implementation that uses one mutex per node is *much* slower. This also makes sense since each time a single node is accessed there will be two function calls—one to lock the node mutex and one to unlock it. Thus, there's considerably more overhead for this implementation.

The inferiority of the implementation that uses one mutex per node persists when we use multiple threads. There is far too much overhead associated with all the locking and unlocking to make this implementation competitive with the other two implementations.

Perhaps the most striking difference between the two tables is the relative performance of the read-write lock implementation and the single-mutex implementation when multiple threads are used. When there are very few `Insert`s and `Delete`s, the read-write lock implementation is far better than the single-mutex implementation. Since the single-mutex implementation will serialize all the operations, this suggests that if there are very few `Insert`s and `Delete`s, the read-write locks do a very good job of allowing concurrent access to the list. On the other hand, if there are a relatively large number of `Insert`s and `Delete`s (for example, 10% each), there's very little difference between the performance of the read-write lock implementation and the single-mutex implementation. Thus, for linked list operations, read-write locks *can* provide a considerable increase in performance, but only if the number of `Insert`s and `Delete`s is quite small.

Also notice that if we use one mutex or one mutex per node, the program is *always* as fast or faster when it's run with one thread. Furthermore, when the number of `Insert`s and `Delete`s is relatively large, the read-write lock program is also faster with one thread. This isn't surprising for the one mutex implementation, since effectively accesses to the list are serialized. For the read-write lock implementation, it appears that when there are a substantial number of write locks, there is too much contention for the locks and overall performance deteriorates significantly.

In summary, the read-write lock implementation is superior to the single mutex and one mutex per node implementations. However, unless the number of `Insert`s and `Delete`s is small, a serial implementation will be superior.

1.9.5 Implementing read-write locks

The original Pthreads specification didn't include read-write locks, so some of the early texts describing Pthreads include implementations of read-write locks (see, for example, [**?**]). A typical implementation[7] defines a data structure that uses two condition variables—one for "readers" and one for "writers"—and a mutex. The structure also contains members that indicate

1. how many readers own the lock, that is, are currently reading,
2. how many readers are waiting to obtain the lock,
3. whether a writer owns the lock, and

[7] This discussion follows the basic outline of Butenhof's implementation [**?**].

4. how many writers are waiting to obtain the lock.

The mutex protects the read-write lock data structure: whenever a thread calls one of the functions (read-lock, write-lock, unlock), it first locks the mutex, and whenever a thread completes one of these calls, it unlocks the mutex. After acquiring the mutex, the thread checks the appropriate data members to determine how to proceed. As an example, if it wants read-access, it can check to see if there's a writer that currently owns the lock. If not, it increments the number of active readers and proceeds. If a writer is active, it increments the number of readers waiting and starts a condition wait on the reader condition variable. When it's awakened, it decrements the number of readers waiting, increments the number of active readers, and proceeds. The write-lock function has an implementation that's similar to the read-lock function.

The action taken in the unlock function depends on whether the thread was a reader or a writer. If the thread was a reader, there are no currently active readers, *and* there's a writer waiting, then it can signal a writer to proceed before returning. If, on the other hand, the thread was a writer, there can be both readers and writers waiting, so the thread needs to decide whether it will give preference to readers or writers. Since writers must have exclusive access, it is likely that it is much more difficult for a writer to obtain the lock. Many implementations therefore give writers preference. Programming Assignment 6 explores this further.

1.10 Caches, Cache-Coherence, and False Sharing[8]

Recall that for a number of years now, computers have been able to execute operations involving only the processor much faster than they can access data in main memory. If a processor must read data from main memory for each operation, it will spend most of its time simply waiting for the data from memory to arrive. Also recall that in order to address this problem, chip designers have added blocks of relatively fast memory to processors. This faster memory is called **cache memory**.

The design of cache memory takes into consideration the principles of **temporal and spatial locality**: if a processor accesses main memory location x at time t, then it is likely that at times close to t it will access main memory locations close to x. Thus, if a processor needs to access main memory location x, rather than transferring only the contents of x to/from main memory, a block of memory containing x is transferred from/to the processor's cache. Such a block of memory is called a **cache line** or **cache block**.

In Section **??**, we saw that the use of cache memory can have a huge impact on shared-memory. Let's recall why. First, consider the following situation: Suppose `x` is a shared variable with the value five, and both thread 0 and thread 1 read `x` from memory into their (separate) caches, because both want to execute the statement

[8] This material is also covered in Chapter **??**. So if you've already read that chapter, you may want to skim this section.

```
my_y = x;
```

Here, `my_y` is a private variable defined by both threads. Now suppose thread 0 executes the statement

```
x++;
```

Finally, suppose that thread 1 now executes

```
my_z = x;
```

where `my_z` is another private variable.

What's the value in `my_z`? Is it five? Or is it six? The problem is that there are (at least) three copies of `x`: the one in main memory, the one in thread 0's cache, and the one in thread 1's cache. When thread 0 executed `x++`, what happened to the values in main memory and thread 1's cache? This is the **cache coherence** problem, which we discussed in Chapter **??**. We saw there that most systems insist that the caches be made aware that changes have been made to data they are caching. The line in the cache of thread 1 would have been marked *invalid* when thread 0 executed `x++`, and before assigning `my_z = x`, the core running thread 1 would see that its value of `x` was out of date. Thus, the core running thread 0 would have to update the copy of `x` in main memory (either now or earlier), and the core running thread 1 would get the line with the updated value of `x` from main memory. For further details, see Chapter **??**.

The use of cache coherence can have a dramatic effect on the performance of shared-memory systems. To illustrate this, recall our Pthreads matrix-vector multiplication example: the main thread initialized an $m \times n$ matrix A and an n-dimensional vector $\mathbf{x}$. Each thread was responsible for computing m/t components of the product vector $\mathbf{y} = A\mathbf{x}$. (As usual, t is the number of threads.) The data structures representing A, $\mathbf{x}$, $\mathbf{y}$, m, and n were all shared. For ease of reference, we reproduce the code in Program 1.13.

If T_{serial} is the run time of the serial program and T_{parallel} is the run time of the parallel program, recall that the *efficiency* E of the parallel program is the speedup S divided by the number of threads:

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

Since $S \leq t$, $E \leq 1$. Table 1.5 shows the run-times and efficiencies of our matrix-vector multiplication with different sets of data and differing numbers of threads.

In each case, the total number of floating point additions and multiplications is $64,000,000$; so an analysis that only considers arithmetic operations would predict that a single thread running the code would take the same amount of time for all three inputs. However, it's clear that this is *not* the case. With one thread, the $8,000,000 \times 8$ system requires about 14% more time than the 8000×8000 system, and the $8 \times 8,000,000$ system requires about 28% more time than the 8000×8000 system. Both of these differences are at least partially attributable to cache performance

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```

Program 1.13: Pthreads matrix-vector multiplication

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

Table 1.5: Run-times and efficiencies of matrix-vector multiplication (times are in seconds)

Recall that a *write-miss* occurs when a core tries to update a variable that's not in the cache, and it has to access main memory. A cache profiler (such as Valgrind [**?**]) shows that when the program is run with the $8,000,000 \times 8$ input, it has far more cache write-misses than either of the other inputs. The bulk of these occur in Line 9. Since the number of elements in the vector `y` is far greater in this case (8,000,000 vs. 8000 or 8), and each element must be initialized, it's not surprising that this line slows down the execution of the program with the $8,000,000 \times 8$ input.

Also recall that a *read-miss* occurs when a core tries to read a variable that's not in the cache, and it has to access main memory. A cache profiler shows that when the program is run with the $8 \times 8,000,000$ input, it has far more cache read-misses than either of the other inputs. These occur in Line 11, and a careful study of this program (see Exercise 16) shows that the main source of the differences is due to the reads of `x`. Once again, this isn't surprising, since for this input, `x` has 8,000,000 elements, versus only 8000 or 8 for the other inputs.

It should be noted that there may be other factors that are affecting the relative performance of the single-threaded program with the differing inputs. For example, we haven't taken into consideration whether virtual memory (see Subsection **??**) has affected the performance of the program with the different inputs. How frequently does the CPU need to access the page table in main memory?

Of more interest to us, though, is the tremendous difference in efficiency as the number of threads is increased. The two-thread efficiency of the program with the $8 \times 8,000,000$ input is nearly 20% less than the efficiency of the program with the $8,000,000 \times 8$ and the 8000×8000 inputs. The four-thread efficiency of the program with the $8 \times 8,000,000$ input is nearly 60% less than the program's efficiency with the $8,000,000 \times 8$ input and *more* than 60% less than the program's efficiency with the 8000×8000 input. These dramatic decreases in efficiency are even more remarkable when we note that with one thread the program is much slower with $8 \times 8,000,000$ input. Therefore, the numerator in the formula for the efficiency:

$$\text{Parallel Efficiency} = \frac{\text{Serial Run-Time}}{(\text{Number of Threads}) \times (\text{Parallel Run-Time})}$$

will be much larger. Why, then, is the multithreaded performance of the program so much worse with the $8 \times 8,000,000$ input?

In this case, once again, the answer has to do with cache. Let's take a look at the program when we run it with four threads. With the $8,000,000 \times 8$ input, `y` has 8,000,000 components, so each thread is assigned 2,000,000 components. With the 8000×8000 input, each thread is assigned 2000 components of `y`, and with the $8 \times 8,000,000$ input, each thread is assigned 2 components. On the system we used, a cache line is 64 bytes. Since the type of `y` is **double**, and a **double** is 8 bytes, a single cache line can store 8 **double**s.

Cache coherence is enforced at the "cache-line level." That is, each time any value in a cache line is written, if the line is also stored in another processor's cache, the entire *line* will be invalidated—not just the value that was written. The system we're using has two dual-core processors and each processor has its own cache. Suppose for the moment that threads 0 and 1 are assigned to one of the processors and threads 2 and 3 are assigned to the other. Also suppose that for the $8 \times 8,000,000$ problem all of y is stored in a single cache line. Then every write to some element of y will invalidate the line in the other processor's cache. For example, each time thread 0 updates y[0] in the statement

```
y[i] += A[i][j]*x[j];
```

If thread 2 or 3 is executing this code, it will have to reload y. Each thread will update each of its components $8,000,000$ times. We see that with this assignment of threads to processors and components of y to cache lines, all the threads will have to reload y *many* times. This is going to happen in spite of the fact that only one thread accesses any one component of y—for example, only thread 0 accesses y[0].

Each thread will update its assigned components of y a total of 16,000,000 times. It appears that many if not most of these updates are forcing the threads to access main memory. This is called **false sharing**. Suppose two threads with separate caches access different variables that belong to the same cache line. Further suppose at least one of the threads updates its variable. Then even though neither thread has written to a variable that the other thread is using, the cache controller invalidates the entire cache line and forces the threads to get the values of the variables from main memory. The threads aren't sharing anything (except a cache line), but the behavior of the threads with respect to memory access is the same as if they were sharing a variable, hence the name *false sharing*.

Why is false sharing not a problem with the other inputs? Let's look at what happens with the 8000×8000 input. Suppose thread 2 is assigned to one of the processors and thread 3 is assigned to another. (We don't actually know which threads are assigned to which processors, but it turns out—see Exercise 17—that it doesn't matter.) Thread 2 is responsible for computing

```
y[4000], y[4001], . . . , y[5999],
```

and thread 3 is responsible for computing

```
y[6000], y[6001], . . . , y[7999].
```

If a cache line contains 8 consecutive **double**s, the only possibility for false sharing is on the interface between their assigned elements. If, for example, a single cache line contains

```
y[5996], y[5997], y[5998], y[5999], y[6000], y[6001], y[6002], y[6003],
```

then it's conceivable that there might be false sharing of this cache line. However, thread 2 will access

```
y[5996], y[5997], y[5998], y[5999]
```

at the *end* of its **for** i loop, while thread 3 will access

```
y[6000], y[6001], y[6002], y[6003]
```

at the *beginning* of its **for** i loop. So it's very likely that when thread 2 accesses (say) y[5996], thread 3 will be long done with all four of

```
y[6000], y[6001], y[6002], y[6003].
```

Similarly, when thread 3 accesses, say, y[6003], it's very likely that thread 2 won't be anywhere near starting to access

```
y[5996], y[5997], y[5998], y[5999].
```

It's therefore unlikely that false sharing of the elements of y will be a significant problem with the 8000×8000 input. Similar reasoning suggests that false sharing of y is unlikely to be a problem with the $8,000,000 \times 8$ input. Also note that we don't need to worry about false sharing of A or x, since their values are never updated by the matrix-vector multiplication code.

This brings up the question of how we might avoid false sharing in our matrix-vector multiplication program. One possible solution is to "pad" the y vector with dummy elements in order to insure that any update by one thread won't affect another thread's cache line. Another alternative is to have each thread use its own private storage during the multiplication loop, and then update the shared storage when they're done. See Exercise 19.

1.11 Thread-Safety[9]

Let's look at another potential problem that occurs in shared-memory programming: *thread-safety.* A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

As an example, suppose we want to use multiple threads to "tokenize" a file. Let's suppose that the file consists of ordinary English text, and that the tokens are just contiguous sequences of characters separated from the rest of the text by white space—a space, a tab, or a newline. A simple approach to this problem is to divide the input file into lines of text and assign the lines to the threads in a round-robin fashion: the first line goes to thread 0, the second goes to thread 1, ..., the tth goes to thread t, the $t+1$st goes to thread 0, and so on.

We can serialize access to the lines of input using semaphores. Then, after a thread has read a single line of input, it can tokenize the line. One way to do this is to use the strtok function in string.h, which has the following prototype:

```
char* strtok(
        char*         string       /* in/out */,
        const char*   separators   /* in     */);
```

Its usage is a little unusual: the first time it's called the string argument should be the text to be tokenized, so in our example it should be the line of input. For subsequent calls, the first argument should be NULL. The idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens taken from the

[9] This material is also covered in Chapter ??. So if you've already read that chapter, you may want to skim this section.

cached copy. The characters that delimit tokens should be passed in `separators`. We should pass in the string `" \t\n"` as the `separators` argument.

```
 1  void *Tokenize(void* rank) {
 2     long my_rank = (long) rank;
 3     int count;
 4     int next = (my_rank + 1) % thread_count;
 5     char *fg_rv;
 6     char my_line[MAX];
 7     char *my_string;
 8
 9     sem_wait(&sems[my_rank]);
10     fg_rv = fgets(my_line, MAX, stdin);
11     sem_post(&sems[next]);
12     while (fg_rv != NULL) {
13        printf("Thread %ld > my line = %s", my_rank, my_line);
14
15        count = 0;
16        my_string = strtok(my_line, " \t\n");
17        while ( my_string != NULL ) {
18           count++;
19           printf("Thread %ld > string %d = %s\n", my_rank, count,
20                 my_string);
21           my_string = strtok(NULL, " \t\n");
22        }
23
24        sem_wait(&sems[my_rank]);
25        fg_rv = fgets(my_line, MAX, stdin);
26        sem_post(&sems[next]);
27     }
28
29     return NULL;
30  }  /* Tokenize */
```

Program 1.14: A first attempt at a multithreaded tokenizer

Given these assumptions, we can write the thread function shown in Program 1.14. The main thread has initialized an array of *t* semaphores—one for each thread. Thread 0's semaphore is initialized to 1. All the other semaphores are initialized to 0. So the code in Lines 9 to 11 will force the threads to sequentially access the lines of input. Thread 0 will immediately read the first line, but all the other threads will block in `sem_wait`. When thread 0 executes the `sem_post`, thread 1 can read a line of input. After each thread has read its first line of input (or end-of-file), any additional input is read in lines 24 to 26. The `fgets` function reads a single line of input and lines 15 to 22 identify the tokens in the line. When we run the program with a single thread, it correctly tokenizes the input stream. The first time we run it with two threads and the input

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

the output is also correct. However, the second time we run it with this input, we get the following output.

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

What happened? Recall that `strtok` caches the input line. It does this by declaring a variable to have **static** storage class. This causes the value stored in this variable to persist from one call to the next. Unfortunately for us, this cached string is shared, not private. Thus, thread 0's call to `strtok` with the third line of the input has apparently overwritten the contents of thread 1's call with the second line.

The `strtok` function is *not* thread-safe: if multiple threads call it simultaneously, the output it produces may not be correct. Regrettably, it's not uncommon for C library functions to fail to be thread-safe. For example, neither the random number generator `random` in `stdlib.h` nor the time conversion function `localtime` in time.h is thread-safe. In some cases, the C standard specifies an alternate, thread-safe, version of a function. In fact, there is a thread-safe version of strtok:

```
char* strtok_r(
      char*        string      /* in/out */,
      const char*  separators  /* in     */,
      char**       saveptr_p   /* in/out */);
```

The "_r" indicates the function is *reentrant*, which is sometimes used as a synonym for thread-safe[10]. The first two arguments have the same purpose as the arguments to strtok. The saveptr_p argument is used by strtok_r for keeping track of where the function is in the input string; it serves the purpose of the cached pointer in strtok. We can correct our original Tokenize function by replacing the calls to strtok with calls to strtok_r. We simply need to declare a **char*** variable to pass in for the third argument, and replace the calls in line 16 and line 21 with the calls

```
my_string = strtok_r(my_line, " \t\n", &saveptr);
. . .
my_string = strtok_r(NULL, " \t\n", &saveptr);
```

respectively.

1.11.1 Incorrect programs can produce correct output

Notice that our original version of the tokenizer program shows an especially insidious form of program error: the first time we ran it with two threads, the program produced correct output. It wasn't until a later run that we saw an error. This, unfortunately, is not a rare occurrence in parallel programs. It's especially common in shared-memory programs. Since, for the most part, the threads are running independently of each other, as we noted earlier, the exact sequence of statements executed is nondeterministic. For example, we can't say when thread 1 will first call strtok. If its first call takes place after thread 0 has tokenized its first line, then the tokens identified for the first line should be correct. However, if thread 1 calls strtok before thread 0 has finished tokenizing its first line, it's entirely possible that thread 0 may not identify all the tokens in the first line. Therefore, it's especially important in developing shared-memory programs to resist the temptation to assume that since a program produces correct output, it must be correct. We always need to be wary of race conditions.

[10] However, the distinction is a bit more nuanced; being reentrant means a function can be interrupted and called again (reentered) in different parts of a program's control flow and still execute correctly. This can happen due to nested calls to the function or a trap/interrupt sent from the operating system. Since strtok uses a single static pointer to track its state while parsing, multiple calls to the function from different parts of a program's control flow will corrupt the string — therefore, it is **not** reentrant. It's worth noting that although reentrant functions such as strtok_r can also be thread safe, there is no guarantee reentrant function will *always* be thread safe (and vice versa). It's best to consult the documentation if in doubt.

1.12 Summary

Like MPI, Pthreads is a library of functions that programmers can use to implement parallel programs. Unlike MPI, Pthreads is used to implement shared-memory parallelism.

A **thread** in shared-memory programming is analogous to a process in distributed-memory programming. However, a thread is often lighter-weight than a full-fledged process.

We saw that in Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function. In order to use Pthreads, we should include the `pthread.h` header file, and, when we compile our program, it may be necessary to link our program with the Pthread library by adding `–lpthread` to the command line. We saw that we can use the functions `pthread_create` and `pthread_join`, respectively, to start and stop a thread function.

When multiple threads are executing, the order in which the statements are executed by the different threads is usually nondeterministic. When nondeterminism results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a **race condition**. One of our most important tasks in writing shared-memory programs is identifying and correcting race conditions. A **critical section** is a block of code that updates a shared resource that can only be updated by one thread at a time, so the execution of code in a critical section should, effectively, be executed as serial code. Thus, we should try to design our programs so that they use them as infrequently as possible, and the critical sections we do use should be as short as possible.

We looked at three basic approaches to avoiding conflicting access to critical sections: busy-waiting, mutexes, and semaphores. **Busy-waiting** can be done with a flag variable and a **while** loop with an empty body. It can be very wasteful of CPU cycles. It can also be unreliable if compiler optimization is turned on, so mutexes and semaphores are generally preferable.

A **mutex** can be thought of as a lock on a critical section, since mutexes arrange for *mutually exclusive* access to a critical section. In Pthreads, a thread attempts to obtain a mutex with a call to `pthread_mutex_lock`, and it relinquishes the mutex with a call to `pthread_mutex_unlock`. When a thread attempts to obtain a mutex that is already in use, it *blocks* in the call to `pthread_mutex_lock`. This means that it remains idle in the call to `pthread_mutex_lock` until the system gives it the lock.

A **semaphore** is an **unsigned int** together with two operations: `sem_wait` and `sem_post`. If the semaphore is positive, a call to `sem_wait` simply decrements the semaphore, but if the semaphore is zero, the calling thread blocks until the semaphore is positive, at which point the semaphore is decremented and the thread returns from the call. The `sem_post` operation increments the semaphore; a semaphore can be used as a mutex with `sem_wait` corresponding to `pthread_mutex_lock` and `sem_post` corresponding to `pthread_mutex_unlock`. However, semaphores are more powerful than mutexes since they can be initialized to any nonnegative value. Furthermore, since there is no “ownership” of a semaphore, any thread can “unlock” a locked semaphore. We saw that semaphores can be easily used to implement **producer-consumer synchronization**. In producer-consumer synchronization, a “consumer” thread waits for some condition or data created by a “producer” thread before proceeding. Semaphores are not part of Pthreads. In order to use them, we need to include the `semaphore.h` header file.

A **barrier** is a point in a program at which the threads block until all of the threads have reached it. We saw several different means for constructing barriers. One of them used a **condition variable**. A condition variable is a special Pthreads object that can be used to suspend execution of a thread until a condition has occurred. When the condition has occurred, another thread can awaken the suspended thread with a condition signal or a condition broadcast.

The last Pthreads construct we looked at was a **read-write lock**. A read-write lock is used when it’s safe for multiple threads to simultaneously *read* a data structure, but if a thread needs to modify or *write* to the data structure, then only that thread can access the data structure during the modification.

We recalled that modern microprocessor architectures use caches to reduce memory access times, so typical architectures have special hardware to insure that the caches on the different chips are **coherent**. Since the unit of cache coherence, a **cache line** or **cache block**, is usually larger than a single word of memory, this can have the unfortunate side effect that two threads may be accessing different memory locations, but when the two locations belong to the same cache line, the cache-coherence hardware acts as if the threads were accessing the same memory location. Thus, if one of the threads updates its memory location, and then the other thread tries to read its memory location, it will have to retrieve the value from main memory. That is, the hardware is forcing the thread to act as if it were actually sharing the memory location. Hence, this is called **false sharing**, and it can seriously degrade the performance of a shared-memory program.

Some C functions cache data between calls by declaring variables to be **static** . This can cause errors when multiple threads call the function; since static storage is shared among the threads, one thread can overwrite another thread’s data. Such a function is not **thread-safe**, and, unfortunately, there are several such functions in the C library. Sometimes, however, there is a thread-safe variant.

When we looked at the program that used the function that wasn’t thread-safe, we saw a particularly insidious problem: when we ran the program with multiple threads and a fixed set of input, it sometimes produced correct output, even though the program was erroneous. This means that even if a program produces correct output during testing, there’s no guarantee that it is in fact correct–it’s up to us to identify possible race conditions.

1.13 Exercises

1. When we discussed matrix-vector multiplication we assumed that both m and n, the number of rows and the number of columns, respectively, were evenly divisible by t, the number of threads. How do the formulas for the assignments change if this is *not* the case?

2. If we decide to physically divide a data structure among the threads, that is, if we decide to make various members local to individual threads, we need to consider at least three issues:

(a) How are the members of the data structure used by the individual threads?

(b) Where and how is the data structure initialized?

(c) Where and how is the data structure used after its members are computed?

We briefly looked at the first issue in the matrix-vector multiplication function. We saw that the entire vector `x` was used by all of the threads, so it seemed pretty clear that it should be shared. However, for both the matrix `A` and the product vector `y`, just looking at (a) seemed to suggest that `A` and `y` should have their components distributed among the threads. Let's take a closer look at this.

What would we have to do in order to divide `A` and `y` among the threads? Dividing `y` wouldn't be difficult–each thread could allocate a block of memory that could be used for storing its assigned components. Presumably, we could do the same for `A`–each thread could allocate a block of memory for storing its assigned rows. Modify the matrix-vector multiplication program so that it distributes both of these data structures. Can you "schedule" the input and output so that the threads can read in `A` and print out `y`? How does distributing `A` and `y` affect the run-time of the matrix-vector multiplication? (Don't include input or output in your run-time.)

3. Recall that the compiler is unaware that an ordinary C program is multithreaded, and as a consequence, it may make optimizations that can interfere with busy-waiting. (Note that compiler optimizations should *not* affect mutexes, condition variables, or semaphores.) An alternative to completely turning off compiler optimizations is to identify some shared variables with the C keyword **volatile**. This tells the compiler that these variables may be updated by multiple threads and, as a consequence, it shouldn't apply optimizations to statements involving them. As an example, recall our busy-wait solution to the race condition when multiple threads attempt to add a private variable into a shared variable:

```
/* x and flag are shared, y is private             */
/* x and flag are initialized to 0 by main thread */

y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

It's impossible to tell by looking at this code that the order of the **while** statement and the `x = x + y` statement is important; if this code were single-threaded, the order of these two statements wouldn't affect the outcome of the code. But if the compiler determined that it could improve register usage by interchanging the order of these two statements, the resulting code would be erroneous.

If, instead of defining

```
int flag;
int x;
```

we define

```
int volatile flag;
int volatile x;
```

then the compiler will know that both `x` and `flag` can be updated by other threads, so it shouldn't try reordering the statements.

With the `gcc` compiler, the default behavior is no optimization. You can make certain of this by adding the option –`O0` to the command line. Try running the π calculation program that uses busy-waiting (`pth_pi_busy.c`) without optimization. How does the result of the multithreaded calculation compare to the single-threaded calculation? Now try running it with optimization; if you're using `gcc`, replace the –`O0` option with –`O2`. If you found an error, how many threads did you use?

Which variables should be made volatile in the π calculation? Change these variables so that they're volatile and rerun the program with and without optimization. How do the results compare to the single-threaded program?

4. The performance of the π calculation program that uses mutexes remains roughly constant once we increase the number of threads beyond the number of available CPUs. What does this suggest about how the threads are scheduled on the available processors?

5. Modify the mutex version of the π calculation program so that the critical section is in the **for** loop. How does the performance of this version compare to the performance of the original busy-wait version? How might we explain this?

6. Modify the mutex version of the π calculation program so that it uses a semaphore instead of a mutex. How does the performance of this version compare with the mutex version?

7. Modify the Pthreads hello, world program to launch an unlimited number of threads—you can effectively ignore the `thread_count` and instead call `pthread_create` in an infinite loop (e.g., `for (thread = 0; ; thread++)`).

   Note that in most cases the program will *not* create an unlimited number of threads; you'll observe that the "Hello from thread" messages stop after some time, depending on the configuration of your system. How many threads were created before the messages stopped?

   Observe that while nothing is being printed, the program is still running. To determine why no new threads are being created, check the return value of the call to `pthread_create` (hint: use the `perror` function to get a human-readable description of the problem, or look up the error codes). What is the cause of this bug?

Finally, modify the **for** loop containing pthread_create to detach each new thread using pthread_detach. How many threads are created now?

8. Although producer-consumer synchronization is easy to implement with semaphores, it's also possible to implement it with mutexes. The basic idea is to have the producer and the consumer share a mutex. A flag variable that's initialized to false by the main thread indicates whether there's anything to consume. With two threads we'd execute something like this:

```
while (1) {
   pthread_mutex_lock(&mutex);
   if (my_rank == consumer) {
      if (message_available) {
         print message;
         pthread_mutex_unlock(&mutex);
         break;
      }
   } else { /* my_rank == producer */
      create message;
      message_available = 1;
      pthread_mutex_unlock(&mutex);
      break;
   }
   pthread_mutex_unlock(&mutex);
}
```

So if the consumer gets into the loop first, it will see there's no message available and return to the call to pthread_mutex_lock. It will continue this process until the producer creates the message. Write a Pthreads program that implements this version of producer-consumer synchronization with two threads. Can you generalize this so that it works with 2k threads–odd-ranked threads are consumers and even-ranked threads are producers? Can you generalize this so that each thread is both a producer and a consumer? For example, suppose that thread q "sends" a message to thread $(q+1)$ mod t and "receives" a message from thread $(q-1+t)$ mod t? Does this use busy-waiting?

9. If a program uses more than one mutex, and the mutexes can be acquired in different orders, the program can **deadlock**. That is, threads may block forever waiting to acquire one of the mutexes. As an example, suppose that a program has two shared data structures–for example, two arrays or two linked lists–each of which has an associated mutex. Further suppose that each data structure can be accessed (read or modified) after acquiring the data structure's associated mutex.

(a) Suppose the program is run with two threads. Further suppose that the following sequence of events occurs:

Time	Thread 0	Thread 1
0	pthread_mutex_lock(&mut0)	pthread_mutex_lock(&mut1)
1	pthread_mutex_lock(&mut1)	pthread_mutex_lock(&mut0)

What happens?

(b) Would this be a problem if the program used busy-waiting (with two flag variables) instead of mutexes?

(c) Would this be a problem if the program used semaphores instead of mutexes?

10. Some implementations of Pthreads define barriers. The function

```
int pthread_barrier_init(
      pthread_barrier_t*               barrier_p  /* out */,
      const pthread_barrierattr_t*     attr_p     /* in  */,
      unsigned                         count      /* in  */);
```

initializes a barrier object, `barrier_p`. As usual, we'll ignore the second argument and just pass in `NULL`. The last argument indicates the number of threads that must reach the barrier before they can continue. The barrier itself is a call to the function

```
int pthread_barrier_wait(pthread_barrier_t*  barrier_p  /* in/out */);
```

As with most other Pthreads objects, there is a destroy function

```
int pthread_barrier_destroy(pthread_barrier_t*  barrier_p  /* in/out */);
```

Modify one of the barrier programs from the book's website so that it uses a Pthreads barrier. Find a system with a Pthreads implementation that includes barrier and run your program with various numbers of threads. How does its performance compare to the other implementations?

11. Modify one of the programs you wrote in the Programming Assignments that follow so that it uses the scheme outlined in Section 1.8 to time itself. In order to get the time that has elapsed since some point in the past, you can use the macro `GET_TIME` defined in the header file `timer.h` on the book's website. Note that this will give *wall clock* time, not CPU time. Also note that since it's a macro, it can operate directly on its argument. For example, to implement

```
Store current time in my_start;
```

you would use

```
GET_TIME(my_start);
```

not

```
GET_TIME(&my_start);
```

How will you implement the barrier? How will you implement the following pseudocode?

```
elapsed = Maximum of my_elapsed values;
```

12. Give an example of a linked list and a sequence of memory accesses to the linked list in which the following pairs of operations can potentially result in problems:

    (a) Two deletes executed simultaneously

    (b) An insert and a delete executed simultaneously

    (c) A member and a delete executed simultaneously

    (d) Two inserts executed simultaneously

    (e) An insert and a member executed simultaneously.

13. The linked list operations `Insert` and `Delete` consist of two distinct "phases." In the first phase, both operations search the list for either the position of the new node or the position of the node to be deleted. If the outcome of the first phase so indicates, in the second phase a new node is inserted or an existing node is deleted. In fact, it's quite common for linked list programs to split each of these operations into two function calls. For both operations, the first phase involves only read-access to the list; only the second phase modifies the list. Would it be safe to lock the list using a read-lock for the first phase? And then to lock the list using a write-lock for the second phase? Explain your answer.

14. Download the various threaded linked list programs from the website. In our examples, we ran a fixed percentage of searches and split the remaining percentage among inserts and deletes.

    (a) Rerun the experiments with all searches and all inserts.

    (b) Rerun the experiments with all searches and all deletes.

    Is there a difference in the overall run times? Is insert or delete more expensive?

15. Recall that in C a function that takes a two-dimensional array argument must specify the number of columns in the argument list. Thus it is quite common for C programmers to only use one-dimensional arrays, and to write explicit code for converting pairs of subscripts into a single dimension. Modify the Pthreads matrix-vector multiplication so that it uses a one-dimensional array for the matrix and calls a matrix-vector multiplication function. How does this change affect the run time?

16. Download the source file `pth_mat_vect_rand_split.c` from the book's website. Find a program that does cache profiling (for example, Valgrind [**?**]) and compile the program according to the instructions in the cache profiler documentation. (with Valgrind you will want a symbol table and full optimization (e.g., `gcc −g −O2 . . .`). Now run the program according to the instructions in the cache profiler documentation, using input $k \times (k \cdot 10^6)$, $(k \cdot 10^3) \times (k \cdot 10^3)$, and $(k \cdot 10^6) \times k$. Choose k so large that the number of level 2 cache misses is of the order 10^6 for at least one of the input sets of data.

    (a) How many level 1 cache write-misses occur with each of the three inputs?

    (b) How many level 2 cache write-misses occur with each of the three inputs?

    (c) Where do most of the write-misses occur? For which input data does the program have the most write-misses? Can you explain why?

    (d) How many level 1 cache read-misses occur with each of the three inputs?

    (e) How many level 2 cache read-misses occur with each of the three inputs?

    (f) Where do most of the read-misses occur? For which input data does the program have the most read-misses? Can you explain why?

    (g) Run the program with each of the three inputs, but without using the cache profiler. With which input is the program the fastest? With which input is the program the slowest? Can your observations about cache misses help explain the differences? How?

17. Recall the matrix-vector multiplication example with the 8000×8000 input. Suppose that the program is run with four threads, and thread 0 and thread 2 are assigned to different processors. If a cache line contains 64 bytes or eight **double**s, is it possible for false sharing between threads 0 and 2 to occur for any part of the vector `y`? Why? What about if thread 0 and thread 3 are assigned to different processors–is it possible for false sharing to occur between them for any part of `y`?

18. Recall the matrix-vector multiplication example with an $8 \times 8,000,000$ matrix. Suppose that **double**s use 8 bytes of memory and that a cache line is 64 bytes. Also suppose that our system consists of two dual-core processors.

    (a) What is the minimum number of cache lines that are needed to store the vector `y`?

    (b) What is the maximum number of cache lines that are needed to store the vector `y`?

    (c) If the boundaries of cache lines always coincide with the boundaries of 8-byte **double**s, in how many different ways can the components of `y` be assigned to cache lines?

    (d) If we only consider which pairs of threads share a processor, in how many different ways can four threads be assigned to the processors in our computer? Here we're assuming that cores on the same processor share cache.

(e) Is there an assignment of components to cache lines and threads to processors that will result in no false sharing in our example? In other words, is it possible that the threads assigned to one processor will have their components of `y` in one cache line, and the threads assigned to the other processor will have their components in a different cache line?

(f) How many assignments of components to cache lines and threads to processors are there?

(g) Of these assignments, how many will result in no false sharing?

19. (a) Modify the matrix-vector multiplication program so that it pads the vector `y` when there's a possibility of false sharing. The padding should be done so that if the threads execute in lock-step, there's no possibility that a single cache line containing an element of `y` will be shared by two or more threads. Suppose, for example, that a cache line stores eight `doubles` and we run the program with four threads. If we allocate storage for at least 48 **double**s in `y`, then, on each pass through the **for** `i` loop, there's no possibility that two threads will simultaneously access the same cache line.

    (b) Modify the matrix-vector multiplication so that each thread uses private storage for its part of `y` during the **for** `i` loop. When a thread is done computing its part of `y`, it should copy its private storage into the shared variable.

    (c) How does the performance of these two alternatives compare to the original program? How do they compare to each other?

20. Although `strtok_r` is thread-safe, it has the rather unfortunate property that it gratuitously modifies the input string. Write a tokenizer that is thread-safe and doesn't modify the input string.

1.14 Programming Assignments

1. Write a Pthreads program that implements the histogram program in Chapter **??**.

2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are two feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and its area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

This is called a “Monte Carlo” method, since it uses randomness (the dart tosses).

Write a Pthreads program that uses a Monte Carlo method to estimate π. The main thread should read in the total number of tosses and print the estimate. You may want to use **long long int**s for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

3. Write a Pthreads program that implements the trapezoidal rule. Use a shared variable for the sum of all the threads' computations. Use busy-waiting, mutexes, and semaphores to enforce mutual exclusion in the critical section. What advantages and disadvantages do you see with each approach?

4. Write a Pthreads program that finds the average time required by your system to create and terminate a thread. Does the number of threads affect the average time? If so, how?

5. Write a Pthreads program that implements a “task queue.” The main thread begins by starting a user-specified number of threads that immediately go to sleep in a condition wait. The main thread generates blocks of tasks to be carried out by the other threads; each time it generates a new block of tasks, it awakens a thread with a condition signal. When a thread finishes executing its block of tasks, it should return to a condition wait. When the main thread completes generating tasks, it sets a global variable indicating that there will be no more tasks, and awakens all the threads with a condition broadcast. For the sake of explicitness, make your tasks linked list operations.

6. Write a Pthreads program that uses two condition variables and a mutex to implement a read-write lock. Download the online linked list program that uses Pthreads read-write locks, and modify it to use your read-write locks. Now compare the performance of the program when readers are given preference with the program when writers are given preference. Can you make any generalizations?