Chapter 10 Shared Memory Parallel Computing With Pthreads

Preface

There are two different approaches to multi-threaded programming. One is based on explicit user-defined threads and the other is based in part on user-guided threading support provided by the compiler. The latter is exemplified by *OpenMP*; the former by various threading libraries. This chapter is an introduction to the use of threads in general, specifically covering the POSIX threads library, better known as *Pthreads*. This is a cross-platform library, supported on Solaris, Mac OS, FreeBSD, OpenBSD, and Linux. There are several other threading libraries, including the native threading introduced in C++ 11 through the thread support library, whose API is obtained by including the `<thread>` header file. C++ includes built-in support for threads, mutual exclusion, condition variables, and futures. There is also *Qt Threads*, which are part of the Qt cross-platform C++ toolkit. Qt threads look very much like those from Java. On Windows platforms, there is *winThreads*, which has a C++ binding.

Concepts Covered

Shared memory parallelism, processes, threads,
multi-threading paradigms,
Pthreads, NPTL,
thread properties,
thread cancellation, detached threads,
mutexes, condition variables,
barrier synchronization, reduction algorithm
producer-consumer problem,
reader/writer locks,
thread scheduling, deadlock, starvation

10.1 Introduction

By "shared memory" we mean that the physical processors have access to a shared physical memory. This in turn implies that independent processes running on these processors can access this shared physical memory. The fact that they can access the same memory does not mean that they can access the same logical memory because their logical address spaces are by default made to be disjoint for safety and security. Modern operating systems provide the means by which processes can access the same set of physical memory locations and thereby share data, but that is not the topic of these notes. The intention of these notes is to discuss ***multi-threading***.

In the shared memory model of parallel computing, processes running on separate processors have access to a shared physical memory and therefore they have access to shared data. This shared access is both a blessing and a curse to the programmer. It is a blessing because it makes it possible for processes to exchange information and synchronize actions through shared variables. It is a curse because it makes it possible to corrupt the state of the collection of processes in ways that depend purely on the relative rates of progress of the processes. We call the potential for such corruption a ***race condition***.

A running program, which we call a ***process***, is associated with a set of resources including its memory segments (text, stack, initialized data, uninitialized data, and so on), environment variables, command line arguments, and various properties and data that are contained in kernel resources such as the process and user structures (data structures used by the kernel to manage the processes.) A partial list of the kinds of information contained in these latter structures includes things such as the process's

- IDs, including its process ID, process group ID, user ID, and group ID
- Hardware state
- Memory mappings, such as where all process segments are located in logical memory
- Flags such as set-uid, set-gid
- File descriptors
- Signal masks and dispositions
- Resource limits
- Inter-process communication tools such as message queues, pipes, semaphores, or shared memory.

In short, a process is a fairly "heavy" object in the sense that when a process is created, all of these resources must be created for it, which takes time. The `fork()` system call duplicates almost all of the calling process's resources for the new, child process. A few resources are not duplicated. But after the `fork()` call, the processes are essentially independent execution units. The paradigm for forking a process is shown in Listing 10.1 below.

Listing 10.1: Simple `fork()` program

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

int main(int argc, char* argv[])
{
    pid_t  result;

    result = fork();

    if ( -1 == result ) {
        printf("Error trying to create new process.\n");
        exit(1);
    }
    else if ( 0 == result ) {
        /* child executes this branch */
        printf("Code execute by child process.\n");
    }
    else {
        /* parent executes this branch */
        printf("Code executed by parent process.\n");
    }

    return 0;
}
```

Processes by default are limited in what they can share with each other because they do not share their logical memory spaces. Thus, for example, they do not in general share variables and other objects that they create in memory. To make sharing possible, most operating systems provide an API for sharing memory. For example, in Linux 2.4 and later, and `glibc` 2.2 and later, *POSIX shared memory* is available so that unrelated processes can communicate through shared memory objects. Solaris also supported shared memory, both natively and with support for the later POSIX standard. In addition, processes can share files and messages, and they can send each other signals to synchronize.

The major drawbacks to using processes as a means of multi-tasking is their consumption of system resources and the inability to share variables easily. The `fork()` call itself is a time-consuming call because it has to replicate the memory image of the process. The replication of data is also wasteful because in many cases the child process replaces its code anyway using some form of the `exec()` system call. This was the motivation for the invention of threads.

10.2 Thread Concepts

A ***thread*** is a flow of control (i.e., a sequence of instructions) that can be independently scheduled by the kernel. A process can have multiple threads. A typical process can be thought of as having a single thread of control: each process is executing one instruction at a time. When a program has multiple threads of control, more than one instruction at a time can be executed within a single process, with each thread handling a separate task. Some of the advantages of using threads are:

- ***Asynchronous Processing***. Code to handle asynchronous events can be executed by separate threads. Each thread can then handle its event using a synchronous programming model.
- ***Performance***:
  - Multiple threads can take advantage of multiple cores because they can be independently scheduled by the kernel.
  - Whereas multiple processes have to use mechanisms provided by the kernel to share memory and file descriptors, threads automatically have access to the same memory address space, which is faster and simpler.
  - Even on a single processor machine, performance can be improved by putting calls to system functions with expected long waits in separate threads. This way, just the calling thread blocks, not the whole process, allowing the other threads to make progress[1].
- ***Response Time***. Programs that have both interactive and background tasks, such as animated games, have to handle user input while computing scenes to display. Separate threads can be used to improve response time while performing background tasks.

[1]This depends to some extent on how threads are implemented on the particular system

- ***Program Structure***. Programs can be structured more understandably if separate logical tasks are performed by separate threads.

Threads share certain resources with the parent process and each other, and maintain private copies of other resources. The most important resources *shared* by threads are

- the program's text segment, i.e., its executable code, and
- its global and heap memory.

This implies that threads can communicate through the program's global variables, but it also implies that they have to synchronize their access to these shared resources. To make threads independently schedulable, at the very least they they must have their own stack and register values. They also need some unique number, a thread ID, so that they can be identified.

POSIX requires that each thread have its own distinct

- thread ID
- stack and alternate stack
- stack pointer and registers
- signal mask
- errno value
- scheduling properties
- thread specific data.

On the other hand, in addition to the text and data segments of the process, UNIX threads share

- file descriptors
- environment variables
- process ID
- parent process ID
- process group ID and session ID
- controlling terminal
- user and group IDs (real and effective)
- open file descriptors
- record locks
- signal dispositions
- file mode creation mask (the umask)

- current directory and root directory
- interval timers and POSIX timers
- nice value
- resource limits
- measurements of the consumption of CPU time and resources

To summarize, a thread

- is a single flow of control within a process and uses the process resources;
- duplicates only the resources it needs to be independently schedulable;
- can share the process resources with other threads within the process; and
- terminates if the parent process is terminated;

10.3 Examples of Thread Creation

To make this concrete, below are three examples of thread creation using different libraries. The first uses the thread class introduced in the C++ 11 standard and the second uses the POSIX thread API known as ***Pthreads***.

Listing 10.2: C++ Native `<thread>` Class Example

```
/* This must be built on Linux using
      g++ -std=c++11 -o simplethreaddemo simplethreaddemo.cpp -pthread
   It is implemented using the underlying POSIX library.
*/
#include <thread>
#include <iostream>

void greeting()
{
   std::cout << "Hello world!\n";
}

int main(int argc, char* argv[])
{
  std::thread child(greeting); /* thread executes function named greeting */

  std::cout << "This is the main (parent) thread\n";
  child.join();
  return 0;
}
```

In Listing 10.2 the declaration of a thread in the `main` program causes it to be created and launched. It is given a function name, which is the code that it executes. The function must return `void` and have no parameters. Threads in C++ have limited support and not all implementations of C++ provide the full support natively. For this reason, we will focus on POSIX threads. The following listing shows the analogous POSIX thread example.

Listing 10.3: PThreads Simple Thread Creation Example

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

void * hello_world( void * unused)
{
    printf("The child says, \"Hello world!\"\n");
    pthread_exit(NULL) ;
}

int main( int argc, char *argv[])
{
    pthread_t  child_thread;
    char       *planet  = "Pluto";

    /* Create the thread and launch it. */
    if ( 0 != pthread_create(&child_thread, NULL, hello_world, NULL ) ){
        printf("pthread_create failed.\n");
        exit(1);
    }

    printf("This is the parent thread.\n");
    /* Wait for the child thread to terminate. */
    pthread_join(child_thread, NULL);

    return 0;
}
```

Notice that the declaration of a thread in Pthreads does not launch it. There is an explicit function, `pthread_create()` to launch the thread. Also, the function passed to the thread has a single `void*` parameter. We will go through the details later in these notes.

10.4 Race Conditions

A race condition occurs when two or more threads access some shared resource and the outcome of their sharing it is that the correctness of the computation depends on the order in which they do so. Clearly, if both just read the resource without modifying it, there is no problem. But if one or more can modify it, then a race condition might exist.

Example

Multiple threads execute the thread routine below. Each performs a computation and adds its partial sum to the address pointed to by parameter `sum`, which is passed to it. The increment takes place through an integer cast of `sum` within the thread routine.

```
void * update_count( void  *sum)
{
    int i;
    int  *temp = (int*) sum;
    /* temp is just a cast of sum because sum is of
       type void*  and we cannot do arithmetic with it. */
    int  partial_sum = calc(); /* do some work and return a result */

    for ( i = 0; i < 10000; i++ )
        *temp = *temp + partial_sum; /* Race Condition */
}
```

The problem is that the line marked as a race condition performs addition, which is not a single machine instruction. It is typically translated to something like this:

```
mov register1, @temp
add register1, partial_sum
mov @temp, register1
```

If two different threads execute this code and their computations are interleaved in time, the result can be incorrect. Suppose Thread1 has 5 as the value for `partial_sum` and Thread2 has the value 8, and that `*temp` is 0 initially. They each use a separate register. After both execute, `temp` should be 13. Consider this interleaving in time

```
t0: thread1  executes mov register1, @temp        {register1 == 0 }
       t1: thread1  executes add register1, partial_sum  {register1 == 5 }
       t2: thread2  executes mov register2, @temp        {register2 == 0 }
       t3: thread2  executes add register2, partial_sum  {register2 == 8 }
       t4: thread1  executes mov @temp, register1        {temp   == 5 }
       t5: thread2  executes mov @temp, register2        {temp   == 8 }
```

The final value is 8, not 13. Had they executed in a different order, the result could be 5 or 13. Programs using shared variables are prone to this problem.

10.5 Program Design Using Threads

Threads are suitable for certain types of parallel programming. In general, in order for a program to take advantage of multi-threading, it must be possible to organize it into discrete, independent tasks that can execute concurrently. The first consideration when contemplating using multiple threads is how to decompose the program into such discrete, concurrent tasks. There are other considerations though. Among these are

- How can the load be balanced among the threads so that they no one thread becomes a bottleneck?
- How will threads communicate and synchronize to avoid race conditions?
- What type of data dependencies exist in the problem and how will these affect thread design?
- What data will be shared and what data will be private to the threads?
- How will I/O be handled? Will each thread perform its own I/O for example?

Each of these considerations is important, and to some extent each arises in most programming problems. Determining data dependencies, deciding which data should be shared and which should be private, and determining how to synchronize access to shared data are very critical aspects to the correctness of a solution. Load balancing and the handling of I/O usually affect performance but not correctness.

Knowing how to use a thread library is just the technical part of using threads. The much harder part is knowing how to design the program. There is no magic bullet in so far as that goes and these notes do not pretend to assist you with that task. Their purpose is just to provide the technical background, examples, and guidance.

However, before continuing, we present a few common paradigms for organizing multi-threaded programs.

10.5.1 Master/Worker Paradigm

In this approach, also called the *boss/worker paradigm*, there is a single ***master*** thread and multiple ***worker*** threads. The master thread does the following:

- It manages various tasks that must be performed and hands out tasks to the workers to perform work. It might be creating the tasks itself or creating tasks based on user input.
- It collects the results of worker computations.
- It performs all I/O and sends and receives data from the worker threads so that they do not have to perform I/O.
- If the program is interactive, the master is the thread that interacts with the user.

This paradigm models the way that many servers behave - as the server receives incoming requests, it hands out tasks to worker threads to service these requests. The functions of the master typically include handing out work and collecting results from workers, performing all I/O, and interacting with the user and or other processes.

Within this model two different sub-models emerge. In one sub-model, worker tasks are created dynamically as needed. The following pseudocode illustrates this idea.

```
int main( int argc, char *argv[])
{
        /* The master thread */
    // declare child threads
        {
        while (more work to do) {
        get a job from a job queue;
        switch (job) {
            case X : pthread_create(& child_thread1,
                                    NULL, taskX, ( void*) job) )
            case Y : pthread_create(& child_thread2,
                                    NULL, taskY, ( void*) job) )
             /* ... more cases here */
        }
        }
    /* rest of main thread here */
}

void * taskX() /* Worker to process job of type X */
{
           process job type X
}

void * taskY() /* Worker to process job of type Y */
{
            process job type X
}
```

The problem with this approach is that threads are being created and destroyed frequently, which adds to the total overhead of the program. The alternative is to maintain a ***thread pool***, which is a fixed set of threads created at the start of the program. Rather than creating new threads as jobs arise, the jobs are assigned to available threads. This implies that the master and workers must use a shared data structure, such as a queue, to manage the assignment of jobs to threads.

In general, the master/worker paradigm, whether it uses dynamically created threads or a thread pool, suffers from the possibility that the master is a bottleneck in the computation. A paradigm that avoids this removes the master and lets the workers manage themselves collectively.

A paradigm that is similar to the Master/Worker paradigm is one derived from Google's ***MapReduce*** technology, which Google used in its search engines for several years. In the *MapReduce* paradigm, the master coordinates two types of tasks: map tasks, which transform data from one form to another, and reduce tasks, which essentially perform reductions on the data that they receive. The number of map tasks and reduce tasks can vary over time.

10.5.2 Peer or WorkCrew Paradigm

In the ***peer*** model, tasks are assigned to a finite set of worker threads. Each worker can enqueue subtasks for concurrent evaluation by other workers as they become idle. The peer model is similar to the master/worker model except that after the master creates all the other peer threads when the program starts, the master becomes the another thread in the thread pool, and is thus a peer to the other threads. Whereas the master/worker model uses a stream of input requests to the master, the peer model makes each thread responsible for its own input. A peer knows what its input is in advance, has its own private way of obtaining its input, and/or shares a single point of input with other peers.

The peer model is suitable for applications that have a fixed or well-defined set of inputs, such as matrix operations, parallel database search engines, applications that process grids of data in general, and prime number generators. One problem with this model is that communication costs can be high, and peer performance can degrade if they must frequently synchronize to access shared resources.

An application like the 2D boundary value problem (heat dissipation for example) or generating a MandelBrot set, is suitable for this model, but because the results of each thread's calculations might require the adjustment of the bounds of the next thread's calculations, all threads might have to synchronize afterward to exchange and compare each other's results.

10.5.3 Pipeline

Similar to how pipelining works in a processor, each thread is part of a long chain in a processing factory. Each thread works on data processed by the previous thread and hands it off to the next thread. Conceptually, each thread executes a stage of the pipeline, which is of the form

```
initialize_stage();
while (there_is_data_to_be_processed) {
    get_data_from_previous_stage(data);
    process(data);
    pass_data_to_next_stage(data);
}
```

When designing a program using a pipeline structure, the difficulty is distributing work equally. Extra steps must be taken to ensure non-blocking behavior in this thread model or the program could experience pipeline "stalls." The paradigm is illustrated in the sample code in the listing below.

```
int main( int argc, char *argv[])
{
        /* The master thread */
    // declare child threads
        pthread_create( ... stage1 );
        pthread_create( ... stage2 );
        //  more stages here

        pthread_create( ... stageN );

        wait for all pipeline threads to finish
        clean up
}

void * stage1() /* thread to process pipeline stage 1 */
```

```
{
    get input from previous thread in pipeline
    process stage 1 of input
    pass result to next thread in pipeline
}

void * stage2() /* thread to process pipeline stage 2  */
{
        get input from previous thread in pipeline
    process stage 2 of input
    pass result to next thread in pipeline
}
// and so on
```

10.6 Overview of the Pthreads Library

The Pthreads library is the most widely used threading library and it underpins the design of many other libraries such as Qt threads, winThreads and C++ threads. For this reason, it is the library that these notes explore.

In 1995 the Open Group defined a standard interface for UNIX threads (IEEE POSIX 1003.1c) which they named *Pthreads* (P for POSIX). This standard was supported on multiple platforms, including Solaris, Mac OS, FreeBSD, OpenBSD, and Linux. In 2005, a new implementation of the interface was developed by Ulrich Drepper and Ingo Molnar of Red Hat, Inc. called the *Native POSIX Thread Library* (*NPTL*), which was much faster than the original library, and has since replaced that library. The Open Group further revised the standard in 2008. We will limit our study of threads to the NPTL implementation of Pthreads. To check whether a Linux system is using the NPTL implementation or a different implementation, run the command

```
getconf GNU_LIBPTHREAD_VERSION
```

The Pthreads library provides a very large number of primitives for the management and use of threads; there are 93 different functions defined in the 2008 POSIX standard. Some thread functions are analogous to those of processes. The following table compares the basic process primitives to analogous Pthread primitives.

Process Primitive	**Thread Primitive**	**Description**
`fork()`	`pthread_create()`	Create a new flow of control with a function to execute
`exit()`	`pthread_exit()`	Exit from the calling flow of control
`waitpid()`	`pthread_join()`	Wait for a specific flow of control to exit and collect its status
`getpid()`	`pthread_self()`	Get the id of the calling flow of control
`abort()`	`pthread_cancel()`	Request abnormal termination of the calling flow of control

The Pthreads API can be categorized roughly into the following four groups.

Thread management: This group contains functions that work directly on threads, such as creating, detaching, joining, and so on. This group also contains functions to set and query thread attributes.

Mutexes: This group contains functions for handling critical sections using mutual exclusion. Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Condition variables: This group contains functions that address communications between threads that share a mutex based upon programmer-specified conditions. These include functions to create, destroy, wait and signal based upon specified variable values, as well as functions to set and query condition variable attributes.

Synchronization: This group contains functions that manage read/write locks and barriers.

We will visit these groups in the order they are listed here, not covering any in great depth, but enough depth to write fairly robust programs.

10.7 Thread Management

Each thread in a process has a unique thread identifier, of type `pthread_t`. This identifier is used by various functions in the Pthreads API. The thread id is returned to the caller of `pthread_create()`, and a thread can obtain its own thread identifier using `pthread_self()`. Thread ids are only unique within a single process. Thread ids might be reused by a system after a thread has been terminated (and joined); an application should never try to reference the id of a terminated thread but should instead use whatever id was returned to the caller of `pthread_create()`. In the remainder of this section we examine the fundamental functions involved in managing threads. We start with the `pthread_create()` function.

10.7.1 Creating Threads

The prototype for `pthread_create()` is

```
int pthread_create  ( pthread_t             *thread,
                      const pthread_attr_t *attr,
                      void * (*start_routine)(void *),
                      void                  *arg);
```

This function creates and starts a new thread in the calling process. On successful creation of the new thread, `thread` contains the thread id of the created thread. Unlike `fork()`, this call passes in the third parameter the address of a function to be executed by the new thread. This function must have exactly one argument, of type `void*`, and its return value must also be `void*`. The fourth argument, `arg`, is the argument that will be passed to the `start_routine()` function in the thread.

The second argument is a pointer to a `pthread_attr_t` structure. This structure can be used to define attributes of the new thread. These attributes include properties such as its stack size, scheduling policy, and *joinability* (to be discussed below). If the program does not specifically set values for its members, default values are used instead. A `NULL` value can be passed instead to use the system defaults. To start, it is easiest to accept the default attributes assigned by the library. We will explore changing thread properties in more detail later.

Because the `start_routine()` function is passed just a single argument, if the function needs to be passed more than a pointer-sized variable, we need a way to give the thread access to the data that it needs. Recall that all threads have their own stacks but can share global memory and heap memory. One solution is to make all data that a threads needs to access global, so that the thread has access to it.

When there is just a single thread, this might be an acceptable solution, but when there are many threads, it raises the specter of race conditions, when threads try to modify the same data at the same time. Even if the data is not modified by the threads, it can create too much contention for main memory.

An alternative is to define a structure with all of the state that needs to be accessed within a thread, and to give each thread its own copy of that structure by pass a pointer to that structure as the argument to `pthread_create()`.

Example

A set of `P` threads needs to access a shared array named `data` that contains `length` elements. Each thread must process a contiguous portion of that array. We can define the structure

```
typedef struct _task_data
{
    int first;     /* index of first element in array for this thread */
    int last;      /* index of last element in array for this thread */
    int *array;    /* pointer to start of data array */
    int task_id;   /* program's id of thread */
    int result;    /* whatever the thread computes */
} task_data;
```

and each thread can then be passed a copy of this structure with the values of `first`, `last`, `array`, and `task_id` initialized before the call. The array pointer may or may not be needed; if the array is declared as a global variable, the threads will have access to it. If the array name is a local variable in the main program, then the array must be allocated on the heap and its address passed to the threads in the struct's member variable. As another option, the array could be declared as a ***static local variable*** in the main program and its name can be passed into the threads through the struct.

The program declares an array of `P task_data` structs, one for each thread, and an array to store the thread ids returned by `pthread_create()`:

```
task_data  thread_data[P];
pthread_t  threads[P];
```

The code to initialize the thread data and create the threads could be

```
for ( unsigned int t = 0 ; t < P; t++) {
    thread_data[t].first     = (t*length)/P;
    thread_data[t].last      = ((t+1)*length)/P -1;
    thread_data[t].task_id   = t;
    thread_data[t].num_segments   = length;

    if ( 0 != (rc = pthread_create(&threads[t], NULL, process_array,
                            (void *) &thread_data[t]) ) )  {
        printf("ERROR; %d return code from pthread_create()\n", rc);
        exit(-1);
    }
}
```

This would create P many threads, each executing a function named `process_array`(), each with its own structure containing parameters of its execution. The assignment of array segments follows the same strategy as was described in the notes on the parallel implementation of the Floyd-Warshall algorithm.

Note that we could have created a single array and made the thread id a part of the structure, as in

```
typedef struct _task_data
{
    int first;     /* index of first element in array for this thread */
    int last;      /* index of last element in array for this thread */
    int *array;    /* pointer to start of data array */
    int task_id;   /* program's id of thread */
    pthread_t thread_id; /* system's id for thread */
    int result;    /* whatever the thread computes */
} task_data;
```

Design Decision Regarding Shared Data

The advantage of declaring the data array as a static local variable in the main program is that code is easier to analyze and maintain when there are fewer global variables and potential side effects. Programs with functions that modify global variables are harder to analyze. On the other hand, making it a static local in main and then having to add a pointer to that array in the thread data structure passed to each thread increases thread storage requirements and slows down the program. Each thread has an extra pointer in its stack when it executes, and each reference to the array requires two dereferences instead of one. Which is preferable? It depends what the overall project requirements are. If speed and memory are a concern, use a global array and use good practices in documenting and accessing it. If not, use the static local. Of course if the array's size is not known statically, then it will be created dynamically and the threads will have to access it through a pointer in a thread structure.

10.7.2 Thread Identification

A thread can get its thread ID by calling `pthread_self()`, whose prototype is

```
pthread_t pthread_self(void);
```

This is the analog to `getpid()` for processes, and to `MPI_Comm_rank()`. This function is the only way that the thread can get its ID, because it is not provided to it by the creation call. It is entirely analogous to `fork()` in this respect.

A thread can check whether two thread IDs are equal by calling

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

This returns a non-zero if the two thread IDs are equal and zero if they are not.

10.7.3 Thread Termination

A thread can terminate itself by calling `pthread_exit()`:

```
void pthread_exit(void *retval);
```

This function terminates the calling thread. The `pthread_exit()` function never returns. Analogous to the way that `exit()` returns a value to `wait()`, the return value may be examined from another thread in the same process if it calls `pthread_join()`[2]. The value pointed to by `retval` should not be located on the calling thread's stack, since the contents of that stack are undefined after the thread terminates. It can be a global variable or allocated on the heap. Therefore, if you want to use a locally-scoped variable for the return value, declare it as static within the thread.

It is a good idea for the main program to terminate itself by calling `pthread_exit()`, because if it has not waited for spawned threads and they are still running, if it calls `exit()`, they will be killed. The `exit()` call terminates all threads if any thread of a process calls it. If these threads should not be terminated, then calling `pthread_exit()` from `main()` will ensure that they continue to execute. Listing 10.4 contains a short program that demonstrates this. The thread displays the process id that can be used to kill it from the terminal.

Listing 10.4: Example showing that threads continue to run after main thread calls `pthread_exit()`

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

void * child( void * unused)
{
    pthread_t tid = pthread_self(); /* get thread id  */
    pid_t     pid = getpid();       /* get process id */
    while ( 1 ) {
        sleep(1);
        printf("Thread %lu, part of process %d is still running\n",
                tid, pid);
```

[2]Provided that the terminating thread is joinable.

```
    }
    pthread_exit(NULL) ;
}

int main( int argc, char *argv[])
{
    pthread_t  child_thread;

    /* Create the thread and launch it. */
    if ( 0 != pthread_create(&child_thread, NULL, child, NULL ) ){
        printf("pthread_create failed.\n");
        exit(1);
    }

    sleep(10);
    printf("This is the parent thread. It is about to terminate.\n");
    pthread_exit(NULL);  /* replace this with a call to exit and
                            observe the difference */

    return 0;
}
```

10.7.4 Thread Joining and Joinability

When a thread is created, one of the attributes defined for it is whether it is *joinable* or *detached*. By default, created threads are joinable. If a thread is joinable, another thread can wait for its termination using the function `pthread_join()`. Only threads that are created as joinable can be joined.

Joining is a way for one thread to wait for another thread to terminate, in much the same way that the `wait()` system calls lets a process wait for a child process. When a parent process creates a thread, it may need to know when that thread has terminated before it can perform some task. Joining a thread, like waiting for a process, is a way to synchronize the performance of tasks. It also allows the child thread to pass its status on exiting to the process calling `pthread_join()`.

However joining is different from waiting in one respect: the thread that calls `pthread_join()` must specify the thread ID of the thread for which it waits. In this respect it is like the analogous `waitpid()`. The prototype is

```
int pthread_join(pthread_t thread, void **value_ptr);
```

The `pthread_join()` function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated. If the target thread already terminated, `pthread_join()` returns immediately and successfully.

If `value_ptr` is not `NULL`, then the value passed to `pthread_exit()` by the terminating thread will be available in the location referenced by `value_ptr`, provided `pthread_join()` succeeds.

Some things that cause problems include:

- Multiple simultaneous calls to `pthread_join()` specifying the same target thread have undefined results.
- The behavior is undefined if the value specified by the thread argument to `pthread_join()` does not refer to a joinable thread.
- The behavior is undefined if the value specified by the thread argument to `pthread_join()` refers to the calling thread.
- Failing to join with a thread that is joinable produces a "zombie thread". Each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

The following listing shows a simple program that creates a single thread and waits for it using `pthread_join()`, collecting and printing its exit status.

Listing 10.5: Simple example of thread creation with join

```
void* hello_world( void * world)
{
    static int exitval; /* The exit value cannot be on the stack */

    printf("Hello World from %s.\n", (char*) world);
    exitval = 2;
    pthread_exit((void*) exitval) ;
}

int main( int argc, char *argv[])
{
    pthread_t  child_thread;
    void  *status;
    char  *planet  = "Pluto";

    if ( 0 != pthread_create(&child_thread, NULL,
                             hello_world,( void*) planet) ) {
        perror("pthread_create");
        exit(-1);
    }
    /* Call join passing address of status, which is a pointer to
       void */
    pthread_join(child_thread, (void**) (&status));
    printf("Child exited with status %ld\n", (long) status);
    return 0;
}
```

Any thread in a process can join with any other thread. They are peers in this sense. The only obstacle is that to join a thread, it needs its thread ID.

10.7.5 Detached Threads

Because `pthread_join()` must be able to retrieve the status and thread ID of a terminated thread, this information must be stored someplace. In many Pthread implementations, it is stored in a structure that we will call a *Thread Control Block* (TCB). In these implementations, the entire

TCB is kept around after the thread terminates, just because it is easier to do this. Therefore, until a thread has been joined, this TCB exists and uses memory. Failing to join a joinable thread turns these TCBs into wasted memory.

Sometimes threads are created that do not need to be joined. Consider a process that spawns a thread for the sole purpose of writing output to a file. The process does not need to wait for this thread. When a thread is created that does not need to be joined, it can be created as a *detached thread*. When a detached thread terminates, no resources are saved; the system cleans up all resources related to the thread.

A thread can be created in a detached state, or it can be detached after it already exists. To create a thread in a detached state, you can use the `pthread_attr_setdetachstate()` function to modify the `pthread_attr_t` structure prior to creating the thread, as in:

```
pthread_t      tid;  /* thread ID        */
pthread_attr_t attr; /* thread attribute */

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

/* now create the thread */
pthread_create(&tid, &attr, start_routine, arg);
```

An existing thread can be detached using `pthread_detach()`:

```
int pthread_detach(pthread_t thread);
```

The function `pthread_detach()` can be called from any thread, in particular from within the thread itself! It would need to get its thread ID using `pthread_self()`, as in

```
pthread_detach(pthread_self());
```

Once a thread is detached, it cannot become joinable. It is an irreversible decision. The following listing shows how a main program can exit, using `pthread_exit()` to allow its detached child to run and produce output, even after `main()` has ended. The call to `usleep()` gives a bit of a delay to simulate computationally demanding output being produced by the child.

Listing 10.6: Example of detached child

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

void *thread_routine(void * arg)
{
    int    i;
    int    bufsize = strlen(arg);
    int    fd = 1;
    printf("Child is running...\n");
    for (i = 0; i < bufsize; i++) {
                usleep(500000);
                write(fd, arg+i, 1);
    }
    printf("\nChild is now exiting.\n");
    return(NULL);
}

int main(int argc, char* argv[])
{
    char * buf = "abcdefghijklmnopqrstuvwxyz";
    pthread_t thread;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    if (pthread_create(&thread, NULL, thread_routine, (void *)(buf))) {
               fprintf(stderr, "error creating a new thread \n");
               exit(1);
    }

    printf("Main is now exiting.\n");
    pthread_exit(NULL);
}
```

10.7.6 Example: Calculating Pi

A simple application of the use of threads that does not involve more than thread creation and joining is a multi-threaded solution to the problem of estimating the value of π by using numerical integration. It demonstrates how to pass the required parameters and collect the results.

The program declares a task_data structure globally so that the main thread and child threads all have access to its definition:

```
#include <pthread.h>
/*
   task_data is a structure that contains the data required for
   a thread to compute the sum of the partial results it has been
   delegated to calculate, storing the sum in the struct.  The
   struct contains first, last segments. and their task number
   and total number of segments, and thread id.
*/
typedef struct _task_data
{
    int first;            /* index of first element for task */
    int last;             /* index of last element for task */
    int num_segments;     /* total number of segments to be calculated */
    double sum;           /* sum of values computed by this thread */
    pthread_t thread_id; /* id returned by pthread_create()   */
    int task_num;         /* program's thread id     */
} task_data;
```

The thread function, which each thread uses to approximate πnumerically, is called `approximate_pi` and is given below. Notice that each thread stores its partial result in the sum member of the passed-in parameter, and that each thread computes the areas of a consecutive sequence of approximating rectangles. See Figure 10.1.

```
void* approximate_pi ( void  *thread_data )
{
    double dx, x;
    task_data *t_data = (task_data*) thread_data;
    int   k;

    /* Set dx to the width of each segments */
    dx = 1.0 / (double) t_data->num_segments;

    /* Initialize sum to 0 to be safe */
    t_data->sum = 0;

     for ( k = t_data->first; k <= t_data->last; k++ ) {
        x = dx * ((double)k - 0.5);      /* x is midpoint of segment k */
        t_data->sum += 4.0 / (1.0 + x*x);    /* add new area to sum */
    }

    /* multiply sum by dx because we are computing
       an integral and dx is the differential */
    t_data->sum =  dx * t_data->sum;

    pthread_exit((void*) 0);

}
```

Following is the main program.

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

int main( int argc, char *argv[] )
{
    double total;           /* estimated value of pi       */
    int    num_intervals;   /* number of segments to sum   */
    int    num_threads;     /* number of threads this program will use */

    int    retval;
```

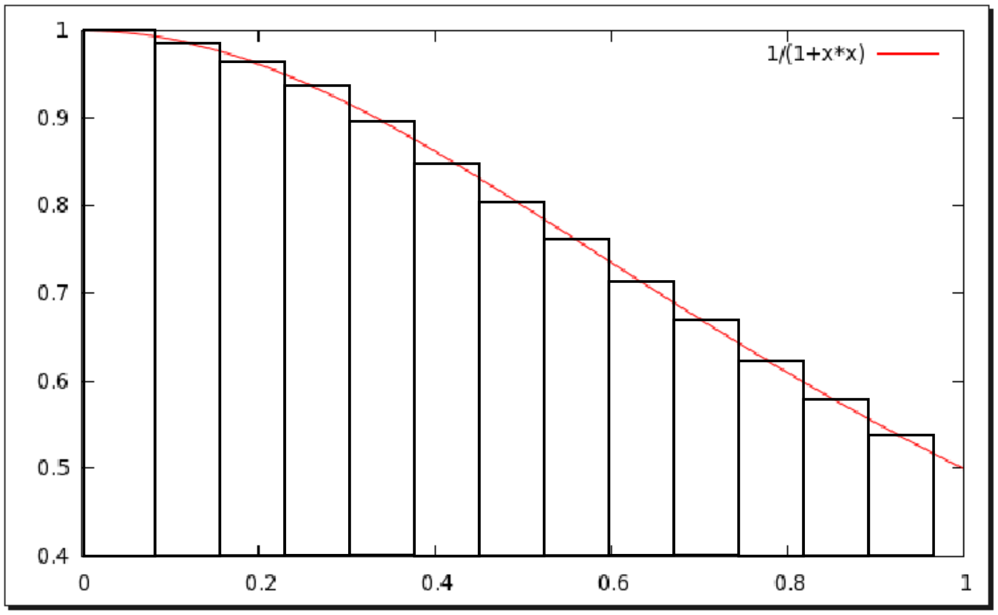

Figure 10.1: Numerical integration of $f(x) = 1/(1 + x^2)$

```
    int      t;

    task_data  *thread_data;  /*  array of thread data */
    pthread_attr_t attr;

    /* Make all threads joinable */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    if ( argc < 3 ) {
        exit(1);
    }

    /* Get command line arguments, convert to ints, and compute
       size of each thread's segment of the array
    */
    num_intervals  = atoi(argv[1]);
    num_threads    = atoi(argv[2]);
    if ( (0 == num_intervals ) || ( 0 == num_threads )) {
        printf("ERROR; insufficient memory\n");
        exit(1);
    }

    /* Allocate the array of task_data structures on the heap.
        This is necessary because the array is not global.   */
    thread_data = calloc( num_threads, sizeof(task_data));
    if  ( thread_data == NULL  )
        exit(1);

    /* Initialize task_data for each thread and create the threads */
    for ( t = 0 ; t < num_threads; t++) {
        thread_data[t].first     = (t*num_intervals)/num_threads;
        thread_data[t].last      = ((t+1)*num_intervals)/num_threads -1;
        thread_data[t].task_num  = t;
        thread_data[t].sum       = 0;
        thread_data[t].num_segments  = num_intervals;

        retval = pthread_create(&(thread_data[t].thread_id), &attr,
                                approximate_pi,
                                (void *) &thread_data[t]);
        if ( retval ) {
            printf("ERROR; return code from pthread_create() is %d\n",
                   retval);
            exit(-1);
        }
    }

   /* Join all threads so that we can add up their partial sums */
    for ( t = 0 ; t < num_threads; t++) {
        pthread_join(thread_data[t].thread_id, (void**) NULL);
    }
    /* Collect partial sums into a final total */
    total = 0;
    for ( t = 0 ; t < num_threads; t++) {
        total += thread_data[t].sum;
    }

    printf("pi is approximated to be %.16f. The error is %.16f\n",
    total, fabs(total - M_PI));
    fflush(stdout);

    /* Free  memory allocated to program */
    free ( thread_data );
    return 0;
}
```

Observation

This program is not ideal because a single thread sums the partial sums obtained by all other threads after the all terminate. It would be faster if the total could be computed in parallel. For example, suppose we modify the program as follows. First we make `total` a global variable and change the `sum` member of the task_data structure so that it can be used as a pointer to `total`:

```
#include <pthread.h>

static double total = 0;

typedef struct _task_data
{
    int first;           /* index of first element for task */
    int last;            /* index of last element for task */
    int num_segments;    /* total number of segments to be calculated */
    double* sum;         /* pointer to sum  computed by all threads */
    pthread_t thread_id; /* id returned by pthread_create()   */
    int task_num;        /* program's thread id     */
} task_data;
```

Next we modify the main program:

```
    /* Declare and calculate dx in the main program
       after getting the number of intervals from command line*/
    double dx = 1.0 / (double) num_intervals;;

   /* Initialize task_data for each thread and create the threads */
    for ( t = 0 ; t < num_threads; t++) {
        thread_data[t].first     = (t*num_intervals)/num_threads;
        thread_data[t].last      = ((t+1)*num_intervals)/num_threads -1;
        thread_data[t].task_num  = t;
        thread_data[t].sum       = &total;  /* CHANGED */
        thread_data[t].num_segments  = num_intervals;

        retval = pthread_create(&(thread_data[t].thread_id), &attr,
                                approximate_pi,
                                (void *) &thread_data[t]);

   /* Join all threads*/
    for ( t = 0 ; t < num_threads; t++) {
        pthread_join(thread_data[t].thread_id, (void**) NULL);
    }

    total = total * dx;

    /* print the value of total */
    printf("pi is approximated to be %.16f. The error is %.16f\n",
    total, fabs(total - M_PI));
```

Lastly, we need to modify `approximate_pi()` because it treats the member `sum` as a `double` and now it is a pointer to a `double`:

```
void* approximate_pi ( void  *thread_data )
{
    double dx, x;
    task_data *t_data = (task_data*) thread_data;
    int   k;

    /* Set dx to the width of each segments */
    dx = 1.0 / (double) t_data->num_segments;

    /* we do not multiply by dx here; instead we do it once in main */
    for ( k = t_data->first; k <= t_data->last; k++ ) {
        x = dx * ((double)k - 0.5);      /* x is midpoint of segment k */
        *(t_data->sum) += 4.0 / (1.0 + x*x);    /* add new area to sum */
    }

    /* multiply sum by dx because we are computing
       an integral and dx is the differential */
    *(t_data->sum) *=  dx;

    pthread_exit((void*) 0);

}
```

This code avoids the main thread's having to compute the total, but is it correct? No, because there is are race conditions in the `approximate_pi()` function in two places where it updates the global `sum` through `t_data->sum`. Soon we will see how to fix this.

10.7.7 Thread Cancellation

Threads can be ***canceled*** as well. Cancellation is roughly like killing a thread. When a thread is canceled, its resources are cleaned up and it is terminated. A thread can request that another thread be canceled by calling `pthread_cancel()`, the prototype for which is

```
int pthread_cancel(pthread_t thread);
```

This is just a request; it is not necessarily honored. When this is called, a cancellation request is sent to the thread given as the argument. Whether or not that thread is canceled depends upon the thread's ***cancelability state*** and ***cancelability type***. A thread can enable or disable cancelability, and it can also specify whether its cancelability type is ***asynchronous*** or ***deferred***. If a thread's cancelability type is asynchronous, then it will be canceled immediately upon receiving a cancellation request, assuming it has enabled its cancelability. On the other hand, if its cancelability is deferred, then cancellation requests are deferred until the thread enters a ***cancellation point***. Certain functions are cancellation points. To be precise, if a thread is cancelable, and its type is deferred, and a cancellation request is pending for it, then if it calls a function that is a cancellation point, it will be terminated immediately. The list of cancellation point functions required by POSIX can be found on the man page for `pthreads` in Section 7.

A thread's cancelability state is enabled by default and can be set by calling `pthread_setcancelstate()`:

```
int pthread_setcancelstate(int state, int *oldstate);
```

The two values are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE. The new state is passed as the first argument and a pointer to an integer to store the old state, or NULL, is the second argument. If a thread disables cancellation, then a cancellation request remains queued until it enables cancellation. If a thread has enabled cancellation, then its cancelability type determines when cancellation occurs.

A thread's cancellation type, which is deferred by default, can be set with pthread_setcanceltype() :

```
int pthread_setcanceltype(int type, int *oldtype);
```

To set the type to asynchronous, pass PTHREAD_CANCEL_ASYNCHRONOUS in the first argument. To make it deferred, pass PTHREAD_CANCEL_DEFERRED.

The only way for the caller of pthread_cancel() to know if the target thread was canceled is to join that thread and check if the returned status is PTHREAD_CANCELED, as in the following code:

```
retval = pthread_join(thread, &res);
if (retval)
    /* join failed - report it */
else
    if (res == PTHREAD_CANCELED)
        printf("Thread was canceled\n");
    else
        printf("Thread was not canceled.\n");
```

10.7.8 Thread Properties

10.7.8.1 Stack Size

The POSIX standard does not dictate the size of a thread's stack, which can vary from one implementation to another. Furthermore, with today's demanding problems, exceeding the default stack limit is not so unusual, and if it happens, the program will terminate, possibly with corrupted data.

Safe and portable programs do not depend upon the default stack limit, but instead, explicitly allocate a large enough stack for each thread by using the pthread_attr_setstacksize() function, whose prototype is

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

The first argument is the address of the thread's attribute structure and the second is the size that you want to set for the stack. This function will fail if the attribute structure does not exist, or if the stack size is smaller than the allowed minimum (PTHREAD_STACK_MIN) or larger than the maximum allowed. See the man page for further caveats about its use.

To get the stack's current size, use

```
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
```

This retrieves the current size of the stack. It will fail of course if `attr` does not reference an existing structure.

The problem when trying to use this function is that it must be passed the attributes structure of the thread. There is no POSIX function to retrieve the attribute structure of the calling thread, but there is a GNU extension, **`pthread_getattr_np()`**. If this extension is not used, the best that the calling thread can do is to get a copy of the attribute structure with which it was created, which may have different values than the one it is currently using. The following listing is of a program that prints the default stack size then sets the new stack size based on a command line argument, and from within the thread, displays the actual stack size it is using, using the GNU **`pthread_getattr_np()`** function. *To save space, some error checking has been removed.*

Listing 10.7: Setting a new stack size (with missing error checking)

```
#define _GNU_SOURCE     /* To get pthread_getattr_np() declaration */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

void *thread_start(void *arg)
{
   size_t         stack_size;
   pthread_attr_t gattr;

   pthread_getattr_np       ( pthread_self(),  &gattr);
   pthread_attr_getstacksize( &gattr,          &stack_size);
   printf("Actual stack size is %ld\n", stack_size);
   pthread_exit(0);
}

int main(int argc, char *argv[])
{
    pthread_t         thr;
    pthread_attr_t    attr;
    int               retval;
    size_t            new_stack_size, stack_size;
    void              *sp;

    if ( argc < 2 ) {
        printf("usage: %s stacksize\n", argv[0] );
        exit(1);
    }

    new_stack_size = strtoul(argv[1], NULL, 0);

    retval = pthread_attr_init(&attr);
    if (retval) {
        exit(1);
    }
    pthread_attr_getstacksize (&attr, &stack_size);
    printf("Default stack size = %ld\n", stack_size);
    printf("New stack size will be %ld\n", new_stack_size);
    retval = pthread_attr_setstacksize(&attr, new_stack_size);
    if ( retval ) {
        exit(1);
    }

    retval = pthread_create(&thr, &attr, &thread_start, NULL);
    if ( retval ) {
        exit(1);
    }

    pthread_join(thr, NULL);
    return(0);
}
```

10.8 Mutexes

10.8.1 Introduction

When multiple threads share the same memory, the programmer must ensure that each thread sees a consistent view of its data. If each thread uses variables that no other threads read or modify, then there are no consistency problems with those variables. Similarly, if a variable is read-only, there is no consistency problem if multiple threads read its value at the same time. The problem occurs when one thread can modify a variable that other threads can read or modify. We saw earlier that this can lead to race conditions. In this case the threads must be synchronized with respect to the shared variable. The segment of code in which this shared variable is accessed within a thread, whether for a read or a write, is called a ***critical section***.

A simple example of a critical section occurs when each thread in a group of threads needs to increment some shared counter, after which it does some work that depends on the value of that counter. The main program would initialize the counter to zero, after which each thread would increment the counter and use it to access the array element indexed by that value. The following code typifies this scenario.

```
void * work_on_ticker( void * counter)
{
    int i;
    int  *ticker = (int*) counter;
    int temp;

    for ( i = 0; i < NUM_UPDATES; i++ ) {
        temp = *ticker;
        /* do something that takes some time here */
        usleep(100000);
        *ticker = temp + 1;
        /* use the ticker to do stuff here with A[*ticker] */
    }
    pthread_exit( NULL );
}
```

If the increment of `*ticker` is not executed in mutual exclusion, some threads may overwrite other threads' array data, and some array elements may remain unprocessed because the ticker skipped over them. You will probably not see this effect if this code is executed on a single-processor machine, as the threads will be time-sliced on the processor, and the likelihood of their being sliced in the middle of the update to the ticker is very small, but if you run this on a multi-processor machine, you will almost certainly see the effect.

There are various ways to enforce mutually exclusive access to an object. A ***mutex*** is one of the provisions of Pthreads for providing mutually exclusive access to critical sections. A *mutex* is like a software version of a lock. Its name derives from "mutual exclusion" because a mutex can only be held, or *owned*, by one thread at a time. The typical use of a mutex is to surround a critical section of code with a call to lock and then to unlock the mutex, as in

```
pthread_mutex_lock  ( &mutex );
/* critical section here */
pthread_mutex_unlock( &mutex );
```

Mutexes are a low-level form of critical section protection, providing the most rudimentary features. They were intended as the building blocks of higher-level synchronization methods. Nonetheless, they can be used in many cases to solve critical section problems. In the remainder of this section, we describe the fundamentals of using mutexes.

10.8.2 Creating and Initializing Mutexes

A mutex is a variable of type `pthread_mutex_t`. It must be initialized before it can be used. There are two ways to initialize a mutex:

1. Statically, when it is declared, using the `PTHREAD_MUTEX_INITIALIZER` macro, as in

   ```
   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
   ```

2. Dynamically, with the `pthread_mutex_init()` routine:

   ```
   int pthread_mutex_init(pthread_mutex_t *mutex,  pthread_mutexattr_t *attr);
   ```

   This function is given a pointer to a mutex and to a *mutex attribute structure*, and initializes the mutex to have the properties of that structure. If one is willing to accept the default mutex attributes, the `attr` argument may be `NULL`.

In both cases, the mutex is initially unlocked. The call

```
pthread_mutex_init(&mutex, NULL);
```

is equivalent to the static method except that no error-checking is done.

10.8.3 Locking a Mutex

To lock a mutex, one uses one of the functions

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

We will begin with `pthread_mutex_lock()`. The semantics of this function are a bit complex, in part because there are different types of mutexes. Here we describe the semantics of *normal* mutexes, which are the default type, `PTHREAD_MUTEX_NORMAL`.

If the mutex is not locked, the call returns with the mutex object referenced by mutex in the locked state with the calling thread as its *owner*. The return value will be 0. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked. If a thread tries to lock a mutex that it has already locked, it causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results. We will discuss the other types of mutexes later.

In short, if several threads try to lock a mutex only one thread will be successful. The other threads will be in a blocked state until the mutex is unlocked by its owner.

If a signal is delivered to a thread that is blocked on a mutex, when the thread returns from the signal handler, it resumes waiting for the mutex as if it had not been interrupted.

The `pthread_mutex_trylock()` function behaves the same as the `pthread_mutex_lock()` function except that it never blocks the calling thread. Specifically, if the mutex is unlocked, the calling thread acquires it and the function returns a 0, and if the mutex is already locked by any thread, the function returns the error value `EBUSY`.

10.8.4 Unlocking a Mutex

The call to unlock a mutex is

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The `pthread_mutex_unlock()` function will unlock a mutex if it is called by the owning thread. If a thread that does not own the mutex calls this function, it is an error. It is also an error to call this function if the mutex is not locked. If there are threads blocked on the mutex object referenced by mutex when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy determines which thread next acquires the mutex. If the mutex is a normal mutex that used the default initialization, there is no specific thread scheduling policy, and the underlying kernel scheduler makes the decision. The behavior of this function for non-normal mutexes is different.

10.8.5 Destroying a Mutex

When a mutex is no longer needed, it should be destroyed using

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

The `pthread_mutex_destroy()` function destroys the mutex object referenced by mutex; the mutex object becomes uninitialized. The results of referencing the mutex object after it has been destroyed are undefined. A destroyed mutex object can be reinitialized using `pthread_mutex_init()`.

10.8.6 Examples Using a Normal Mutex

Two examples will show how threads can use mutexes to protect their updates to a shared, global variable. The first example will demonstrate how multiple threads can increment a shared counter that serves as an index into a global array, so that no two threads access the same array element. Each thread will then modify that array element. In the second example, the update to the shared variable is on the back-end of the problem. Each thread is given an equal-size segment of two arrays, computes a function of this pair of segments, and adds the value of that function to a shared, global accumulator.

Example 1

Suppose that we want a function which, when given an integer `N` and an array `roots` of size `N`, stores the square roots of the first `N` non-negative integers into `roots`. A sequential version of this function would execute a loop of the form

```
for ( i = 0; i < N; i++ )
    roots[i] = sqrt(i);
```

To make this program run faster when there are multiple processors available, we distribute the work among multiple threads. Let P be the number of threads that will jointly solve this problem. Each thread will compute the square roots of a set of `N`/P integers. These integers are not necessarily consecutive. The idea is that each thread concurrently iterates a loop N times, incrementing a shared, global counter mutually exclusively in each iteration. In each iteration, the thread computes the square root of the current counter value and stores it in an array of roots at the position indexed by the counter value.

The program is in Listing 10.8. All of the multi-threading is opaque to the main program because it is encapsulated in a function. This way it can be ported easily to a different application.

To simplify the program, the array size and number of threads are hard-coded as macros in the program. This is easily changed.

Listing 10.8: A multi-threaded program to compute the first N square roots.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pthread.h>
#include <errno.h>
#include <math.h>
```

```
#define  NUM_THREADS       20                     /* Number of threads */
#define  NUMS_PER_THREAD   50                     /* Number of roots per thread */
#define  SIZE  (NUM_THREADS*NUMS_PER_THREAD)  /* Total roots to compute */

/* Declare a structure to pass multiple variables to the threads in the
   pthread_create() function and for the thread routine to access in its single
   argument.
*/
typedef struct  _thread_data
{
    int      count;           /* shared counter, incremented by each thread */
    int      size;            /* length of the roots array */
    int      nums_per_thread; /* number of roots computed by each thread */
    double*  roots;           /* pointer to the roots array */

} thread_data;

pthread_mutex_t update_mutex;  /* Declare a global mutex */


/**************************************************************************
                          Thread and Helper Functions
**************************************************************************/

/** handle_error(num,mssge)
 *  A convenient error handling function
 *  Prints to standard error the system message associated with errno num
 *  as well as a custom message, and then exits the program with EXIT_FAILURE
 */
void handle_error(int num, char *mssge)
{
    errno = num;
    perror(mssge);
    exit(EXIT_FAILURE);
}

/** calc_square_roots()
 *  A thread routine that calculates the square roots of N  integers
 *  and stores them in an array. The integers are not necessarily consecutive;
 *  as it depends how the threads are scheduled.
 *  @param  [out] double data->roots[] is the array in which to store the roots
 *  @param  [inout] int data->count is the first integer whose root should be
 *                                     calculated
 *  This  increments data->count N times.
 *
 *  Loops to waste time a bit so that the threads may be scheduled out of order.
 */
void * calc_square_roots( void * data)
{
    int  i, j;
    int  temp;
    int  size;
    int  nums_to_compute;
    thread_data *t_data = (thread_data*) data;

    size            = t_data->size;
    nums_to_compute = t_data->nums_per_thread;

    for ( i = 0; i < nums_to_compute; i++ ) {
        pthread_mutex_lock (&update_mutex);   /* lock mutex   */
        temp = t_data->count;
        t_data->count = temp + 1;
        pthread_mutex_unlock (&update_mutex); /* unlock mutex */

        /* updating the array can be done outside of the CS since temp is
           a local variable to the thread. */
        t_data->roots[temp] = sqrt(temp);

        /* idle loop */
        for ( j = 0; j < 1000; j++ )
            ;
    }
    pthread_exit( NULL );
}

/** compute_roots()
 *  computes the square roots of the first num_threads*roots_per_thread many
 *  integers. It hides the fact that it uses multiple threads to do this.
 */
void compute_roots( double sqrts[],  int size, int num_threads )
{
    pthread_t     threads[num_threads];
    int           t;
    int           retval;
    static thread_data  t_data;

    t_data.count = 0;
    t_data.size  = size;
    t_data.nums_per_thread = size / num_threads;
    t_data.roots = &sqrts[0];

    /* Initialize the mutex */
    pthread_mutex_init(&update_mutex, NULL);

    /* Initialize task_data for each thread and then create the thread */
    for ( t = 0 ; t < num_threads; t++) {
        retval = pthread_create(&threads[t], NULL, calc_square_roots,
                               (void *) &t_data);
        if ( retval )
            handle_error( retval, "pthread_create");
    }

    /* Join all threads and then print sum */
    for ( t = 0 ; t < num_threads; t++)
        pthread_join(threads[t], (void**) NULL);
}
```

```
/*****************************************************************************
                            Main Program
*****************************************************************************/

int main( int argc, char *argv[])
{
    int    t;
    double roots[SIZE];

    memset((void*) &roots[0], 0, SIZE * sizeof(double));
    compute_roots(roots, SIZE, NUM_THREADS );

    for ( t = 0 ; t < SIZE; t++)
        printf("Square root of %5d is %6.3f\n", t, roots[t]);
    return 0;

}
```

A slightly different approach to this program is to allow each thread to compute as many roots as it can, as if the threads were in a race with each other. If the threads were scheduled on asymmetric processors, some being much faster than others, or if some threads had faster access to memory than others, so that they could do more work per unit time, then it would be advantageous to let these threads do more, rather than limiting them to a fixed number of roots to compute. This is the basis for the variation of `calc_square_roots()` from Listing 10.8 found in Listing 10.9.

The function in Listing 10.9 lets each thread iterate from 0 to `size` but it checks in each iteration whether the value of the counter has exceeded the array size, and if it has, that thread terminates. It has an extra feature that is used by the main program and requires a bit of extra code outside of the function – it stores the id of the thread that computed the root in a global array that can be printed to see how uniformly the work was distributed.

Listing 10.9: A "greedy" thread function.

```
/*
   This function also stores the id of the thread that computed each
   root in a global array so that the main program can print these
   results. If it did not do this, there would be no need for the
   lines marked with /****.
*/
void * calc_square_roots( void * data)
{
    int   i, j;
    int   temp;            /* local copy of counter */
    int   size;            /* local copy of size of roots array */
    int   nums_to_compute; /* local copy of number of roots to compute */
    thread_data *t_data = (thread_data*) data;

    int   my_id;           /**** unique id for this thread */

    /* Copy to local copies for faster access */
    size            = t_data->size;
    nums_to_compute = t_data->nums_per_thread;

    /* Each thread gets a unique thread_id by locking this mutex,
       capturing the current value of tid, assigning it to its own
       local variable and then incrementing it.
    */
    pthread_mutex_lock(&id_mutex);   /**** lock mutex   */
    my_id = tid;                     /**** copy tid to local my_id */
    tid++;                           /**** increment tid for next thread*/
    pthread_mutex_unlock(&id_mutex); /**** unlock mutex */

    i = 0;
    while ( i < size ) {
        pthread_mutex_lock (&update_mutex);   /* lock mutex   */
        temp = t_data->count;
        t_data->count = temp + 1;
        pthread_mutex_unlock (&update_mutex); /* unlock mutex */

        /* Check if the counter exceeds the roots array size */
        if ( temp >= size )
            break;

        /* updating the arrays can be done outside of the CS since temp
           and my_id are local variables to the thread. */
        t_data->roots[temp] = sqrt(temp);

        /* Store the id of the thread that just computed this root. */
        computed_by[temp] =  my_id;   /**** store the id */

        /* idle loop */
        for ( j = 0; j < 1000; j++ )
            ;
        i++;
    }
    pthread_exit( NULL );
}
```

Example 2

The second example, in Listing 10.10, computes the inner product of two vectors V and W by partitioning V and W into subvectors of equal sizes and giving the subproblems to separate threads. Assume for simplicity that V and W are each of length N and that the number of threads, P, divides N without remainder and let $s = N/P$. The actual code does not assume anything about N and P. The main program creates P threads, with ids $0, 1, 2, ... P-1$. The thread with id k computes the inner product of $V[k \cdot s \cdots (k+1) \cdot s - 1]$ and $W[k \cdot s \cdots (k+1) \cdot s - 1]$ and stores the result in a temporary variable, temp_sum. It then locks a mutex and adds this partial sum to the global variable sum and unlocks the mutex afterward.

This example uses the technique of declaring the vectors and the sum as static locals in the main program.

Listing 10.10: Mutex example: Computing the inner product of two vectors.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libintl.h>
#include <locale.h>
#include <math.h>
#include <errno.h>

#define  NUM_THREADS    20

typedef struct _task_data
{
   int          first;
   int          last;
   double       *a;
   double       *b;
   double       *sum;
} task_data;


pthread_mutex_t mutexsum; /* Declare the mutex globally */

/*****************************************************************************
                         Thread and Helper Functions
*****************************************************************************/
void usage(char *s)
{
        char *p = strrchr(s, '/');
        fprintf(stderr,
                "usage: %s length datafile1 datafile2  \n", p ? p + 1 : s);
}

void handle_error(int num, char *mssge)
{
    errno = num;
    perror(mssge);
    exit(EXIT_FAILURE);
}

/**
   This function computes the inner product of the sub-vectors
   thread_data->a[first..last] and thread_data->b[first..last],
   adding that sum to thread_data->sum within the critical section
   protected by the shared mutex.
*/
void*  inner_product( void  *thread_data )
{
    task_data *t_data;
    int        k;
    double      temp_sum = 0;

    t_data     = (task_data*) thread_data;

    for ( k = t_data->first; k <= t_data->last; k++ )
        temp_sum += t_data->a[k] * t_data->b[k];

    pthread_mutex_lock (&mutexsum);
    *(t_data->sum) += temp_sum;
    pthread_mutex_unlock (&mutexsum);

    pthread_exit((void*) 0);
}

/*****************************************************************************
                                    Main Program
*****************************************************************************/

int main( int argc, char *argv[])
{
    static double   *a_vector;
    static double   *b_vector;
    FILE            *fp;
    float           x;
    int             num_threads = NUM_THREADS;
    int             length;
    int             segment_size;
    static double   total;
    int             k;
    int             retval;
    int             t;
    pthread_t       *threads;
    task_data       *thread_data;
    pthread_attr_t  attr;


    if ( argc < 4 ) { /* Check usage */
        usage(argv[0]);
        exit(1);
    }

    /* Get command line args, no input validation here */
    length       = atoi(argv[1]);
    a_vector     = calloc( length, sizeof(double));
    b_vector     = calloc( length, sizeof(double));

    /* Zero the two vectors */
    memset(a_vector, 0, length*sizeof(double));
    memset(b_vector, 0, length*sizeof(double));

    /* Open the first file, do check for failure and read the numbers
       from the file. Assume that it is in proper format
    */
    if ( NULL == (fp = fopen(argv[2], "r")) )
```

```
        handle_error(errno, "fopen");
    k = 0;
    while (( fscanf(fp, " %f ", &x) > 0 ) && (k < length) )
        a_vector[k++] = x;
    fclose(fp);

    /* Open the second file, do check for failure and read the numbers
       from the file. Assume that it is in proper format
    */
    if ( NULL == (fp = fopen(argv[3], "r")) )
        handle_error(errno, "fopen");
    k = 0;
    while (( fscanf(fp, " %f ", &x) > 0 ) && (k < length) )
        b_vector[k++] = x;
    fclose(fp);


    /* Allocate the array of threads and task_data structures*/
    threads     = calloc( num_threads, sizeof(pthread_t));
    thread_data = calloc( num_threads, sizeof(task_data));
    if ( threads == NULL || thread_data == NULL  )
        exit(1);

    /* Compute the size each thread will get */
    segment_size = (int) ceil (length*1.0 / num_threads);

    /* Initialize the mutex */
    pthread_mutex_init(&mutexsum, NULL);

    /* Get ready -- initialize the thread attributes */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    /* Initialize task_data for each thread and then create the thread */
    for ( t = 0 ; t < num_threads; t++) {
        thread_data[t].first     = t*segment_size;
        thread_data[t].last      = (t+1)*segment_size -1;
        if ( thread_data[t].last > length -1 )
            thread_data[t].last  = length - 1;
        thread_data[t].a         = &a_vector[0];
        thread_data[t].b         = &b_vector[0];
        thread_data[t].sum       = &total;

        retval = pthread_create(&threads[t], &attr, inner_product,
                              (void *) &thread_data[t]);
        if ( retval )
            handle_error( retval, "pthread_create");
    }

    /* Join all threads and print sum */
    for ( t = 0 ; t < num_threads; t++) {
        pthread_join(threads[t], (void**) NULL);
    }
    printf("The array total is %8.2f\n", total);

    /* Free all memory allocated to program */
    free ( threads );
    free ( thread_data );
    free ( a_vector );
    free ( b_vector );

    return 0;
}
```

Example 3

We fix the "broken" program that estimates the value of π. The problem is that we need to protect the unprotected update to the total that we had in that program to eliminate the race condition. We use a mutex to do this. We need to modify The code follows.

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

static double total = 0;       /* The shared variable updated by threads */
pthread_mutex_t update_mutex;  /* Declare a global mutex */

typedef struct _task_data
{
    int first;            /* index of first element for task */
    int last;             /* index of last element for task */
    int num_segments;     /* total number of segments to be calculated */
    double *sum;          /* pointer to total updated by all threads */
    pthread_t thread_id;  /* id returned by pthread_create()   */
    int task_num;         /* program's thread id     */
} task_data;

/* Print usage statement */
void usage(char *s)
{
        char *p = strrchr(s, '/');
        fprintf(stderr,
                "usage: %s num_intervals  numthreads \n", p ? p + 1 : s);
}

void* approximate_pi ( void  *thread_data )
{
    double dx, x;
    task_data *t_data;
    int   k;
    t_data     = (task_data*) thread_data;

    /* Set dx to the width of each segment */
    dx = 1.0 / (double) t_data->num_segments;

    for ( k = t_data->first; k <= t_data->last; k++ ) {
        x = dx * ((double)k - 0.5); /* x is midpoint of segment i */
        double y = 4.0 / (1.0 + x*x);
        pthread_mutex_lock (&update_mutex);     /* lock mutex    */
        *(t_data->sum) += y;    /* add new area to sum */
        pthread_mutex_unlock (&update_mutex);   /* unlock mutex    */
    }
    pthread_exit((void*) 0);
}


int main( int argc, char *argv[] )
{
    int     num_intervals;  /* number of segments to sum    */
    int     num_threads;    /* number of threads this program will use */
    int     retval;
    int     t;
    double dx;
    task_data      *thread_data;  /* dynamically allocated array of thread data */
    pthread_attr_t attr;

    /* Make all threads joinable */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    if ( argc < 3 ) {
        usage(argv[0]);
        exit(1);
    }

    num_intervals  = atoi(argv[1]);
    num_threads    = atoi(argv[2]);
    if ( (0 == num_intervals ) || ( 0 == num_threads )) {
        printf("ERROR; insufficient memory\n");
        exit(1);
    }

    /* Set dx to the width of each segment */
    dx = 1.0 / (double) num_intervals;



    /* Allocate the array of task_data structures on the heap.
        This is necessary because the array is not global.   */
    thread_data = calloc( num_threads, sizeof(task_data));

    if  ( thread_data == NULL  )
        exit(1);

    /* Initialize the mutex */
    pthread_mutex_init(&update_mutex, NULL);

    /* Initialize task_data for each thread and then create the thread */
    for ( t = 0 ; t < num_threads; t++) {
        thread_data[t].first     = (t*num_intervals)/num_threads;
        thread_data[t].last      = ((t+1)*num_intervals)/num_threads -1;
        thread_data[t].task_num  = t;
        thread_data[t].sum       = &total; /* sum points to total */
        thread_data[t].num_segments  = num_intervals;

        retval = pthread_create(&(thread_data[t].thread_id), &attr,
                                 approximate_pi, (void *) &thread_data[t]);
        if ( retval ) {
            printf("ERROR; return code from pthread_create() is %d\n", retval);
            exit(-1);
        }
    }

   /* Join all threads so that we can add up their partial sums */
    for ( t = 0 ; t < num_threads; t++) {
        pthread_join(thread_data[t].thread_id, (void**) NULL);
    }

    total = total*dx;
    printf("pi is approximated to be %.16f. The error is %.16f\n",
    total, fabs(total - M_PI));
    fflush(stdout);

    pthread_mutex_destroy(&update_mutex);
    /* Free all memory allocated to program */
    free ( thread_data );
    return 0;
}
```

This program has no race conditions. On the other hand, because the critical section

```
    *(t_data->sum) += y;
```

protected by the locking and unlocking of the mutex provides the mutual exclusion, it implies that if many threads are at the same point in their code, many might get blocked waiting for the mutex to be unlocked and this can reduce performance.

To make this clear, suppose that there are 10,000 intervals and 20 threads, each computing the area of 500 rectangles. In the above solution, the critical section is executed a total of 10,000 times, once for each rectangle, and these updates must take place sequentially. From a performance standpoint, this is not very efficient.

Is there another solution that can improve performance?

There is. Using this example, suppose that we revisit the very first solution and let the threads produce their partial sums in the `sum` member variable passed to each thread on its stack:

```
t_data->sum += 4.0 / (1.0 + x*x);
```

Recall that there was an array of `thread_data` structs, one for each thread, and `t_data` was a cast of that structure in the `approximate_pi()` function:

```
t_data    = (task_data*) thread_data;
```

The `main` program had a loop to add the partial sums and store them in total:

```
/* Collect partial sums into a final total */
total = 0;
for ( t = 0 ; t < num_threads; t++)
    total += thread_data[t].sum;
```

With 10,000 intervals and 20 threads, the running time for adding up the areas is 500 local sums followed by a loop of 20 sums, which was executed in mutual exclusion because only the main thread performed the final summation.

What if, instead of the main program doing this, we could let the threads perform a parallel reduction? Then instead of an $O(p)$ loop to perform the addition in `main()`, we could have a $O(\log p)$ summation performed by the threads. Pthreads does not have a reduction function, but we could implement it. The problem is that we cannot start the reduction until all threads have finished their tasks, so we need a way to detect this. Soon we will see that Pthreads has a barrier synchronization instruction that will allow us to program this.

10.8.7 Other Types of Mutexes

The type of a mutex is determined by the mutex attribute structure used to initialize it. There are four possible mutex types:

`PTHREAD_MUTEX_NORMAL`

`PTHREAD_MUTEX_ERRORCHECK`

`PTHREAD_MUTEX_RECURSIVE`

`PTHREAD_MUTEX_DEFAULT`

The default type is always `PTHREAD_MUTEX_DEFAULT`, which is usually equal to `PTHREAD_MUTEX_NORMAL`. To set the type of a mutex, use

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

passing a pointer to the `mutexattr` structure and the type to which it should be set. Then you can use this `mutexattr` structure to initialize the mutex.

There is no function that, given a mutex, can determine the type of that mutex. The best one can do is to call

```
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                              int *restrict type);
```

which retrieves the mutex type from a `mutexattr` structure. But, since there is no function that retrieves the `mutexattr` structure of a mutex, if you need to retrieve the type of the mutex, you must access the `mutexattr` structure that was used to initialize the mutex to know the mutex type.

When a normal mutex is accessed incorrectly, undefined behavior or deadlock results, depending on how the erroneous access took place. A thread will deadlock if it attempts to re-lock a mutex that it already holds. But if the mutex type is `PTHREAD_MUTEX_ERRORCHECK`, then error checking takes place instead of deadlock or undefined behavior. Specifically, if a thread attempts to re-lock a mutex that it has already locked, the `EDEADLK` error is returned, and if a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error is also returned.

Recursive mutexes, i.e., those of type `PTHREAD_MUTEX_RECURSIVE`, can be used when threads invoke recursive functions or when for some other reason, they need to repeatedly lock the mutex. Basically, the mutex maintains a counter. When a thread first acquires the lock, the counter is set to one. Unlike a normal mutex, when a recursive mutex is re-locked, rather than deadlocking, the call succeeds and the counter is incremented. This is true regardless of whether it is a call to `pthread_mutex_trylock()` or `pthread_mutex_lock()`. A thread can continue to re-lock the mutex, up to some system-defined number of times. Each call to unlock the mutex by that same thread decrements the counter. When the counter reaches zero, the mutex is unlocked and can be acquired by another thread. Until the counter is zero, all other threads attempting to acquire the lock will be blocked on calls to `pthread_mutex_lock()`. A thread attempting to unlock a recursive mutex that another thread has locked is returned an error. A thread attempting to unlock an unlocked recursive mutex also receives an error.

Listing 10.11 contains an example of a program with a recursive mutex. It does not do anything other than print some diagnostic messages.

Listing 10.11: A program that uses a recursive mutex.

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

#define  NUM_THREADS     5   /* Fixed number of threads */

pthread_mutex_t    mutex;
int                counter = 0;

void bar(int tid);

void foo(int tid)
```

```
{
    pthread_mutex_lock(&mutex);
    printf("Thread %d: In foo(); mutex locked\n",  tid);
    counter++;
    printf("Thread %d: In foo(); counter = %d\n", tid, counter);
    bar(tid);
    pthread_mutex_unlock(&mutex);
    printf("Thread %d: In foo(); mutex unlocked\n", tid);
}

void bar(int tid)
{
    pthread_mutex_lock(&mutex);
    printf("Thread %d: In bar(); mutex locked\n", tid);
    counter = 2*counter;
    printf("Thread %d: In bar(); counter = %d\n", tid, counter);
    pthread_mutex_unlock(&mutex);
    printf("Thread %d: In bar(); mutex unlocked\n", tid);
}

void * thread_routine( void * data )
{
    int t =  (int) data;
    foo(t);
    pthread_exit(NULL);
}

/****************************************************************************
                                    Main Program
****************************************************************************/

int main( int argc, char *argv[])
{
    int          retval;
    int          t;
    pthread_t    threads[NUM_THREADS];
    pthread_mutexattr_t     attr;
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&mutex, &attr);

    /* Initialize task_data for each thread and then create the thread */
    for ( t = 0 ; t < NUM_THREADS; t++) {
        if ( 0 != pthread_create(&threads[t], NULL, thread_routine,
                                 (void *) t)  ) {
            perror("Creating thread");
            exit(EXIT_FAILURE);
        }
    }

    for ( t = 0 ; t < NUM_THREADS; t++)
        pthread_join(threads[t], (void**) NULL);

    return 0;
}
```

10.9 Condition Variables

Mutexes are not sufficient to solve all synchronization problems efficiently. One problem is that they do not provide a means for one thread to signal another[3]. Consider the classical *producer-consumer problem*. In this problem, there are one or more "producer" threads that produce data that they place into a shared, finite pool of buffers, and one or more "consumer" threads that consume the data in those buffers. We think of the data as being "consumed" because once it is read, no other thread should be able to read it. Synchronization was needed to ensure that

- different producers did not write into the same buffer,
- producers did not overwrite buffers when no empty buffers were available,
- different consumers did not read from the same buffer, and
- consumers did not try to read from empty buffers.

Suppose that the data chunks are fixed size, each fitting into exactly one buffer, and that the buffer pool has N buffers. A consumer thread needs to be able to retrieve a data chunk from a buffer as long as one available, but if all buffers are empty, it should wait until one is non-empty. Assume the following shared variables:

```
int  in = 0;  /* index of next empty buffer */
int out = 0;  /* index of next full buffer  */
const int NUM_BUFFERS = N; /* number of buffers */
```

A producer essentially executes an infinite loop of the form:

```
producer()
{
    item next_item; /* holds next item produced */

    while (true) {
        /* produce an item and store into next_item */
        next_item = produce_new_item();

        /* keep testing whether buffer pool is full */
        while (((in + 1) % NUM_BUFFERS) == out)
            ; /* do nothing because buffer pool is full */

        /* buffer[in] is not full */
        buffer[in] = next_item;
        in = (in + 1) % NUM_BUFFERS; /* advance in */
    }
}
```

[3]You might think that a call to `pthread_mutex_unlock()` can be used to signal another thread that is waiting on a mutex. This is not the way that a mutex can be used. The specification states that if a thread tries to unlock a mutex that it has not locked, undefined behavior results.

and a consumer executes an infinite loop of the form

```
consumer()
{
    item next_item; /* for storing item retrieved from buffer */

    while (true) {
        /* keep testing whether all buffers are empty */
        while (in == out)
            ; /* do nothing because all buffers are empty */

        /* buffer is not empty  */
        next_item = buffer[out];
        out = (out + 1) % NUM_BUFFERS;

      /* consume the item that was copied into next_item */
      consume_item(next_item);
    }
}
```

In the above code, both producers and consumers execute ***busy-waiting loops*** in which they repeatedly check some condition until it is false. They continuously check whether a buffer is non-empty or empty. This is an inefficient solution that wastes CPU cycles. Therefore, for efficiency, a consumer should block itself if all buffers are empty. Similarly, a producer thread should be able to write a chunk into an empty buffer but block if all buffers are full.

These two conditions, no empty buffers and no full buffers, require that consumers be able to signal producers and vice versa when the a buffer changes state from empty to full and full to non-empty. In short, this type of problem requires that threads have the ability to signal other threads when certain conditions hold.

Condition variables solve this problem. They allow threads to wait for certain conditions to occur and to signal other threads that are waiting for the same or other conditions. Consider a version of the producer-consumer problem with a single producer and a single consumer. The producer thread would need to execute something like the following pseudo-code:

```
1. generate data to store into the buffer
2. try to lock a mutex (blocking if it is locked)
3. if all buffers are full (bufferpool is full)
4.    atomically release the mutex and wait for the condition "bufferpool is not full"
5. when the bufferpool is not full:
6. re-acquire the mutex lock
7. add the data to an empty buffer
8. unlock the mutex
9. signal the consumer that there is data in the bufferpool
```

Steps 4, 5, and 9 involve condition variables. The above pseudo-code would become

```
generate data_chunk to store into the buffer;
pthread_mutex_lock(&buffer_mutex);
if ( bufferpool_is_full() ) {
     pthread_cond_wait(&bufferpool_has_space, &buffer_mutex);
}
add data chunk to buffer;
pthread_mutex_unlock(&buffer_mutex);
pthread_cond_signal(&data_is_available);
```

The logic of the above code is that

1. A producer first locks a mutex to access the shared bufferpool. It may get blocked at this point if the mutex is locked already, but eventually it acquires the lock and advances to the following if-statement.
2. In the if-statement, it then tests whether the boolean predicate "bufferpool_is_full" is true.
3. If so, it blocks itself on the condition variable named `bufferpool_has_space`. Notice that the call to block on a condition variable has a second argument which is a mutex. This is important. Condition variables are only used in conjunction with mutexes. When the thread calls this function, ***the mutex lock is taken away from it***, freeing the lock, and the thread instead gets blocked on the condition variable.
4. Now assume that when a consumer empties a slot in the buffer, it issues a signal on the condition variable `bufferpool_has_space`. When this happens, the producer is woken up and re-acquires the mutex *in a single atomic step*. In other words, the magic of the condition variable is that when a process is blocked on it and is later signaled, it is given back the lock that was taken away from it.
5. The producer thread next adds its data to the buffer, unlocks the mutex, and signals the condition variable `data_is_available`, which is a condition variable on which the consumer might be waiting in case it tried to get data from an empty buffer.

An important observation is that the thread waits on the condition variable `bufferpool_has_space` only within the true-branch of the if-statement. A thread should make the call to `pthread_cond_wait()` only when it has ascertained that the logical condition associated with the condition variable is false (so that it is guaranteed to wait.) It should never call this unconditionally. Put another way, *associated with each condition variable is a programmer-defined boolean predicate that should be evaluated to determine whether a thread should wait on that condition.*

We now turn to the programming details.

10.9.1 Creating and Destroying Condition Variables

A condition variable is a variable of type `pthread_cond_t`. Condition variable initialization is similar to mutex initialization. There are two ways to initialize a condition variable:

1. Statically, when it is declared, using the `PTHREAD_COND_INITIALIZER` macro, as in

   ```
   pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
   ```

2. Dynamically, with the `pthread_cond_init()` routine:

   ```
   int pthread_cond_init(pthread_cond_t *restrict cond,
                         const pthread_condattr_t *restrict attr);
   ```

   - This function is given a pointer to a condition variable and to a condition attribute structure, and initializes the condition variable to have the properties of that structure. If the `attr` argument is `NULL`, the condition is given the default properties. Attempting to initialize an already initialized condition variable results in undefined behavior.
   - The call

     ```
     pthread_cond_init(&cond, NULL);
     ```

     is equivalent to the static method except that no error-checking is done.
   - On success, `pthread_cond_init()` returns zero.

Because the condition variable must be accessed by multiple threads, it should either be global or it should be passed by address into each thread's thread function. In either case, the main thread should create it.

To destroy the condition variable, use

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

The `pthread_cond_destroy()` function destroys the given condition variable `cond` after which it becomes, in effect, uninitialized. A thread can only destroy an initialized condition variable if no threads are currently blocked on it. Attempting to destroy a condition variable on which other threads are currently blocked results in undefined behavior.

10.9.2 Waiting on Conditions

There are two functions that a thread can call to wait on a condition, an ***untimed wait*** and a ***timed wait***:

```
int pthread_cond_wait     (pthread_cond_t  *restrict cond,
                            pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t  *restrict cond,
                            pthread_mutex_t *restrict mutex,
                            const struct timespec *restrict abstime);
```

The meaning of the `restrict` qualifier will be explained shortly. Before a thread calls either of these functions, it must first lock the `mutex` argument, otherwise the effect of the call is undefined. Calling either function causes the following two actions to take place atomically:

1. `mutex` is released, and
2. the thread is blocked on the condition variable `cond`.

In the case of the untimed `pthread_cond_wait()`, the calling thread remains blocked in this call until some other thread signals `cond` using either of the two signaling functions described in Section 10.9.3 below. The signal wakes up the blocked thread and the call returns with the value zero, with `mutex` locked and owned by the now-unblocked thread.

In the case of `pthread_cond_timedwait()`, the calling thread remains blocked in this call until

- either some other thread signals `cond`, or
- the absolute time specified by `abstime` is passed.

In either case the effect is the same as that of `pthread_cond_wait()`, but if the time specified by `abstime` is passed first, the call returns with the error `ETIMEDOUT`, otherwise it returns zero.

The `restrict` Qualifier in C

> The precise definition of the `restrict` qualifier is complex. Simply put, when a pointer variable is *restrict-qualified* (it has the `restrict` qualifier), the object that it points to can only be accessed through that pointer in that program; there cannot be another way to access the object. This allows a compiler to optimize code more efficiently.

Condition variables hold no state; they have no record of how many signals have been received at any given time. Therefore, if a thread T_1 signals a condition `cond` before another thread T_2 issues a wait on `cond`, thread T_2 will still wait on `cond` because the signal will have been lost; it is not saved. Only a signal that arrives after a thread has called one of the wait functions can wake up that waiting thread. This is why we need to clarify the sense in which `pthread_cond_wait()` is atomic.

When a thread T_1 calls `pthread_cond_wait()`, as in

```
pthread_cond_wait(&buf_cond, &buf_mutex);
```

the mutex `buf_mutex` is unlocked and then the thread is blocked on the condition variable `buf_cond`. It is possible for another thread T_2 to acquire the mutex after thread T_1 has released it, but before T_1 is blocked on the condition. If a third thread T_3 signals this condition variable after this mutex has been acquired by T_2, then thread T_1 will respond to the signal as if it had taken place after it had been blocked. This means that it will re-acquire the mutex from T_2 as soon as it can and its call to `pthread_cond_wait(&buf_cond, &buf_mutex)` will return.

The fact that a thread returns from a wait on a condition variable does not imply anything about the boolean predicate associated with this condition variable. It might be true or false. This might occur for reasons such as the following:

- The blocked thread is awakened due to a signal delivered to it because of a programming error.

- Several threads are waiting for the same signal and they take turns acquiring the mutex, in which case any one of them can then modify the condition they all waited for.
- A Pthreads library implementation is permitted to issue *spurious wakeups* to a waiting thread without violating the Pthreads standard. This can occur when Pthreads programs are run on multi-processors.

Therefore, calls to wait on condition variables should be inside a loop, not in a simple if-statement. For example, the above producer code should be written as

```
generate data_chunk to store into the buffer;
pthread_mutex_lock(&buffer_mutex);
while ( bufferpool_is_full() ) {
     pthread_cond_wait(&bufferpool_has_space, &buffer_mutex);
}
add data chunk to buffer;
pthread_mutex_unlock(&buffer_mutex);
pthread_cond_signal(&data_is_available);
```

It is in general safer to code with a loop rather than an if-statement, because if you made a logic error elsewhere in your code and it is possible that a thread can be signaled even though the associated predicate is not true, then the loop prevents the thread from being woken up erroneously.

10.9.3 Waking Threads Blocked on Conditions

A thread can send a signal on a condition variable in one of two ways:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Both of these functions unblock threads that are blocked on a condition variable. The difference is that `pthread_cond_signal()` unblocks (at least) *one* of the threads that are blocked on the condition variable whereas `pthread_cond_broadcast()` unblocks *all* threads blocked by the condition variable. Under normal circumstances, `pthread_cond_signal()` will unblock a single thread, but implementations of this function may at times wake up more than one, if more than one are waiting. As noted above, this is allowed by the POSIX standard; these are the spurious wake-ups mentioned above. Both functions return zero on success or an error code on failure.

Other points to remember about these two functions include:

- When multiple threads blocked on a condition variable are all unblocked by a broadcast, the order in which they are unblocked depends upon the scheduling policy. As noted in Section 10.9.2 above, when they become unblocked, they re-acquire the mutex associated with the condition variable. Therefore, the order in which they re-acquire the mutex is dependent on the scheduling policy. Only one can hold the mutex at a given time of course.

- Although any thread can call `pthread_cond_signal()` or `pthread_cond_broadcast()` on a condition variable `cond`, only a thread that has locked the mutex associated with the condition variable `cond` should make this call, otherwise the scheduling of threads will be unpredictable, even knowing the scheduling policy.

The `pthread_cond_broadcast()` function has several applications. One use is when a producer thread produces many items and stores them in successive buffer locations, but consumers only remove one at a time for processing. When the buffer is empty and consumers are all waiting on a condition variable such as data_available, a producer that fills many buffer locations all at once can call `pthread_cond_broadcast()` to wake up the waiting consumers. Another application is to implement a form of barrier synchronization, which is described in Section 10.10 below.

Listing 10.12: Example using `pthread_cond_broadcast`

```
#define   NUM_THREADS         8
#define   NUM_ITERATIONS      4

pthread_mutex_t   mutex;
pthread_cond_t    ready   = PTHREAD_COND_INITIALIZER;

pthread_mutex_t   count_mutex;
int               client_count;

/*****************************************************************
                          Thread Functions
*****************************************************************/

void *client( void * data)
{
    int i;
    pthread_t id =  (pthread_t) data ;

    for (i = 1; i <= NUM_ITERATIONS; i++ ) {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&ready, &mutex);
        pthread_mutex_unlock(&mutex);
        usleep(200000);  /* delay a bit to simulate work */
    }

    pthread_mutex_lock(&count_mutex);
    client_count--;
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
}

void *server( void * data)
{
    while ( 1 ) {
        pthread_mutex_lock(&count_mutex);
        int temp = client_count;
        pthread_mutex_unlock(&count_mutex);
        if ( temp > 0 ) {
            pthread_cond_broadcast(&ready);
        }
        else
            break;

        usleep(1000000); /* delay a bit to simulate work */
    }
    pthread_exit(NULL);
}

/*****************************************************************************
                                  Main Program
*****************************************************************************/

int main(int argc, char* argv[])
{
    long        t;
    pthread_t   threads[NUM_THREADS];
    pthread_t   server_id;
    pthread_mutexattr_t    attr;


    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK);
    pthread_mutex_init(&mutex, &attr);
    pthread_mutex_init(&count_mutex, NULL);

    /* Create the threads that will acquire a mutex and then wait on the
       condition variable. */

    client_count = NUM_THREADS;
    for ( t = 0 ; t < NUM_THREADS; t++)
        pthread_create(&threads[t], NULL, client, (void*) t );

    /* Create the thread that will broadcast on the condition variable */
    pthread_create(&server_id, NULL, server, NULL);

    /* Main thread waits for the others to exit. */
    for ( t = 0 ; t < NUM_THREADS; t++)
        pthread_join(threads[t], (void**) NULL);
    pthread_join(server_id, (void**) NULL);

    /* Clean-up */
    pthread_mutexattr_destroy(&attr);
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&ready);
    pthread_exit (NULL);
}
```

10.9.4 Condition Attributes

The only attributes that conditions have are the process-shared attribute and the clock attribute. These are advanced topics that are not covered here. There are several functions related to condition attributes, specifically the getting and setting of these properties, and they are described by the respective man pages:

```
int pthread_condattr_destroy   ( pthread_condattr_t *attr);
int pthread_condattr_init      ( pthread_condattr_t *attr);
int pthread_condattr_getclock  ( const pthread_condattr_t *restrict attr,
                                 clockid_t *restrict clock_id);
int pthread_condattr_setclock  ( pthread_condattr_t *attr,
                                 clockid_t clock_id);
int pthread_condattr_getpshared( const pthread_condattr_t *restrict attr,
                                 int *restrict pshared);
int pthread_condattr_setpshared( pthread_condattr_t *attr,
                                 int pshared);
```

10.9.5 Example

Listing 10.13 contains a multi-threaded solution to the single-producer/single-consumer problem that uses a mutex and two condition variables. For simplicity, it is designed to terminate after a fixed number of iterations of each thread. It sends output messages to a file named `prodcons_mssges` in the working directory. The buffer routines add a single integer and remove a single integer from a shared global buffer. The calls to these functions in the producer and consumer are within the region protected by the mutex `buffer_mutex`.

The consumer logic is a bit more complex because the producer may exit when the buffer is empty. Therefore, the consumer thread has to check whether the producer is still alive before it blocks itself on the condition `data_available`, otherwise it will hang forever without terminating, and so will `main()`.

It is not enough for the producer to set the flag `producer_exists` to zero when it exits, because the consumer might check its value just prior to the producer's setting it to zero, and seeing `producer_exists` == 1, block itself on the `data_available` condition. That is why the producer executes the lines

```
pthread_mutex_lock(&buffer_mutex);
producer_exists = 0;
pthread_cond_signal(&data_available);
pthread_mutex_unlock(&buffer_mutex);
```

when it exits. It first locks the `buffer_mutex`. If the consumer holds the lock, it will block until the consumer releases the lock. This implies that either the consumer has just acquired the mutex and is about to block itself on the `data_available` condition or that it is getting data from the buffer and will unlock the mutex soon. In either case, the consumer will release the lock and the producer will set `producer_exists` to zero and then signal `data_available`. If the consumer blocked itself on `data_available`, then the signal will wake it up, it will see that `producer_exists` is zero, and it will exit. If it was getting data from the buffer and then released the mutex lock, after which the producer acquired it, then when it gets it again, `producer_exists` will be zero, and it will exit if the buffer is empty.

Listing 10.13: Single-producer/single-consumer multithreaded program.

```
#include <sys/time.h>
#include <sys/types.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <errno.h>

/****************************************************************************
                                   Global, Shared Data
****************************************************************************/

#define   NUM_ITERATIONS   500   /* number of loops each thread iterates */
#define   BUFFER_SIZE      20    /* size of buffer */

/* buffer_mutex controls buffer access */
pthread_mutex_t  buffer_mutex     = PTHREAD_MUTEX_INITIALIZER;

/*  space_available is a condition that is true when the buffer is not full */
pthread_cond_t   space_available  = PTHREAD_COND_INITIALIZER;

/*  data_available is a condition that is true when the buffer is not empty */
pthread_cond_t   data_available   = PTHREAD_COND_INITIALIZER;

int       producer_exists;   /* true when producer is still running */
FILE      *fp;               /* log file pointer for messages       */

/****************************************************************************
                                   Buffer Object
****************************************************************************/

int buffer[BUFFER_SIZE]; /* the buffer of data -- just ints here */
int bufsize;             /* number of filled slots in buffer */

void add_buffer(int data)
{
    static  int  rear = 0;
    buffer[rear] = data;
    rear = (rear + 1) % BUFFER_SIZE;
    bufsize++;
}

int get_buffer()
{
    static  int  front = 0;
    int i;
    i = buffer[front];
    front= (front + 1) % BUFFER_SIZE;
    bufsize--;
    return i ;
}

/*****************************************************************************
                              Error Handling Function
*****************************************************************************/

void handle_error(int num, char *mssge)
{
    errno = num;
    perror(mssge);
    exit(EXIT_FAILURE);
}

/****************************************************************************
                                Thread Functions
*****************************************************************************/

void *producer( void * data)
{
    int i;
    for (i = 1; i <= NUM_ITERATIONS; i++ ) {
        pthread_mutex_lock(&buffer_mutex);
        while ( BUFFER_SIZE == bufsize ) {
            pthread_cond_wait(&space_available,&buffer_mutex);
        }
        add_buffer(i);
        fprintf(fp,"Producer added %d to buffer; buffer size = %d.\n",
                i, bufsize);
        pthread_cond_signal(&data_available);
        pthread_mutex_unlock(&buffer_mutex);
    }

    pthread_mutex_lock(&buffer_mutex);
    producer_exists = 0;
    pthread_cond_signal(&data_available);
    pthread_mutex_unlock(&buffer_mutex);

    pthread_exit(NULL);
}


void *consumer( void * data )
{
    int i;
    for (i = 1; i <= NUM_ITERATIONS; i++ ) {
        pthread_mutex_lock(&buffer_mutex);
        while ( 0 == bufsize ) {
            if ( producer_exists ) {
                pthread_cond_wait(&data_available,&buffer_mutex);
            }
            else {
                pthread_mutex_unlock(&buffer_mutex);
                pthread_exit(NULL);
            }
        }
        i = get_buffer();
        fprintf(fp,"Consumer got data element %d; buffer size = %d.\n",
                i, bufsize);
        pthread_cond_signal(&space_available);
        pthread_mutex_unlock(&buffer_mutex);
    }
    pthread_exit(NULL);
}

/*****************************************************************************
                                               Main Program
*****************************************************************************/

int main(int argc, char* argv[])
{
    pthread_t producer_thread;
    pthread_t consumer_thread;

    producer_exists = 1;
    bufsize = 0;

    if ( NULL == (fp = fopen("./prodcons_mssges", "w")) )
        handle_error(errno, "prodcons_mssges");

    pthread_create(&consumer_thread, NULL, consumer, NULL);
    pthread_create(&producer_thread, NULL, producer, NULL);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    fclose(fp);
    return 0;
}
```

10.10 Barrier Synchronization

10.10.1 Motivation

Some types of parallel programs require that the individual threads or processes proceed in a lockstep manner, each performing a task in a given phase and then waiting for all other threads to complete their tasks before continuing to the next phase. This is typically due to mutual dependencies on the data written during the previous phase by the threads. Many simulations have this property. One simple example is a multithreaded version of Conway's *Game of Life.*

The *Game of Life* simulates the growth of a colony of organisms over time. Imagine a finite, two-dimensional grid in which each cell represents an organism. Time advances in discrete time steps,

t_0, t_1, t_2, ad infinitum. Whether or not an organism survives in cell (i, j) at time t_{k+1} depends on how many organisms are living in the adjacent surrounding cells at time t_k. Whether or not an organism is born into an empty cell (i, j) is also determined by the state of the adjacent cells at the given time. The exact rules are not relevant.

A simple method of simulating the progression of states of the grid is to create a unique thread to simulate each individual cell, and to create two grids, `A` and `B`, of the same dimensions. The initial state of the population is assigned to grid `A`. At each time step t_k, the thread responsible for cell (i, j) would perform the following task:

1. For cell `A[i,j]`, examine the states of each of its eight neighboring cells `A[m,n]` and set the value of `B[i,j]` accordingly.
2. When all other cells have finished their step 1, copy `B[i,j]` to `A[i,j]`, and repeat steps 1 and 2.

Notice that this solution requires that each cell wait for all other cells to reach the same point in the code. This could be achieved with a combination of mutexes and condition variables. The main program would initialize the value of a counter variable, `count`, to zero. Assuming there are `N` threads, each would execute a loop of the form

```
loop forever {
    update cell (i,j);

    pthread_mutex_lock (&update_mutex);
    count++;
    while ( count < N )
        pthread_cond_wait(&all_threads_ready,&update_mutex);
    /* count reached N so all threads proceed */
    pthread_cond_broadcast( &all_threads_ready);
    count --;
    pthread_mutex_unlock (&update_mutex);
    pthread_mutex_lock (&count_mutex);
    while ( count > 0 )
        pthread_cond_wait(&all_threads_at_start, &count_mutex);
    pthread_cond_broadcast( &all_threads_at_start);
    pthread_mutex_unlock (&count_mutex);
}
```

After each thread updates its cell, it tries to acquire a mutex named `update_mutex`. The cell that acquires the mutex increments `count` and then waits on a condition variable named `all_threads_ready` associated with the predicate `count < N`. As it releases `update_mutex`, the next thread does the same, and so on until all but one thread has been blocked on the condition variable. Eventually the `N`*th* thread acquires the mutex, increments `count` and, finding `count == N`, issues a broadcast on `all_threads_ready`, unblocking all of the waiting threads, one by one.

One by one, each thread then decrements `count`. If each were allowed to cycle back to the top of the loop, this code would not work, because one thread could quickly speed around, increment `count` so that it equaled `N` again even though the others had not even started their updates. Instead, no thread is allowed to go back to the top of the loop until `count` reaches zero. This is achieved by using a second condition variable, `all_threads_at_start`. All threads will block on this condition except the one that sets the value of `count` to zero when it decrements it. When that happens, every thread is unblocked and they all start this cycle all over again.

Now as you can see, this adds so much serial code to the parallel algorithm that it defeats the purpose of using multiple threads in the first place. In addition, it ignores the possibility of spurious wake-ups and would be even more complex if these were taken into account. Fortunately, there is a simpler solution; the Pthread library has a ***barrier synchronization*** primitive that solves this synchronization problem efficiently and elegantly.

A ***barrier synchronization point*** is an instruction in a program at which the executing thread must wait until all participating threads have reached that same point. If you have ever been in a guided group of people being taken on a tour of a facility or an institution of some kind, then you might have experienced this type of synchronization. The guide will wait for all members of the group to reach a certain point, and only then will he or she allow the group to move to the next set of locations.

10.10.2 PThreads Barriers

The Pthreads implementation of a barrier lets the programmer initialize the barrier to the number of threads that must reach the barrier in order for it to be opened. A barrier is declared as a variable of type `pthread_barrier_t`. The function to initialize a barrier is

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
       const pthread_barrierattr_t *restrict attr, unsigned count);
```

It is given the address of a barrier, the address of a barrier attribute structure, which may be `NULL` to use the default attributes, and a *positive* value `count`. The `count` argument specifies the number of threads that must reach the barrier before any of them successfully return from the call. If the function succeeds it returns zero.

A thread calls

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

to wait at the barrier given by the argument. When the required number of threads have called `pthread_barrier_wait()` specifying the barrier, the constant `PTHREAD_BARRIER_SERIAL_THREAD` is returned to exactly one unspecified thread and zero is returned to each of the remaining threads. At this point, the barrier is reset to the state it had as a result of the most recent `pthread_barrier_init()` function that referenced it. Some programs may not need to take advantage of the fact that a single thread received the value `PTHREAD_BARRIER_SERIAL_THREAD`, but others may find it useful, particularly if exactly one thread has to perform a task when the barrier has been reached. One can check for errors at the barrier with the code

```
retval = pthread_barrier_wait(&barrier);
if ( PTHREAD_BARRIER_SERIAL_THREAD != retval &&  0 != retval )
    pthread_exit((void*) 0);
```

which will force a thread to exit if it did not get one of the non-error values.

Finally, a barrier is destroyed using

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

which destroys the barrier and releases any resources used by it. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to `pthread_barrier_init()`. The results are undefined if `pthread_barrier_destroy()` is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

10.10.3 Example

Consider the problem of adding the elements of an array of N numbers, where N is extremely large. The serial algorithm would take $O(N)$ steps. Suppose that a processor has P subprocessors and that we want to use P threads to reduce the total running time of the problem. Assume for simplicity that N is a multiple of P. We can decompose the array into P segments of N/P elements each and let each thread sum its set of N/P numbers. But then how can we collect the partial sums calculated by the threads?

Let us create an array, `sums`, of length P. The partial sum computed by thread `k` is stored in `sums[k]`. To compute the sum of all numbers, we let the main program add the numbers in the sums array and store the result in `sums[0]`. In other words, we could execute a loop of the form

```
for ( i = 1; i < P; i++)
    sums[0] += sums[i];
```

This would run in time proportional to the number of threads. Alternatively, we could have each thread add its partial sum directly to a single accumulator, but we would need to serialize this by enclosing it in a critical section. The performance is the same, since there would still be P sequential additions.

Another solution is to use a ***parallel reduction algorithm*** to add the partial sums. In Chapter 3 we introduced parallel reduction. Given a set of n values $a_0, a_1, a_2, ..., a_{n-1}$, and any associative binary operator $\oplus$, ***reduction*** is the process of computing $a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1}$. A *parallel reduction algorithm* is a binary divide-and-conquer solution. We have been using parallel reductions based on a binomial tree communication pattern. We will do the same here. Before the reduction can begin, each thread must have computed its partial sum. Therefore, every thread must wait at a barrier until all other threads have also computed their partial sums. At this point the parallel reduction algorithm proceeds in stages.

Parallel Reduction Algorithm

> The set of thread ids is divided in half. Every thread in the lower half has a *mate* in the upper half, except possibly one odd thread. For example, if there are 100 threads, then thread 0 is mated to thread 50, thread 1 to thread 51, and so on, and thread 49 to thread 99. In each stage, each thread in the lower half of the set adds its mate's sum to its own. At the end of each stage, the upper half of threads is no longer needed, so

the set is cut in half. The lower half becomes the new set and the process is repeated. For example, there would be 50 threads numbered 0 to 49, with threads 0 through 24 forming the lower half and threads 25 to 49 in the upper half. As this happens, the partial sums are being accumulated closer and closer to `sums[0]`.

Eventually the set becomes size 2, and thread 0 adds `sums[0]` and `sums[1]` into `sums[0]`, which is the sum of all array elements. This approach takes $O(\log(P))$ steps. The entire running time is thus $O((N/P)+\log(P))$.

Listing 10.14 contains the code.

Listing 10.14: Reduction algorithm with barrier synchronization.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libintl.h>
#include <locale.h>
#include <math.h>

/*****************************************************************************
                              Data Types and Constants
*****************************************************************************/

double      *sum;        /* array of partial sums of data          */
double      *array;      /* dynamically allocated array of data    */
int         num_threads; /* number of threads this program will use */
pthread_barrier_t barrier;

/*
   a task_data structure contains the data required for a thread to compute
   the sum of the segment of the array it has been delegated to total, storing
   the sum in its cell in an array of sums. The data array and the sum array
   are allocated on the heap. The threads get the starting addresses of each,
   and their task number and the first and last entries of their segments.
*/
typedef struct _task_data
{
    int first;      /* index of first element for task */
    int last;       /* index of last element for task */
    int task_id;    /* id of thread */
} task_data;


/*****************************************************************************
                           Thread and Helper Functions
*****************************************************************************/

/* Print usage statement */
void usage(char *s)
{
        char *p = strrchr(s, '/');
        fprintf(stderr,
                 "usage: %s arraysize  numthreads \n", p ? p + 1 : s);
}

/**
   The thread routine.
*/
void  *add_array( void * thread_data )
{
    task_data *t_data;
    int    k;
    int    tid;
    int    half;
    int    retval;

    t_data  = (task_data*) thread_data;
    tid     = t_data->task_id;

    sum[tid] = 0;
    for ( k = t_data->first; k <= t_data->last; k++ )
        sum[tid] += array[k];

    half = num_threads;
    while ( half > 1 ) {
        retval = pthread_barrier_wait(&barrier);
        if ( PTHREAD_BARRIER_SERIAL_THREAD != retval &&
             0 != retval )
           pthread_exit((void*) 0);

        if ( half % 2 == 1 && tid == 0 )
            sum[0] = sum[0] + sum[half-1];
        half = half/2; // integer division
        if ( tid < half )
            sum[tid] = sum[tid] + sum[tid+half];
    }

    pthread_exit((void*) 0);

}

/*****************************************************************************
                                     Main Program
*****************************************************************************/
int main( int argc, char *argv[])
{
    int          array_size;
    int          size;
    int          k;
    int          retval;
    int          t;
    pthread_t   *threads;
    task_data   *thread_data;
    pthread_attr_t attr;

    /* Instead of assuming that the system creates threads as joinable by
```

```
       default, this sets them to be joinable explicitly.
    */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    if ( argc < 3 ) {
        usage(argv[0]);
        exit(1);
    }

    /* Get command line arguments, convert to ints, and compute size of each
       thread's segment of the array
    */
    errno = 0;
    array_size = strtol(argv[1], '\0', 0);
    if (errno != 0 )
        exit(1);

    errno = 0;
    num_threads = strtol(argv[2], '\0', 0);
    if (errno != 0 )
        exit(1);

    if ( 0 >= num_threads || 0 >= array_size ) {
        usage(argv[0]);
        exit(1);
    }
    size = (int) ceil(array_size*1.0/num_threads);

    /* Allocate the array of threads, task_data structures, data and sums */
    threads     = calloc( num_threads, sizeof(pthread_t));
    thread_data = calloc( num_threads, sizeof(task_data));
    array       = calloc( array_size,  sizeof(double));
    sum         = calloc( num_threads, sizeof(double));

    if ( threads == NULL || thread_data == NULL ||
         array == NULL || sum == NULL )
        exit(1);

    /* Synthesize array data here */
    for ( k = 0 ; k < array_size; k++ )
        array[k] = (double) k;


    /* Initialize a barrier with a count equal to the number of threads */
    pthread_barrier_init(&barrier, NULL, num_threads);

    /* Initialize task_data for each thread and then create the thread */
    for ( t = 0 ; t < num_threads; t++) {
        thread_data[t].first    = t*size;
        thread_data[t].last     = (t+1)*size -1;
        if ( thread_data[t].last > array_size -1 )
            thread_data[t].last  = array_size - 1;
        thread_data[t].task_id  = t;
        retval = pthread_create(&threads[t], &attr, add_array,
                                (void *) &thread_data[t]);
        if ( retval ) {
            printf("ERROR; return code from pthread_create() is %d\n", retval);
            exit(-1);
        }
    }

    /* Join all threads so that we can add up their partial sums */
    for ( t = 0 ; t < num_threads; t++) {
        pthread_join(threads[t], (void**) NULL);
    }

    pthread_barrier_destroy(&barrier);

    printf("The array total is %7.2f\n", sum[0]);

    /* Free all memory allocated to program */
    free ( threads );
    free ( thread_data );
    free ( array );
    free ( sum );

    return 0;
}
```

Although the solution in Listing 10.14 is asymptotically faster than the solution in which the threads add their partial sums to a running total in a critical section, it may not be faster in practice, because the final accumulation of partial sums must wait until all threads have calculated their partial sums. If the number of threads is very large, and there is one very slow thread, then the $\log(P)$ steps will be delayed until the slow thread completes. On the other hand, if the other solution is used, then all threads will have added their partial sums to the total while the slow thread was still working, and when it finishes, a single addition will complete the task. The performance gain of this reduction algorithm depends upon the threads running on symmetric processors.

This same strategy can be used in the pi estimation program. The code is similar to the above.

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

int               num_threads; /* number of threads this program will use */
pthread_barrier_t barrier;
double            *sum;

typedef struct _task_data
{
    int first;          /* index of first element for task */
    int last;             /* index of last element for task */
    int num_segments;     /* total number of segments to be calculated */
    pthread_t thread_id; /* id returned by pthread_create()    */
    int task_num;         /* program's thread id      */
} task_data;

/* Print usage statement */
void usage(char *s)
{
        char *p = strrchr(s, '/');
        fprintf(stderr,
                "usage: %s num_intervals  numthreads \n", p ? p + 1 : s);
}

void* approximate_pi ( void  *thread_data )
{
    double dx, x;
    int half;
    int retval;
    task_data *t_data;
    int   k;
    t_data    = (task_data*) thread_data;

    /* Set dx to the width of each segments */
    dx = 1.0 / (double) t_data->num_segments;

    int tid = t_data->task_num;

    /* Initialize sum for this thread */
    sum[tid] = 0;

    /* Compute partial sum in sum[tid] */
    for ( k = t_data->first; k <= t_data->last; k++ ) {
        x = dx * ((double)k - 0.5); /* x is midpoint of segment i */
        sum[tid] += 4.0 / (1.0 + x*x);    /* add new area to sum */
    }
    sum[tid] =  dx * sum[tid];

    /* this thread has finished computed its partial sum */
    half = num_threads;
    while ( half > 1 ) {
        retval = pthread_barrier_wait(&barrier);
        if ( PTHREAD_BARRIER_SERIAL_THREAD != retval &&
             0 != retval )
           pthread_exit((void*) 0);

        if ( half % 2 == 1 && t_data->task_num == 0 )
            sum[0] = sum[0] + sum[half-1];
        half = half/2;
        if ( t_data->task_num < half )
            sum[tid] = sum[tid] + sum[tid + half];
    }
    pthread_exit((void*) 0)
}

int main( int argc, char *argv[] )
{
    int     num_intervals;  /* number of segments to sum    */
    int     retval;
    int     t;

    task_data      *thread_data;  /* dynamically allocated array of thread data */
    pthread_attr_t attr;

    /* Make all threads joinable */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    if ( argc < 3 ) {
        usage(argv[0]);
        exit(1);
    }

    /* Get command line arguments, convert to ints, and compute size of each
       thread's segment of the array
    */
    num_intervals  = atoi(argv[1]);
    num_threads    = atoi(argv[2]);
    if ( (0 == num_intervals ) || ( 0 == num_threads )) {
        printf("ERROR; insufficient memory\n");
        exit(1);
    }

    /* Allocate the array of task_data structures on the heap.
        This is necessary because the array is not global.    */
    thread_data = calloc( num_threads, sizeof(task_data));

    /* Allocate the sum array */
    sum = calloc( num_threads, sizeof(double));

    /* Initialize a barrier with a count equal to the number of threads */
    pthread_barrier_init(&barrier, NULL, num_threads);

    if  ( thread_data == NULL  )
        exit(1);
    /* Initialize task_data for each thread and then create the thread */
    for ( t = 0 ; t < num_threads; t++) {
        thread_data[t].first       = (t*num_intervals)/num_threads;
        thread_data[t].last        = ((t+1)*num_intervals)/num_threads -1;
        thread_data[t].task_num    = t;
        thread_data[t].num_segments  = num_intervals;

        retval = pthread_create(&(thread_data[t].thread_id), &attr,
                                approximate_pi, (void *) &thread_data[t]);
        if ( retval ) {
            printf("ERROR; return code from pthread_create() is %d\n", retval);
            exit(-1);
        }
    }

   /* Join all threads  */
    for ( t = 0 ; t < num_threads; t++) {
        pthread_join(thread_data[t].thread_id, (void**) NULL);
    }

    printf("pi is approximated to be %.16f. The error is %.16f\n",
   sum[0], fabs(sum[0] - M_PI));

    /* Free all memory allocated to program */
    free ( thread_data );
    free ( sum );
    return 0;
}
```

10.11 Reader/Writer Locks

10.11.1 Introduction

A mutex has the property that it has just two states, locked and unlocked, and only one thread can lock it at a time. For many problems this is fine, but for many others, it is not. Consider a problem in which one thread updates a database of some kind and multiple threads look up information in that database. For example, a web search engine might consist of thousands of “reading” threads that need to read the database of search data to deliver pages of search results to client browsers, and other “writing” threads that crawl the web and update the database with new data. When the database is not being updated, the reading threads should be allowed simultaneous access to the database, but when a writing thread is modifying the database, it needs to do so in mutual exclusion, at least on the parts of it that are changing.

To support this paradigm, POSIX provides *reader/writer locks.* Multiple readers can lock a reader/writer lock without blocking each other, but blocking writers from accessing it, and when a single writer acquires the lock, it obtains exclusive access to the resource; in this case any thread, whether a reader or a writer, will be blocked if it attempts to acquire the lock while a writer holds the lock.

Clearly, reader/writer locks allow for a higher degree of parallelism than does a mutex. Unlike mutexes, they have three possible states:

- locked in read mode,
- locked in write mode, and
- unlocked.

Multiple threads can hold a reader/writer lock in read mode, but only a single thread can hold a reader/writer lock in write mode.

Think of a reader/writer lock as the key to a large room. If the reader/writer lock is not currently held by any thread and a reader acquires it, then it enters the room and leaves a guard at the door. If an arriving thread wants to write, the guard makes it wait on a line outside of the door until the reader leaves the room, or possibly later. All arriving writers will wait on this line while the reader is in the room. If an arriving thread wants to read, whether or not it is let into the room depends on how Pthreads has been configured.

Some systems support a Pthreads option known as the *Thread Execution Scheduling,* or *TES,* option. This option allows the programmer to control how threads are scheduled. If the system does not support this option, and a reader arrives at the door, and there are writers standing in line, it is up to the implementation as to whether the reader must stand at the end of the line, behind the waiting writer(s), or can be allowed to enter the room immediately. If *TES* is supported, then the decision is based on which scheduling policy is in force. If either *FIFO*, *round-robin*, or *sporadic*[4] scheduling is in force, then an arriving reader will stand in line behind all writers (and any readers who have set their priorities higher than the arriving reader's.)

These decisions about who must wait for whom when threads are blocked on a lock can lead to unfair scheduling and even starvation. A detailed discussion of this topic is outside of the scope of this chapter, but you should at least have the understanding that, if the implementation gives arriving readers precedence over writers that are blocked when a reader has the lock, then *a steady stream of readers could prevent a writer from ever writing.* This is not good. Usually, a writer has something important to do, updating information, and it should be given priority over readers. This is why the *TES* option allows this type of behavior, and why some implementations always give waiting writers priority over waiting readers. For this reason, it is also possible that a stream of writers will starve all of the readers, so if for some reason, there must be multiple writers, *the code itself must ensure that they do not starve the readers*, using mutexes and conditions to prevent this possibility.

10.11.2 Using Reader/Writer Locks

It is natural that, as a result of their increased complexity, there are more functions for locking and unlocking reader/writer locks than for manipulating simple mutexes. The prototypes for the functions in the API related to these locks, listed by category, are:

[4]This is also an option to PThreads that may not be available in a given implementation.

Initialization and destruction:

```
int    pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                   const pthread_rwlockattr_t *restrict attr);
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
int    pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

[5]

Locking for reading:

```
int   pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int   pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int   pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                   const struct timespec *restrict abstime);
```

Locking for writing:

```
int   pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int   pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int   pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                   const struct timespec *restrict abstime);
```

Unlocking:

```
int   pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Working with attributes:

```
int   pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int   pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int   pthread_rwlockattr_getpshared(const pthread_rwlockattr_t
                   *restrict attr, int *restrict pshared);
int   pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                    int pshared);
```

As with all of the other locks and synchronization objects described here so far, the first step is to initialize the reader/writer lock. This is done using either the function `pthread_rwlock_init()` or the initializer macro `PTHREAD_RWLOCK_INITIALIZER`, which is equivalent to using `pthread_rwlock_init()` with a `NULL` second argument. There are not many attributes that can be configured; the process-shared attribute is not required to be implemented by a POSIX-compliant system, and there are no others that can be modified. Therefore, it is fine to accept the defaults.

Notice that a thread wishing to use the lock for reading uses a different set of primitives than one that wants to write. For reading, a thread can use `pthread_rwlock_rdlock()`, which has the semantics described in the introduction above. If you do not want the thread to block in those

[5]The `restrict` qualifier in C was introduced in C99 to assist in compiler optimization. It has the following meaning: Objects referenced through a restrict-qualified pointer have a special association with that pointer. All references to that object must directly or indirectly use the value of this pointer. In the absence of this qualifier, other pointers can alias this object.

cases where it might, use `pthread_rwlock_tryrdlock()`, which will return the error value `EBUSY` whenever it would block.

The `pthread_rwlock_timedrdlock()` function is like the `pthread_rwlock_rdlock()` function, except that, if the lock cannot be acquired without blocking, the wait is terminated when the specified timeout expires. The timeout expires when the *absolute time* specified by `abstime` passes, as measured by the real time clock (`CLOCK_REALTIME`) or if the absolute time specified by `abstime` has already been passed at the time of the call. Note that the time specification is not an interval, but what you might call "clock time", as the system perceives it. The `timespec` data type is defined in the `<time.h>` header file. The function does not fail if the lock can be acquired immediately, and the validity of the `abstime` parameter is not checked if the lock can be acquired immediately.

The same statements apply to the three functions for acquiring a writer lock, and so they are not repeated. As for unlocking, there is only one function to unlock. It does not matter whether the thread holds the lock for reading or writing – it calls `pthread_rwlock_unlock()` in either case.

10.11.3 Further Details

This section answers some more subtle, advanced questions about reader/writer locks.

- If the calling thread already holds a shared read lock on the reader/writer lock, another read lock can be successfully acquired by the calling thread. If more than one shared read lock is successfully acquired by a thread on a reader/writer lock, that thread is required to successfully call `pthread_rwlock_unlock()` a matching number of times.
- Some implementations of Pthreads will allow a thread that already holds an exclusive write lock on a reader/writer lock to acquire another write lock on that same lock. In these implementations, if more than one exclusive write lock is successfully acquired by a thread on a reader/writer lock, that thread is required to successfully call `pthread_rwlock_unlock()` a matching number of times. In other implementations, the attempt to acquire a second write lock will cause deadlock.
- If while either of `pthread_rwlock_wrlock()` or `pthread_rwlock_rdlock()` is waiting for the shared read lock, the reader/writer lock is destroyed, then the `EDESTROYED` error is returned.
- If a signal is delivered to the thread while it is waiting for the lock for either reading or writing, if a signal handler is registered for this signal, it runs, and the thread resumes waiting.
- If a thread terminates while holding a write lock, the attempt by another thread to acquire a shared read or exclusive write lock will not succeed. In this case, the attempt to acquire the lock does not return and will deadlock. If a thread terminates while holding a read lock, the system automatically releases the read lock.
- If a thread calls `pthread_rwlock_wrlock()` and currently holds a shared read lock on the reader/writer lock and no other threads are holding a shared read lock, the exclusive write request is granted. After the exclusive write lock request is granted, the calling thread holds both the shared read and the exclusive write lock for the specified reader/writer lock.
- In an implementation in which a thread can hold multiple read and write locks on the same reader/writer lock, if a thread calls `pthread_rwlock_unlock()` while holding one or more

shared read locks and one or more exclusive write locks, the exclusive write locks are unlocked first. If more than one outstanding exclusive write lock was held by the thread, a matching number of successful calls to `pthread_rwlock_unlock()` must be completed before all write locks are unlocked. At that time, subsequent calls to `pthread_rwlock_unlock()` will unlock the shared read locks.

10.11.4 Example

The program in Listing 10.15 demonstrates the use of reader/writer locks. It would be very simple if we did not attempt to prevent starvation, either of readers or writers, but this program uses the GNU extension to the standard, `pthread_rwlockattr_setkind_np()`, which can be used to change the priorities given to readers and writers. The call

```
pthread_rwlockattr_setkind_np(&rwlock_attributes,
                              PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP);
```

sets the reader/writer attribute so that it gives preference to waiting writers, meaning that as long as there is a writer waiting, when the lock becomes available, it will be given to the next waiting writer. In contrast,

```
pthread_rwlockattr_setkind_np(&rwlock_attributes,
                              PTHREAD_RWLOCK_PREFER_READER_NP);
```

sets the reader/writer attribute so that it gives preference to waiting readers.

The program uses barrier synchronization to ensure that no thread enters its main loop until after all threads have been created. Without the barrier, the first threads that would be created in the main program would get the lock first, and if these are writers, the readers would starve.

In the listing below, writers are given preference. This being the case, if the number of writers is changed to be greater than one, they will starve the readers whenever the first writer grabs the lock, because there will always be at least one writer waiting. If the sleep() in the writer code outside of the critical section is lengthened enough, then there is a chance that the readers will not be starved. This program can be used to experiment with the likelihood of starvation.

Listing 10.15: Reader/writer locks: A simple example.

```
#define  _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

/*****************************************************************************
                          Data Types and Constants
*****************************************************************************/

#define            NUM_READERS  10
#define            NUM_WRITERS  1
```

```
pthread_rwlock_t  rwlock;          /* the reader/writer lock */
pthread_barrier_t barrier;         /* to try to improve fairness */

int          done;                 /* to terminate all threads */
int          num_threads_in_lock;  /* for the monitor code */


/*****************************************************************************
                           Thread and Helper Functions
*****************************************************************************/
/** handle_error(num,mssge)
   Prints to standard error the system message associated with error number num
   as well as a custom message, and then exits the program with EXIT_FAILURE
*/
void handle_error(int num, char *mssge)
{
    errno = num;
    perror(mssge);
    exit(EXIT_FAILURE);
}

/** reader()
 *  A reader repeatedly gets the lock, sleeps a bit, and then releases the lock,
 *  until done becomes true.
 */
void *reader(void * data)
{
    int rc;
    int t = (int) data;

    /* Wait here until all threads are created */
    rc = pthread_barrier_wait(&barrier);
    if ( PTHREAD_BARRIER_SERIAL_THREAD != rc &&  0 != rc )
        handle_error( rc, "pthread_barrier_wait");

    /* repeat until user says to quit */
    while ( ! done ) {
        rc = pthread_rwlock_rdlock(&rwlock);
        if ( rc ) handle_error( rc, "pthread_rwlock_rdlock");
        printf("Reader %d got the read lock\n", t);
        sleep(1);
        rc = pthread_rwlock_unlock(&rwlock);
        if ( rc ) handle_error( rc, "pthread_rwlock_unlock");
        sleep(1);
    }
    pthread_exit(NULL);
}

/** writer()
 *  A writer does the same thing as a reader -- it repeatedly gets the lock,
 *  sleeps a bit, and then releases the lock, until done becomes true.
 */
void *writer(void * data)
{
    int rc;
    int t = (int) data;

    /* Wait here until all threads are created */
    rc = pthread_barrier_wait(&barrier);
    if ( PTHREAD_BARRIER_SERIAL_THREAD != rc &&  0 != rc )
        handle_error( rc, "pthread_barrier_wait");

    /* repeat until user says to quit */
    while ( ! done ) {
        rc = pthread_rwlock_wrlock(&rwlock);
        if ( rc ) handle_error( rc, "pthread_rwlock_wrlock");
        printf("Writer %d got the write lock\n", t);
        sleep(2);

        rc = pthread_rwlock_unlock(&rwlock);
        if ( rc ) handle_error( rc, "pthread_rwlock_unlock");
        sleep(2);
    }
    pthread_exit(NULL);
}

/****************************************************************************
                                    Main Program
****************************************************************************/

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_READERS+NUM_WRITERS];
    int retval;
    int t;
    unsigned int num_threads = NUM_READERS+NUM_WRITERS;

    done                = 0;
    printf("This program will start up a number of threads that will run \n"
           "until you enter a character. Type any character to quit\n");

    pthread_rwlockattr_t  rwlock_attributes;
    pthread_rwlockattr_init(&rwlock_attributes);
    /* The following non-portable function is a GNU extension that alters the
       thread priorities when readers and writers are both waiting on a rwlock,
       giving preference to writers.
    */
    pthread_rwlockattr_setkind_np(&rwlock_attributes,
           PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP);
    pthread_rwlock_init(&rwlock, &rwlock_attributes );

    /* Initialize a barrier with a count equal to the numebr of threads */
    retval = pthread_barrier_init(&barrier, NULL, num_threads);
    if ( retval ) handle_error( retval, "pthread_barrier_init");

    for ( t = 0 ; t < NUM_READERS; t++) {
        retval = pthread_create(&threads[t], NULL, reader, (void *)t);
        if ( retval ) handle_error( retval, "pthread_create");
    }

    for ( t = NUM_READERS ; t < NUM_READERS+NUM_WRITERS; t++) {
        retval = pthread_create(&threads[t], NULL, writer, (void *)t);
        if ( retval )  handle_error( retval, "pthread_create");
    }

    getchar();
    done = 1;

    for ( t = 0 ; t < NUM_READERS+NUM_WRITERS; t++)
        pthread_join(threads[t],  NULL);

    return 0;
}
```

Notes.

- The messages printed by the various `printf` statements will not necessarily appear in the order in which `printf` was called by the threads.

10.12 Topics Not Covered

Any serious multi-threaded program must deal with signals and their interactions with threads. The man pages for the various thread-related functions usually have a section on how signals interact with those functions. Spin locks are another synchronization primitive not discussed here; they have limited use. Real-time threads and thread scheduling, where supported, provide the means to control how threads are scheduled for more accurate performance control. Thread keys are a way to create thread-specific data that is visible to all threads in the process.