Chapter 10 Threads

Concepts Covered

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

We saw in Chapter 8 that a process is associated with a set of resources including its memory segments (text, stack, initialized data, uninitialized data), environment variables and command line arguments, and various properties and data that are contained in kernel resources such as the process and user structures. A partial list of the kinds of information contained in these structures includes things such as the process's

- IDs such as process ID, process group ID, user ID, and group ID
- Hardware state
- Memory mappings, such as where process segments are located
- 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.

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. The `fork()` system call duplicates some, but not all, of the calling process's resources. Some of them are shared between the parent and child process.

Processes by default are limited in what they can share with each other because they do not share their memory spaces. Thus, for example, they do not in general share variables and other objects that they create in memory. Most operating systems provide an API for sharing memory though. 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 biggest drawback to using processes as a means of multi-tasking is their consumption of system resources. This was the motivation for the invention of threads.

10.2 Thread Concepts

A *thread* is a flow of control (think sequence of instructions) that can be independently scheduled by the kernel. A typical UNIX process can be thought of as having a single thread of control: each process is doing only one thing at a time. When a program has multiple threads of control, more than one thing at a time can be done within a single process, with each thread handling a separate task. Some of the advantages of this are that

- Code to handle asynchronous events can be executed by a separate thread. Each thread can then handle its event using a synchronous programming model.
- 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, and not the whole process.
- Response time of interactive programs can be improved by splitting off threads to handle user input and output.

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

In UNIX, POSIX requires that each thread will have its own distinct

- thread ID
- stack and an 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
- 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 Programming 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 able to be organized into discrete, independent tasks which can execute concurrently. The first consideration when considering 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 write a parallel program. These notes are not intended to assist you in that task. Their purpose is just to provide the technical background, with pointers here and there. However, before continuing, we present a few common paradigms for organizing multi-threaded programs.

Thread Pool, or Boss/Worker Paradigm

In this approach, there is a single *boss* thread that dispatches threads to perform work. These threads are part of a worker thread pool which is usually pre-allocated before the boss begins dispatching threads.

Peer or WorkCrew Paradigm

In the WorkCrew 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 boss/worker model except that once the worker pool has been created, the boss becomes the another thread in the thread pool, and is thus, a peer to the other threads.

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. You must be careful to equally distribute work and take extra steps to ensure non-blocking behavior in this thread model or you could experience pipeline "stalls."

10.4 Overview of the Pthread Library

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.

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 by 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.

10.5 Thread Management

10.5.1 Creating Threads

We will start with the `pthread_create()` function. The prototype is

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

This function starts a new thread with thread ID `*thread` as part of the calling process. On successful creation of the new thread, `thread` contains its thread ID. Unlike `fork()`, this call passes the address of a function, `start_routine()`, to be executed by the new thread. This "start" function has exactly one argument, of type `void*`, and returns a `void*`. The fourth argument, `arg`, is the argument that will be passed to `start_routine()` 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. We will examine thread properties in more detail later.

Because `start_routine()` has just a single argument, if the function needs access to more than a simple variable, the program should declare a structure with all state that needs to be accessed within the thread, and pass a pointer to that structure. For example, if a set of threads is accessing a shared array and each thread will process a contiguous portion of that array, you might want to define a structure such as

```
typedef struct _task_data
{
    int first;    /* index of first element for task */
    int last;     /* index of last element for task */
    int *array;   /* pointer to start of array */
    int task_id;  /* id of thread */
} task_data;
```

and start each thread with the values of `first`, `last`, and `task_id` initialized. The array pointer may or may not be needed; if the array is a global variable, the threads will have access to it. If it is declared in the main program, then its address can be part of the structure. Suppose that the array is declared as a static local variable named `data_array` in the main program. Then a code fragment to initialize the thread data and create the threads could be

```
task_data  thread_data[NUM_THREADS];
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].array     = &data_array[0];
    thread_data[t].task_id   = t;

    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 `NUM_THREADS` many threads, each executing `process_array`(), each with its own structure containing parameters of its execution.

10.5.1.1 Design Decision Regarding Shared Data

The advantage of declaring the data array as a local variable in the main program is that it makes it easier to analyze and maintain the code when there are fewer global variables and side effects. Programs with functions that modify global variables are harder to analyze. On the other hand, making it a 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 and use good practices in documenting and accessing it. If not, use the static local.

10.5.2 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. 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.

10.5.3 Thread Termination

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

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

This function kills the 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()`[1]. 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. If these threads should not be terminated, then calling `pthread_exit()` from `main()` will ensure that they continue to execute.

[1] Provided that the terminating thread is joinable.

10.5.4 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.

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, making it more like `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 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 example that creates a single thread and waits for it using `pthread_join()`, collecting and printing its exit status.

Listing 10.1: Simple example of thread creation with join

```
int exitval;

void* hello_world( void * world)
{
    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);
    }
    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.

10.5.5 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 waste 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 10.2: 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);
}
```

10.5.6 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 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`.

10.5.7 Thread Properties

10.5.7.1 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 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 threads 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 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 it 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 10.3: 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);
}
```

10.6 Mutexes

10.6.1 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. 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;
```

```
    for ( i = 0; i < NUM_UPDATES; i++ ) {
        *ticker = *ticker + 1;
        /* use the ticker to do stuff here with A[*ticker] */
    }
    pthread_exit( NULL );
}
```

Without any synchronization to force the increment of `*ticker` to be 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.

A *mutex* is one of the provisions of Pthreads for providing mutual exclusive access to critical sections. A *mutex* is like a software version of lock. Its name derives from "mutual exclusion" because a mutex can only be held, or *owned*, by one thread at a time. Like a binary semaphore, 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.

10.6.2 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.

10.6.3 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`.

10.6.4 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.

10.6.5 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()`.

10.6.6 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 10.4. 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 10.4: 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 eacxh 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 10.4 found in Listing 10.5.

The function in Listing 10.5 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 fo 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 10.5: 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 current 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 10.6, 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 10.6: 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;
}
```

10.6.7 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 result, 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. 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 relocked, rather than deadlocking, the call succeeds and the counter is incremented. 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 10.7 contains an example of a program with a recursive mutex. It does not do anything other than print some diagnostic messages.

Listing 10.7: 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;
}
```

10.7 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[2]. 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 buffer, and one or more "consumer" threads that consume the data in that buffer. We think of the data as being consumed because once it is read, no other thread should be able to read it; it is discarded, like the data in a pipe or a socket.

Suppose that the data chunks are fixed size and that the buffer can store N chunks. A consumer thread needs to be able to retrieve a data chunk from the buffer as long as one available, but if the buffer is empty, it should wait until one becomes available. Although it is possible for a consumer to busy-wait in a loop, continuously checking whether the buffer is non-empty, this is an inefficient

[2]In case you are thinking that a call to `pthread_mutex_unlock()` can be used to signal another thread that is waiting on a mutex, recall that 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.

solution that wastes CPU cycles. Therefore, a consumer should block itself if the buffer is empty. Similarly, a producer thread should be able to write a chunk into the buffer if it is not full but otherwise block until a consumer removes a chunk.

These two buffer-full and buffer-empty conditions require that consumers be able to signal producers and vice versa when the buffer changes state from empty to non-empty 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 the 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*
3. *if the buffer is full*
4. *atomically release the mutex and wait for the condition "buffer is not full"*
5. *when the buffer is not full:*
6. *re-acquire the mutex lock*
7. *add the data to the buffer*
8. *unlock the mutex*
9. *signal the consumer that there is data in the buffer*

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 ( buffer_is_full() ) {
     pthread_cond_wait(&buffer_has_space, &buffer_mutex);
}
add data chunk to buffer;
pthread_mutex_unlock(&region_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 buffer. It may get blocked at this point if the mutex is locked already, but eventually it acquires the lock and advances to the if-statement.
2. In the if-statement, it then tests whether the boolean predicate "buffer_is_full" is true.
3. If so, it blocks itself on a condition variable named `buffer_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 `buffer_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 `buffer_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.

10.7.1 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 upon which other threads are currently blocked results in undefined behavior.

10.7.2 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);
```

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 10.7.3 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.

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 calling thread. This is why we need to clarify the sense in which `pthread_cond_wait()` is atomic.

When a thread T calls `pthread_cond_wait()`, the mutex is unlocked and then the thread is blocked on the condition variable. It is possible for another thread to acquire the mutex after thread T has released it, but before it is blocked. If a thread signals this condition variable after this mutex has been acquired by another thread, then thread T 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 as soon as it can and the call 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 is because a thread can return from a call to either of these functions due to a *spurious wakeup*. A spurious wakeup might occur, for example, if a signal is delivered to the blocked thread. It can also occur under certain conditions when a multi-threaded program is running on a multiprocessor. 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 properly be written as

```
generate data_chunk to store into the buffer;
pthread_mutex_lock(&buffer_mutex);
while ( buffer_is_full() ) {
     pthread_cond_wait(&buffer_has_space, &buffer_mutex);
}
add data chunk to buffer;
pthread_mutex_unlock(&region_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.

10.7.3 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 inadvertently wake up more than one, if more than one are waiting. Both 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 10.7.2 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.
- Although any thread can call `pthread_cond_signal(&cond)` or `pthread_cond_broadcast(&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.

10.7.4 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);
```

10.7.5 Example

Listing 10.8 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 was about to block 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 10.8: 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;
}
```

10.8 Barrier Synchronization

10.8.1 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++;
    if ( 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);
    if ( 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.

10.8.2 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.

10.8.3 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 *reduction algorithm* to add the partial sums. A *reduction algorithm* is like a divide-and-conquer solution. Each thread computes its partial sum and then waits at a barrier until all other threads have also computed their partial sums. At this point the algorithm proceeds in stages.

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 10.9 contains the code.

Listing 10.9: 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
    */
    array_size  = atoi(argv[1]);
    num_threads = atoi(argv[2]);
    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 numebr 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 10.9 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.

10.9 Reader/Writer Locks

10.9.1 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; 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 read/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[3] 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 discussion of this topic is really outside of the scope of these notes, but you should at least have the intuition 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

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

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.

10.9.2 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 simple mutexes. The prototypes for the functions in the API related to these locks, listed by category, are:

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);
```

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 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.

10.9.3 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.

10.9.4 Example

The program in Listing 10.10 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. It uses barrier synchronization to ensure that no thread enters its main loop until all threads have at least been created. Without the barrier, the threads that are created first in the main program will always get the lock first, and if these are writers, the readers will starve.

If the number of writers is changed to be greater than one, they will starve the readers whenever the first writer grabs the lock. This is because writers are given priority over readers in the code below.

Listing 10.10: 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;
}
```

10.10 Other 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 thoze 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.