3 Synchronization

"That's right!" said the Tiger-lily. "The daisies are worst of all. When one speaks, they all begin together, and it's enough to make one wither to hear the way they go on!"

—Lewis Carroll, Through the Looking-Glass

To write a program of any complexity using threads, you'll need to share data between threads, or cause various actions to be performed in some coherent order across multiple threads. To do this, you need to synchronize the activity of your threads.

Section 3.1 describes a few of the basic terms we'll be using to talk about thread synchronization: critical section and invariant.

Section 3.2 describes the basic Pthreads synchronization mechanism, the mutex.

Section 3.3 describes the condition variable, a mechanism that your code can use to communicate changes to the state of invariants protected by a mutex.

Section 3.4 completes this chapter on synchronization with some important information about threads and how they view the computer's memory.

3.1 Invariants, critical sections, and predicates

"I know what you're thinking about," said Tweedledum; "but it isn't so, nohow."
"Contrariwise," continued Tweedledee, "If it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

—Lewis Carroll, Through the Looking-Glass

Invariants are assumptions made by a program, especially assumptions about the relationships between sets of variables. When you build a queue package, for example, you need certain data. Each queue has a queue header, which is a pointer to the first queued data element. Each data element includes a pointer to the next data element. But the data isn't all that's important—your queue package relies on relationships between that data. The queue header, for example,
must either be NULL or contain a pointer to the first queued data element. Each data element must contain a pointer to the next data element, or NULL if it is the last. These relationships are the invariants of your queue package.

It is hard to write a program that doesn’t have invariants, though many of them are subtle. When a program encounters a broken invariant, for example, if it dereferences a queue header containing a pointer to something that is not a valid data element, the program will probably produce incorrect results or fail immediately.

Critical sections (also sometimes called “serial regions”) are areas of code that affect a shared state. Since most programmers are trained to think about program functions instead of program data, you may well find it easier to recognize critical sections than data invariants. However, a critical section can almost always be translated into a data invariant, and vice versa. When you remove an element from a queue, for example, you can see the code performing the removal as a critical section, or you can see the state of the queue as an invariant. Which you see first may depend on how you’re thinking about that aspect of your design.

Most invariants can be “broken,” and are routinely broken, during isolated areas of code. The trick is to be sure that broken invariants are always repaired before “unsuspecting” code can encounter them. That is a large part of what “synchronization” is all about in an asynchronous program. Synchronization protects your program from broken invariants. If your code locks a mutex whenever it must (temporarily) break an invariant, then other threads that rely on the invariant, and which also lock the mutex, will be delayed until the mutex is unlocked—when the invariant has been restored.

Synchronization is voluntary, and the participants must cooperate for the system to work. The programmers must agree not to fight for (or against) possession of the bailing bucket. The bucket itself does not somehow magically ensure that one and only one programmer bails at any time. Rather, the bucket is a reliable shared token that, if used properly, can allow the programmers to manage their resources effectively.

“Predicates” are logical expressions that describe the state of invariants needed by your code. In English, predicates can be expressed as statements like “the queue is empty” or “the resource is available.” A predicate may be a boolean variable with a TRUE or FALSE value, or it may be the result of testing whether a pointer is NULL. A predicate may also be a more complicated expression, such as determining whether a counter is greater than some threshold. A predicate may even be a value returned from some function. For example, you might call select or poll to determine whether a file is ready for input.
3.2 Mutexes

"How are you getting on?" said the Cat, as soon as there was mouth enough for it to speak with. Alice waited till the eyes appeared, and then nodded. "It's no use speaking to it," she thought, "till its ears have come, or at least one of them."

—Lewis Carroll, Alice's Adventures in Wonderland

Most threaded programs need to share some data between threads. There may be trouble if two threads try to access shared data at the same time, because one thread may be in the midst of modifying some data invariant while another acts on the data as if it were consistent. This section is all about protecting the program from that sort of trouble.

The most common and general way to synchronize between threads is to ensure that all memory accesses to the same (or related) data are "mutually exclusive." That means that only one thread is allowed to write at a time—others must wait for their turn. Pthreads provides mutual exclusion using a special form of Edsger Dijkstra's semaphore [Dijkstra, 1968a], called a mutex. The word mutex is a clever combination of "mut" from the word "mutual" and "ex" from the word "exclusion."

Experience has shown that it is easier to use mutexes correctly than it is to use other synchronization models such as a more general semaphore. It is also easy to build any synchronization models using mutexes in combination with condition variables (we'll meet them at the next corner, in Section 3.3). Mutexes are simple, flexible, and can be implemented efficiently.

The programmers' bailing bucket is something like a mutex (Figure 3.1). Both are "tokens" that can be handed around, and used to preserve the integrity of the concurrent system. The bucket can be thought of as protecting the bailing critical section—each programmer accepts the responsibility of bailing while holding the bucket, and of avoiding interference with the current bailer while not holding the bucket. Or, the bucket can be thought of as protecting the invariant that water can be removed by only one programmer at any time.

Synchronization isn't important just when you modify data. You also need synchronization when a thread needs to read data that was written by another thread, if the order in which the data was written matters. As we'll see a little later, in Section 3.4, many hardware systems don't guarantee that one processor will see shared memory accesses in the same order as another processor without a "nudge" from software.
Consider, for example, a thread that writes new data to an element in an array, and then updates a `max_index` variable to indicate that the array element is valid. Now consider another thread, running simultaneously on another processor, that steps through the array performing some computation on each valid element. If the second thread "sees" the new value of `max_index` before it sees the new value of the array element, the computation would be incorrect. This may seem irrational, but memory systems that work this way can be substantially faster than memory systems that guarantee predictable ordering of memory accesses. A mutex is one general solution to this sort of problem. If each thread locks a mutex around the section of code that's using shared data, only one thread will be able to enter the section at a time.

Figure 3.2 shows a timing diagram of three threads sharing a mutex. Sections of the lines that are above the rounded box labeled "mutex" show where the associated thread does not own the mutex. Sections of the lines that are below the center line of the box show where the associated thread owns the mutex, and sections of the lines hovering above the center line show where the thread is waiting to own the mutex.

Initially, the mutex is unlocked. Thread 1 locks the mutex and, because there is no contention, it succeeds immediately—thread 1's line moves below the center
of the box. Thread 2 then attempts to lock the mutex and, because the mutex is already locked, thread 2 blocks, its line remaining above the center line. Thread 1 unlocks the mutex, unblocking thread 2, which then succeeds in locking the mutex. Slightly later, thread 3 attempts to lock the mutex, and blocks. Thread 1 calls `pthread_mutex_trylock` to try to lock the mutex and, because the mutex is locked, returns immediately with `EBUSY` status. Thread 2 unlocks the mutex, which unblocks thread 3 so that it can lock the mutex. Finally, thread 3 unlocks the mutex to complete our example.

### 3.2.1 Creating and destroying a mutex

A mutex is represented in your program by a variable of type `pthread_mutex_t`. You should never make a copy of a mutex, because the result of using a copied mutex is undefined. You can, however, freely copy a pointer to a mutex so that various functions and threads can use it for synchronization.
Most of the time you'll probably declare mutexes using extern or static storage class, at 'file scope,' that is, outside of any function. They should have "normal" (extern) storage class if they are used by other files, or static storage class if used only within the file that declares the variable. When you declare a static mutex that has default attributes, you should use the `PTHREAD_MUTEX_INITIALIZER` macro, as shown in the `mutex_static.c` program shown next. (You can build and run this program, but don't expect anything interesting to happen, since main is empty.)

```
#define static.c
1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5 * Declare a structure, with a mutex, statically initialized. This
6 * is the same as using pthread_mutex_init, with the default
7 * attributes.
8 */
9 typedef struct my_struct_tag {
10    pthread_mutex_t mutex; /* Protects access to value */
11    int value; /* Access protected by mutex */
12 } my_struct_t;
13
14 my_struct_t data = {PTHREAD_MUTEX_INITIALIZER, 0};
15
16 int main (int argc, char *argv[])
17 {
18    return 0;
19 }
```

Often you cannot initialize a mutex statically, for example, when you use `malloc` to create a structure that contains a mutex. Then you will need to call `pthread_mutex_init` to initialize the mutex dynamically, as shown in `mutex_dynamic.c`, the next program. You can also dynamically initialize a mutex that you declare statically—but you must ensure that each mutex is initialized before it is used, and that each is initialized only once. You may initialize it before creating any threads, for example, or by calling `pthread_once` (Section 5.1). Also, if you need to initialize a mutex with nondefault attributes, you must use dynamic initialization (see Section 5.2.1).
/* Define a structure, with a mutex.
 */

typedef struct my_struct_tag {
  pthread_mutex_t mutex; /* Protects access to value */
  int value; /* Access protected by mutex */
} my_struct_t;

int main (int argc, char *argv[])
{
  my_struct_t *data;
  int status;

  data = malloc (sizeof (my_struct_t));
  if (data == NULL)
    errno_abort ("Allocate structure");
  status = pthread_mutex_init (&data->mutex, NULL);
  if (status != 0)
    err_abort (status, "Init mutex");
  status = pthread_mutex_destroy (&data->mutex);
  if (status != 0)
    err_abort (status, "Destroy mutex");
  (void)free (data);
  return status;
}

It is a good idea to associate a mutex clearly with the data it protects, if possible, by keeping the definition of the mutex and data together. In mutex_static.c and mutex_dynamic.c, for example, the mutex and the data it protects are defined in the same structure, and line comments document the association.

When you no longer need a mutex that you dynamically initialized by calling pthread_mutex_init, you should destroy the mutex by calling pthread_mutex_destroy. You do not need to destroy a mutex that was statically initialized using the PTHREAD_MUTEX_INITIALIZER macro.

You can destroy a mutex as soon as you are sure no threads are blocked on the mutex.

It is safe to destroy a mutex when you know that no threads can be blocked on the mutex, and no additional threads will try to lock the mutex. The best way to know this is usually within a thread that has just unlocked the mutex, when program logic ensures that no threads will try to lock the mutex later. When a thread locks a mutex within some heap data structure to remove the structure from a list and free the storage, for example, it is safe (and a good idea) to unlock and destroy the mutex before freeing the storage that the mutex occupies.
3.2.2 Locking and unlocking a mutex

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

In the simplest case, using a mutex is easy. You lock the mutex by calling either `pthread_mutex_lock` or `pthread_mutex_trylock`, do something with the shared data, and then unlock the mutex by calling `pthread_mutex_unlock`. To make sure that a thread can read consistent values for a series of variables, you need to lock your mutex around any section of code that reads or writes those variables.

You cannot lock a mutex when the calling thread already has that mutex locked. The result of attempting to do so may be an error return, or it may be a self-deadlock, with the unfortunate thread waiting forever for itself to unlock the mutex. (If you have access to a system supporting the UNIX98 thread extensions, you can create mutexes of various types, including reentrant mutexes, which allow a thread to relock a mutex it already owns. The mutex type attribute is discussed in Section 10.1.2.)

The following program, `alarm_mutex.c`, is an improved version of `alarm_thread.c` (from Chapter 1). It lines up multiple alarm requests in a single "alarm server" thread.

```c
12-17 The `alarm_t` structure now contains an absolute time, as a standard UNIX
18 `time_t`, which is the number of seconds from the UNIX Epoch (Jan 1 1970 00:00)
19 to the expiration time. This is necessary so that `alarm_t` structures can be sorted
20 by "expiration time" instead of merely by the requested number of seconds. In
21 addition, there is a `link` member to connect the list of alarms.
```

The `alarm_mutex` mutex coordinates access to the list head for alarm requests, called `alarm_list`. The mutex is statically initialized using default attributes, with the `PTHREAD_MUTEX_INITIALIZER` macro. The list head is initialized to `NULL`, or empty.

```c
#include <pthread.h>
#include <time.h>
#include "errors.h"
```

* The "alarm" structure now contains the `time_t` (time since the
  `time_t`, in seconds) for each alarm, so that they can be
* sorted. Storing the requested number of seconds would not be
* enough, since the "alarm thread" cannot tell how long it has
* been on the list.
*/
```