Introduction

The central components of the thread API are briefly covered in this chapter. Each aspect will be detailed in further detail in the following sections, along with examples of how to use the API. However, the subsequent parts present the ideas of locks and condition variables more slowly and with more examples; hence, this chapter is best utilized as a reference.

What thread creation and control interfaces should the OS provide? What should these interfaces look like in terms of usability and convenience of use?

Thread Creation

To design a multi-threaded program, you must first generate new threads, so a thread creation interface is required. Easy in POSIX:

A little difficult (especially if you've never used function pointers in C), but not too horrible. Thread, attr, start routine, and arg. The first thread is a pointer to a pthread t structure, which we must initialize with pthread new().

The second option, attr, specifies the thread's properties. Setting the stack size or the thread's scheduling priority are other examples. A separate pthread attr init() call initializes each attribute. In most circumstances, the defaults are fine, so we'll just pass in NULL.

The final input asks: in which function should this thread start? This is a function pointer in C. So this one says to expect: a function (start procedure) that takes a single void * argument and returns a void * value (i.e., a void pointer).

Instead of a void pointer, this routine's declaration would be:

Instead, the routine would take a void reference as an argument and return an integer:

Finally, the fourth argument, arg, is the argument supplied to the thread's start method. Why do we need void pointers? We can send any type of argument to the function start routine, and the thread can return any type of result by using a void pointer as an argument.

Consider the case in Figure 27.1. We just create a thread with two arguments and a self-defined type (myarg t). Once constructed, the thread can simply cast its parameters to the desired type.

That's it! Creating a thread adds another live executing entity with its own call stack to the program's address space. So much fun!

Thread Completion

Creating a thread is illustrated above. But what if you want to wait for a thread to finish? To wait for completion, you must use the pthread join routine ().

It takes two arguments. The first is of type pthread t and specifies the waiting thread. This variable is initialized by the thread creation routine (when you feed it a pointer to pthread create()); you can use it to wait for the thread to terminate.

The second argument is a reference to the expected return value. Because the pthread join() procedure alters the value of the supplied in argument, you need to pass in a pointer to that value, not just the value itself.

Consider another example (Figure 27.2, page 4). First, we establish a new thread and provide it some arguments using the myarg t structure. The myret t type returns values. After the thread has done operating, the main thread returns from the pthread join() routine1, and we can access the thread's returned values, particularly whatever is in myret t.

Observations on this example: first, we don't always have to pack and unpack arguments. For example, to construct a thread with no arguments, we can supply NULL as an argument. If we don't care about the return value, we can pass NULL to pthread join().

Second, a single value (e.g., a long, long int) does not require an argument. Figure 27.3 (page 5) illustrates. In this situation, we don't need to organize arguments and return values.

Third, we should notice that values returned from a thread must be handled with care. Never return a pointer to a thread's call stack. What do you expect to happen? (Consider!) This is a modified version of Figure 27.2's harmful code.

An unallocated variable will produce undesirable outcomes. You may be startled when you print out the values you thought you returned. So give it a shot and see!

Finally, you may have noticed that using pthread create() followed by pthread join() is an odd approach to create a thread. A procedure call is an easier technique to do the same action. We normally create multiple threads and wait for them to finish; otherwise, threads are useless.

Not all multi-threaded programming uses the join routine. The main thread accepts requests and passes them to the workers endlessly. These programs may not need to join. To ensure that all work is completed before departing or going on to the next step of computation, a parallel program may typically employ join.

Locks

After thread creation and join, the POSIX threads library's mutual exclusion functions are perhaps the most useful. The following is the most basic set of procedures to employ for this purpose:

The procedures should be simple to use. Locks are useful when a vital part of code needs to be protected to ensure proper operation. You can probably guess the code:

The code is intended to acquire the lock if no other thread has it when pthread mutex lock() is called. If another thread already holds the lock, the thread attempting to acquire it will not return from the call until it obtains it (implying that the thread holding the lock has released it via the unlock call). Many threads may be waiting in the lock acquisition function at the same time; only the thread that has the lock should call unlock.

Sadly, this code has two fundamental flaws. The first issue is improper startup. All locks must be initialized properly to ensure that they have the correct values and work properly when locked and unlocked.

POSIX threads can initialize locks in two ways. Using PTHREAD MUTEX INITIALIZER, for example:

This restores the lock's default values and makes it usable. The dynamic way is to call pthread mutex init() as follows:

This function takes two arguments: the lock's address and an optional set of attributes. Learn about the properties; NULL uses the defaults. Both methods work, but we prefer the latter. When you're done with the lock, call pthread mutex destroy(); see the manual page for more.

The code above also fails to verify error codes when calling lock and unlock. These functions, like every library call in a UNIX system, can fail! If your code doesn't properly check error codes, it could enable numerous threads into a key part. Use wrappers that claim that the function succeeded, as shown in Figure 27.4. More advanced (non-toy) programs should check for failure and do something appropriate when a call fails.

The pthreads library's lock functions are not the only ones that interact with locks. Other interesting routines:

These two calls help get locks. The timedlock variant of getting a lock returns after a timeout or after acquiring the lock, whichever comes first.

Thus, a timedlock with a timeout of 0 is a trylock. However, as we’ll see in later chapters, there are some situations when avoiding becoming caught in a lock acquisition method might be useful (e.g., when we study deadlock).

Condition Variables

A condition variable is also a crucial component of any threads library, including POSIX threads. Condition variables are useful when threads must signal each other in order to proceed. Programs desiring to interact in this fashion employ two main routines:

To use a condition variable, one must also have a lock linked with it. This lock should be held when invoking either of the above functions.
The first routine, pthread cond wait(), puts the calling thread to sleep and waits for another thread to notify it that something has changed in the program. Usage examples:

After initializing the lock and condition, the code above checks that the variable ready is not zero. So the thread just sleeps till another thread wakes it up.

The code to wake a thread in another thread is as follows:

Observations about this code sequence: First, while signaling (or changing the global variable ready), the lock is always held. This prevents us from introducing race conditions into our code.

Second, the wait call requires a lock as a second parameter, whereas the signal call requires simply a condition. The reason for this is that putting the calling thread to sleep releases the lock. If it didn't, how could the other thread get the lock and wake it up? While the waiting thread is running between the lock acquisition at the start of the wait sequence and the lock release at the end, it re-acquires the lock.

Lastly, the waiting thread rechecks the condition in a while loop rather than an if. While we'll cover condition variables in more detail in a later chapter, using a while loop is the safest option. Some pthread implementations may wake up a waiting thread without rechecking the condition, causing the waiting thread to think the situation has changed when it hasn't. It's safer to think about waking up as a hint rather than a fact.

It's tempting to use a flag instead of a condition variable and related lock to indicate between two threads. For example, we could change the above waiting code to look like this:

The signaling code would be:

Don't do it for the reasons below. First, it frequently fails (spinning for a long time just wastes CPU cycles). Two flaws: it's prone to Recent research demonstrates that utilizing flags (as above) to synchronize between threads is surprisingly easy to get wrong; in that study, about half of these ad hoc synchronizations were incorrect! Finally, don’t be lazy; use condition variables even when you can avoid them.

Don’t worry (yet) about condition variables; we’ll examine them in depth in a later chapter. Until then, knowing they exist and how and why they are employed should sufficient.

Compiling and Running

The code examples in this chapter are all fairly simple to set up and run. You must include the header pthread.h in your code to compile them. The -pthread flag must be added to the link line to explicitly link with the pthreads library.

To compile a simple multi-threaded application, for example, simply perform the following:

You've successfully constructed a concurrent application as long as main.c has the pthreads header. It's a different thing whether it works or not, as it always is.

Thread API Guidelines

When utilizing the POSIX thread library (or any other thread library) to develop a multi-threaded program, keep in mind the following. Them:

- Be simple. Above all, thread locking and signaling code should be as basic as feasible. Bugs result from thread interactions. Avoid thread interactions. Keep the amount of thread interactions to a minimal. Each encounter should be well-planned and built using tried-and-true methods (many of which we will learn about in the coming chapters).
- Set locks and condition variables. Failure to do so will result in code that works sometimes and fails in bizarre ways.
- Verify return codes. Of course, this is true for any C or UNIX programming. If you don't, you'll probably (a) scream, (b) rip out some hair, or (c) both.
- Take caution while passing arguments to threads and returning values. In particular, passing a reference to a stack-allocated variable is probably improper. It has its own stack. Remember that each thread has its own stack. So, a locally allocated variable inside a thread's function is essentially private to that thread; no other thread can access it rapidly. To transfer data between threads, values must be in the heap or another globally accessible locale.
- Always use condition variables for inter-thread communication. • Use the manual pages instead of a simple flag. The pthread man pages on Linux, in particular, go into great length about many of the points raised here. Pay attention!

Summary

We’ve covered thread creation, mutual exclusion via locks, and signaling and waiting using condition variables. You only need patience and care to develop strong and efficient multi-threaded programs!

We’ll close this chapter with some tips for writing multi-threaded programming (see the aside on the following page for details). For further information, use man -k pthread on a Linux machine to see over one hundred APIs that make up the whole interface. Less advanced multi-threaded programs should be able to use the concepts taught here. The challenging logic of constructing concurrent systems is difficult with threads, not the APIs. Learn more below.