**Chapter 27: Interlude: Thread API**

This chapter covers the main portions of the thread API.

**27.1 Thread Creation**

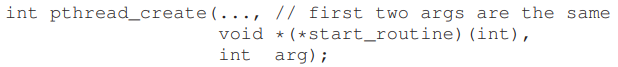
In C, the declaration of threads is

Text

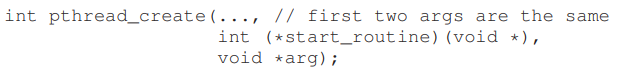
Description automatically generated

The first argument is a pointer to a pthread\_t. We will use this structure to interact with this thread. Thus, we pass it to pthread\_create() to initialize it.

The second argument is to specify any attributes this thread might have. For example, it contains information about the scheduling priority. An attribute is initialized with a separate call to pthread\_attr\_init(). The third argument is the function pointer. We pass a function name that is passed a single void pointer and return a void pointer. If it requires an int, the declaration will look like this:



If the function returns an int, it will look like this:



The reason we use void pointer is that we can pass in any type of argument and return any type of result.

Text, letter

Description automatically generated

Here we just create a thread that is passed two arguments, packaged into a single type we define ourselves (myarg\_t). The thread, once created, can simply cast its argument to the type it expects and thus unpack the arguments as desired.

**27.2 Thread Completion**

How to wait for a thread to complete? We use pthread\_join()



The first argument is used to specify which thread to wait. The second argument is a pointer to the return value you expect to get back. Because the routine can return anything, it is defined to return a pointer to void; because the pthread join() routine changes the value of the passed in argument, you need to pass in a pointer to that value, not just the value itself.

Text

Description automatically generated

In the above code, we created a single thread. Once the thread is finished running, the main thread, which has been waiting inside of the pthread\_join() routine , then returns, and we can access the values returned from the thread, namely whatever is in myret\_t.

Often, we don’t have to do all of this painful packing and unpacking of arguments. For example, if we just create a thread with no arguments, we can pass NULL in as an argument when the thread is created. Similarly, we can pass NULL into pthread\_join() if we don’t care about the return value.

In addition, if we are just passing a single value, we don’t have to package it up as an argument (Figure 27.3).

Third, we should note that one has to be extremely careful with how values are returned from a thread. Specifically, never return a pointer which refers to something allocated on the thread’s call stack. This is because when it returns, the value is automatically deallocated

Text, letter

Description automatically generated

There is an easier way to create threads, which is procedure call. Also, note that not all code that is multi-threaded uses the join routine. Long-lived programs may not need to join.

**27.3 Locks**

**Critical** **sections** are secured by **locks**. The most basic pair of routines to use for this purpose is provided by the following:



A thread will acquire the lock and enter the critical section. Other threads are not allowed to grab the lock. The one that holds the lock should call unlock.

Unfortunately, this code is broken, in two important ways. The first problem is a lack of proper initialization. All locks must be properly initialized in order to guarantee that they have the correct values to begin with and thus work as desired when lock and unlock are called.

With POSIX threads, there are two ways to initialize locks:

1. The first is using PTHREAD\_MUTEX\_INITIALIZER:

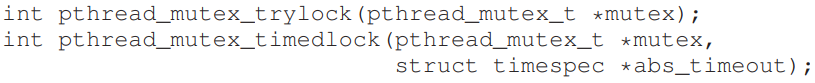


1. The dynamic way to do it is to make a call to pthread\_mutex\_init():



We usually use the dynamic method. Always check if the routine to initiate lock successfully (assert).

There are two other routines of interest:



The trylock turns failure if the lock is already held. The timedlock version returns after a timeout or after acquiring a lock, whichever happen first. (timedlock with timeout 0 is a trylock)

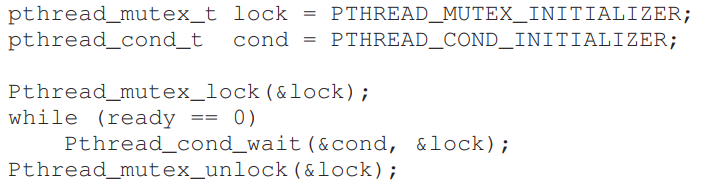
**27.4 Condition Variables**

Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue. Two primary routines are used by programs wishing to interact in this way:



To use a condition variable, one has to in addition have a lock that is associated with this condition. When calling either of the above routines, this lock should be held.

The first routine (wait), puts the calling thread to sleep and this waits for some other thread to signal it. A typical usage looks like this:



In this code, after initialization of the relevant lock and condition, a thread checks to see if the variable ready has yet been set to something other than zero. If not, the thread simply calls the wait routine in order to sleep until some other thread wakes it.

The code to wake a thread, which would run in some other thread, looks like this:

Text

Description automatically generated

When signaling, we always make sure to have the lock held to make sure not getting into race condition.

The wait call takes a lock as its second parameter, whereas the signal call only takes a condition. This is because the wait call releases the lock when putting said caller to sleep. However, before returning after being woken, the pthread\_cond\_wait() re-acquires the lock, thus ensuring that any time the waiting thread is running between the lock acquire at the beginning of the wait sequence, and the lock release at the end, it holds the lock.

The waiting thread re-checks the condition in a while loop.

Doing the above interactions between threads without locks are not a good choice as it performs poorly in many cases and error prone.

**27.5 Compiling and Running**

Compile with **-pthread**