Introduction to Posix Threads

Asynchronous Programming in the Unix/Linux Environment

Class ESC-308
Wednesday, April 16, 8:30 am

Doug Abbott
Silver City, NM
doug@intellimetrix.us

Intellimetrix
COMPUTING FOR SCIENCE AND INDUSTRY

www.intellimetrix.us

Abstract

The heavyweight "process model", historically used by Unix systems, including Linux, to split a large system into smaller, more tractable pieces doesn't always lend itself to embedded environments owing to substantial computational overhead. POSIX threads, also known as Pthreads, is a multithreading API that looks more like what embedded programmers are used to but runs in a Unix/Linux environment. This class introduces Posix Threads and shows you how to use threads to create more efficient, more responsive programs.

Introduction

The primary focus of this class is the *asynchronous programming paradigm*, commonly called multitasking. This technique has been employed successfully for at least the past quarter century to build highly responsive, robust computer systems that do everything from flying space shuttles to decoding satellite TV programs. While multitasking operating systems have been common in the world of embedded computing, it is only fairly recently—within the past decade—that multitasking has made its way into the Unix/Linux world in the form of "threads". Specifically, in this class, we'll be looking at the standard thread API known as POSIX 1003.1c, Posix Threads, or Pthreads for short.

From the perspective of an embedded systems developer familiar with off-the-shelf real-time operating systems, Linux appears to be unnecessarily complex. Much of this complexity derives from the protected memory environment in which Unix evolved. So we should start by reviewing the concept of Linux "processes". Then we'll see how threads differ, but at the same time, how the threads approach to multitasking is influenced by its Unix heritage.

The basic structural element in Linux is a *process* consisting of executable code and a collection of *resources* like data, file descriptors and so on. These resources are fully protected such that one process can't directly access the resources of another. In order for two processes to communicate with each other, they must use the inter-process communication mechanisms defined by Linux such as shared memory regions or pipes.

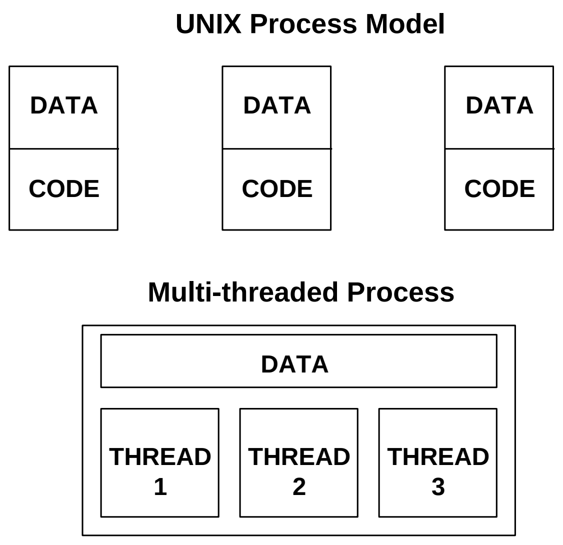


Figure 1: Processes vs. Threads

This is all well and good as it establishes a high degree of protection in the system. An errant process will most likely be detected by the operating system and thrown out before it can do any damage. But there's a price to be paid in terms of excessive overhead in creating processes and using the inter-process communication mechanisms.

Protected memory systems are divided into *User Space* and *Kernel Space*. Normal applications execute as processes in fully protected User Space. The operating system kernel executes in Kernel Space. This means that every time a kernel service is called, read() or write() for example, the system must jump through some hoops to switch from User Space to Kernel Space and back again. Among other things, data buffers must be copied between the two spaces.

A *thread* on the other hand is code only. Well, ok it's code and a *context*, which is for all practical purposes a stack that can store the state of the thread when it isn't executing. Threads only exist within the context of a process and all threads in one process share its resources. Thus all threads have equal access to data memory and file descriptors. This model is sometimes called *lightweight multi-tasking* to distinguish it from the UNIX process model.

The advantage of lightweight tasking is that intertask communication is more efficient. The drawback of course is that any task can clobber any other task's data.

Historically, most off-the-shelf multitasking operating systems, such as VRTX and VxWorks, have used the lightweight multitasking model. Recently, as the cost of processors with memory protection hardware has plummeted, and the need for reliability has increased, many vendors have introduced protected mode versions of their operating systems.

The Interrupt

Let us digress for a moment to consider the essence of the asynchronous programming paradigm. In real life, "events" often occur *asynchronously* while you're engaged in some other activity. The alarm goes off while you're sleeping. The boss rushes into your office with some emergency while you're deep in thought on a coding problem. A telemarketer calls to sell you insurance while you're eating dinner.

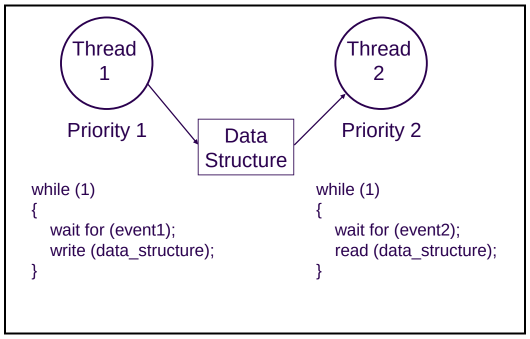
In all these cases you are "interrupted" from what you were doing and are forced to respond. How you respond depends on the nature and source of the interrupt. You chew out the telemarketer, slam the phone down and go back to eating dinner. The interruption is short if nonetheless irritating. You stop your coding to listen to the boss's perceived emergency. You may have to drop what you're doing and go do something else. When the alarm goes off, there's no question you're going to do something else. It's time to get up.

In computer terms, an interrupt is the processor's response to the occurrence of an event. . The event "interrupts" the current flow of instruction execution invoking another stream of instructions that services the event. When servicing is complete, control normally returns to where the original instruction stream was interrupted. But under supervision of the operating system, control may switch to some other activity.

Interrupts are the basis of high performance, highly responsive computer systems. Perhaps not surprisingly they are also the cause of most of the problems in real-time programming.

Consider a pair of threads acting in a simple producer/consumer paradigm. Thread 1 produces data that it writes to a global data structure. Thread 2 consumes the data from the same structure. Thread 1 has higher priority than Thread 2 and we'll assume that this is a preemptive system. (Set aside for the moment that I haven't defined what preemptive is. It should become clear.)

Thread 1 produces data in response to some event, i.e. data is available from the source, an A/D converter perhaps. The event is signaled by an interrupt. The interrupt may be from the A/D converter saying that it has finished a conversion, or it may simply be the timer tick saying that the specified time interval has elapsed. This leads to some problems.



```
while (1)
{
   wait for (event1);
   write (data_structure);
}
```

```
while (1)
{
   wait for (event2);
   read (data_structure);
}
```

Figure 2: Interrupts casue problems

The problem should be relatively obvious. The event signaling "data ready" may occur while Thread 2 is in the middle of reading the data structure. Thread 1 *preempts* Thread 2 and writes new data into the structure. When Thread 2 resumes, it finds inconsistent data.

The Multitasking Paradigm

This then is the essence of the real-time programming problem; managing asynchronous events such that they can be serviced only when it is safe to do so.

Fundamentally, multitasking is a paradigm for safely and reliably handling asynchronous events. Beyond that it is also a useful way to break a large problem down into a set of much smaller, more tractable problems that may be treated independently. Each part of the problem is implemented as a *thread*. Each thread does one thing to keep it simple. Then we pretend that all the threads run in parallel.

To reiterate, threads are the way in which multitasking is implemented in Unix-like systems. With this background we can now begin exploring the world of Posix threads.

The Threads API

Creating a Thread

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(*start_ routine) (void *), void
*arg);
void pthread_exit (void *retval);
int pthread_join (pthread_t thread, void **thread_return);
pthread_t pthread_self (void);
int sched_yield (void);
```

The mechanism for creating and managing a thread is analogous to creating and managing a process. The pthread_create() function is like fork() except that the new thread doesn't return from pthread_create() but rather begins execution at start_routine(), which takes one void * argument and returns void * as its value. The arguments to pthread_create() are:

- pthread_t – A *thread object* that represents or identifies the thread. pthread_create() initializes this as necessary..
- Pointer to a thread *attribute* object. More on that later.
- Pointer to the start routine.
- Argument to be passed to the start routine when it is called.

A thread may terminate by calling pthread_exit(). The argument to pthread_exit() is the start routine's return value.

In much the same way that a parent process can wait for a child to complete by calling waitpid(), a thread can wait for another thread to complete by calling pthread_join(). The arguments to pthread_join() are the ID of the thread to wait on and a place to store the thread's return value. The calling thread is blocked until the target thread terminates.

A thread can determine its own ID by calling pthread_self(). Finally, a thread can voluntarily yield the processor by calling sched_yield().

```
/*        Thread Example   */
#include <pthread.h>
#include "errors.h"

/*
 * Thread start routine.
 */
void *thread_routine (void *arg)
{
   printf ("%s\n"), arg);
   return arg;
}

main (int argc, char *argv[])
{
   pthread_t thread_id;
   void *thread_result;
   int status;

   status = pthread_create (&thread_id, NULL,
       thread_routine, (void *)argv[0]);
   printf ("New thread created\n");

   status = pthread_join (thread_id, thread_result);
   printf ("Thread returned %p\n", thread_result);
   return 0;
}
```

Figure 3: Sample Thread Program

Note that most of the functions above return an int value. This reflects the threads approach to error handling. Rather than reporting errors in the global variable errno, threads functions report errors through their return value. This is because errno is global and visible to all threads. This makes it susceptible to the same kind of preemption problem we saw earlier in discussing interrupts.

Figure 3 is a simple example of creating a thread. To make it simpler, error return status has been ignored. This is a thread version of the traditional Hello World program.

All thread programs must include the header file pthread.h. This program creates a thread, passing as the argument the first element of argv passed to main. After telling us it created a thread, the main function waits for the thread to terminate and then outputs the returned value.

The thread simply prints its argument as a string and returns the argument as its value. Note that a thread may terminate by simply returning rather than calling pthread_exit().

Figure 4 illustrates the life cycle of a thread as represented by a state machine. A thread is "born" by the pthread_create() function, which places it in the ready state. A thread runs when it is scheduled. The running thread may be blocked because it must wait for some resource, or it may be preempted either because a higher priority thread is ready to run or its timeslice has expired.

When the resource becomes available a blocked thread transitions to the Ready state and will eventually be scheduled to run again. Finally, a thread is terminated when it is done or another thread requests its cancellation.

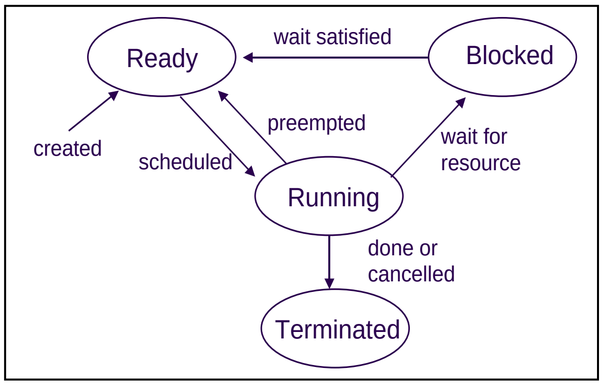


Figure 4: Thread State Machine

Thread Termination

A thread may be terminated either voluntarily or involuntarily. A thread terminates itself either by simply returning or by calling pthread_exit(). In the latter case, all *cleanup handlers* that the thread registered by calls to pthread_cleanup_push() are called prior to termination.

A thread may be involuntarily terminated if another thread *cancels* it. The cleanup handlers are also called in this case. We'll return to the notion of cleanup handlers and thread cancellation later.

Threads have an *attribute* called *detach state*. The detach state determines whether or not a thread can be joined when it terminates. The default detach state is PTHREAD_CREATE_JOINABLE meaning that the thread can be joined on termination. The alternative is PTHREAD_CREATE_DETACHED, which means the thread can't be joined.

Joining is useful for two reasons: either you need the thread's return value, or you need to be sure the thread has terminated before proceeding. Otherwise it's better to create the thread detached. The resources of a joinable thread can't be recovered until another thread joins it whereas a detached thread's resources can be recovered as soon as it terminates. Most multitasking kernels have no concept of a joinable task. All tasks are detached.

Attribute Objects

POSIX provides an open-ended mechanism for extending the API through the use of *attribute objects*. For each pthread object there is a corresponding attribute object. This attribute object is effectively an extended argument list to the related object create function. A pointer to an attribute object is always the second argument to a create function. If this argument is NULL the create function uses appropriate default values.

An important philosophical point is that all pthread objects are considered to be "opaque". This means that you never directly access members of the object itself. All access is through API functions that get and set the member fields of the object.

This allows new arguments to be added to a pthread object by simply defining a corresponding pair of get and set functions for the API.

Here is part of the attribute API for thread objects:

```
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);
int pthread_attr_getdetachstate (pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate  (pthread_attr_t *attr, int detachstate);
```

Before it can be used, an attribute object must be initialized. Then any of the attributes defined for that object may be set or retrieved with the appropriate functions. This must be done *before* the attribute object is used in a call to pthread_create(). If necessary, an attribute object can also be "destroyed". Note that a single attribute object can be used in the creation of multiple threads.

For threads there is only one required attribute, the detach state that we met earlier.

Thread Scheduling Policies

There are several optional thread attributes that have to do with scheduling policy, that is, how threads are scheduled relative to one another. These attributes and the strategies they represent are only present if the constant _POSIX_THREAD_PRIORITY_SCHEDULING is defined. Without this constant, Pthreads is *not* real-time and simply falls back on the default Linux scheduling policy.

A threads implementation that defines _POSIX_THREAD_PRIORITY_SCHEDULING must also define a structure, sched_param, with at least one member, sched_priority, an integer. Interestingly, here is one place where a structure member is made explicitly visible. There is not a pair of functions for setting and getting sched_priority. You simply read and write the structure member directly. Note that sched_param is declared with a struct and not as a typedef.

Here is the API for thread scheduling attributes:

```
int pthread_attr_setschedparam (pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedparam (const pthread_attr_t *attr, struct sched_param *param);
int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy (const pthread_attr_t *attr, int *policy);
int pthread_attr_setinheritsched (pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched (const pthread_attr_t *attr, int *inherit);

int pthread_setschedparam (pthread_t pthread, int *policy, const struct sched_param *param);
int pthread_getschedparam (pthread_t pthread, int *policy, struct sched_param *param);
int sched_get_priority_max (int policy);
int sched_get_priority_min (int policy);
```

Pthreads defines two real-time scheduling policies that can be applied on a per-thread basis:

- SCHED_FIFO – A thread runs until another thread of higher priority becomes ready or until it voluntarily blocks. When a thread with SCHED_FIFO scheduling policy becomes ready, it runs immediately if its priority is higher than that of the running thread.
- SCHED_RR – Much like SCHED_FIFO with the addition that a SCHED_RR thread can be *preempted* by another SCHED_FIFO or SCHED_RR thread of the same or higher priority after a specified time interval or *timeslice.*

In Linux, the real-time scheduling policies may only be set by processes with superuser privileges. When you wish to explicitly set the scheduling policy or parameters of a thread, you must also set the *inheritsched* attribute. By default inheritsched is set to PTHREAD_INHERIT_SCHED, meaning that newly created threads will inherit the scheduling policies of the thread that created them. To be able to change scheduling policies you set inheritsched to PTHREAD_EXPLICIT_SCHED.

When threads with SCHED_FIFO or SCHED_RR scheduling policies block waiting for a resource, they wait up in *priority* order. That is, if several threads are waiting on a resource, the one with the highest priority is scheduled when the resource becomes available. If several threads of the same priority are waiting, the one that has been waiting longest will be scheduled (FIFO order).

You can get the minimum and maximum priority values for each of the real-time scheduling policies through the functions sched_get_priority_min() and sched_get_priority_max(). You can also get and set the scheduling policies of a running thread through pthread_getschedparam() and pthread_setschedparam().

Finally, Pthreads defines another scheduling policy, SCHED_OTHER, which is "not defined." In practice, this becomes the default Linux scheduling policy. In most cases it is a "fairness" algorithm that gradually raises the priority of low priority waiting threads until they run.

Sharing Resources

One of the primary functions of any operating system is to manage access to global resources by multiple threads/tasks/processes to avoid conflicts. This is a major issue in event-driven systems where threads may compete asynchronously for access to a global resource.

Consider the following scenario. Two threads of the same priority are each running the same code fragment:

```
printf ("I am thread %d\n", n);
```

Let's assume they're operating under the SCHED_RR scheduling policy. In the absence of any kind of synchronizing mechanism, the result could be something like "II a amm ThThrreeadad 12".

What is needed is some way to regulate access to the printer so that only one task can use it at a time. In Pthreads, that mechanism is called a *mutex*, which is short for "mutual exclusion". A mutex acts like a key to control access to a resource. Only the thread that has the key can use the resource. In order to use the resource (in this case a printer) a thread must first *acquire* the key (mutex) by calling an appropriate Pthreads service. If the key is available, that is the resource (printer) is not currently in use by someone else, the thread is allowed to proceed. Following its use of the printer, the thread releases the mutex so another thread may use it.

If however, the printer is in use, the thread is blocked until the thread that currently has the mutex releases it. Any number of threads may try to acquire the mutex while it is in use. All of them will be blocked. The waiting tasks are queued in priority order.

Mutex API

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *mutex_attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);

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

The Pthreads mutex API follows much the same pattern as the thread API. There is a pair of functions to initialize and destroy mutex objects and a set of functions to act on the mutex objects. This also shows an alternate way to initialize statically allocated Pthreads objects. PTHREAD_MUTEX_INITIALIZER provides the same default values as pthread_mutex_init().

Two operations may be performed on a mutex: lock and unlock. The lock operation causes the calling thread to block if the mutex is not available. trylock allows you to test the state of a mutex without blocking. If the mutex is available trylock returns success and locks the mutex. If the mutex is not available it returns EBUSY.

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

```
#ifdef _POSIX_THREAD_PROCESS_SHARED
int pthread_mutexattr_getpshared (pthread_mutexattr_t *attr, int *pshared);
int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared);
#endif

int pthread_mutexattr_getkind_np (pthread_mutexattr_t *attr, int *kind);
int pthread_mutexattr_setkind_np (pthread_mutexattr_t *attr, int kind);
```

Mutex attributes follow the same basic pattern as thread attributes. There is a pair of functions to create and destroy a mutex attribute object. We'll defer discussion of the pshared attribute until later. There are some other attributes we'll take up shortly.

Linux implements an interesting *non-portable* extension to the Pthreads mutex. The Pthreads standard explicitly allows non-portable extensions. The only requirement is that any symbol that is non-portable must have "_np" appended to its name.

What happens if a thread should attempt to lock a mutex that it has already locked? Normally the thread would simply hang up. Linux offers a way out of this dilemma. The "kind" attribute alters the behavior of a mutex when a thread attempts to lock a mutex that it has already locked:

<u>Fast (PTHREAD_MUTEX_FAST_NP)</u>. This is the default type. If a thread attempts to lock a mutex it already holds it is blocked and thus effectively deadlocked. The fast mutex does no consistency or sanity checking and so it is faster.

<u>Recursive (PTHREAD_MUTEX_RECURSIVE_NP)</u>. A recursive mutex allows a thread to successfully lock a mutex multiple times. It counts the number of times the mutex was locked and requires the same number of calls to the unlock function before the mutex goes to the unlocked state.

<u>Error checking (PTHREAD_MUTEX_ERRORCHECK_NP)</u>. If a thread attempts to recursively lock an error checking mutex, the lock function returns immediately with the error code EDEADLK. Furthermore, the unlock function returns an error if it is called by a thread other than the current owner of the mutex.

Note the "_NP" in the constant names.

Priority Inversion

Solving the resource sharing problem with mutexes sometimes leads to other problems. Consider a scenario involving three threads. Threads 1 and 2 each require access to a common resource protected by a mutex. Thread 1 has the highest priority and Thread 2 has the lowest. Thread 3, which has no interest in the resource, has a "middle" priority.

The situation is illustrated graphically in figure 5. Thread 2 is currently executing and locks the mutex. So it gets the resource. Next an interrupt occurs which makes Thread 1 ready. Since Thread 1 has higher priority, it preempts Thread 2 and executes until it attempts to lock the resource mutex.

Thread 1 blocks and Thread 2 regains control. So far everything is working as we would expect. Even though Thread 1 has higher priority, it simply has to wait until Thread 2 is finished with the resource.

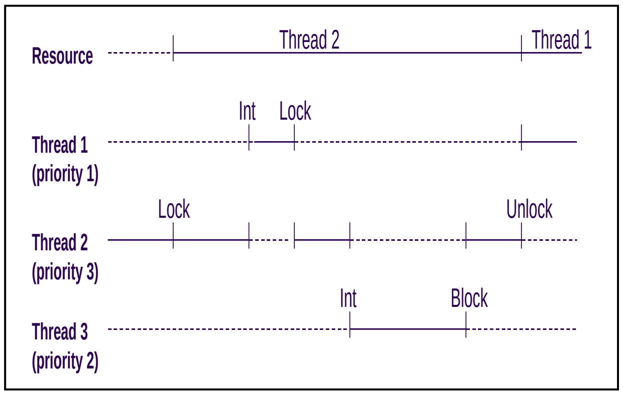


Figure 5: Priority Inversion

The problem arises if Thread 3 should become ready while Thread 2 has the resource locked. Thread 3 preempts Thread 2. This situation is called *priority inversion* because a lower priority thread (Thread 3) is effectively preventing a higher priority thread (Thread 1) from executing.

There are a couple of possible solutions to this problem. One is called *priority inheritance*. If a higher priority thread attempts to lock a mutex that is already locked, the priority of the thread owning the mutex is temporarily raised to that of the thread attempting to lock the mutex. This prevents any lower priority thread from preempting the owner until it unlocks the mutex, at which time its priority reverts to its normal value.

The other solution is called *priority ceiling*. In this approach, when a thread locks a mutex, its priority is immediately raised to that of the highest priority thread that will ever try to lock the mutex. This is considered to be more efficient because it can avoid some unnecessary context switches. Note in figure 5 that Thread 1 is switched into the processor only to be blocked a short time later when it attempts to lock the mutex. The priority ceiling protocol would prevent this.

These strategies are optional values for a mutex attribute called *protocol*.

```
#if defined (_POSIX_THREAD_PRIO_PROTECT) || defined
(_POSIX_THREAD_PRIO_INHERIT)
int pthread_mutexattr_getprotocol (const pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr, int protocol);
#endif

#ifdef _POSIX_THREAD_PRIO_PROTECT
int pthread_mutexattr_getprioceiling (const pthread_mutexattr_t *attr, int *ceiling);
int pthread_mutexattr_setprioceiling (pthread_mutexattr_t *attr, int ceiling);
int pthread_mutex_getprioceiling (const pthread_mutex_t *mutex, int *ceiling);
int pthread_mutex_setprioceiling (pthread_mutex_t * mutex, int ceiling);
#endif
```

Protocol is an optional attribute for Pthread mutexes. The values for mutex protocol are:

- PTHREAD_PRIO_NONE – Default. Thread priorities are not modified by locking the mutex.
- PTHREAD_PRIO_INHERIT – Use the priority inheritance protocol.
- PTHREAD_PRIO_PROTECT – Use the priority ceiling protocol.

If PTHREAD_PRIO_PROTECT is selected, you can set the priority ceiling value either in the mutex attribute or in the mutex itself.

Conditional Variable

There are many situations where one thread needs to notify another thread about a change in status to a shared resource protected by a mutex. For this we use a *conditional variable*, or just condition for short. A conditional variable must always have a mutex associated with it.

Threads that wish to be notified of the change of status wait on the conditional variable with the associated mutex *locked*. A thread that detects the change *signals* the conditional variable causing one of the waiting threads to wake up. A conditional variable operates much like a semaphore with the additional feature that it solves a potential race condition involving a mutex.

Consider a queue where one thread, Thread 1, reads the queue and another thread, Thread 2, writes it. Clearly each thread requires exclusive access to the queue and so we protect it with a mutex.

Thread 1 will lock the mutex and then see if the queue has any data. If it does, Thread 1 reads the data and unlocks the mutex. However, if the queue is empty, Thread 1 needs to block somewhere until Thread 2 writes some data. It also sets a flag so that the next thread to write to the queue recognizes that someone is blocked on it waiting for data. Thread 1 must unlock the mutex before blocking or else Thread 2 would not be able to write. But there's a gap between the time Thread 1 unlocks the mutex and blocks. During that time, Thread 2 may execute and not recognize that anyone is blocking on the queue.

The conditional variable solves this problem by waiting (blocking) with the mutex locked. Internally, the conditional wait function unlocks the mutex allowing Thread 2 to proceed. When the conditional wait returns, the mutex is again locked.

There's an important distinction between mutexes and conditional variables. A mutex is used to guarantee exclusive access to a resource. A conditional variable is used to signal that something has happened.

Condition API

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *cond_attr);
int pthread_cond_destroy (pthread_cond_t *cond);

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const struct
timespec *abstime);
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);

int pthread_condattr_init (pthread_condattr_t *attr);
int pthread_condattr_destroy (pthread_condattr_t *attr);

#ifdef _POSIX_THREAD_PROCESS_SHARED
int pthread_condattr_getpshared (pthread_condattr_t *attr, int *pshared);
int pthread_condattr_setpshared (pthread_condattr_t *attr, int pshared);
#endif
```

The basic operations on a conditional variable are *signal* and *wait*. A thread may execute a *timed wait* such that if the specified time interval expires before the condition is signaled, the wait returns with an error. A thread may also *broadcast* a condition. This wakes up all threads waiting on the condition. Figure 7 is an example of using a conditional variable with a queue

Threads, Processes and "Kernel Entities"

```
int pthread_attr_getscope (const pthread_attr_t *attr, int *contentionscope);
int pthread_attr_setscope (const pthread_attr_t *attr, int contentionscope);
```

Besides scheduling policy and parameters, there are a couple of other dimensions to real-time scheduling. *Contention Scope* is a description of how threads compete for resources. The contention scope attribute for a thread has two possible values:

- PTHREAD_SCOPE_PROCESS: A thread competes for resources only with other threads in the same process. Processes are scheduled by the kernel using the normal scheduling algorithm and some other scheduler manages the scheduling of threads within each process.
- PTHREAD_SCOPE_SYSTEM: A thread competes for resources with *all* threads in the system. Thus the kernel is involved in thread scheduling.

Process scope is "cheap" in the sense that thread scheduling need not involve a system call with the corresponding switch to kernel space. The thread context switch can be managed entirely in User Space by standard C library facilities like setjmp() and longjmp(). On the other hand, since the process scope thread scheduler has no visibility into, or control over, thread scheduling in other processes, a high priority thread in one process could be preempted by a low priority thread in another process. System scope may offer lower performance owing to the necessary kernel call, but it is more predictable because all threads are competing on the same basis.

Process scope raises an interesting question. What happens if a User Space thread calls, for example, read()? Normally, a process that calls read() is blocked until the read data is ready. Does this mean that all threads in the process are blocked? In some simple threads implementations the answer is yes. But a fully conforming Posix Threads implementation must not block all threads when one thread calls a blocking system service. In practice this means that the system library must be "thread aware" and make use of features like non-blocking I/O or POSIX 1.b asynchronous I/O to avoid blocking.

Another way to look at the issue of scope is to consider the mapping between threads and "kernel entities". A kernel entity is the level of abstraction that usually exists between Posix threads and the processor. In a typical Unix/Linux system the kernel entity is a process. User Space threads, sometimes called a "library implementation", represent a *many-to-one* mapping in that multiple threads exist within one schedulable kernel entity or process. By contrast, threads managed under system scope, also called "kernel threads", typically represent a *one-to-one* mapping in that each thread is represented to the system by its own kernel entity. This is not to say that each thread is an entirely separate process. Multiple threads may exist within a single process space but, for scheduling purposes, the kernel treats each thread as a separate schedulable "entity".

Most of the prevalent threads implementation use a one-to-one mapping scheme making use of the clone() system call to create a kernel schedulable thread entity.

This issue of scope brings us back to the *pshared* attribute of mutexes and conditions. If a Pthreads implementation supports the _POSIX_THREAD_PROCESS_SHARED option, you can set the pshared attribute of a mutex or condition variable to the value PTHREAD_PROCESS_SHARED. This allows the object to be accessed by threads in different processes. Of course the object must be created in global memory accessible to all processes. The default value for pshared is PTHREAD_PROCESS_PRIVATE.

Thread Cancellation

```
int pthread_cancel (pthread_t thread);
int pthread_setcancelstate (int state, int *oldstate);
int pthread_setcanceltype (int type, int *oldtype);
void pthread_testcancel (void);
void pthread_cleanup_push (void (*routine)(void *), void *arg);
void pthread_cleanup_pop (int execute);
```

We'll wrap up our brief look at Posix threads by considering the issue of thread cancellation and "cleaning up". Many threads run in an infinite loop. As long as the system is powered up, the thread is running, doing its thing. Some threads start up, do their job and finish. But there are also circumstances where it's useful to allow one thread to terminate another thread involuntarily. Perhaps the user presses a CANCEL button to stop a long search operation. Maybe the thread is part of a redundant numerical algorithm and is no longer needed because another thread has solved the problem. The Pthreads cancellation mechanism provides for the orderly shutdown of threads that are no longer needed.

But cancellation must be done with care. You don't just arbitrarily stop a thread at any point in its execution. Suppose it has a mutex locked. If you terminate a thread that has locked a mutex, it can never be unlocked. Pthreads allows each thread to manage its own termination. So when you cancel a thread you're usually not stopping it immediately, you're politely asking the thread to terminate itself as soon as it's safe or convenient.

Pthreads supports three cancellation modes encoded as two bits called *cancellation state* and *cancellation mode* (see Figure 6). A thread may choose to disable cancellation because it is performing an operation that must be completed. The default mode is Deferred meaning that cancellation can only occur at specific points, called *cancellation points*, where the program tests whether the thread has been requested to terminate. Most functions that can block for an unbounded time, such as waiting on a condition variable or reading or writing a file should be cancellation points and are defined as such in the Posix specification. The function pthread_testcancel() allows you to create your own cancellation points. It simply returns immediately if the thread has not been requested to terminate.

Mode	State	Type	Meaning
Off	Disabled	N/A	Cancellation remains pending until enabled.
Deferred	Enabled	Deferred	Cancellation occurs at next *cancellation point.*
Asynchronous	Enabled	Asynchronous	Cancellation may occur at any time.

Figure 6: Cancellation modes

While asynchronous cancellation mode might seem like a good idea, it is rarely safe to use. That's because, by definition you don't know what state the thread is in when it gets the cancellation request. It may have just called pthread_mutex_lock(). Is the mutex locked? Don't know. So while asynchronous cancellation mode is in effect, you can't safely acquire any shared resources.

Thread Cleanup

When a thread is requested to terminate itself, there may be some things that need to be "cleaned up" before the thread can safely terminate. It may need to unlock a mutex or return a dynamically allocated memory buffer for example. That's the role of *cleanup handlers*. Every thread conceptually has a stack of active cleanup handlers. Handlers are pushed on the stack by pthread_cleanup_push() and executed in reverse order when the thread is terminated or calls pthread_exit(). A cleanup handler takes one void * argument.

The most recently pushed cleanup handler can be popped off the stack with pthread_cleanup_pop(). Often the functionality of a cleanup handler is needed whether or not the thread terminates. The execute argument specifies whether or not a handler is executed when it's popped. A non-zero value means execute.

Figure 8 shows the read thread (the main function) from figure 7 with a cleanup handler added. We assume the default deferred cancellation mode. Note that pthread_cleanup_pop() is used to unlock the mutex rather than the normal mutex unlock function. Is it really necessary to push and pop the cleanup handler on every pass through the while loop? It is if there is a cancellation point in the section called "do something with the data" where the mutex is unlocked. This thread can only invoke the cleanup handler if it has the mutex locked. If there are no cancellation points while the mutex is unlocked then it's safe to move the push cleanup call outside the loop. In that case we don't really need pop cleanup. Note also that pop cleanup can only be called from the same function that called push cleanup.

Threads Implementation in Linux

Until kernel version 2.6, the most prevelant threads implementation was LinuxThreads. It has been around since about 1996 and by the time development began on the 2.5 kernel it was generally agreed that a new approach was needed to address the limitations in LinuxThreads. Among these limitations, the kernel represents each thread as a separate process, giving it a unique process ID, even though many threads exist within one process entity. This causes compatibility problems with other thread implementations. There's a hard coded limit of 8192 threads per process, and while this may seem like a lot, there are some problems that can benefit from running thousands of threads.

The result of this new development effort is the Native Posix Threading Library or NPTL, which is now the standard threading implementation in 2.6 series kernels. It too is a one-to-one model, but takes advantage of improvements in the kernel that were specifically intended to overcome the limitations in LinuxThreads. The clone() call was extended to optimize thread creation. There's no longer a limit on the number of threads per process, and the new fixed time scheduler can handle thousands of threads without excessive overhead. A new synchronization mechanism, the Fast Mutex or "futex", handles the non-contention case without a kernel call.

In tests on an IA-32, NPTL is reportedly able to start 100,000 threads in two seconds. By comparison, this test under a kernel without NPTL would have taken around 15 minutes.

Both flavors of real-time Linux, RTLinux and RTAI, implement Pthreads as real-time tasks executing in kernel space.

Summary

At first glance, the Posix Threads API looks like it was invented by a committee. Well, it was. Nevertheless, a lot of thought went into its design, which becomes evident if you study it carefully. There are more features in Pthreads than we've covered here and probably more than you'll ever use. But used properly and creatively, Pthreads is a powerful tool for building highly reliable, responsive embedded applications.

Resources

Internet

http://pauillac.inria.fr/~xleroy/linuxthreads/ This is an older site. It has a very good FAQ and a number of links to other threads resources. The Pthreads implementation that this site originally described has been integrated into libc 6.
www.gnu.org/directory/libs/c/pth.html The Portable Threads Library. This is a User Space implementation using a non-preemptive priority based scheduler. The API is not Posix Threads but a Posix wrapper is available.
www.rtai.org RTAI includes a Posix API that runs in kernel space.

Books

Butenhof, David R., *Programming with POSIX Threads,* Addison-Wesley, 1997. A thorough and well-written guide to Posix threads and asynchronous programming in general.

```
/* Conditional Variable Example */
#include <pthread.h>

typedef struct my_queue_tag {
   pthread_mutex_t     mutex;          /* Protects access to queue */
   pthread_cond_t      cond;           /* Signals change to queue */
   int                 get, put;       /* Queue pointers */
   unsigned char       empty, full;    /* Status flags */
   int                 q[Q_SIZE]       /* the queue itself        */
} my_queue_t;

my_queue_t data = {
   PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0, 0, 1, 0};

void *write_thread (void *arg)
{
   while (1)
   {
      /* wait for the data source to supply new data */

      pthread_mutex_lock (&data.mutex);
      /* write data to queue */
      if (data.empty)
      {
         data.empty = 0;
         pthread_cond_signal (&data.cond);
      }
      pthread_mutex_unlock (&data.mutex);
   }
}

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

   status = pthread_create (&write_thread_id, NULL, write_thread, NULL);

   while (1)
   {
      pthread_mutex_lock (&data.mutex);
      if (queue is empty)
      {
         data.empty = 1;
         pthread_cond_wait (&data.cond, &data.mutex);
      }
      /* read data from queue */
      pthread_mutex_unlock (&data.mutex);

      /* do something with the data */
   }
}
```

Figure 7: Conditional Variable Example

```
/* Cleanup Handler Example */
#include <pthread.h>

typedef struct my_queue_tag {
   pthread_mutex_t      mutex;           /* Protects access to queue */
   pthread_cond_t       cond;            /* Signals change to queue */
   int                  get, put;        /* Queue pointers */
   unsigned char        empty, full;     /* Status flags */
   int                  q[Q_SIZE]        /* the queue itself          */
} my_queue_t;

my_queue_t data = {
   PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0, 0, 1, 0};

void cleanup_handler (void *arg)
/*
   Unlocks the mutex associated with a queue
*/
{
   int status;
   pthread_mutext_t *m = &((my_queue_t *)arg)->mutex;

   status = pthread_mutex_unlock (m);
}

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

   status = pthread_create (&write_thread_id, NULL, write_thread, NULL);

   while (1)
   {
      pthread_cleanup_push (cleanup_handler, (void *) &data);
      pthread_mutex_lock (&data.mutex);
      if (queue is empty)
      {
         data.empty = 1;
         pthread_cond_wait (&data.cond, &data.mutex);
      }
      /* read data from queue */
      pthread_cleanup_pop (1);

      /* do something with the data */
   }
}
```

Figure 8: Cleanup Handler Example