Lecture 18 — Concurrency & Synchronization in POSIX

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 3, 2018

Concurrency with pthreads

Earlier we saw pthreads & how we can use them: multithreaded programs.

There are, accordingly, pthread routines for working with mutual exclusion.

Mutexes and pthreads

pthread_mutex_init: used to create a new mutex variable and returns it, with type pthread_mutex_t.

It takes an optional parameter, the attributes (the details of which are not important at the moment, but relate mostly to priorities).

We can initialize it using null, or:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

Mutexes and pthreads

When created, by default, the mutex is unlocked.

There are three methods related to using the mutex; two to lock it and one to unlock it, all of which take as a parameter the mutex to (un)lock.

The unlock method, `pthread_mutex_unlock` is self-explanatory.

The two kinds of lock are `pthread_mutex_lock`, which is blocking, and `pthread_mutex_trylock`, which is nonblocking.

Attempting to unlock a mutex that is not currently locked is an error.

Also an error if one thread attempts to unlock a mutex owned by another.

Destroying the Mutex

To destroy a mutex, there is a method `pthread_mutex_destroy`.

It cleans up a mutex and should be used when finished with it.

If attributes were created with `pthread_mutexattr_init` they should be destroyed with `pthread_mutexattr_destroy`.

Destruction Aborted

An attempt to destroy the mutex may fail if the mutex is currently locked.

The specification says that destroying an unlocked mutex is okay, but attempting to destroy a locked one results in undefined behaviour.

Undefined behaviour is, in the words of the internet, the worst thing ever.

It means code might work some of the time or on some systems, but not others.

Or it could work fine for a while and then break suddenly later.

Condition Variables

Condition variables are another way to achieve synchronization.

A condition variable allows synchronization based on the value of the data.

Instead of locking a mutex, checking a variable, and then unlocking the mutex, we could achieve the same goal without constantly polling.

We can think of condition variables as “events” that occur.

Condition Variables: Events

An event is similar to, but slightly different from, a counting semaphore.

When an event occurs we can:

- Signal either one thread waiting for that event to occur, or
- Broadcast (signal) to all threads waiting for the event.

If a thread signals a condition variable that an event has occurred, but no thread is waiting for that event, the event is “lost”.

Condition Variables and pthreads

To create a `pthread_cond_t` (condition variable type), the function is `pthread_cond_init`.

To destroy them, `pthread_cond_destroy`.

As before, we can initialize them with attributes, and there are functions to create and destroy the attribute structures, too.

Condition Variables as a Building Block

Condition variables are a building block and must be paired with a mutex.

To wait on a condition variable, the function `pthread_cond_wait` takes two parameters: the condition variable and the mutex.

This routine should be called only while the mutex is locked.

Condition Variables as a Building Block

It will automatically release the mutex while it waits for the condition.

When the condition is true then the mutex will be automatically locked again so the thread may proceed.

The programmer then unlocks the mutex when the thread is finished with it.

Condition Variables

There is `pthread_cond_signal` that signals a provided condition variable.

There is also `pthread_cond_broadcast` that signals all threads waiting.

It is (almost always) a logical error to signal or broadcast on a condition variable before some thread is waiting on it.

Condition Variable Example

[This code is too big for slides; see notes.]

Monitors

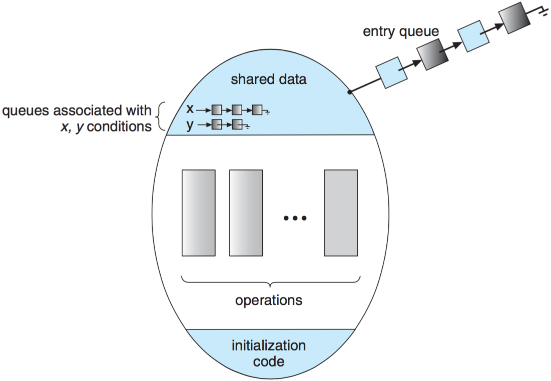
A condition variable can be used to create a monitor.

Analogy: in object-oriented programming we package up data and functions inside a class to make errors less likely and to improve the design.

When we use a monitor we are packaging up the shared data and operations on that data to avoid problems of synchronization and concurrency.

Monitors

Goal: make it so that programmers do not need to code the synchronization part directly, making it less likely a programmer makes an error.



Monitor Example

Suppose there are processes *P* and *Q* and a condition variable *x*.

If *P* signals on *x* when *Q* is waiting, there are two things we can do:
signal-and-wait or signal-and-continue.

Signal-and-continue seems logical, in one sense: *P* is already running, so why do the extra work to switch?

On the other hand, by the time *P* exits the monitor on its own, the condition *Q* was waiting for might no longer be true.

Some programming languages opt for a compromise: after signalling, *P* must immediately leave the monitor.

Monitors and `synchronized`

The idea of monitors should be familiar to you if you have used the Java `synchronized` keyword.

In Java we can declare a method to be `synchronized`.
Then there is a lock created around that method.

Only one thread can be inside that method at a time.

If a second would like to call that method on the same instance, it will be placed in the entry set for the lock.

Java Synchronization

In Java we can make a method `synchronized` or define a block as synchronized:

```
public synchronized void doSomething() {
    // Synchronized area
}
```

```
public void exampleMethod() {
    synchronized( object ) { // Lock must be acquired to enter
           // Critical section
    } // Lock is automatically released.
}
```

UNIX Concurrency Mechanisms

UNIX also has semaphores (not just in the pthreads).

The semaphore in UNIX is general, and maintains a few pieces of extra data:

- The value of the semaphore.
- Process ID of the last process to operate it.
- Number of processes waiting on a value greater than the current value.
- Number of processes waiting for the value to be 0.

UNIX Semaphores

UNIX semaphores are technically created in sets of one or more.

It is possible to manage semaphores en masse with a system call (`semctl`).

This then performs the requested operation on all of the semaphores in the set.

There is also a `sem_op` system call that takes as an argument, a list of semaphore operations, each defined on one of the semaphores in the set.

When `sem_op` is used, the indicated operations are performed one at a time.

UNIX `sem_op`

The meanings of `sem_op` can be:

- `sem_op` positive
- `sem_op` 0
- `sem_op` negative and its absolute value $\leq$ to the semaphore value
- `sem_op` negative and its absolute value $>$ the semaphore value

Linux Kernel Concurrency Mechanisms

The Linux kernel provides operations that are guaranteed to execute atomically, to avoid simple race conditions.

Uniprocessor system: CPU can’t be interrupted until the operation is finished.

Multiprocessor system: the variable is locked until the operation is finished.

Linux Kernel Concurrency Mechanisms

There are two types of atomic operations:

- Those that operate on integers; and
- Those that operate on a single bit in a bitmap.

On some architectures, the atomic operations are translated into uninterruptible assembly instructions.

On others, the memory bus is locked to ensure the operation is atomic.

Linux Kernel Concurrency Mechanisms

Rather than just using an integer for atomic integer operations, there is a defined type `atomic_t`.

This ensures that only atomic operations can be used on this data type and that atomic operations can only be used on that data type.

This prevents programming errors and hides architecture-specific implementation details.

Linux Atomic Operations

Function	Description
`ATOMIC_INIT( int i )`	At declaration, initialize an `atomic_t` to `i`
`int atomic_read( atomic_t *v )`	Read the integer value of `v`
`void atomic_set( atomic_t *v, int i )`	Set `v` equal to `i`
`void atomic_add( int i, atomic_t *v )`	Add `i` to `v`
`void atomic_sub( int i, atomic_t *v )`	Subtract `i` from `v`
`void atomic_inc( atomic_t *v )`	Add 1 to `v`
`void atomic_dec( atomic_t *v )`	Subtract 1 from `v`
`int atomic_sub_and_test( int i, atomic_t *v )`	Subtract `i` from `v`; return true if 0; otherwise false
`int atomic_add_negative( int i, atomic_t *v )`	Add `i` to `v`; return true if negative; otherwise false
`int atomic_dec_and_test( atomic_t *v )`	Decrement `v` by 1; return true if 0; otherwise false
`int atomic_inc_and_test( atomic_t *v )`	Increment `v` by 1; return true if 0; otherwise false
`void set_bit( int n, void *addr )`	Set the n^{th} bit starting from `addr`
`void clear_bit( int n, void *addr )`	Clear the n^{th} bit starting from `addr`
`void change_bit( int n, void *addr )`	Flip the value of the n^{th} bit starting from `addr`
`int test_and_set_bit( int n, void *addr )`	Set n^{th} bit starting from `addr`; return previous value
`int test_and_clear_bit( int n, void *addr )`	Clear n^{th} bit starting from `addr`; return previous value
`int test_and_change_bit( int n, void *addr )`	Flip n^{th} bit starting from `addr`; return previous value
`int test_bit( int n, void *addr )`	Return value of n^{th} bit starting from `addr`

Linux Spinlocks

Another technique for protecting a critical section in Linux is the spinlock.

This is a handy way to implement constant checking to acquire a lock.

Unlike semaphores where the process is blocked if it fails to acquire the lock, a thread will constantly try to acquire the lock.

The implementation is an integer that is checked by a thread.

Linux Spinlocks

This is very inefficient; it would be better to let another thread execute.

Except where the amount of time waiting on the lock is less than it would take to block the process, switch to another, and unblock it when the value changes.

```
spin_lock( &lock )
    /* Critical Section */
spin_unlock( & lock )
```

Linux Reader-Writer Spinlock

There are reader-writer-spinlocks.

Allow multiple readers but give exclusive access to a writer.

This is implemented as a 24-bit reader counter and an unlock flag:

Counter	**Flag**	**Interpretation**
0	1	The spinlock is released and available.
0	0	The spinlock has been acquired for writing.
n ($n > 0$)	0	The spin lock has been acquired for reading by *n* threads.
n ($n > 0$)	1	Invalid state.