Lecture 21 — Condition Variables, Monitors, Atomic Types

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

May 2, 2021

Condition Variables

Condition variables are another way to achieve synchronization.

Condition Variables

How do we know if some condition has been fulfilled?

Signal on a semaphore?

Lock mutex, read variable(s), unlock mutex?

Or: a Condition Variable.

Condition Variable

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

What differentiates it: broadcast!

We have the option, when an event occurs, to signal either one thread waiting for that event to occur, or to broadcast (signal) to all threads waiting for the event.

Condition Variable Syntax

```
pthread_cond_init( pthread_cond_t *cv, pthread_condattr_t *attributes );
pthread_cond_wait( pthread_cond_t *cv, pthread_mutex_t *mutex );
pthread_cond_signal( pthread_cond_t *cv );
pthread_cond_broadcast( pthread_cond_t *cv );
pthread_cond_destroy( pthread_cond_t *cv );
```

As with other pthread functions we've seen there are create and destroy calls.

Signal is self-explanatory; but broadcast is new and wait looks weird!

Wait, in Two Parts

Condition variables are always used in conjunction with a mutex.

`pthread_cond_wait` takes the condition variable and a mutex.

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

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.

Then, manually unlock when finished.

HEY GUYS!!! GUESS WHAT!!!

`pthread_cond_broadcast` signals all threads waiting on that condition variable.

It's this "broadcast" idea that makes the condition variable more interesting than the simple "signalling" pattern we covered much earlier on.

Condition Variable Example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define COUNT_LIMIT 12

int count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void* inc_count( void* arg ) {
  for (int i = 0; i < 10; i++ ) {
    pthread_mutex_lock( &count_mutex );
    count++;
    if ( count == COUNT_LIMIT ) {
      printf( "Condition␣Fulfilled!\n" );
      pthread_cond_signal( &count_threshold_cv );
      printf( "Sent␣signal.\n" );
    }
    pthread_mutex_unlock( &count_mutex );
  }
  pthread_exit( NULL );
}
```

Condition Variable Example

```
void* watch_count( void *arg ) {
  pthread_mutex_lock( &count_mutex );
  if ( count < COUNT_LIMIT ) {
    pthread_cond_wait( &count_threshold_cv, &count_mutex );
    printf( "Watcher␣has␣woken␣up.\n" );
    /* Do something useful here now that condition is fulfilled. */
  }
  pthread_mutex_unlock( &count_mutex );
  pthread_exit( NULL );
}
```

Condition Variable Example

```
int main( int argc, char **argv ) {
  pthread_t threads[3];

  pthread_mutex_init( &count_mutex, NULL );
  pthread_cond_init ( &count_threshold_cv, NULL );

  pthread_create( &threads[O], NULL, watch_count, NULL );
  pthread_create( &threads[1], NULL, inc_count, NULL );
  pthread_create( &threads[2], NULL, inc_count, NULL );

  for ( int i = O; i < NUM_THREADS; i++ ) {
    pthread_join(threads[i], NULL);
  }

  pthread_mutex_destroy( &count_mutex );
  pthread_cond_destroy( &count_threshold_cv );
  pthread_exit( NULL );
}
```

Anybody Listening?

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

This is sometimes called the “lost wakeup problem”, because threads don’t get woken up if they weren’t waiting for this.

That’s usually an error, but it might be acceptable.

Can We Apply This To What We Know?

The condition variable with broadcast can be used to replace some of the synchronization constructs we've seen already.

What patterns could be replaced?

Broadcast Barrier

Consider the barrier pattern from earlier.

There are *n* threads and we wait for the last one to arrive.

```
1. wait( mutex )
2. count++
3. if count == n
4.     post( barrier )
5. end if
6. post( mutex )
7. wait( barrier )
8. post( barrier )
```

```
1. wait( mutex )
2. count++
3. if count < n
4.     cond_wait( barrier, mutex )
5. else
6.     cond_broadcast( barrier )
7. end if
8. post( mutex )
```

The `wait` takes place before the post on `mutex`. That looks strange, doesn't it?

Spidey Sense Tingling?

We give up the mutex `lock` when we wait on the condition variable.

The fact that we don't get to the unlock statement first does not cause a problem.

So we are alright.

The last thread doesn't wait on the condition at all because there's no need to!

It knows that it is last and there's nothing to wait for so it should proceed.

Barrier Pattern Code Example

```
int count;
pthread_mutex_t lock;
pthread_cond_t cv;

void barrier( ) {
  pthread_mutex_lock( &lock );
  count++;
  if ( count < NUM_THREADS ) {
    pthread_cond_wait( &cv, &lock );
  } else {
    pthread_cond_broadcast( &cv );
  }
  pthread_mutex_unlock( &lock );
}
```

Will this work?

Uh… wait… this isn't the right kind.

What's a Monitor?

A condition variable can be used to create a monitor, a higher level synchronization construct.

This is a bit like in OOP the idea of a class.

When we use a monitor we are packaging up the shared data and operations on that data.

Eliminate the need to write synchronization code ourselves.

Did We Forget Something?

```
void foo( ) {
  pthread_mutex_lock( &l );
  /* Read some data */
  if ( condition ) {
    printf( "Cannot continue due to reasons...\n" );
    return;
  }
  /* More stuff */

  pthread_mutex_unlock( &l );
}
```

Control Flow Problems

There is control flow that could lead to exiting this function `foo` without unlocking the mutex `l`.

It looks super obvious in this case, but in real life it's harder to see!

Java Anyone?

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

In Java we can declare a method to be `synchronized`.

There is a lock created around that method. Locking and unlocking is handled automatically.

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

Synchronized Blocks

in Java one can also define a block as synchronized:

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

This sort of “automatic” locking and releasing is intended to simplify the process of writing multithreaded code.

This isn’t an endorsement of Java...

Monitors don’t have to be written in Java (or similar).

They do frequently appear in Object-Oriented languages because the concepts are familiar enough.

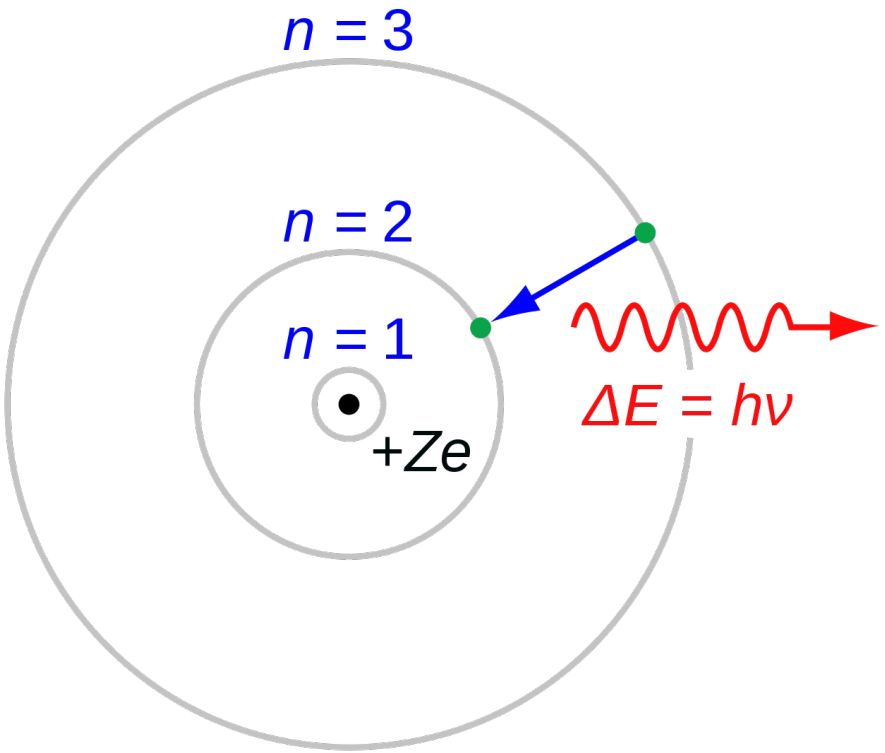

Image Credit: Wikipedia user JabberWok

A testament to how good humans are at blowing things up…

Atomic Types

Frequently we have a code pattern that looks something like this:

```
pthread_mutex_lock( lock );
shared_var++;
pthread_mutex_unlock( lock );
```

It seems like a lot of work to lock and unlock the mutex, no?

Thinking back to the “test and set” type of instruction from earlier, wouldn’t it be nice if we could do that sort of thing for something like incrementing a variable?

Hey glibc, Could You Please...

The GNU (Linux) standard C library (`glibc`) provides operations that are guaranteed to execute atomically, to avoid simple race conditions

Where possible, the compiler will try to turn these into uninterruptible hardware instructions.

Otherwise a function that has locking will be used to implement the atomic nature.

You Can't Take It With You

These are, however, glib specific, and not necessarily available or portable.

In the C11 (2011) standard, atomic types were finally introduced as part of the language specification.

In the specification, we see `type` as the type.

In its place you would use an `int` for an integer.

A valid type is one that 1, 2, 4, or 8 bytes in length (integral or pointer).

To set a value:

```
type __sync_lock_test_and_set( type *ptr, type value );
```

The following functions are used to swap two values, only if the old value matches the expected (i.e., what was provided as the second argument):

```
bool __sync_bool_compare_and_swap( type *ptr, type oldval, type newval );
type __sync_val_compare_and_swap( type *ptr, type oldval, type newval );
```

Change and Return

The following functions perform the operation and return the *old* value:

```
type __sync_fetch_and_add( type *ptr, type value );
type __sync_fetch_and_sub( type *ptr, type value );
type __sync_fetch_and_or( type *ptr, type value );
type __sync_fetch_and_and( type *ptr, type value );
type __sync_fetch_and_xor( type *ptr, type value );
type __sync_fetch_and_nand( type *ptr, type value );
```

The following functions perform the operation and return the *new* value:

```
type __sync_add_and_fetch( type *ptr, type value );
type __sync_sub_and_fetch( type *ptr, type value );
type __sync_or_and_fetch( type *ptr, type value );
type __sync_and_and_fetch( type *ptr, type value );
type __sync_xor_and_fetch( type *ptr, type value );
type __sync_nand_and_fetch( type *ptr, type value );
```

Reading is... Fun?

Interestingly, for x86 there is no atomic read operation.

The (normal) read itself is atomic for 32-bit-aligned data.

This behaviour is specific to x86 and we try to avoid that.

If we do rely on this, however, we could get an out-of-date value.

If you want to really be sure you did get the latest, you can use one of the above functions and add or subtract 0.

Use Tool Appropriately

```
struct point {
  volatile int x;
  volatile int y;
};
__sync_lock_test_and_set( p1->x, 0 );
__sync_lock_test_and_set( p1->y, 0 );

/* Somewhere else in the program */
__sync_lock_test_and_set( p1->x, 25 );
__sync_lock_test_and_set( p1->y, 30 );
```

Does this work?

Use Tool Appropriately

Although the set of each of x and y is atomic, the operation as a whole is not.

We could see invalid states, like (25, 0) or (0, 30).

You Spin Me Right Round...

Another common 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.

When would we want this behaviour?

When It's Worth It!

It would be better to let another thread execute.

Except when the amount of time waiting on the lock might be small.

Specifically, 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 )
```

In addition to the regular spinlock, there are *reader-writer-spinlocks*.

Like the readers-writers problem discussed earlier, the goal is to allow multiple readers but give exclusive access to a writer.

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.