

# Lecture 23 — Condition Variables, Monitors, Atomic Types

Jeff Zarnett  
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering  
University of Waterloo

March 23, 2019



Condition variables are another way to achieve synchronization.

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.

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.

---

```
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!

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 );
}
```

---

---

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

    for ( int i = 0; i < NUM_THREADS; i++ ) {
        pthread_join(threads[i], NULL);
    }

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

---

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?

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?

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.



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 );  
}
```

---

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!

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  
}
```

---

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.

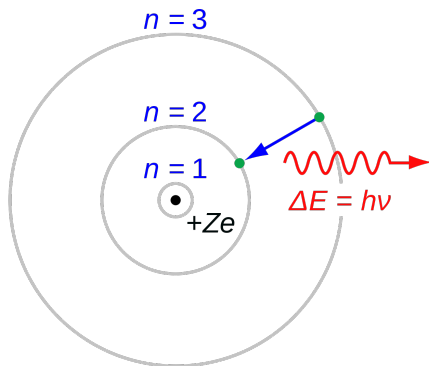


Image Credit: Wikipedia user JabberWok

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

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 Kernel, Could You Please...

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

In a uniprocessor system, the CPU cannot be interrupted until the operation is finished.

In a multiprocessor system, the variable is locked until the operation is finished.

There are two types of atomic operations: those that operate on integers and those that operate on a single bit in a bitmap.

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

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

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

These are Linux-specific and not portable (necessarily).

---

```
struct point {  
    atomic_t x;  
    atomic_t y;  
};  
atomic_set( p1->x, 0 );  
atomic_set( p1->y, 0 );  
  
/* Somewhere else in the program */  
atomic_set( p1->x, 25 );  
atomic_set( p1->y, 30 );
```

---

Does this work?

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

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

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?

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.